Testing in context: Efficiency and executability

Lihua Duan
University of Windsor

Testing in Context: Efficiency and Executability

by

Lihua Duan

A Dissertation
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy at the
University of Windsor

Windsor, Ontario, Canada
2009

Your file Votre référence
ISBN: 978-0-494-57572-7
Our file Notre référence
ISBN: 978-0-494-57572-7

Declaration of Co-Authorship / Previous Publication

I. Co-Authorship Declaration

I hereby declare that this dissertation incorporates material that is the result of joint research undertaken in collaboration with my advisor, Dr. Jessica Chen. The collaboration is covered in Chapter 7 of the dissertation.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my dissertation, and have obtained written permission from each of the co-author(s) to include the above material(s) in my dissertation.

I certify that, with the above qualification, this dissertation, and the research to which it refers, is the product of my own work.

II. Declaration of Previous Publication

This dissertation includes one original paper that have been previously published/submitted for publication in peer reviewed conference proceedings, as follows:

Dissertation Chapter	Chapter 7
Publication title /full citation	Lihua Duan and Jessica Chen. An Approach to Testing with Embedded Context using Model Checker. Proc. of the Ninth International Conference on Formal Engineering Methods (ICFEM'08), Lecture Notes in Computer Science Vol. 5256, pp. 66-85, 2008. Springer-Verlag.
Publication status	published

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my dissertation. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my dissertation does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my dissertation,

Abstract

Testing each software component in isolation is not always feasible. We consider testing a deterministic Implementation Under Test (IUT) together with some other correctly implemented components as its context. One of the essential issues of testing in context is test executability problem, i.e., tests generated solely from the specification of the IUT may not be executable due to the uncontrollable interaction between the IUT and its context. On the other hand, generating a test sequence from the abstract specifications of a stateful IUT and its context often suffers from the well-known state explosion problem. In this dissertation, we solve the problem of generating a minimal-length test sequence from a given specification of a stateful IUT and its embedded context. By adopting model checking techniques, we avoid the state explosion problem during test generation and avoid the test executability problem during testing in context.

Keywords: finite state machines, conformance testing, test generation, testing in context, test sequences.

This work is dedicated to my parents, Zhanyi Duan and Yubing Lian, who have been teaching me the value of life.

Acknowledgments

First and foremost, I would like to express my heartfelt thanks to my advisor, Dr. Jessica Chen, for her invaluable guidance, extensive time, extreme patience, and enthusiastic encouragement during my entire graduate studies. Without her help, the work presented here would not have been possible. I appreciate her for all she has done for me and it will always be held deeply in my memory.

I would like to thank my internal committee members, Dr. Arunita Jaekel, Dr. Jianguo Lu, and Dr. Huapeng Wu for attending a series of my research seminars and giving their precious comments and suggestions to the research work through all my Ph.D study. I would like also to express my appreciation to my external examiner, Dr. Wuwei Shen, for his zealous help of examining my dissertation work.

My special thanks go to the secretary at the School of Computer Science, Ms. Mandy Dumouchelle, for her consistent help.

Last, but not least, I would like to thank all my friends for their support and encouragement for my studies.

Contents

List of Tables

List of Figures

1 Introduction

Finite State Machines (FSMs) have been widely used to model the abstract behavior of sequential circuits [32, 74], lexical analysis systems [51], and more recently, communications protocols [1, 11, 63, 85]. Furthermore, some more expressive specification languages such as Specification and Description Languages (SDLs) [50], Estelle [49], and Statecharts [69] are based on *extensions* of FSMs. The demand of ensuring the correctness of computer systems motivates the research on *conformance testing* in a setting where system specifications are given in FSMs or FSM-based languages [1, 6, 11, 32, 38, 59, 63, 77, 80, 85].

Given an *implementation under test* (IUT) for which we can only observe its input/output behavior, conformance testing can be conducted to improve our confidence that this implementation *conforms* to its specification. Conformance testing is often carried out by i) constructing from the specification of the system a *test sequence*, which is an input sequence with an expected output sequence; ii) applying the input portion of this sequence to the IUT, which is considered as a *black box*, according to the given test architecture; and iii) determining whether the actual output sequence is produced as expected.

Given specification M describing the expected behavior of the IUT, we can imagine that the IUT behaves according to a certain abstract machine N in the same format. In this setting, conformance testing amounts to establishing the correspondence between M and N. In doing so, it is essential to understand the fault models. In the following, for convenience, we also use the IUT to represent the implementation FSM N.

A faulty IUT falls into one of the following categories.

- output faults: the IUT produces an incorrect output in response to an input in a state.
- transfer faults: the IUT ends at an incorrect state after applying an input sequence.
- hybrid faults: the IUT has both output faults and transfer faults.

Based on these fault models, a series of fault coverage criteria and test generation methods have been proposed, see [53, 56] for comprehensive surveys on this topic.

Testing an IUT in isolation is not always feasible in the unit testing. There are situations when we have to test the IUT together with some other components. As pointed out in [73], this can be the case when the IUT is an embedded component of a complex system, called a *context* of the IUT, only through which the IUT can be accessed. As another example, suppose we want to test a web-based implementation WS_1, which makes use of web service WS_2. Due to the difficulty in providing input and observing output all encapsulated according to certain protocol such as SOAP [91], testing WS_1 invokes the necessity of activating WS_2. Here, again, WS_2 is considered as the *context* of WS_1. In general, the context can be the system components, the drivers, the stubs, the test beds, and so on.

Obviously, it is worthful to study how to test an IUT within its context. Petrenko et al. first presented a test generation framework for an embedded IUT whose communication with the environment has to be carried out through its context [72, 73]. In particular, the problems of *test executability* and *fault propagation* are addressed in the presence of the context. The test executability problem describes the situation where a test sequence generated from a given specification solely without taking into account the behavior of the context may not be executable when testing in context, and the fault propagation problem describes the situation where the faults of the IUT are masked by the context. In [23, 24, 57, 70], different approaches are discussed for solving the problem of translating internal tests derived for an embedded component into external observable tests of the entire system.

Different from their application domain, our work is applicable to an IUT with an embedded context, i.e. it does not communicate with any component other than $\mathcal{I}$. In our current work, we consider the problem of FSM-based deterministic testing on $(\mathcal{I}, \mathcal{I}_c)$ which is an IUT implementation $\mathcal{I}$ together with a correct context implementation $\mathcal{I}_c$. The communication port between $\mathcal{I}$ and $\mathcal{I}_c$ is not *controllable* but *observable*. This means that the tester can neither provide input to the IUT using this port nor stop an input from the context to the IUT. It can, however, observe all the input from and all the output to the context. The specification of $\mathcal{I}$ is given in terms of an FSM. $\mathcal{I}_c$ can be either stateless or stateful. When $\mathcal{I}_c$ is stateless, its specification can be given in form of a set of $\langle request, response \rangle$ pairs. When $\mathcal{I}_c$ is stateful, we assume that it is specified by a specification language or a structural modelling language. In both cases, we present methods to generate a minimal-length test sequence that can be used to test $(\mathcal{I}, \mathcal{I}_c)$ without encountering test executability problem during testing.

When the context is stateful, the existing test generation techniques of testing in context often suffer from the state explosion problem. This is caused by requiring the computation of the product of several auxiliary components in addition to the model of the specification of $\mathcal{I}$. The ultimate goal of our work is to avoid generating the operational model of the given specification of $\mathcal{I}_c$ (if a higher level specification is provided) and constructing the global model of $\mathcal{I}$ and $\mathcal{I}_c$. In order to do so, we employ model checking tools to retrieve necessary information from the context specification so that test sequences for $(\mathcal{I}, \mathcal{I}_c)$ can be generated. The idea of using a model checking tools to generate tests is not new. In the literature, various applications have presented. Ammann et al. combined model checking with *mutation analysis* to generate test cases [2]: after a specification model is mutated by applying mutation operators, a model checker generates counter-examples to distinguish the mutant models from the original specification model, and thus test cases are derived. Gargantini and Heitmeyer presented a technique to construct test sequences upon a special class of so-called *Software Cost Reduction* requirements, by using a model checker [26]. In order to save memory from a huge predefined test suite, Tretmans and de Vries [17] used model checker SPIN to generate tests *during testing* for non-deterministic stateful systems. How to generate test cases according to some *data flow test selection criteria* is discussed in [45]. In [75], Goltz et al. used a model checker to generate a shortest distinguishing sequence of an EFSM.

Note that it is straightforward to extend our work to a more general case where the embedded context consists of a set of components, each having its own port to communicate with $\mathcal{I}$. In terms of applying model checking tools for test generation, we have added one more example along this line of research, particularly for testing in context.

We consider conformance testing of *deterministic* systems in this dissertation. The readers who are interested in conformance testing of *non-deterministic* systems should refer to [25, 35, 36, 58, 71, 82, 96].

The rest of the dissertation is organized as follows. In Chapter 2, we give a brief introduction to FSMs and the related notations and terminologies that will be used later on, followed by a discussion of the fault models of FSMs. The main issues of testing in context are addressed in Chapter 3. Among those issues, four widely used fault coverage criteria together with some existing test generation and optimization techniques are discussed in Chapter 4. In Chapter 5, test executability problem is explained. Test generation techniques for an IUT with stateless and stateful embedded context are presented in Chapter 6 and 7, respectively. In the end, we conclude our work with some final remarks.

2 Finite state machines and related fault models

There are various formalisms to describe the expected behavior of a stateful system. Suitable for different levels of abstractions, they range from formal specification languages such as process algebras, to structural/operational modelling languages such as (input/output) *labelled transition systems (LTSs)* and *Finite State Machines (FSMs).* Here, we use FSMs to show the main issues related to testing in context.

2.1 Finite state machines

There are two types of FSMs: Mealy machines [61] and Moore machines [66]. The difference between them lies in how an output is determined: For the former, an output is determined by the current state and an input; while for the latter, an output is determined by the current state alone (not directly by an input). Usually, the number of states in a Moore machine is greater than or equal to that in an equivalent Mealy machine. We adopt Mealy machines since they are more natural to model software systems. As mentioned in the Introduction, we consider deterministic FSMs. In order to explicitly associate each input and output with a *port* (an *interface* to communicate with a certain component), we use *n-port FSM* to describe the abstract behavior of the systems with n ports.

Definition 1 (Finite state machines) *A deterministic n-port Finite State Machine (also called finite state machine for short) is defined by a tuple* $(S, I, O, \delta, \lambda, s_0)$.

- *S is a finite set of states where $s_0 \in S$ is its initial state.*

- *$I = \bigcup_{i=1}^{n} I_i$, where I_i is the input alphabet of port i $(i = 1, \ldots, n)$.*

  Being abstract, these input symbols encapsulate the information of the communication channels. Thus, without loss of generality, we can assume that the input symbols at different ports are distinct, i.e. $I_i \cap I_j = \emptyset$ for $i \neq j$.

- *$O = \Pi_{i=1}^{n} O_i$ where O_i is the output alphabet of port i $(i = 1, \ldots, n)$.*

  Each $o \in O$ is a vector of outputs denoted by $o = \langle o_1, \ldots, o_n \rangle$ where $o_i \in O_i$ for $i = 1, \ldots, n$. We do not consider the order in which we observe output o_i and o_j at different ports. When there is no output at a port i, we use a distinct symbol $-$ to denote it.

- *δ is the transition function that maps $S \times I$ to S, and*

  λ is the output function that maps $S \times I$ to O.

A "slow environment" assumption is usually used in the literature. That is, whenever an input reaches the system, the system will always prompt the output for it before the second input can reach the system.

The inputs and the outputs are abstract symbols. The discussions on data types and complicate data structures in the inputs and outputs are beyond the scope of this dissertation.

Note that functions λ and δ can be partial, i.e., it is possible that there exists $i \in I$ for some $s \in S$ such that $\lambda(s, i) = null$ and $\delta(s, i) = null$. We will use $\delta(s, x) = null$ to denote that there is no image of δ for the given state s of S and the given input x of I. In this case, we also have $\lambda(s, x) = null$. Furthermore, we extend the input of λ and δ from an input alphabet to a sequence of input alphabets with their meanings obtained straightforwardly from the original ones.

For simplicity, we assume the number of states of M is n and the states of M are enumerated, giving $S = \{s_0, \ldots, s_{n-1}\}$.

A *transition* t is defined by a tuple $(s_1, s_2, x/y)$ in which s_1 is the *starting state*, x is the input, $s_2 = \delta(s_1, x)$ is the *ending state*, and $y = \lambda(s_1, x)$ is the output. The input/output x/y is called the *label* of t. Note that when an FSM has only one port or there is exactly one output for each transition of an FSM, we use a single output alphabet instead of an output vector to denote the output for simplicity.

Let t_i be a transition for $1 \leq i \leq k$. A *path* $\rho = t_1\ t_2\ \ldots\ t_k$ is a finite sequence of transitions such that for $k \geq 2$, the ending state of t_i is the starting state of t_{i+1} for all $1 \leq i \leq k-1$. A *tour* is a path whose starting state and ending state are the same. For convenience, we use *start*(ρ), *end*(ρ), *label*(ρ), and *in*(ρ) to denote the starting state, the ending state, the label, and the input portion of the label of ρ, respectively.

Let ρ_1 and ρ_2 be two paths of M. When *end*(ρ_1) and *start*(ρ_2) are the same, we use $\rho_1\rho_2$ to denote the *concatenation* of ρ_1 and ρ_2. For clarity, sometimes we also use $\rho_1 \circ \rho_2$ for $\rho_1\rho_2$. For $\rho_1 = (s_1, s_h, T_1)$ and $\rho_2 = (s_h, s_r, T_2)$, we have $\rho = \rho_1 \circ \rho_2 = (s_1, s_r, T_1 \circ T_2)$.

A state $s \in S$ is *reachable* if there exists a path starting from s_0 and ending at s. We consider FSMs where all states are reachable.

An FSM is *completely specified* if functions λ and δ are *total*; otherwise, it is *partially specified*. When an FSM M is not completely specified, it is possible to make M completely specified. Two typical ways of doing so are named after [16].

- *angelic completion*: for any $(s, x) \notin domain(\delta)$, add transition $(s, s, x/null)$.
- *demonic completion*: i) add an erroneous state s_{err}; ii) for any $(s, x) \notin domain(\delta)$, add transition $(s, s_{err}, x/null)$; and iii) for any $x \in I$, add transition $(s_{err}, s_{err}, x/null)$.

The completion, however, slightly changes the meaning of the FSM and is not always acceptable.

Two states s_i and s_j are *equivalent* if, for every input sequence σ, $\lambda(s_i, \sigma) = \lambda(s_j, \sigma)$. If $\lambda(s_i, \sigma) \neq \lambda(s_j, \sigma)$ then σ *distinguishes* between s_i and s_j. An FSM M is *minimal* if every state can be reached from the initial state of M and no two states of M are equivalent. Since only deterministic FSMs are considered, we can easily obtain a minimal FSM from any given FSM [27, 46]. In the following, we assume that all given FSMs are minimal.

When the specification of an IUT is given in the form of an FSM, we would like to automatically generate an efficient and effective *test sequence* from it. Here, a *test sequence* refers to an input sequence, which is typically obtained from a path of the given specification FSM. That is, our goal is usually to find a path in the given specification FSM such that

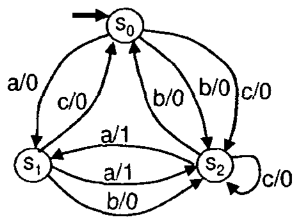


Figure 1: An example FSM M_0

transition	starting state	ending state	label
t_0	s_0	s_1	$a/0$
t_1	s_0	s_2	$b/0$
t_2	s_0	s_2	$c/0$
t_3	s_1	s_2	$a/1$
t_4	s_1	s_2	$b/0$
t_5	s_1	s_0	$c/0$
t_6	s_2	s_1	$a/1$
t_7	s_2	s_0	$b/0$
t_8	s_2	s_2	$c/0$

Table 1: Transitions in M_0

the input portion of this path is the desired test sequence and the output portion of this path is the expected output sequence.

Example 1 An example 1-port FSM M_0 is given in Figure 1. Here, $S = \{s_0, s_1, s_2\}$, $I = \{a, b, c\}$, and $O = \{0, 1\}$. Transitions in M_0 are listed in Table 1. ◇

2.2 Fault models

Fault models can serve as a guide to test generation and fault coverage analysis, as claimed in [89]. When a specification M and its IUT have the same input alphabet and output alphabet, faulty IUTs can be classified into three types.

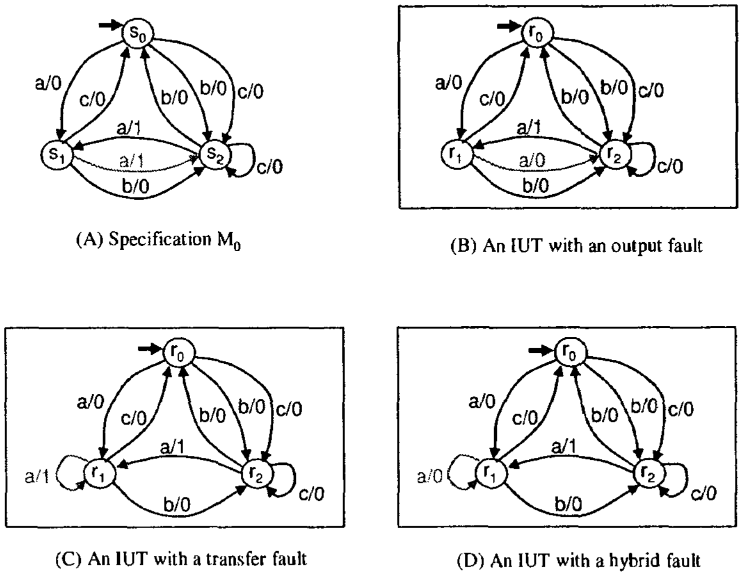
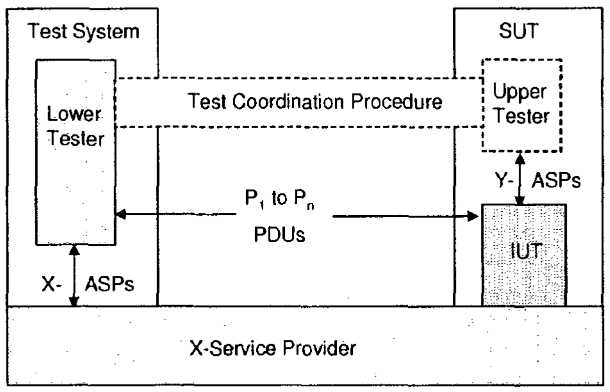

(A) Specification M_0
(B) An IUT with an output fault
(C) An IUT with a transfer fault
(D) An IUT with a hybrid fault

Figure 2: Faulty IUTs of example FSM M_0

- An IUT has only *output faults* if M can be obtained from the IUT by changing the outputs of one or more transitions in the IUT.
- An IUT has only *transfer faults* if M can be obtained from the IUT by changing the ending states of one or more transitions in the IUT.
- An IUT has *hybrid faults* if M can be obtained from the IUT by changing the outputs and/or the ending states of one or more transitions in the IUT.

Here, we do not consider the fault type of *extra states*, i.e., the number of states of the implementation FSM will not exceed that of the specification FSM. We argue that this assumption is reasonable. As we know, the purpose of the conformance testing is to ensure that the behavior of the implementation *conforms to* the behavior specified by the specification. The existence of the extra states means the existence of the extra behavior which is not specified by the specification, and thus will not be tested.

Example 2 Figure 2 shows three faulty IUTs of M_0. The shaded area surrounding an IUT represents the black box where only inputs and outputs can be observed. When transition $(s_1, s_2, a/1)$ in M_0 is concerned, IUTs shown in Figure 2(B), (C), and (D) have an output



(N) - ASP: abstract N-service primitive, an implementation-independent description of an interaction between a service-user and a service-provider at an (N)-service boundary.
PDU: protocol data unit.

Figure 3: A test architecture of distributed systems [48]

fault, a transfer fault, and a hybrid fault, respectively. ◇

Based on the above fault models, we want to *automatically* generate an *effective and efficient* test sequence from a given specification, i.e., it is desirable to generate a test sequence as short as possible while detecting as many faults as possible. This is known as the *fault coverage* problem and the *test optimization* problem. We will discuss the existing solutions to these two problems in Chapter 4.

3 Main issues in testing in context

Ideally, an IUT can be tested in isolation, i.e., a tester can apply a desired input directly to the IUT and observe the actual output produced by the IUT directly. In practice, however, it is not always feasible: the IUT is often tested through its environment, called *context*. For example, in the distributed test architecture shown in Figure 3, the underlying network is the context of the IUT since it has to be used when the lower tester interacts with the IUT.

According to the ways of the interactions among the tester, the IUT, and the context, the architectures of testing in context can be classified into three types as shown in Fig-

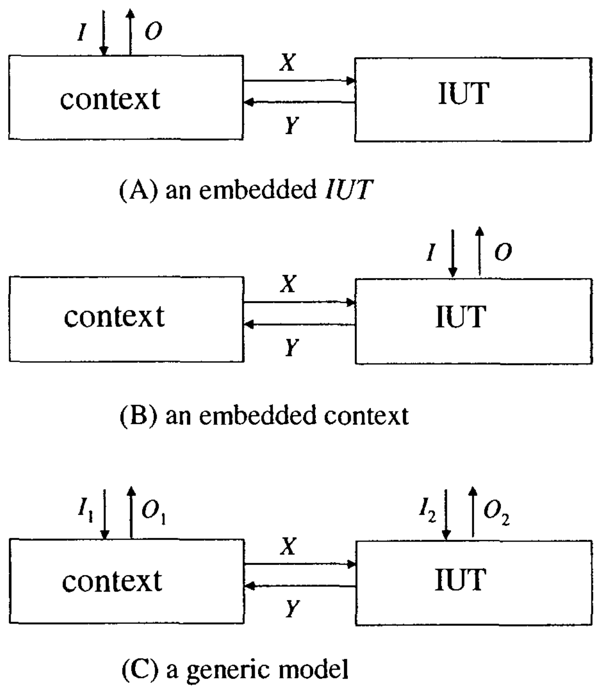


Figure 4: Context-based testing: types of test architecture

ure 4. Here, for simplicity, we demonstrate these architectures by treating the context as one component even though it may include multiple components instead. Petrenko et al. considered the situation where the IUT is an embedded component whose communication with the environment has to be carried out through its context, as shown in Figure 4(A). In [72, 73], they presented a framework of testing an embedded component in context. Along this direction, different approaches [23, 57, 70] are discussed for solving the problem of translating internal tests derived for an embedded component into external observable tests of the entire system. Different from their test architecture, we consider how to test an IUT that is associated with an embedded context as shown in Figure 4(B) [21].

The existence of the context may introduce additional problems for testing. In the previous example, when the underlying network is not transparent, in the sense that it has its own behavior, it is possible that both the inputs applied from the lower tester and the outputs produced by the IUT are modified by the underlying network. Consequently, the validity of the testing is problematic. Thus, the behavior of the context of the IUT has to be considered for the test generation. In the following, we first introduce the *external equivalence* for context-based systems and then discuss the main issues in this setting.

3.1 External equivalence

Let $\mathcal{K}$ be a system consisting of a finite set of FSMs $M_i = (S_i, s_{i,0}, X_i, Y_i, \delta_i, \lambda_i)$, where $i = 1, \ldots, k$. Here, we assume all the actions in X_i, Y_i are distinct. Suppose $I \subseteq \bigcup_i X_i$ and $O \subseteq \bigcup_i Y_i$ are the sets of the *external inputs* and the *external outputs* regarding the entire system, respectively. For $\mathcal{K}$ to be a meaningful system, we have the following assumptions.

- $I \cap (\bigcup_i Y_i) = \emptyset$, i.e., an external input cannot be produced by any component FSM.
- $O \cap (\bigcup_i X_i) = \emptyset$, i.e., an external output cannot be accepted by any component FSM.
- $(\bigcup_i Y_i) \setminus O \subseteq (\bigcup_i X_i) \setminus I$, i.e., any internal output should be accepted by some component FSM.

We say FSM M_i *communicates with* FSM M_j if there exists an internal action in set $Y_i \cap X_j$. The communication among component FSMs can be either *synchronous* or *asynchronous*. We assume that the communication channels are reliable. A global FSM of a asynchronous/synchronous communication can be composed by performing reachability analysis [7, 62, 90, 93, 97]. In black-box testing, we are particularly interested in synchronous composition, where all the internal actions are hidden and only external inputs and outputs are indicated. In the following, we use operator $\times$ to denote the synchronous product of component FSMs.

In the realm of deterministic FSMs, two FSMs are *trace equivalent* if for any input sequence, they produce the same output sequence in response. For testing in context, *external equivalence* is defined by taking into account the behavior of the context. Note that the following definition is adopted from [73] with slight modification.

Definition 2 (External equivalence) *Let S_1 and S_2 be two FSMs, and C their context FSM. S_1 is externally equivalent to S_2 w.r.t. C, denoted by $S_1 =_C S_2$, if $S_1 \times C$ is trace equivalent to $S_2 \times C$.*

The goal of testing in context is to ensure that the IUT is *externally equivalent* to its specification in a given context. Test generation aiming at ensuring *external equivalence* needs to take into account the behavior of the context.

3.2 Main issues

Many issues arise for testing in context.

- stateless v.s. stateful. When the IUT and its context are stateful, test generation may suffer from the state explosion problem. Specification languages such as FSMs, Extended FSMs, and Labeled Transition Systems are often used to specify stateful systems. In this work, we consider the situation where the stateful IUT is specified by an FSM while its embedded context is either stateless or stateful. In the latter case, we require that the context be specified by a specification language that can be translated to *Extended Finite State Machines* (EFSMs), which is a concise specification formalism that allows the use of variables.
- deterministic v.s. non-deterministic. Studies on both deterministic testing and non-deterministic testing have practical significance and confront different challenges. We are particularly interested in test generation techniques of deterministic systems because two benefits can be provided.
  - A high level confidence on the correctness of the IUT can be ensured by applying a test sequence whose length is polynomial to the size of the IUT. For example, trace equivalence can be guaranteed with a *checking sequence* whose length is polynomial to the number of transitions under certain conditions. The detailed discussion on this regard is in Chapter 4. On the other hand, test sequences of non-deterministic systems are often of infinite length [82] or much more costly by requiring to repeat the testing for many times with the fairness hypothesis that all the possible paths with non-deterministic choices are tested, which it actually can not be always guaranteed.
  - If we can derive from deterministic testing that components I_1 and I_2 are trace equivalent to their respective specification P_1 and P_2, then under certain circum-

stances we know $I_1||I_2$ is a correct implementation of $P_1||P_2$ without performing further *integration testing* or *system testing* which involves nondeterministic system specifications.

In this work, we consider the case when both the IUT and its context are deterministic.

- test criteria. Many fault coverage criteria have been proposed in the literature. In Chapter 4, four widely used criteria for FSM-based test sequence generation are explained, namely, the T-method, the U-method, the D-method, and the W-method. In this work, we choose the T-method in Chapter 6 and the W-method in Chapter 7 to demonstrate our proposed methods. Other criteria are applicable with proper adaptation.

- optimization on the efficiency. It is always desirable to reduce the time complexity to generate tests and to reduce the time to carry out the testing while a certain desired fault coverage is satisfied. The existing optimization techniques for different fault coverage criteria are discussed in Chapter 4.

- internal observer v.s. external observer only. An internal observer is an observer who can passively observe the interactions between IUT and its context although it has no control on them. An external observer is actually a tester who has the control to give the inputs and observe the outputs. When an internal observer is available, better testing results can be achieved by making use of the knowledge obtained by observing the internal interactions. We assume the internal observer is available.

- fault propagation. The problem of *fault propagation* describes the fact that some faults of the IUT can be tolerated by the context. There are two types of reasons resulting in the tolerance: i) the selected test sequence is incomplete in the sense that it can not distinguish all the faulty implementation; and ii) some faults are *intrinsically* tolerated by the context such that *no tester* can detect them. Note that the fault propagation problem does not exist when an internal observer is available. Thus, we do not consider fault propagation as we assume the internal observer.

- test executability. *Test executability* problem describes the situation where a test sequence generated solely from a given specification without taking into account the behavior of the context may not be executable when we carry out testing in context. This problem is caused by the uncontrollable interaction between IUT and its context during testing. This problem is crucial in testing in context, and we have a detailed discussion in Chapter 5.

4 Fault coverage and test optimization

It is well known that the exhaustive testing is impossible in practice, and a tester has to make a tradeoff between the fault coverage and the cost. When the IUT has certain properties, it is possible to utilize these properties to maximize the fault coverage. These desirable properties include the reliable reset, the existence of some special input sequences which can be used to identify the states in the IUT, etc. In this chapter, we introduce four typical test generation methods along with the corresponding optimization techniques. Namely, these methods are the T-method [68], the U-method [1, 63, 76], the D-method [29, 32, 85], and the W-method [11]. Actually, these methods can also serve as fault coverage criteria.

4.1 Graph representations of FSMs

Since most of the algorithms for test sequence generation are based on some well-known algorithms in graph theory, in the following, we introduce the graph representation of an FSM and several typical problems in graph theory.

Each FSM M has a graph representation $G = (V, E, L)$, in which a state of M is represented by a vertex from V and a transition of M is represented by an edge from E. We use G_M to denote the graph representation of FSM M, where state s_i is represented by vertex v_i, and transition from s_i to s_j with label x/y is represented by edge $(v_i, v_j, x/y)$.

Terminologies and notations defined for FSMs are naturally extended to their graph representations.

A digraph is *strongly connected* if for any ordered pair of vertices (v_i, v_j) there is a path from v_i to v_j. When G is strongly connected, a *Postman Tour* of G is a tour which contains every edge of E at least once. Given digraph $G = (V, E, cost)$, where *cost* is a cost function that associates each edge in E with a cost, we say G is a weighted digraph. The *Chinese Postman Problem (CPP)* is to find the minimum-cost Postman Tour in a strongly connected (weighted) digraph. Given a strongly connected $G = (V, E)$ and $E_1 \subseteq E$, a *Rural Postman Tour* is a tour which contains each edge in E_1 at least once. The *Rural Chinese Postman Problem (RPP)* is to find a Rural Postman Tour with minimum cost. CPP has a polynomial time solution while RPP is in general NP-hard. Various sophisticated heuristics have been proposed in the literature for RPP (see e.g. [22]).

4.2 T-method

The fault coverage criterion specified by the T-method [68] is as follows.

- T-method: The corresponding path of the generated test sequence in the specification FSM M should contain each transition in M at least once.

According to the T-method, a transition is tested to be correct when its output is correct in response to the corresponding input. If a faulty IUT has only output faults, test sequences generated with the T-method are capable of detecting any faults; otherwise, a faulty IUT may not be distinguished. The advantage of the T-method is that shorter test sequences are generated.

Clearly, the optimization problem of generating a minimal-length test sequence can be reduced to Chinese Postman Problem (CPP).

Example 3 Given specification M_0 in Figure 1, a tour ϱ is found:

$$\varrho = t_0 t_3 t_6 t_5 t_1 t_8 t_6 t_4 t_7 t_2 t_7$$

Then a test sequence of length 11 can be derived by concatenating the inputs of ϱ: $\chi = in(\varrho) = aaacbcabbcb$, whose expected output sequence is 01100010000.

The faulty IUTs in Figure 2(B) and (D) can be distinguished by χ since the actual output sequences are 00100010000 and 00000010000, respectively. However, the faulty IUT in Figure 2(C) cannot be distinguished since it yields the same output sequence as expected. $\diamondsuit$

states	UIO sequence
s_0	IntFromU
s_1	ReqFromL $\circ$ RspFromL
s_2	RspFromL $\circ$ PerRspFromU
s_3	PerRspFromU $\circ$ RspFromL
s_4	ReqFromL $\circ$ RspFromL $\circ$ PerRspFromU

Table 2: UIO sequences for each states in M_1

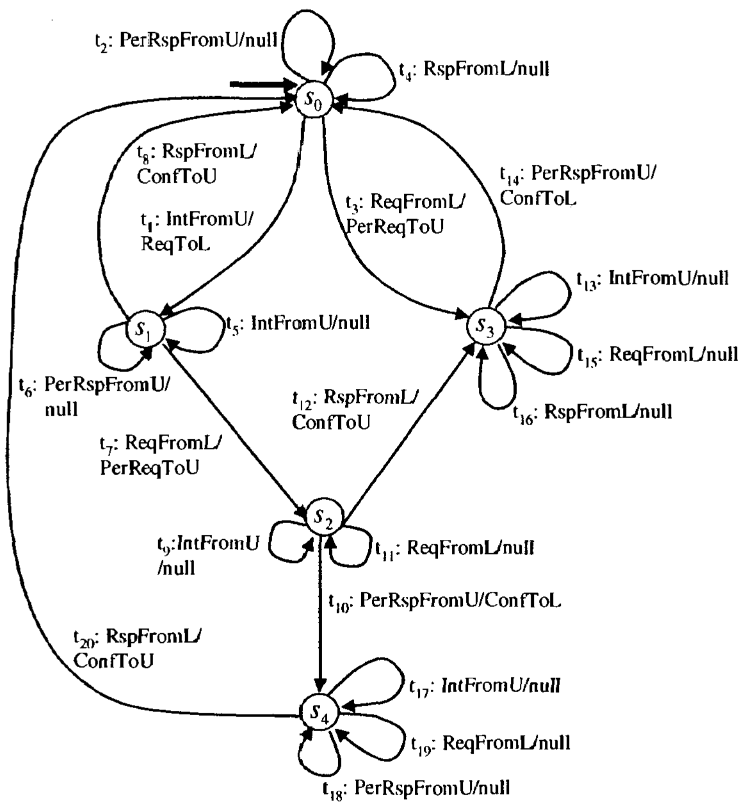
4.3 U-method

The U-method can be applied to a special class of FSMs that have a *Unique Input/Output sequence* (UIO sequence) for each of their states. Given an FSM M, a UIO sequence of a state s is an input sequence such that the corresponding output sequence obtained by applying this input sequence at s in M is unique from those obtained by applying this input sequence at any other state. We use *UIO*$_i$ to denote the UIO sequence for state s_i. Formally,

Definition 3 (UIO sequences) *Given an FSM $M = (S, I, O, \delta, \lambda, s_0)$, an input sequence* UIO$_i$ *is a UIO sequence of state s_i if for any $s_j \in S$, $s_j \neq s_i$ implies $\lambda(s_j, \mathrm{UIO}_i) \neq \lambda(s_i, \mathrm{UIO}_i)$.*

Example 4 We present here a protocol for establishing service connection, which is commonly used in peer-to-peer systems. In this protocol, any participant, upon receiving a request from its user, can initiate a connection with any other peer participant by issuing a *connection request*. The connection will not be established until the confirmations from all peer participants are received. Each confirmation represents the permission from another participant. For simplicity, we consider such a protocol with two participants.

Note that the *connection requests* can be issued concurrently by both participants. That is, the two participants may issue the requests at about the same time. Consequently, it is possible that each participant receives a *connection request* from the other participant right after it has sent out its own request and yet before it receives the confirmation from its



S = { s_0, s_1, s_2, s_3, s_4 }
I = { IntFromU, PerRspFromU, ReqFromL, RspFromL }
O = { ConfToU, PerReqToU, ConfToL, ReqToL }
Note: Symbol "null" means no output is produced.

Figure 5: FSM M_1 of the connection establishment protocol for one participant

partner. In this case, in order to establish a connection, each participant should respond to the request from its partner as well as receive the confirmation from its partner for its own request.

The specification FSM $M_1 = (S, I, O, \delta, \lambda, s_0)$ of a participant in this protocol is shown in Figure 5. The service primitives and their symbolic representations for each participant in this protocol are listed below.

- *IntFromU*: user's intention for establishing a service connection;
- *ReqToL*: message to request the partner to establish connection;
- *RspFromL*: response from the partner for service connection;
- *ConfToU*: confirmation of the service connection to the user;
- *ReqFromL*: request from the partner for service connection;
- *PerReqToU*: request for the user's permission for service connection;
- *PerRspFromU*: user's permission for a service connection;
- *ConfToL*: confirmation of the service connection to the partner.

Suppose process A is a participant of this connection establishment protocol modeled by M_0. I/O pair *IntFromU/ReqToL* means that upon receipt of message *IntFromU*, A will send a request to its partner for the connection establishment. I/O pair *ReqFromL/PerReqToU* represents that when A receives message *ReqFromL*, it will send a request to its user asking for permission.

Table 2 shows the shortest UIO sequences for each state.

$\diamond$

Not every FSM has a UIO sequence for each of its states, and the problem of finding UIO sequences for an FSM is very hard [55]. For a given specification M, the following decision problems are proven to be PSPACE-complete: i) whether a specific state s of M have a UIO sequence; ii) whether all states of M have UIO sequences; iii) whether some of the states of M have UIO sequences. Furthermore, even though a state has a UIO sequence, it is possible that this UIO sequence is of exponential length. In this case, there is no value for testing purpose. Note that these are the worst case result. In practice (e.g. communication protocols), short UIO sequences exist for most cases and can be found quickly [56]. Discussions on finding the UIO sequences from a given FSM can be found in [12, 28, 18, 47, 56, 76].

For conformance testing, UIO_i can be used to *verify* whether an IUT is in a state corresponding to state s_i since the desired output sequence is supposed to be produced when applying UIO_i in s_i. This property can be used to tackle the transfer faults in IUTs in the sense that the ending states of transitions can be verified with UIO sequences. Thus, the U-method is inspired.

The fault coverage criterion specified by the U-method [1, 63, 76] is as follows.

- U-method: The corresponding path of the generated test sequence in the specification FSM M should contain each transition in M with its ending state in the implementation FSM *verified.*

Example 5 Suppose we want to generate a test sequence from specification M_1 in Figure 5 with the U-method. For simplicity, we only consider two transitions, namely, t_9 and t_{20}, as examples.

Let $\rho_9 = t_9t_{12}t_{14}$ and $\rho_{20} = t_{20}t_1$ be two paths in M_1. Since the input portion of $label(t_{12}t_{14})$ is a UIO sequence for the ending state of transition t_9, the ending state of t_9 is verified by applying the input portion of ρ_9. That is, ρ_9 can be used to test the correctness of t_9. We call such a path a *test segment* of t_9. Similarly, ρ_{20} is a test segment of t_{20}. Using transfer sequence $t_1t_7t_{10}$ to connect these two test segments, we get

$$\rho = t_9t_{12}t_{14}t_1t_7t_{10}t_{20}t_1$$

which is a path containing both test segments.

Analogously, a path ϱ in M can be found containing the test segments of all the transitions in M_1. Then the input sequence $in(\varrho)$ is a desired test sequence satisfying the U-method. ◇

4.3.1 Test optimization

As the U-method is effective to detect the transfer faults in the IUTs, it is appealing to study on how to minimize the lengths of the generated test sequences. In the literature, a lot of contribution has been made in this regard [1, 9, 10, 20, 33, 34, 63, 76, 78, 95], and the main ideas of these work are to maximize the *overlaps* among the test segments and to reduce the use of the transfer sequences connecting test segments. In the following, we explain some latest results along this approach.

In [33], Hierons proposed the notion of the *invertible transitions* [1]. A transition $(s_i, s_j, x/y)$ is *invertible* if it is the only transition entering state s_j with input x and output y. In the example FSM M_1, t_1, t_2, t_3 are invertible transitions while t_8 and t_{20} are not because both t_8 and t_{20} end at s_0 with the same label $RspFromL/ConfToU$.

The existence of invertible transitions in existing protocol descriptions has been the major source of the recent success in reducing the lengths of the generated U-sequences. This is based on the following observation ([63, 33]):

$\mathcal{O}$) If t is an invertible transition and UIO_i is a UIO sequence of $end(t)$, then the input sequence $in(t) \circ UIO_i$ is a UIO sequence for $start(t)$.

Suppose that t is an invertible transition, and $t\sigma$ is a test segment for t in the sense that σ is a path induced by applying the UIO sequence of state $end(t)$ at $end(t)$. Now if t' is a transition adjacent to t in the sense that $end(t') = start(t)$, then path $t't\sigma$ is a test segment for t'. As $t't\sigma$ contains test segments for both t' and t, we say there is an overlap between test segment $t't\sigma$ and test segment $t\sigma$. By using invertible transitions, the overlap between test segments is increased. It follows that the length of the generated U-sequence can be reduced.

Some heuristic algorithms have been proposed in [63, 33] to maximize the use of invertible transitions to reduce the lengths of the U-sequences. In doing so, the notion of *invertible transition* is extended to that of *invertible sequence* [34]. A path ρ is an *invertible sequence* if it is the only path with label $label(\rho)$ that ends at $end(\rho)$. That is, for any path

[1] A similar notion called *non-converging edge* was defined on the digraphs that represent the FSMs ([63]).

ρ', $start(\rho) \neq start(\rho')$ implies $end(\rho) \neq end(\rho')$ or $label(\rho) \neq label(\rho')$. Clearly, when the length of an invertible sequence is 1, it is actually an *invertible transition.*

Similar to $\mathcal{O}$), we have the following result [34]:

$\mathcal{O}'$) If ρ is an invertible sequence and UIO_i is a UIO sequence of $end(\rho)$, then the input sequence $in(\rho) \circ UIO_i$ is a UIO sequence of $start(\rho)$.

Note that the additional UIO for $start(\rho)$ obtained from $\mathcal{O}'$) may be longer than the given UIO sequence for $start(\rho)$. For the example in Figure 5, $t_{10}t_{20}$ is an invertible sequence ending at s_0. We know that $UIO_0 = IntFromU$ and $UIO_2 = RspFromL \circ PerRspFromU$. By using invertible sequence $t_{10}t_{20}$, we have another UIO sequence for s_2:

$$UIO_2' = PerRspFromU \circ RspFromL \circ IntFromU.$$

Although this newly found UIO sequence is longer than the given one, it may help to reduce the *total* length of a U-sequence since the test segment it produced has an *overlap* with other test segment(s). Let us use ρ_i to denote the test segment formed by concatenating t_i and the path induced by applying the originally given UIO sequence of $end(t_i)$ at $end(t_i)$. Consider the two test segments for transitions t_9 and t_{20} in M_0. We have $\rho_9 = t_9t_{12}t_{14}$ and $\rho_{20} = t_{20}t_1$. Using transfer sequence $t_1t_7t_{10}$ to connect these two test segments, we get

$$\rho = t_9t_{12}t_{14}t_1t_7t_{10}t_{20}t_1$$

which is a path containing both test segments. The length of ρ is 8. If we use the UIO sequence derived according to $\mathcal{O}'$), one of the test segments for t_9 is $\rho_9' = t_9t_{10}t_{20}t_1$ which contains ρ_{20}. In this case, ρ_9' can be used to verify both t_{20} and t_9 and its length is only 4. With this observation, a heuristic algorithm was given in [34] to use the invertible sequences to reduce the length of U-sequences.

As from $\mathcal{O}$) an optimal solution was derived for finding a minimal-length U-sequence in the special case when *all* transitions in M are invertible, now for general FSMs which may contain both invertible transitions and non-invertible ones, $\mathcal{O}'$) leads to the following idea:

a') Determine a minimal-length path $\varrho = t_0\sigma_1t_1\sigma_2t_2\ldots\sigma_kt_k\sigma_0$, where for $0 \leq i \leq k$, σ_it_i is an invertible sequence and for each $t \in M$, there exists i $(0 \leq i \leq k)$ such that

$t_i = t$. Without loss of generality, we assume t_0 is a transition starting from the initial state s_0.

b') Obtain ρ by removing σ_0 from ϱ and appending path ρ' induced by applying the UIO sequence of $end(t_k)$ at state $end(t_k)$.

Then, $in(\rho)$ can be used as the desired test sequence. This is formally introduced below.

Definition 4 (proximate test path) *Let M be a given FSM. Suppose t_i is a transition in M and σ_i is a path in M ($0 \le i \le k$). A proximate test path of M is $\varrho = t_0\sigma_1 t_1\sigma_2 t_2 \dots \sigma_k t_k \sigma_0$ such that:*

- *t_0 is a transition starting from the initial state s_0;*
- *$\forall i \in \{0, \dots, k\}$, $\sigma_i t_i$ is an invertible sequence;*
- *$\forall t \in M$, $\exists i$ ($0 \le i \le k$) such that $t = t_i$.*

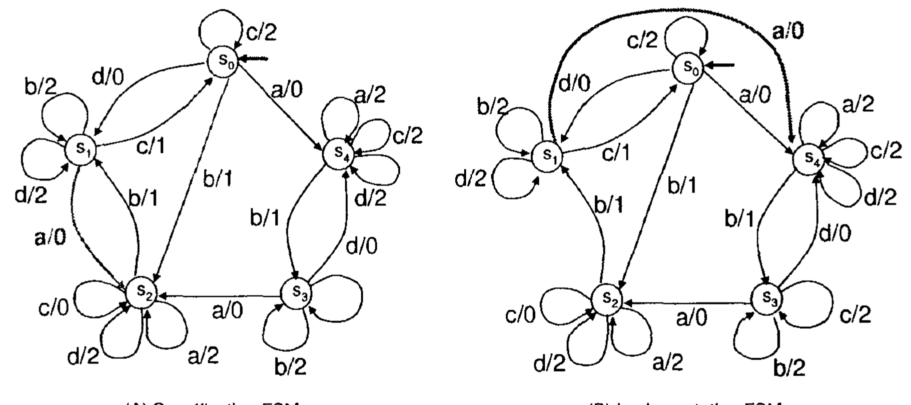
Let $\varrho = t_0\sigma_1 t_1\sigma_2 t_2 \dots \sigma_k t_k \sigma_0$ be a proximate test path of a given specification M, where t_i is a transition and σ_i is a path in M such that $\sigma_i t_i$ is an invertible sequence for $1 \le i \le k$. If $end(t_k) = s_m$, then $in(t_0\sigma_1 t_1\sigma_2 t_2 \dots \sigma_k t_k) \circ UIO_m$ is a test sequence satisfying the U-method.

The algorithms on how to find a *minimal-length* proximate test path of M is explained in details in [20]. Given specification M_1 in Figure 5, a test sequence of length 26 can be generated by the approach in [20] compared with that of length 72 by the approach in [1] and that of length 31 by the approaches in [63, 33].

4.3.2 Weakness and strength of the U-method

The U-method does not support the *full* fault coverage due to the following two main reasons.

i) It does not check whether the starting states of transitions are correct. That is, when a transition starts from a wrong state in the IUT, no mechanism from the U-method intends to check it directly.



(A) Specification FSM (B) Implementation FSM

Input alphabet I = {a, b, c, d}, Output alphabet O = {0, 1, 2}
UIO sequence of each state:
UIO_0 = da; UIO_1 = c; UIO_2 = c; UIO_3 = db; UIO_4 = bd.
a possible test sequence generated by the U-method:
bcbcdabacccbcdcabdbdbdbdbacbcaabdcbddbdbbdbcdbaacdcbbcddccda
the corresponding expected output sequence:
101100100001101010101010100110210210210120120102020121021200

Figure 6: Illustration of the weakness of the U-method

ii) The uniqueness of the output sequence in response to a UIO sequence in the specification does not guarantee the uniqueness of that in the IUT. In a faulty IUT, it is possible that there are other states such that the same output sequence is produced by applying the UIO sequence in those states. Consequently, the state verification fails.

Example 6 A faulty IUT (Figure 6(B)) of the specification shown in Figure 6(A) has a transfer fault for transition $(s_1, s_2, a/0)$: instead of ending at state s_2, it ends at state s_4 in the IUT. The IUT passes the testing with a test sequence generated by the U-method since the actual output sequence produced by the faulty IUT is the same as the expected one with this test sequence. In this case, the testing fails to detect the above transfer fault. $\diamond$

The strength of the U-method is that it can achieve a *satisfactory* fault coverage with an *acceptable* cost [92, 79, 67]: On one hand, compared to the T-method, the U-method is much more effective in detecting faults; On the other hand, the U-method generates much shorter test sequences and are less restrictive than those methods (e.g. the D-method and the W-method) supporting the full fault coverage. For example, the U-method does not require a completely specified specification, reliable reset, a distinguishing sequence, etc.

Due to the benefits the U-method provides, it is desirable to incorporate the U-method with the characterization sets (see Chapter 4.5 for the definition), which exist for all the minimal FSMs, such that the U-method is applicable to all the FSMs. For example, in [37], Hierons proposed a technique to generate a minimal-length test sequence satisfying the U-method with a characterization set.

4.4 D-method

The D-method is applicable to a special class of FSMs that have a *distinguishing sequence* (DS) [27, 52]. Given an FSM M, a *distinguishing sequence* is an input sequence D with the following characteristics: the output sequences produced by M in response to D in different states of M are all different. Formally,

Definition 5 (distinguishing sequences) *Given an FSM* $M = (S, I, O, \delta, \lambda, s_0)$, *an input sequence* D *is a distinguishing sequence of* M *if for all* $s_i, s_j \in S$, $s_i \neq s_j$ *implies* $\lambda(s_i, D) \neq \lambda(s_j, D)$.

Not every FSM has a DS. It is a PSPACE-complete problem to determine whether a given FSM has a DS [55]. The classical algorithms of finding a DS are of exponential time as discussed in [27, 52].

Clearly, a DS is a UIO sequence applicable to all the states. The existence of a DS of an FSM implies the existence of a UIO sequence for each state of the FSM; but the reverse is not true.

Example 7 In Figure 1, a distinguishing sequence for M_0 is $D = aba$: when we apply this input sequence to states s_0, s_1 and s_2, the output sequences are 001, 100, and 101, respectively. They are all different. $\diamond$

Two FSMs M_1 and M_2 are *equivalent* if and only if for every state of M_1 there is an equivalent state of M_2 and vice versa. An input sequence is a *checking sequence* of M if and only if it distinguishes between M and any FSM that has the same sets of input and output alphabets as M but is not equivalent to M. Clearly, a checking sequence is a special test sequence that guarantees the full fault coverage.

Assume that the IUT behaves like some (unknown) FSM N with the number of states no greater than that of the specification M. Since M and N are deterministic and minimal, determining whether N is equivalent to M can be achieved by establishing isomorphism between M and N. More precisely,

- for each state s in M, we identify a state r in N that *corresponds to s*.
- for each transition $t = (s_1, s_2, x/y)$ in M, we verify that there exists a transition $t' = (r_1, r_2, x/y)$ in N which starts from a state corresponding to s_1, ends at a state corresponding to s_2, and gives the same output y upon the same input x.

A checking sequence is designed to help us to achieve the above two goals. With respect to these goals, the construction of a checking sequence usually involves two steps: one for *state identification* and one for *transition verification* [56].

The purpose of *state identification* is to build a one-to-one correspondence between the states in M and those in N. State identification using UIOs is possible but it turns out to be hard and less practical [40]. A characterization set (which is discussed in Chapter 4.5) is easier to find than a distinguishing sequence, yet a test suite generated using a characterization set [11] is usually much longer than that generated using a distinguishing sequence in terms of total length of the test sequences [8, 29, 32, 85]. Of course, DS can also be used for state verification in the sense of verifying the ending states of transitions.

Recall that $D = aba$ is a distinguishing sequence of M_0 in Figure 1. If we apply D to the IUT of M_0 and observe 001, then we know that the state of N before we apply D *corresponds to s_0*, which is the only state in M_0 that gives output sequence 001 in response to input sequence aba. Similarly, if we apply D to the IUT several times (at different states of N) and observe 001, 100, and 101, then we know that N has (at least) three states and the states of N before we apply each D correspond to s_0, s_1 and s_2 respectively.

Example 8 Suppose N_1 shown in Figure 7(B) is an FSM describing the behavior of a possible implementation of specification M_0 in Example 2.1.

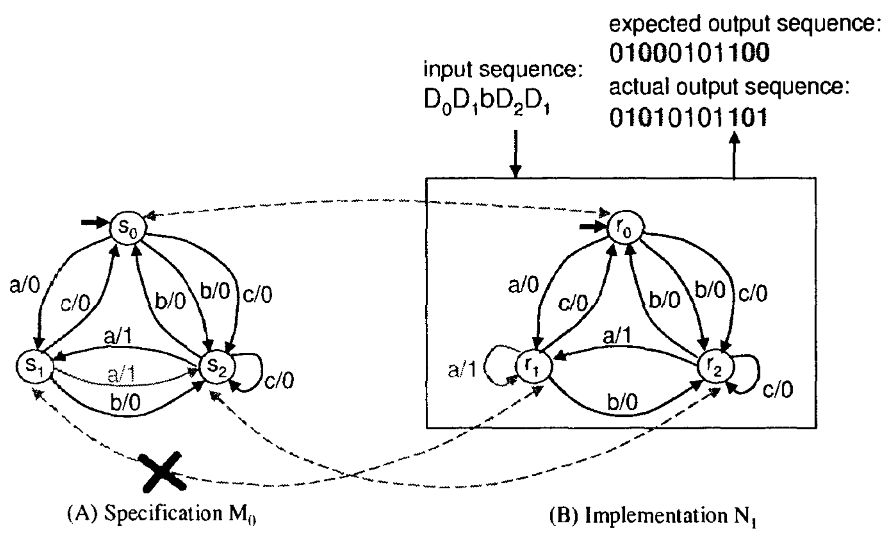


Figure 7: An illustration of the necessity of state identification

Recall that for M_0 in Figure 1, we have distinguishing sequence $D = aba$. As the dashed arrows in Figure 7 show, we have r_0 corresponds to s_0 and r_2 corresponds to s_2 in the sense that $\lambda(s_0, D) = \lambda(r_0, D) = 001$ and $\lambda(s_2, D) = \lambda(r_2, D) = 101$. However, $\lambda(s_1, D) \neq \lambda(r_1, D)$: M_0 (at s_1) and N_1 (at r_1) give different output sequences 100 and 101 respectively in response to input sequence D. In other words, at r_1, the implementation FSM N_1 does not behave like M_0.

We can detect that N_1 is a faulty implementation of M_0 on the stage of state identification: by applying D to the IUT, we fail to find a state in N_1 which produces output sequence 100 in response to D, as s_1 does.

$\diamond$

When a distinguishing sequence D of M is given, sometimes a prefix of D is sufficient in helping us identify a state in N with a state in M. For example, in Figure 1, state s_0 is the only one that gives output 0 in response to input a. Thus, we can simply use a (which is a prefix of D) as input to the IUT to identify a state in N that corresponds to s_0. We will use D_i to denote the *prefix distinguishing sequence for* s_i. It is the shortest prefix of D that is sufficient to distinguish state s_i from others, i.e., for any state s_j where $s_i \neq s_j$, $\lambda(s_i, D_i) \neq \lambda(s_j, D_i)$. In Figure 1, with $D = aba$, the prefix distinguishing sequences for

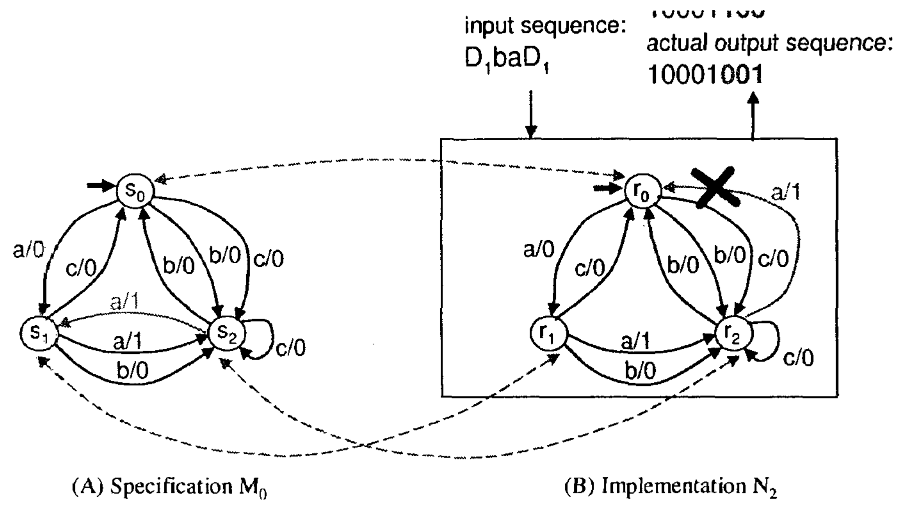


Figure 8: An illustration of the necessity of transition verification

states s_0, s_1, and s_2 are: $D_0 = a$ and $D_1 = D_2 = aba$. In the following, we consider the situation when the prefix distinguishing sequence D_i is given for all $i < n$, where n is the number of states in M.

An *α'-sequence* [39] is an input/output sequence used to identify some states in N with D or D_i $(0 \leq i < n)$. A set of α'-sequences that can jointly identify all the states of N is called an *α'-set.* These two terminologies are evolved from similar but more restrictive terminologies α-sequences and α-set [85], respectively.

Example 9 In the previous example, let $\rho_0 = (s_0, s_1, a/0)$, $\rho_1 = (s_1, s_1, aba/100)$, and $\rho_2 = (s_2, s_1, aba/101)$ be the paths induced by applying D_0, D_1, and D_2 to s_0, s_1, s_2, respectively. Let $\rho = \rho_0 \circ \rho_1 \circ (s_1, s_2, b/0) \circ \rho_2 \circ \rho_1$. Then, $label(\rho) = aabababaaba/01000101100$ is an α'-sequence, and $\{label(\rho)\}$ is an α'-set. In fact, we can use this α'-sequence to identify all states in N_1: When we apply the input portion of $label(\rho)$, i.e., $D_0D_1bD_2D_1 = aabababaaba$, to N_1, if the expected output sequence 01000101100 is produced, then we can conclude that there are three distinct states in N_1 corresponding to those in M_0. However, the actual output sequence produced is 01010101101 which is different from the expected one. Consequently, the one-to-one correspondence cannot be found between the states in M_0 and those in N_1. Thus, we can conclude that N_1 is a faulty implementation.

◇

Example 10 Suppose that the FSM N_2 shown in Figure 8(B) describes another implementation of specification M_0.

Using the same α'-sequence $aababababaaba/01000101100$ on N_2, we observe output sequence 01000101100 as expected upon the input sequence $aababababaaba$. Thus, we conclude that there are three states in N_2 corresponding to s_0, s_1, and s_2 in M_0, respectively.

Later on we will show that even though N_2 passes the test for state identification, it fails the test for transition verification. ◇

Suppose that the state identification has been achieved. We can use this knowledge to investigate the structure of N to determine whether it is equivalent to the specification FSM M. This can be realized by transition verification which builds the one-to-one correspondence between the transitions in M and those in N. More precisely, for each transition $t = (s_i, s_j, x/y)$ in M, we verify the existence of a corresponding transition t' in N. This is basically achieved by the following three steps: i) lead N to the state corresponding to s_i; ii) verify the label of t' by applying x to N to check whether the output is y; and iii) verify whether the ending state of t' corresponds to s_j.

Steps ii) and iii) are usually realized by including *β-sequences* into checking sequence construction. A *β-sequence* of transition $t = (s_i, s_j, x/y)$ is the input/output sequence $x/y \circ D_j/\lambda(s_j, D_j)$. For example, in the FSM M_0 in Figure 1, the β-sequence of $t = (s_2, s_1, a/1)$ is $a/1 \circ D_1/\lambda(s_1, D_1) = aaba/1100$.

When N is led to such a state r that its correspondence with a state in M can be derived, typically via state identification, we say r is *recognized*. A state r in N is *verified* if we apply an input sequence, typically a (prefix) distinguishing sequence, to N at this state in order to check the output sequence to confirm the correspondence between r and a state in M. If an input sequence allows us to lead N to a state *recognized* as s_i, check its output y in response to input x, and subsequently *verify* that the ending state corresponds to s_j, then we say transition $t = (s_i, s_j, x/y)$ is *verified* in this input sequence. The formal definitions of the notions can be found in [85].

Suppose we want to verify the correspondence between transition $t = (s_2, s_1, a/1)$ in M_0 and some $t' = (r_2, r_1, a/y)$ in N_2 in Figure 8(B). To apply the β-sequence of t, we first need to make sure that the current state in N_2 is recognized as s_2. This can be realized by making use of the result of state identification. Recall that we use α'-sequence $D_0D_1bD_2D_1/\lambda(s_0, D_0D_1bD_2D_1) = aababababaaba/01000101100$ to identify states in N_2. This implies that if the current state in N_2 is recognized as s_0, then after applying input sequence D_0D_1b and observing correct output sequence 01000, we know the current state of N_2 is recognized as s_2. This is because we have already checked the output of $D_0D_1bD_2$ for state identification. Similarly, if the current state of N_2 is recognized as s_1, then after applying input sequence D_1b and observing correct output sequence 1000, we know the current state of N_2 is recognized as s_2. In fact, whether the current state in N_2 corresponds to s_1 or not is also known after we apply input sequence D_1b: We just need to check whether the output sequence in response to D_1 is 100.

Now we use this knowledge to lead N_2 to a state recognized as s_2. Suppose N_2 is currently in state r_0. We apply input a to N_2, and r_1 is *supposed* to be reached. Next, we apply D_1b on N_2. If the expected output sequence 1000 is produced in response, we can conclude that: i) before applying D_1b, N_2 was *indeed* in a state corresponding to s_1; and ii) after applying D_1b, a state recognized as s_2 *is* reached.

Having reached a state recognized as s_2, we are ready to use β-sequence to test whether the label and the ending state of t' are correct. We apply input a to N_2, and output 1 is produced as expected. That is, the label of t' is correct. Finally, we verify that the ending state in N_2 corresponds to s_1 by applying D_1: The expected output sequence is 100 while the actual output sequence is 001.

In summary, we use input sequence D_1baD_1 to verify the correspondence between transition t and t', where D_1b is used to lead N_2 to reach r_2, and the last D_1 is used to verify the ending state of t'. This is shown in Figure 8. Since the expected output sequence is 010001100 and the actual one is 010001001, there does not exist a transition in N_2 corresponding to t. Therefore, we conclude that N_2 is a faulty implementation of M_0.

Let n be the number of states in a given FSM M, and p the size of the input alphabet. According to [88], when a DS exists, the lower bound of the length of the generated checking sequence is $\Omega(pn^3)$; and an algorithm is given to find a checking sequence of length $O(p^2n^4 \log(qn))$. By making use of the prefix distinguishing sequences, Lee et al. proved in [56] that a checking sequence of length $O(pn^3)$ can be found.

As shown above, when there exists a DS for a given FSM M, the D-method can be used to generate a checking sequence for M; and under a different condition (which will be discussed in Chapter 4.5), the W-method also applies.

The fault coverage criterion for checking sequences (the D-method and the W-method) [11, 32, 29, 85] is as follows.

- checking sequences: The corresponding path of the generated checking sequence in the specification FSM M should contain each transition in M with its starting state in the implementation FSM *identified* and its ending state in the implementation FSM *verified*.

When generating a checking sequence, it generally requires a completely specified specification FSM.

4.4.1 The test optimization approach in [39]

Besides its advantage of guaranteeing a full fault coverage, the use of checking sequences for unit testing also provides an additional benefit for the *integration testing* or *system testing.* In the *unit testing*, if we can derive from deterministic testing that components I_1 and I_2 are trace equivalent to their respective specifications P_1 and P_2, then without performing the *integration testing* or *system testing*, we know that the integration of I_1 and I_2 is a correct implementation of the parallel composition of P_1 and P_2.

As the checking sequence usually suffers from too long a length, researchers are interested in the optimization techniques to reduce the testing cost in terms of the lengths of the generated checking sequence [8, 19, 39, 40, 81, 85, 86, 94]. Among these pieces of work, [39, 85, 86] consider how to reduce the length of the checking sequence by reducing the length of α-sequences or by increasing the chances of overlaps among α-sequences (or α'-sequences) and β-sequences. [8, 81] focus on how to reduce the length of the checking sequence by exempting some transitions from being verified under certain conditions.

[19, 94] introduce *alternative* β-sequences to expand the selection pool of test subsequences for transition verification such that the chance of the maximum overlaps is increased. Here, we explain two typical approaches in [39] and [86] on this topic. Note that the reduction techniques presented in [8, 19, 81, 94] can work together with those in [39] and [86] under certain circumstance.

In the work of Hierons and Ural [39], a checking sequence is generated in two steps: i) construct an auxiliary graph $G' = (V', E')$ from G_M; ii) find an RPP tour ϱ in G', and then a checking sequence can be easily derived from ϱ.

According to [39] as well as some other work [8, 41, 81, 85], a set of α'-sequences that can form an α'-set was first constructed. Each α'-sequence will be used to identify a (sub)set of states in the implementation FSM. Suppose an α'-sequence ϱ can be used to identify those states in the implementation FSM that correspond to $s_1, \ldots, s_k$ $(k \geq 1)$. ϱ can be considered as some subsequences concatenated together, where each subsequence starts with an input/output sequence corresponding to the (prefix) distinguishing sequence of s_i for some $1 \leq i \leq k$. That is, each subsequence has form $T_i = D_i/\lambda(s_i, D_i) \circ I_i$, where I_i is a possibly *null* input/output sequence called *transfer sequence.* Input/output sequence T_i is called a *T-sequence.* According to the explanation of [39], $I_i = \phi$ for all $i \in \{0, \ldots, n-1\}$. In this setting, an α'-sequence is actually a concatenation of T-sequences.

Example 11 Consider the FSM M_0 in Figure 1. As we explained before, the prefix distinguishing sequences for each state are: $D_0 = a$ and $D_1 = D_2 = aba$; and $\alpha'_1 = D_0D_1bD_2D_1/\lambda(s_0, D_0D_1bD_2D_1)$ is an α'-sequence. Let $T_0 = D_0/\lambda(s_0, D_0) = a/0$, $T_1 = D_1b/\lambda(s_1, D_1b) = abab/1000$, and $T_2 = D_2/\lambda(s_2, D_2) = aba/101$. We have that $\alpha'_1 = T_0T_1T_2T_1$. $\diamondsuit$

In the following, we explain the checking sequence construction technique presented in [39]. Let P_i and R_j be the paths in $G_M = (V, E)$ induced by α'-sequence α'_i and T-sequence T_j respectively. The auxiliary graph $G' = (V', E')$ is constructed from G_M as follows.

- $V' = V \cup U'$ is a set of vertices, where
  - $U' = \{v'_i \mid v_i \in V\}$;

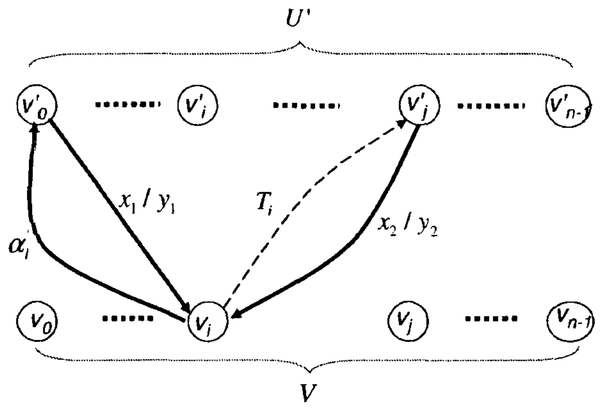


Figure 9: Illustration of the construction of G'. The nodes in U' and V are on the top and at the bottom respectively. The solid bold arrows and the dashed arrows denote the edges in $E_{\alpha'} \cup E_C$ and E_T, respectively.

- $E' = E_C \cup E_{\alpha'} \cup E_T \cup E''$ is a set of edges, where
  - $E_C = \{(v'_i, v_j, x/y) \mid e = (v_i, v_j, x/y) \in E\}$;
  - $E_{\alpha'} = \{(start(P_i), (end(P_i))', \alpha'_i) \mid \alpha'_i \in \alpha'\text{-set}\}$;
  - $E_T = \{(start(R_i), (end(R_i))', T_i) \mid T_i$ is a T-sequence$\}$;
  - E'' is a subset of $\{(v'_i, v'_j, x/y) \mid e = (v_i, v_j, x/y) \in E\}$, such that G' is strongly connected and $G'' = (U', E'')$ is *acyclic*, i.e. it does not contain any cycle.

In order to get a shortest-length checking sequence, we can find an RPP tour ϱ in G' such that each edge in $E_{\alpha'} \cup E_C$ is traversed at least once.

α'-sequences and T-sequences are represented in G' as edges in $E_{\alpha'} \subset E'$ and $E_T \subset E'$, respectively. Each transition of M is represented as an edge in $E_C \subset E'$, and ultimately these edges will be contained in the RPP tour ϱ. The edges in $E_T \cup E_{\alpha'}$ start from the ending vertices of the edges in E_C so the ending states represented by these vertices are identified by either T-sequences or α'-sequences. The set of edges E'' is included in G' to increase the connectivity of the vertices in G'.

Figure 9 illustrates the construction of auxiliary digraph $G' = (V \cup U', E_C \cup E_{\alpha'} \cup E_T \cup E'')$ from M. For any $s_i \in S$, we introduce two vertices in G': vertex v'_i in U' (shown on the top) and vertex v_i in V (shown at the bottom). Suppose $P_l = (v_i, v_0, \alpha'_l)$ and $R_i = (v_i, v_j, T_i)$ are two paths in M, where α'_l is an α'-sequence and T_i is a T-sequence. P_l is represented in G' by an edge (solid bold arrow) from v_i to v'_0 with label α'_l. R_i is represented by an edge (dashed arrow) from v_i to v'_j with label T_i. Suppose $e = (v_0, v_i, x_1/y_1)$ and $e' = (v_j, v_i, x_2/y_2)$ are two edges in G_M. e is represented in G' by an edge (solid bold arrow) from v'_0 to v_i with label x_1/y_1 and e' is represented by an edge (solid bold arrow) from v'_j to v_i with label x_2/y_2.

To determine an RPP tour ϱ in G' such that each edge in $E_{\alpha'} \cup E_C$ is traversed at least once, we assign the cost of each edge in G' to be the number of input/output pairs in its label. This relates the minimal-cost of an RPP tour with the minimal-length of the checking sequence derived from it. It is formally proved in [39] that the input portion of *label*(ϱ) *is* a checking sequence.

Example 12 Given an FSM in Figure 1, the prefix distinguishing sequences for each state are: $D_0 = a$ and $D_1 = D_2 = aba$. Based on these D_is, an α'-set for M_0 is $\{\alpha'_1\}$, where $\alpha'_1 = D_0D_1bD_2D_1/\lambda(s_0, D_0D_1bD_2D_1)$. $T = \{T_0, T_1, T_2\}$, where $T_0 = D_0/\lambda(s_0, D_0)$, $T_1 = D_1b/\lambda(s_1, D_1b)$, and $T_2 = D_2/\lambda(s_2, D_2)$.

A checking sequence generated by the approach in [39] is of length 48. Combining this approach with the technique presented in [19], a checking sequence of length 45 can be found. The length of the generated checking sequence can be further reduced to 42 when the techniques in [8, 81] are considered. $\diamondsuit$

4.4.2 The test optimization approach in [86]

In [86], Ural et al. considered to reduce the lengths of the generated checking sequences by identifying and eliminating the *overlap* among test segments for state identification and transition verification. Let $P_1 = \rho_1 \circ \rho$ and $P_2 = \rho \circ \rho_2$ be two paths in a graph G, when $\rho \neq \epsilon$, we say P_1 *overlaps* P_2 *with* ρ. In particular, if *label*(ρ) has D as a prefix of its input portion, we say P_1 *D-overlaps* P_2 *with* ρ.

Example 13 Let G_{M_0} be the graph representation of M_0 shown in Figure 1. Recall that $D = aba$. $P_1 = (v_1, v_1, bD/\lambda(s_1, bD))$ and $P_2 = (v_2, v_0, DD/\lambda(s_2, DD))$ are two paths in

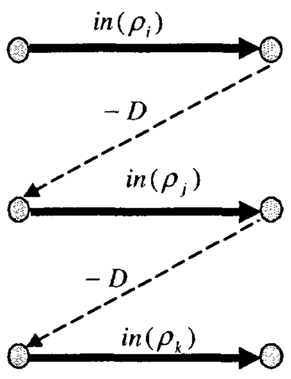


Figure 10: An illustration of the construction of $G^* = (V^*, E^*)$. Paths in $A \cup B$ are represented by bold solid arrows, and D-overlaps among paths in $A \cup B$ are represented by the dashed arrows.

G_{M_0}. Since $\rho = (v_2, v_1, D/\lambda(s_2, D))$ is both a suffix of P_1 and a prefix of P_2, P_1 overlaps P_2 with ρ. Furthermore, since the input portion of *label*(ρ) is D, we also have P_1 D-overlaps P_2 with ρ. $\diamond$

Let $A = \{(v_i, \delta(s_i, DD), DD/\lambda(s_i, DD)) | v_i \in V\}$ be a set of paths induced by applying consecutively twice distinguishing sequence D in each vertex in G_M. Let $B = \{(v_i, \delta(s_i, xD), xD/\lambda(s_i, xD)) | v_i \in V, x \in X\}$ be the set of paths corresponding to the β-sequences of each edge in G_M. Apparently, the labels of the paths in A can be used for state identification and the labels of the paths in B can be used for transition verification. Let ϱ be a path that contains all the paths in $A \cup B$. Note that it is possible that the digraph induced by paths in $A \cup B$ is not strongly connected: We may use some transfer edges in G_M to connect paths in $A \cup B$. When the graph induced by these transfer edges is acyclic, the input portion of *label*(ϱ) is a checking sequence. The goal of the work in [86] is to maximize the D-overlaps among the paths in $A \cup B$ for checking sequence generation.

Like many other existing methods, the method presented in [86] uses two steps to solve the optimization problem of checking sequence generation: i) from the given $G_M = (V, E)$, construct an auxiliary digraph $G^* = (V^*, E^*)$ such that the D-overlaps among the paths in $A \cup B$ are explicitly expressed; and ii) find an RPP tour in G^*.

G^* is defined by augmenting additional vertices and edges to G_M. Figure 10 shows the key ideas on this augmentation. For each path ρ in $A \cup B$, we add two new vertices with a new edge between them. This edge represents ρ and is labelled $in(\rho)$. When looking for an RPP tour, we require that all such new edges be traversed at least once. Apparently, from this tour we can easily derive a path that contains all the paths in $A \cup B$ as we desired.

When ρ_i D-overlaps ρ_j, this D-overlap is explicitly expressed by adding a new edge starting from the ending vertex of the edge representing ρ_i and ending at the starting vertex of the edge representing ρ_j. The label of the edge that represents the D-overlap between the two paths is $-D$. This negative label can be used to remove the overlapped part when ρ_i is concatenated with ρ_j.

The cost of each edge in G^* is defined according to the length of its label: the cost of an edge with label $in(\rho)$ is $|in(\rho)|$, and the cost of an edge with label $-D$ is $-|D|$. A path ϱ that contains all the edges representing paths in $A \cup B$ with the minimum cost can then be found, and $in(\varrho)$ is the desired checking sequence with minimal length.

Example 14 For FSM M_0 in Figure 1, a checking sequence generated by the approach in [86] is of length 47. Combining this approach with the technique presented in [19], a checking sequence of length 44 can be found. The length of the generated checking sequence can be further reduced to 41 when the techniques in [8, 81] are considered. $\diamondsuit$

4.5 W-method

Just like the D-method, the W-method [11] is to generate a checking sequence from a given specification FSM M for a full fault coverage testing. When an IUT can be reset to the initial state correctly at any time, we say the IUT has a *reliable reset* property. With this property, the W-method can be used to generate a checking sequence without requiring the existence of a distinguishing sequence of M. In order to realize state identification and transition verification, the W-method uses a *characterization set*, which exists in any minimal FSM.

A characterization set consists of input sequences that can distinguish between the behavior of every pair of states of M. Formally,

Definition 6 (characterization sets) *Given an FSM M = $(S, I, O, \delta, \lambda, s_0)$, a set*

W of input sequences is a characterization set of M if for any $s_i, s_j \in S$, $s_i \neq s_j$ implies $\exists w \in W$ such that $\lambda(s_i, w) \neq \lambda(s_j, w)$.

Clearly, if M has UIO sequences for all of its states or a distinguishing sequence D, then both $\{UIO_i \mid s_i \in S\}$ and $\{D\}$ are characterization sets. How to find a characterization set from M is discussed in [27].

Suppose a specification $M = (S, I, O, \delta, \lambda, s_0)$ and its characterization set W are given. Let $n_{k,s}$ denote a node at level k with label s in a tree. The concatenation of two sequences is extended to two sets of sequences. More precisely, let A, B be two sets of sequences, $A \circ B = \{a \circ b \mid a \in A, b \in B\}$. The core part of the W-method is given below.

```
1) Construct a testing tree $T$ from $M$.
    1.1) Let the root of $T$ be $n_{0,s_0}$, $S_T = \{s_0\}$, and $k = 0$.
    1.2) For each node $n_{k,s}$, if $s \notin S_T$ or $k == 0$, for each $s' \in \{\delta(s,x) \mid x \in I\}$, add
         node $n_{k+1,s'}$, add an edge from $n_{k,s}$ to $n_{k+1,s'}$ with label $x$ where $s' = \delta(s,x)$.
         Let $S_T = S_T \cup \{s\}$.
    1.3) If $S_T \neq S$, let k = k+1 and go to step 1.2.
2) Let $\Phi$ be the set of the labels of the paths from the root to each node of $T$. $\chi = \Phi \circ W$
   is the desired test suite.
```

The construction of testing tree T takes $O(|S| \cdot |I|)$ time. Each edge of T corresponds to a transition of M, and thus the number of edges of T is $|S| \cdot |I|$ when M is completely specified. In step 2), $\chi = \Phi \circ W$ means that for each transition, there exist test sequences in χ such that both its starting state and its ending state are identified with characterization set W. Here, the generated checking sequence χ is actually a set of test sequences, whose corresponding paths in M start from the initial state. After applying each of the test sequences, we reset the IUT to the initial state.

Under the aforementioned assumptions, the above algorithm is optimal in terms of the size of the generated test suite [11].

Example 15 Figure 11(A) shows an example FSM M_2 from [11] with slight modifications. Let $W = \{a, b\}$ be a characterization set of M_2. Applying a at each state of M_2, we obtain output 0 at s_0 and s_1, output 2 at s_2, and output 3 at s_3. This means that s_2 and s_3 can be identified with a. Similarly, s_0 and s_1 can be identified with b. Combining the above results, all the states of M_2 can be identified by W.

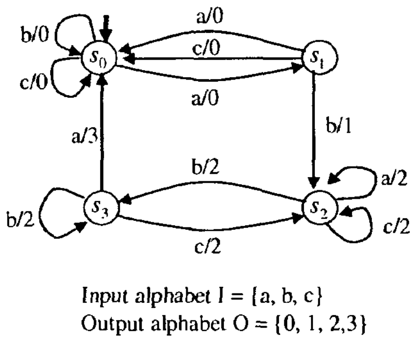
The testing tree T (Figure 11(C)) is used for transition verification. Suppose we want to verify transition $t = (s_3, s_2, c/2)$ in M_2, which is represented by edge $(n_{3,s_3}, n_{4,s_2}, c)$ in T. The labels of the paths from the root to n_{3,s_3} and n_{4,s_2} are *abb* and *abbc*. By concatenating *abb* with input sequences in W, we obtain two test sequences *abba* and *abbb*, whose expected output sequences are 0123 and 0122 respectively. When the actual output sequences are produced as expected, it is guaranteed that the IUT reaches a state corresponding to s_3 after applying *abb* at the initial state. With this knowledge, we can achieve two goals by making use of test sequences generated from $\{abbc\} \circ W$: i) verify the label of t; and ii) the IUT reaches a state corresponding to s_2 after applying *abbc* at the initial state. Thus, t is verified in the IUT.

Since each transition of M_2 is represented by an edge in T, all the transitions of M_2 can be verified in the same way. $\diamondsuit$

4.6 Summary

In this chapter, we have discussed the fault coverage and the optimization techniques related to the T-method, the U-method, the D-method, and the W-method. From the viewpoint of the fault coverage, from coarser to finer, we have the T-method, the U-method, and the checking sequences which include the D-method and the W-method. Of course, the cost in terms of the length of a generated test sequence for a finer fault coverage criterion, in general, is higher. Readers are referred to [79] for an experimental study on the comparison of the fault coverage and the cost of these four methods.

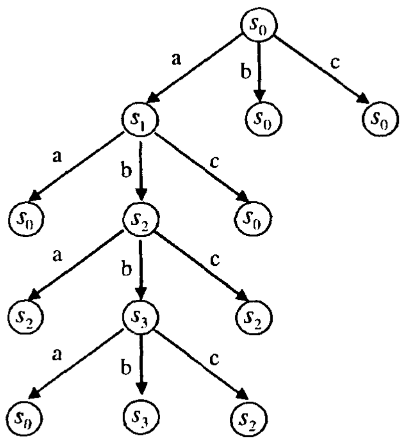
Although the checking sequences are the most costly, they support full fault coverage. This provides the possibility of avoiding the *integration testing* or the *system testing* by leaving the insurance of the correctness of the integrated system to the *formal verification* [5, 13, 14, 31, 44, 60], which is a well studied research area with many available state-of-the-art supporting tools such as SPIN and SMV. This approach is also adopted by many other state-based conformance testing techniques. For example, in [87], Tretmans et al.



(A) An example FSM M_2

state	a	b
s_0	0	0
s_1	0	1
s_2	2	2
s_3	3	2

(B) A characterization set of M_2



(C) Testing tree of M_2

Figure 11: An example FSM M_2

presented the conditions of avoiding integration testing of labelled transition systems w.r.t. the *ioco conformance relation* [82, 83, 84].

The integration testing of the FSM-based systems usually do not support the full fault coverage. Interested readers are referred to [30, 54] for more details.

Another issue of fault coverage criteria is their relationship with *trace equivalence*. *Trace equivalence* is a widely used equivalence relation for stateful systems in many research fields such as automata theory [46], process algebra [42, 64]. A *trace* is the corresponding i/o sequence of a path. We say two processes are *trace equivalent* if they have the same set of all possible traces. In the realm of deterministic FSMs, two FSMs are *trace equivalent* if for any input sequence, they produce the same output sequence in response. Clearly, when there exists a cycle in an FSM, the size of the set of all possible traces is infinite. It turns out that an infinite-length test sequence may be generated from a given FSM if we directly adopt the above definition for test generation. This is undesirable. With *state identification* techniques from the FSM-based testing, it is possible to generate a finite-length test sequence that can ensure the trace equivalence between the specification FSM and the IUT. Such a test sequence is the so-called *checking sequence* generated by the D-method and the W-method to support full fault coverage.

5 Test executability problem

In this chapter, we explain the *test executability* problem [72, 73] in details.

Test executability problem describes the situation where a test sequence generated solely from a given specification without taking into account the behavior of the context may not be executable when we carry out testing in context. There are two causes of this problem.

- *improper order of tests.* In this case, we have the so-called *test translation* problem, i.e., a test sequence generated in isolation may not be feasible in testing in context due to the improper order of inputs. In the example FSM $\mathcal{S}_1$ shown in Figure 12, transition $(s_1, s_2, x_2/o_2)$ cannot be tested by applying test sequence $\chi_1 = i_1x_2i_1x_1$. This is because after the tester inputs i_1 to the IUT, the IUT produces an output y_1 and sends to its context. In response, the context returns x_1 instead of x_2 as expected

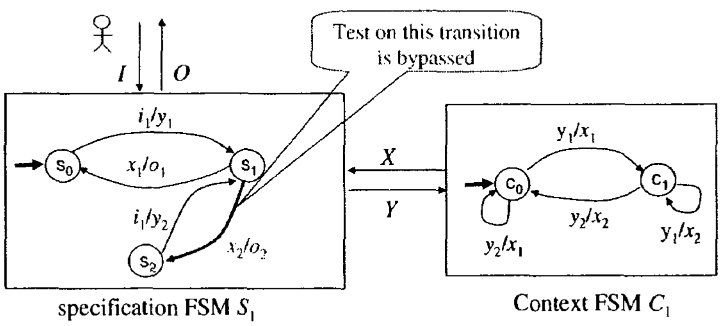


Figure 12: An illustration of test executability problem caused by improper order of test sequence

to the IUT. Therefore, χ_1 is not executable.

- *intrinsically non-executable transitions.* The existence of intrinsically non-executable transitions originates from the fact that the specification of each individual FSM is usually designed separately and thus it does not consider how to trigger each transition in the context. As a result, some transitions specified in the specification cannot be executed in any circumstance, i.e., some transitions of the IUT are not testable due to the restriction imposed by the context. In the example FSM S_2 of an IUT shown in Figure 13, since x_2 can not be produced by C_2 with any (external) input sequence when the IUT is in s_0, transitions $(s_1, s_2, x_2/o_2)$ and $(s_2, s_1, i_1/y_2)$ can not be triggered. We call these transitions intrinsically non-executable in context C_2.

6 Test generation with stateless embedded context

In this chapter, we consider test generation of an IUT with embedded context in this setting: i) The IUT is stateful, deterministic, and specified by an FSM; ii) The context is stateless, and specified by a set of $\langle request, response\rangle$ pairs; and iii) The context may include several components. Our goal is to generate minimal-length test sequence while avoiding test executability problem when carrying out the testing whenever such a sequence exists.

An example application is a web application that makes use of web services as shown in Figure 14. Very often, the functionality of web services is known and stateless. We consider

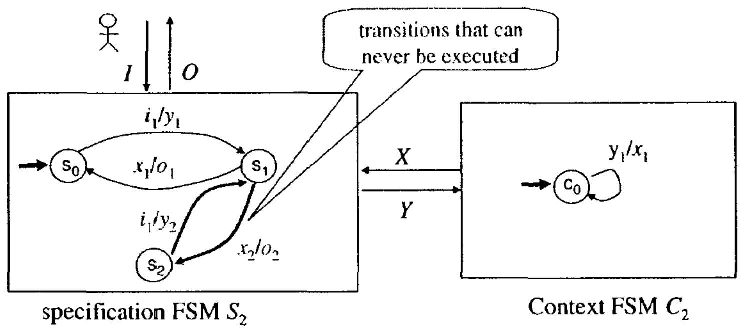


Figure 13: An illustration of test executability problem caused by intrinsically non-executable transitions

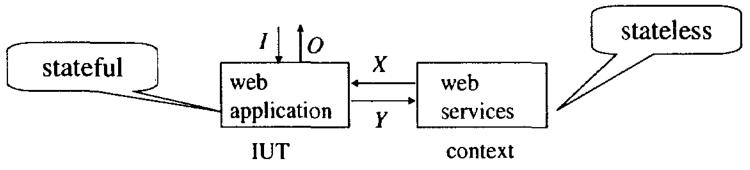
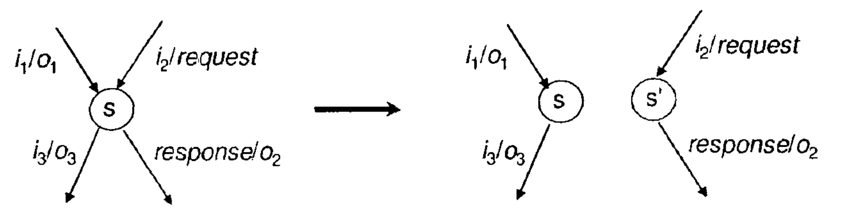


Figure 14: An example application of stateless embedded context

the test generation of this web application without encountering test executability problem during testing.

6.1 Solving test executability problem

Due to the existence of the context, the executability problem occurs when the input of a transition is from the context instead of the tester. In this case, the traverse history has to be taken into consideration. In Figure 15(A), suppose that state s is entered by executing the transition with input i_2 from the tester and an output *request* is sent to its stateless context. The context responds *request* with message *response*, which will be the actual input to the IUT. For test generation, the execution of these two adjacent transitions has to be enforced to avoid executability problem. In doing so, we can split state s by adding a new state s' to isolate the transition pairs involving the interaction with the context from other transitions (See Figure 15(Figure 15(B)). Thus, any test sequence generated from the resulting graph will not encounter test executability problem at this point. An advantage



(A) The original specification of the IUT (B) The derived specification of the IUT

Figure 15: An illustration of solving executability problem for an IUT with stateless embedded context

of this technique is that the computation of the synchronization product of the IUT and its context can be avoided.

How to derive an auxiliary $\mathcal{S}'$ from a given specification FSM $\mathcal{S}$ of IUT is given in Algorithm 1. The time complexity of the algorithm is $O(|S||T|)$, where $|S|$ and $|T|$ are the number of the states and transitions in $\mathcal{S}$. When $\mathcal{S}'$ is constructed, the test generation problem of an IUT together with its stateless embedded context is reduced to the test generation problem of an isolated IUT. Consequently, the aforementioned test techniques, such as the T-method and the U-method, and their optimization techniques for testing isolated IUT are applicable.

Proposition 1 *Let $\mathcal{S}$ and $\mathcal{C}$ be the specifications of an IUT and its context. Let $\mathcal{S}'$ be the auxiliary FSM constructed by Algorithm 1. Then $\mathcal{S}'$ is externally equivalent to $\mathcal{S}$ w.r.t. $\mathcal{C}$.*

PROOF. Since $\mathcal{S}$ and $\mathcal{C}$ are deterministic, $\mathcal{S} \times \mathcal{C}$ is deterministic. According to Algorithm 1, $\mathcal{S}'$ is deterministic, and thus $\mathcal{S}' \times \mathcal{C}$ is deterministic. Therefore, to prove $\mathcal{S}'$ is external equivalent to $\mathcal{S}$ w.r.t. $\mathcal{C}$ is equivalent to prove that for any input sequence, $\mathcal{S}' \times \mathcal{C}$ and $\mathcal{S} \times \mathcal{C}$ produce the same output sequence in response. In other words, for any path in $\mathcal{S} \times \mathcal{C}$, there exists a path $\mathcal{S}' \times \mathcal{C}$ such that these two paths have the same label; and vice verse.

Let ρ be a path in $\mathcal{S} \times \mathcal{C}$ and σ its corresponding local path in $\mathcal{S}$. Suppose $label(\sigma) = i_1/o_1 \circ i_2/o_2 \circ \ldots \circ i_l/o_l$ for some integer $l \geq 1$. The output o_j $(1 \leq j \leq l-1)$ in $label(\sigma)$ can be classified into two cases: i) o_j is an external output at the environment/tester port; or ii) o_j is an internal output at the context port, i.e., $o_j = req_h$ for some h $(1 \leq h \leq k)$.

Algorithm 1 Construction of an auxiliary FSM $\mathcal{S}'$

```
1: Input: FSM $\mathcal{S}$, $\mathcal{C} = \{\langle req_1, resp_1\rangle, \ldots, \langle req_k, resp_k\rangle\}$.
2: Output: FSM $\mathcal{S}'$.
3: $\mathcal{S}' = \mathcal{S}$;
4: for each state $s$ in $\mathcal{S}$ do
5:   if there exist transitions $t = (s_1, s, i/req_j)$ and $t' = (s, s_2, resp_j/o)$ in $\mathcal{S}'$, where $j \in \{1, \ldots, k\}$, $i \in I$ and $o \in O$ then
6:     add a new state $s'$ into $\mathcal{S}'$;
7:     remove transitions $t$ and $t'$ from $\mathcal{S}'$;
8:     add transitions $(s_1, s', i/req_j)$ and $(s', s_2, resp_j/o)$ to $\mathcal{S}'$;
9:   end if
10: end for
11: output $\mathcal{S}'$;
```

For case i), the corresponding transitions with labels $i_j/o_j \circ i_{j+1}/o_{j+1}$ in $\mathcal{S}$ remain adjacent in $\mathcal{S}'$. For case ii), since $\mathcal{C}$ is stateless and σ is derived from $\mathcal{S} \times \mathcal{C}$, we have $i_{j+1} = resp_h$ where $resp_h$ is the unique response message for req_h. According to the way that we construct $\mathcal{S}'$, the corresponding transitions with labels $i_j/o_j \circ i_{j+1}/o_{j+1}$ in $\mathcal{S}$ are transformed to two transitions adjacent upon a newly introduced state in $\mathcal{S}'$.

As we see, for any j, no matter o_j is in case i) or case ii), there exist adjacent transitions in S' whose labels are i_j/o_j and i_{j+1}/o_{j+1} respectively. Thus, for σ in S, there exists a local path with *label*(σ) in $\mathcal{S}'$ and in turn there exists a path with *label*(ρ) in $\mathcal{S}' \times \mathcal{C}$ for any path ρ in $\mathcal{S} \times \mathcal{C}$.

According to Algorithm 1, for any local path in $\mathcal{S}'$, it is obvious that there exists a local path with the same label in $\mathcal{S}$. Consequently, for any path in $\mathcal{S}' \times \mathcal{C}$, there exists a path $\mathcal{S} \times \mathcal{C}$ with the same label.

Therefore, we have $\mathcal{S}'$ is external equivalent to $\mathcal{S}$ w.r.t. $\mathcal{C}$. $\square$

6.2 Reducing the use of the context

Tests involving the use of context may be very costly, especially when the context is distributed. For example, invoking web services is more time-consuming than executing local transitions; Some web service providers charge fees according to the number of web service invokes. In this case, it is desirable to reduce the use of the context during testing. This problem can be reduced to classic problems in graph theory.

Suppose we consider the T-method for test generation of an IUT with stateless embedded context. Algorithm 2 gives an algorithm that generates an optimal test sequence that traverses each transition at least once and the number of the invokes of the context of the IUT is minimal. We use $G_{\mathcal{S}'} = (V, E)$ to denote the graph representation of FSM $\mathcal{S}'$, which is the auxiliary FSM generated from a given specification FSM $\mathcal{S}$ of the IUT by Algorithm 1. Clearly, $|V|$ is linear to the number of states in $\mathcal{S}$ and $|E|$ is equal to the number of transitions in $\mathcal{S}$.

weight is a weight function which assigns a weight to each edge. When an edge e has an output sending to the port of the context, we assign a very large weight to it; otherwise, $weight(e) = 1$. Here, we use ∞ to denote a very large number. In implementation, we can use $|E|$ instead. Since CPP is to find a tour which traverses each edge at least once with minimal weight, the use of the edges representing the invokes of the context will be minimized.

Assigning a weight to each edge of graph $G_{\mathcal{S}'}$ is linear to its size $|E|$. Consequently, the time complexity of the algorithm is determined by the CPP algorithm, whose best known implementation is in $O(|V|^2|E|^3\log(|V|))$.

Here, we use the T-method as an example to show how to use a weighted graph to reduce the communication with the context. Of course, other test criteria can also be applied.

6.3 An application

In the following, we use a simplified online flight reservation system as a running example to show how to generate a test sequence for an IUT with stateless embedded context.

A partial specification of the IUT is given in Figure 16. In the initial state *ready*, a customer can either login to change his/her reservations or query the list of the available

Algorithm 2 Test generation of the minimal number of context invokes with the T-method

```
1: Input: $G_{S'} = (V, E)$.
2: Output: a test sequence with minimal number of context invokes generated by the T-method.
3: Initialize function $weight : E \rightarrow \mathcal{N}$;
4: for each edge $e = (v, v', i/o)$ in $G_S$ do
5:   if $o$ sends to the context then
6:     $weight(e) = \infty$;
7:   else
8:     $weight(e) = 1$;
9:   end if
10: end for
11: let $\rho$ be the path obtained by applying CPP on the resulting graph;
12: output the input portion of $label(\rho)$;
```

flights without login. Suppose a customer can make three types of changes, namely, *change a seat*, *postpone a flight*, and *cancel a flight.* There are two ways to make these changes: One is to request a change form and fill in the change details in the form, and the other is via a web page that displays the current status of the reserved flight. The latter case needs to query the status of the reservation in advance.

The IUT requests for two kinds of services provided by its context: *status query of a flight* and *flight availability query.* We abstract the service requests and responses as symbols, and assume these services are stateless. The specification of the context is $\{\langle serviceReqStatus, respStatus\rangle, \langle serviceReqFlight, respFlight\rangle\}$.

Note that whenever the IUT requests a service from its context, the service provider, it enters state *wait* for the response. A test executability problem occurs if a test sequence contains *queryStatus* $\circ$ *respFlight* or *queryFlightList* $\circ$ *respStatus.* To avoid this problem we split state *wait* into two states: *waitS* and *waitF* to denote the wait for the response of *queryStatus* and *queryFlightList*, respectively. The corresponding FSM derived by applying Algorithm 1 on the example is shown in Figure 17.

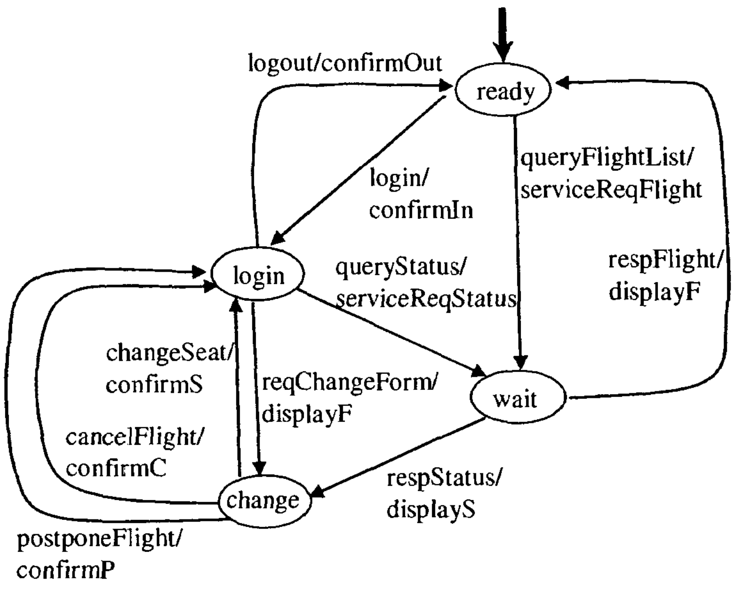


Figure 16: An example: online flight reservation system

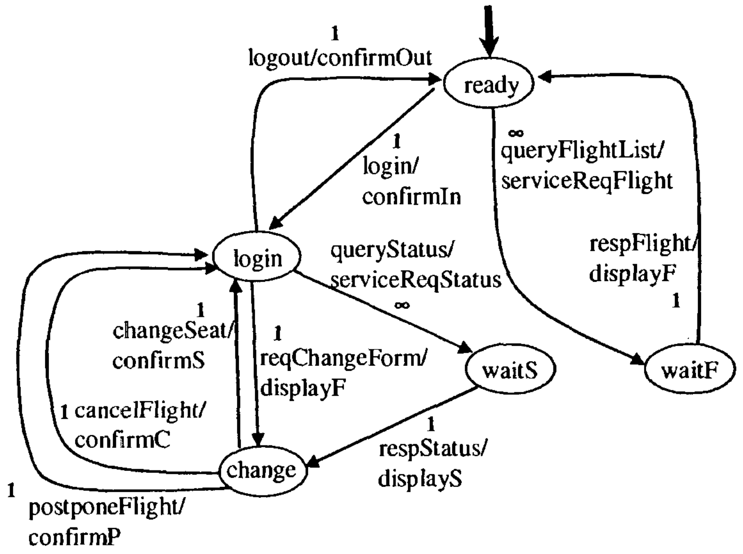


Figure 17: An example with split states and weights

When the service requests to the context is very costly, it is desirable to reduce their uses during testing. According to Algorithm 2, the weight associated with the corresponding edge for each transition is labelled in Figure 17. Note that very large weights are assigned to the transitions involving the service requests to the context. When we want to test transition $(change, login, postponeFlight/confirmP)$ from the initial state *ready*, instead of using the input sequence *login* $\circ$ *queryStatus* $\circ$ *respStatus* (whose total weight is ∞), we use the input sequence *login* $\circ$ *requestChangeForm* (whose total weight is 2) to reach state *change*.

7 Test generation with stateful embedded context[2]

For simplicity, we assume the FSM for an IUT has two ports: one for communicating with its context, called the *context port*; and the other for communicating with the rest part of its environment simulated by a tester, called *environment port*. For clarity, we will use

- I and O as the IUT's input and output at the environment port;
- X and Y as the IUT's input and output at the context port.

The behavior of the IUT is thus given as $\mathcal{S} = \langle S, s_0, I \cup X, O \cup Y, \lambda_s, \delta_s \rangle$.

In order to focus on the major functionality and allow the flexibility for *don't care* cases, the specification of an IUT is usually partially specified in practice. In this case, it is suitable to consider *trace pre-order* $\preceq$ instead of *trace equivalence* in testing in context. $\mathcal{S} \preceq_{\mathcal{C}} \mathcal{M}$ holds if any (input/output) trace allowed by $\mathcal{S} \times \mathcal{C}$ are implemented, yet a trace not specified in $\mathcal{S} \times \mathcal{C}$ may or may not be implemented.

We assume that the specification FSM $\mathcal{S}$ is free from *internal-port-cycles*. An *internal-port-cycle* in an FSM is a path $(s_1, s_2, i_1/o_1)\ (s_2, s_3, i_2/o_2)\ \dots\ (s_k, s_{k+1}, i_k/o_k)$ $(k \geq 2)$ such that $s_1 = s_{k+1}$, and $i_j \notin I$ for all $1 \leq j \leq k$. An *internal-port-cycle* represents a possibly infinite internal communications between the IUT and its context, which is

[2]This chapter is the result of joint research with Dr. Jessica Chen and extracted from the paper titled "An Approach to Testing with Embedded Context using Model Checker", which was published in the proceedings of the Ninth International Conference on Formal Engineering Methods (ICFEM'08), LNCS 5256, pp. 66-85, 2008. Springer-Verlag.

normally considered as a design error called *livelock* [3, 65]. How to guarantee that the design specifications are free from such logical errors can be carried out by formally verifying the correctness of the design specifications.

An input sequence generated from $\mathcal{S}$ cannot be served as an input sequence to test the IUT in its context $\mathcal{I}_c$, as we cannot control the IUT's context port. To take the context into consideration, a possible approach is to develop a testing technique to check whether $\mathcal{M}$ conforms to $\mathcal{S}$ *within context* $\mathcal{C}$ w.r.t. trace pre-order, instead of checking whether $\mathcal{M}$ conforms to $\mathcal{S}$ w.r.t. trace pre-order. That is, we compare the model representing the actually behavior of $(\mathcal{I}, \mathcal{I}_c)$ with the one specifying its expected behavior. Just like we assume that the actual behavior of the IUT can be described by an FSM for testing the IUT in isolation, we assume that the actual behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be described by an FSM.

The model representing the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be derived from the specification of the IUT and that of the context. Suppose that the context specification $\mathcal{C}$ is given as a 1-port FSM. Of course, if it is given in a specification language with higher level of abstraction, we consider its equivalent FSM model. Let

$$\mathcal{C} = \langle C, c_0, \bar{Y}, \bar{X}, \lambda_c, \delta_c \rangle$$

be the specification FSM of the context where $\bar{X} = \{\bar{x} \mid x \in X\}$ and $\bar{Y} = \{\bar{y} \mid y \in Y\}$ are the output and input symbols of $\mathcal{C}$ to communicate with $\mathcal{S}$: $\bar{x}$ and $\bar{y}$ are executed simultaneously with x and y respectively, representing the communications between the IUT and its context. Here we have ignored those actions internal to the context component.

Note that since we have the slow environment assumption, it makes no difference to use synchronous or asynchronous communication mode between the IUT and its context. For simplicity, we consider synchronous communication.

Given $\mathcal{S}$ and $\mathcal{C}$ as the above defined 2-port and 1-port FSMs, the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be described as a synchronous product FSM $\mathcal{S} \times \mathcal{C}$ defined on $\mathcal{S}$ and $\mathcal{C}$ as $\langle S', (s_0, c_0), I, ((O \times Y) \cup X)^*, \lambda, \delta \rangle$. It has only one port with the tester/environment for input. A global state consists of a local state of $\mathcal{S}$ and a local state of $\mathcal{C}$. $S' \subseteq S \times C$ is a set of global states reachable from (s_0, c_0) in the sense that for any $(s, c) \in S'$, there exists an input sequence $\sigma \in I^*$ such that $\delta((s_0, c_0), \sigma) = (s, c)$.

$((O \times Y) \cup X)^*$ is a set of outputs from the tester's viewpoint. As we mentioned in the Introduction, we assume that even though the input/output between the IUT and its context is not controllable, they are observable. Thus, corresponding to each input from the environment, the tester will observe a sequence of outputs which is composed of those outputs $\langle o, y \rangle$ of the transitions in $\mathcal{S}$ ($\langle o, y \rangle \in O \times Y$) and those input x from its context ($x \in X$).

A transition in $\mathcal{S} \times \mathcal{C}$ is derived from a path in $\mathcal{S}$ and a path in $\mathcal{C}$. More precisely, we have transition $((s_1, c_1), (s_2, c_2), i/o)$ in $\mathcal{S} \times \mathcal{C}$, and thus $\lambda((s_1, c_1), i) = o$ and $\delta((s_1, c_1), i) = (s_2, c_2)$, only if we have

$$\lambda_s(s_1, i_1 \dots i_k) = o_1 \dots o_k,\ \delta_s(s_1, i_1 \dots i_k) = s_2,$$

$$\lambda_c(c_1, i'_1 \dots i'_h) = o'_1 \dots o'_h,\ \delta_c(c_1, i'_1 \dots i'_h) = c_2;$$

for $h, k \geq 1$ such that

$$k = h,\ i = i_1,\ o = o_1 \circ i_2 \circ o_2 \dots \circ i_k \circ o_k,$$

$$i'_j = \overline{c(o_j)} \text{ for } 1 \leq j \leq k,\ i_{j+1} = \overline{o'_j} \text{ for } 1 \leq j \leq k-1,\ o'_k = -;$$

or

$$k = h+1,\ i = i_1,\ o = o_1 \circ i_2 \circ o_2 \dots \circ i_k \circ o_k,$$

$$i'_j = \overline{c(o_j)} \text{ for } 1 \leq j \leq k-1,\ i_{j+1} = \overline{o'_j} \text{ for } 1 \leq j \leq k-1,$$

$$o_k = \langle *, - \rangle \text{ where } * \text{ can be any output including -;}$$

Otherwise, $\lambda((s_1, c_1), i) = null$ and $\delta((s_1, c_1), i) = null$. Here $c(o)$ represents the output of o at the context port. In the following, when there is no confusion, we will drop the subscripts of λ and δ.

Since there is no *internal-port-cycle* in $\mathcal{S}$, the above defined product FSM fully describes the expected behavior of the IUT with its context using the slow environment feature. Furthermore, as we assume that $\mathcal{S}$ and $\mathcal{C}$ are minimal and deterministic, the above defined synchronous product of them is also minimal and deterministic.

Once we have a product FSM specification for the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$, it is straightforward to generate a suitable test suite from this product FSM in order to test whether trace pre-order holds between this specification and the implementation FSM of $(\mathcal{I}, \mathcal{I}_c)$. This approach, however, requires that the FSM specification of $\mathcal{I}_c$ be available, and the global model of $(\mathcal{I}, \mathcal{I}_c)$ be calculated, which brings out the state explosion problem. In the present work, we consider using model checker as an auxiliary tool to retrieve necessary information from a context specification in order to generate test sequences. We do not require that the product of $\mathcal{S}$ and $\mathcal{C}$ be actually constructed. In particular, if the specification of the expected behavior of $\mathcal{I}_c$ is given in a specification language of a higher level of abstraction, we do not need to construct its operational model neither.

7.1 The proposed method

To check whether a trace pre-order relation holds between $\mathcal{S} \times \mathcal{C}$ and the implementation FSM of $(\mathcal{I}, \mathcal{I}_c)$, we need to generate a complete test suite to *identify* all the states in $\mathcal{S} \times \mathcal{C}$ using a distinguishing sequence, and *verify* all the transitions in $\mathcal{S} \times \mathcal{C}$ using the same distinguishing sequence. Since the context implementation is known to be correct, we actually only need to generate test sequences to verify *some* of the transitions in $\mathcal{S} \times \mathcal{C}$. Consequently, we can look for a distinguishing sequence that is capable of distinguishing only a subset of states in $\mathcal{S} \times \mathcal{C}$. In this chapter, we characterize such a subset of transitions and a subset of states.

Definition 7 **($\mathcal{R}$ covers T)** *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R} \subseteq T$. $\mathcal{R}$ covers T if for any transition $((s_1, c_1), (s_2, c_2), i/o) \in T$, there exists a transition $t = ((s_1, c_1'), (s_2, c_2'), i/o)$ in $\mathcal{R}$ where $(s_1, c_1), (s_2, c_2), (s_1, c_1')$, and (s_2, c_2') are states in $\mathcal{S} \times \mathcal{C}$, i is an input of $\mathcal{S} \times \mathcal{C}$ and o is an output of $\mathcal{S} \times \mathcal{C}$.*

The transitions in $\mathcal{S} \times \mathcal{C}$ can be partitioned into different groups according to the local states of $\mathcal{S}$ in their starting states, the local states of $\mathcal{S}$ in their ending state, and their input/output pairs. The above definition actually requires that the subset of transitions $\mathcal{R}$ contain at least one representative transition from each of the partitions. The intuition behind is this: Since $\mathcal{S}$ and $\mathcal{C}$ are deterministic, given two states s_1 and s_2 in $\mathcal{S}$, an input i and an output o in $\mathcal{S} \times \mathcal{C}$, there exists exactly one path ρ in $\mathcal{S}$ from s_1 to s_2 with input/output sequence $i_1/o_1 \circ i_2/o_2 \circ \ldots \circ i_k/o_k$ such that $i = i_1$ and $o = o_1 \circ i_2 \circ o_2 \circ \ldots \circ i_k \circ o_k$.

According to the definition of synchronous product, for any states c_1, c_2 in $\mathcal{C}$, if transition $t = ((s_1, c_1), (s_2, c_2), i/o) \in T$, then t is constructed from this path. Consider all such transitions in one partition $G(s_1, s_2, i, o)$. To check that each transition in $G(s_1, s_2, i, o)$ is correctly implemented, we only need to make sure that path ρ is correctly implemented in the sense that there exists a path ρ' in $\mathcal{M}$ which starts from a state identified as s_1, ends at a state verified as s_2, and correctly gives output o in response to input i. Since the context is correct, this implies that all transitions in partition $G(s_1, s_2, i, o)$ are correctly implemented. While any transition in $G(s_1, s_2, i, o)$ can be used to generate a test sequence for the above purpose, we require that the subset $\mathcal{R}$ of transitions contains one transition from each partition $G(s_1, s_2, i, o)$.

As we consider only transitions in such a subset of transitions $\mathcal{R}$ that covers the total set of transitions in $\mathcal{S} \times \mathcal{C}$, we only need a distinguishing sequence to identify all the states appeared as the starting or ending states in the transitions in $\mathcal{R}$, denoted by $states(\mathcal{R})$. In the following, we show that we can further weaken this requirement: it is sufficient to have a distinguishing sequence that can identify, among the states in $states(\mathcal{R})$, all those with different local states of $\mathcal{S}$.

Definition 8 (distinguishing sequence on $\mathcal{S}$ over $\mathcal{W}$) *Let $\mathcal{W}$ be a subset of reachable states in $\mathcal{S} \times \mathcal{C}$. An input sequence $D = i_1 \circ \tilde{x}_1 \circ i_2 \circ \tilde{x}_2 \ldots \circ i_k \circ \tilde{x}_k$ for $i_j \in I$, $\tilde{x}_j \in X^*$ $(1 \leq j \leq k)$ is a distinguishing sequence on $\mathcal{S}$ over $\mathcal{W}$ if*

- *For any state s, $s' \in S$, $s \neq s'$ implies $\lambda(s, D) \neq \lambda(s', D)$.*
- *For any $(s_1, c_1) \in \mathcal{W}$ and for any h $(1 \leq h \leq k)$, the input sequence of X^* obtained from $\lambda((s_h, c_h), i_h)$ by removing all output of Y is $\tilde{x}_h$. Here for $2 \leq h \leq k$, $(s_h, c_h) = \delta((s_1, c_1), i_1 \circ i_2 \ldots \circ i_{h-1})$.*

The above definition can be viewed as an extension of the normal definition of distinguishing sequence of an FSM: A distinguishing sequence of $\mathcal{S}$ over $\emptyset$ is actually the original definition of distinguishing sequence on $\mathcal{S}$ without considering any context.

Note that we do not require an input sequence to distinguish all the states in $\mathcal{S} \times \mathcal{C}$, but a subset of states of interest expressed in $\mathcal{W}$. This brings out two benefits: i) an increased possibility of the existence of a distinguishing sequence; ii) when there exist distinguishing sequences, a possibly shorter one which contributes to the reduction of the cost for carrying out the test.

Now we show that in order to generate from $\mathcal{S} \times \mathcal{C}$ a complete test suite w.r.t. trace pre-order, it is sufficient to consider a subset $\mathcal{R}$ of transitions as long as $\mathcal{R}$ covers its set T of transitions, with a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.

Note that while previous work on this topic for testing in isolation requires *reliable reset*, i.e. the IUT can be reset to its initial state at any time, here we assume that the IUT can be reset to its initial state at any time and its context will be reset at the same time.

Similar to previous work, we assume a bound on the number of states in the implementation FSM of the IUT. When we test an IUT with a context, since the input to the IUT from the context is not *controllable*, the description of the IUT can be considered as a 1-port FSM from the tester's viewpoint. As a consequence, some of the states in a given 2-port FSM are not *stable* in the sense that after an input from the tester/environment, the IUT will never stay in any of those states waiting for the next input from the tester/environment. For testing in context, we consider only stable states: When we say that the number of states in the implementation FSM of the IUT is no more than the number of states in the specification FSM of the IUT, we refer to those states that appear to be the starting states of some transitions with input at the environment port.

With the above assumptions, we present the following result:

Proposition 2 *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$ and $\mathcal{R} \subseteq T$. Let $\mathcal{T}$ be a test suite derived from $\mathcal{S} \times \mathcal{C}$. If*

- *$\mathcal{R}$ covers T,*
- *there exists an input sequence D such that D is a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$, and $\forall t = ((s_1, c_1), (s_2, c_2), i/o) \in \mathcal{R}$, there exists an input sequence σ such that $\sigma \circ D \in \mathcal{T}$, $\sigma \circ i \circ D \in \mathcal{T}$, and* path($\sigma$) *is a path in $\mathcal{S} \times \mathcal{C}$ from (s_0, c_0) to (s_1, c_1),*

then $\mathcal{T}$ of $\mathcal{S} \times \mathcal{C}$ is complete w.r.t. trace pre-order.

This proposition indicates that a desired test suite can be generated by finding a transition set $\mathcal{R}$ and a distinguishing sequence D such that $\mathcal{R}$ covers T and D is a distinguishing sequence over $states(\mathcal{R})$. In the next chapter, we will show how to find $\mathcal{R}$ and D with a model checker.

7.2 Test generation using model checking tools

Model checking tools such as SPIN [44], SMV [15], UPPAAL [4] are originally designed to verify the correctness of design specifications. Recent years have seen trends in applying model checking tools to assist the test generation procedures (see e.g. [73, 57, 72, 70, 23, 24]). When we use a model checker to verify a system model against some required property, a counter-example will be returned if the system model is not correct w.r.t. the property being checked. Making use of this functionality of model checkers, we can characterize a desired test sequence as a property. We use a model checker to verify the negation of this property, called *trap property*, against a system specification. When this trap property is violated, a counter-example returned by the model checker actually serves as a desired test sequence. Following this line of research, we present here another example of using model checkers to generate test sequences in conformance testing with context.

To avoid constructing synchronous product of $\mathcal{S}$ and $\mathcal{C}$, the specifications of the IUT and its context are given to a model checker as a system specification. The specification FSM of the IUT can be straightforwardly translated into any formal specification language accepted model checking tools. For its context, we do not restrict it to be given in a particular specification language or a particular model, as long as it can be translated into a specification language accepted by the adopted model checker. In the following, we use *Spec* to denote the specification for the composition of the IUT and its context given in the specification language of the chosen model checker.

We explain below how to make use of the specification FSM of an IUT and a model checker (with *Spec*) to derive a test suite of the IUT and its context that is complete with respect to trace pre-order.

7.2.1 Finding transitions in $\mathcal{R}$

As we explained in Chapter 7.1, we need to find a subset $\mathcal{R}$ of transitions in $\mathcal{S} \times \mathcal{C}$ such that $\mathcal{R}$ covers T where T is the set of transitions in $\mathcal{S} \times \mathcal{C}$. Since the synchronous product FSM for the IUT and its context is not available, we analyze $\mathcal{S}$ and derive $\mathcal{R}$ via a model checker. Algorithm 3 shows an algorithm to use a model checker to determine a transition set $\mathcal{R}$ such that $\mathcal{R}$ covers T.

Algorithm 3 To find a transition set $\mathcal{R}$

```
1: Input: $\mathcal{S}$, Spec.
2: Output: a set $V$ of pairs of transitions in $\mathcal{S} \times \mathcal{C}$ and input sequences in $I^*$, $\mathcal{R}$.
3: Let $\Phi$ contains all composable paths in $\mathcal{S}$;
4: Let $V = \emptyset$;
5: for each path $\rho$ in $\Phi$ do
6:   define a formula $\phi$ to express the non-existence of a path in Spec which contains a
     subpath which is equal to $\rho$ when all its transitions from the context are ignored.
7:   use model checker to verify formula $\phi$ in Spec;
8:   if formula $\phi$ is violated then
9:     add $(t, \sigma)$ to $V$, where (i) $t \in \mathcal{S} \times \mathcal{C}$ is a transition derived by $\rho$ and a path in $\mathcal{C}$
       defined by the counter-example returned from the model checker; and (ii) $\sigma$ is an
       input sequence in $I^*$ derived from the counter-example that defines a path from
       $(s_0, c_0)$ to the starting state of $t$;
10:  end if
11: end for
12: Let $\mathcal{R} = \{t \mid (t, \sigma) \in V\}$;
13: return $V$ and $\mathcal{R}$;
```

A path $\rho = (s_1, s_2, i_1/o_1) \circ (s_2, s_3, i_2/o_2) \circ \ldots \circ (s_k, s_{k+1}, i_k/o_k)$ in $\mathcal{S}$ is *composable* if $i_1 \in I$, $i_j \in X$ for $2 \leq j \leq k$, and $\delta(s_{k+1}, i) \neq$ *null* for some $i \in I$. According to the definition of synchronous product, any transition $t = ((s, c), (s', c'), i/o) \in T$ is constructed from some composable path. On the other hand, not all composable paths in $\mathcal{S}$ can be used to define a transition in $\mathcal{S} \times \mathcal{C}$. Those that can be used to define a transition in $\mathcal{S} \times \mathcal{C}$ are called *executable paths*. Recall that transitions of T in partition $G(s, s', i, o)$ share the same local state s of the IUT in its starting state, the same local state s' of the IUT in its ending state, and the same input i and output o. Each executable path is actually uniquely used to define all transitions in one of the partitions.

Now, as we want to derive a set $\mathcal{R}$ of transitions that contains at least one (arbitrary) transition in each partition, we can use an executable path ρ in $\mathcal{S}$ to request the model checker to find an arbitrary transition of T that represents the partition uniquely determined by ρ. This can be done as follows: Use temporal logic formula to express such a property that there exists a subpath which is equal to ρ when all its transitions from the context are ignored. Request the model checker to verify the trap property, i.e. the negation of the above property. If ρ is used to define a transition t in $\mathcal{S} \times \mathcal{C}$, then the model checker will detect the violation of the trap property, returning a path in *Spec* from which we can derive a transition in the partition of ρ. Note that in addition to the transition in T, we also derive from the counter-example an input sequence in I^* which defines a path from (s_0, c_0) to the starting state of t. This input sequence will be used later on to construct a test suite.

As statically we do not know which composable path is executable, we simply ask the model checker to check all composable paths. If a composable path is not executable, the model checker will prove the trap property. In this case, we do not need to record any information.

Since $\mathcal{S}$ is finite and free from internal-port-cycles, the number of composable paths in $\mathcal{S}$ is finite and the computation of Φ is in polynomial time. Consequently, the time complexity of Algorithm 3 depends on that of the model checking algorithms used by the model checker. See e.g. [15] for the discussions on the complexity of model checking algorithms. In fact, optimization techniques of model checking have been well studied in recent years to enhance its applicability. Thus, the practicality of Algorithm 3 is endorsed.

According to Algorithm 3, we have the following result.

Proposition 3 *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R}$ the set of transitions obtained from Algorithm 3. We have $\mathcal{R}$ covers T.*

7.2.2 Finding a distinguishing sequence

Algorithms for finding a distinguishing sequence of an FSM are well-discussed in the literature. See [56] for a good survey on this topic. However, finding a distinguishing sequence of an FSM in context is much more complicated. Due to the fact that a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$ must be calculated with both the specification of the IUT and that of its context, while synchronous product FSM of them is not available, we will apply model checker again. In [75], the authors presented an approach to generating a distinguishing sequence of an EFSM with UPPAAL model checker [4]. Here, we adopt the idea of this approach to generate a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.

Algorithm 4 To find a distinguishing sequence over $states(\mathcal{R})$

```
1: Input: Spec, $\mathcal{R}$.
2: Output: a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.
3: for each state $(s, c)$ in $states(\mathcal{R})$ do
4:     create a variant of Spec with $(s, c)$ as its initial state;
5: end for
6: create a monitor process to synchronize all variants in the sense that a variant can only
   accept an input if all others accept the same input simultaneously;
7: define a formula $\phi$ to express the property that there does not exist an input sequence
   such that the corresponding output sequences produced by any two variants with dif-
   ferent local states of $\mathcal{S}$ as their initial states are all different;
8: request model checker to verify $\phi$ in Spec;
9: if model check detects a violation then
10:    Let $D$ be the input sequence derived from the counter-example returned by the model
       checker;
11:    return $D$;
12: else
13:    return "There does not exist any distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$";
14: end if
```

Algorithm 4 shows an algorithm for this purpose. Initially, for each state $(s, c) \in$

$states(\mathcal{R})$, we create a variant of $\mathcal{S}$ with s as its initial state and a variant of $\mathcal{C}$ with c as its initial state. Then by making use of a special *monitor* process, we request all the processes that represent these variants of $\mathcal{S}$ to synchronize all their actions on accepting input from both the environment port and the context port so that they will always accept the same input at the same time. For any two variants whose local states of $\mathcal{S}$ in their initial states are different, if the output sequences produced upon a same input sequence are all different, then the input sequence can be used as a desired distinguishing sequence D on $\mathcal{S}$ over $states(\mathcal{R})$.

As we know, not every FSM has a distinguishing sequence, In our setting, we cannot guarantee either their existence. However, as distinguishing sequences very often exist in real-life examples, the distinguishing sequences in our setting also exist in many application examples.

The problem of finding a distinguishing sequence is PSPACE-hard by itself [56]. Algorithm 4 reduces the problem to an application of model checking tools. This allows us to benefit from important features that they provide, such as the efficient partial order reduction and OBDD, and thus, reduce the actual cost for the computation.

Finally, with V and D, a test suite $\mathcal{T}$ is obtained: For each $(t, \sigma) \in V$, add both $\sigma \circ D$ and $\sigma \circ i \circ D$ to $\mathcal{T}$, where i is the input of t.

7.3 An application

In the following, we use Inter-library Loan System (ILS) as a running example and we use SPIN [43] as a supporting model checker to show how to use the proposed technique to generate a complete test suite w.r.t. trace pre-order for testing in context.

SPIN targets the efficient verification of a system model against the required properties on-the-fly. Here, the system model is described in Promela [43] and the required system properties are often expressed in Linear Temporal Logic (LTL) formulas. As a matter of fact, a design specification expressed in many other specification languages such as FSM and EFSM can be easily translated into a Promela model.

A simplified ILS consists of two components: a borrowing library and a lending library. A user at the borrowing library can search a book in the lending library. When a book is found, the user can choose either to purchase the book or to issue a loan request. The lending library will always grant the purchase of the book; however, the allowance of the loan of the book depends both on the availability of the required book and on the length of the waiting list. There are three cases: i) if the book is available, the loan request will be granted; ii) if the book is unavailable but the waiting list is not full, the lending library will ask the user if he/she wants to make a reservation; and iii) if the waiting list is full, the lending library will tell the user that the book is unavailable.

Suppose that the borrowing library is the IUT and the lending library is its context. The specification $\mathcal{S}$ of the IUT has two ports: *portUser* and *portContext*. Port *portUser* represents the interface of the borrowing library with the environment/tester, and port *portContext* represents the interface of the borrowing library with its context, the lending library. The semantics of service primitives used in ILS can be inferred by their symbolic representations. For example, *searchBook* is an input primitives at *portUser* to represent a user's action of searching a book; *loanAccptd* is an input primitives at *portContext* to represent that a user's request of a book loan is accepted.

Figure 18, Figure 19, and Figure 20 give the specification FSM $\mathcal{S}$ of the borrowing library, the specification extended FSM and the Promela model of the lending library $\mathcal{C}$, respectively. Suppose that the number of available books is 3, and the length of the waiting list for a book reservation cannot exceed 3. Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R} \subseteq T$. To find $\mathcal{R}$ such that $\mathcal{R}$ *covers* T and to find a distinguishing sequence over $states(\mathcal{R})$, we need to translate FSM $\mathcal{S}$ and the behavior of a user of the ILS into Promela processes. Thus, there are three processes in the Promela model of ILS: *User*, *Borrower* and *Lender*, which represent the specifications of the environment/tester, the borrowing library, and the lending library, respectively. To establish the communication among these processes, there are four channels.

- *fromUser*: a channel through which *Borrower* receives inputs from *User*;
- *ToUser*: a channel through which *Borrower* sends outputs to *User*;
- *fromLender*: a channel through which *Borrower* receives inputs from *Lender*;
- *ToLender*: a channel through which *Borrower* sends outputs to *Lender*.

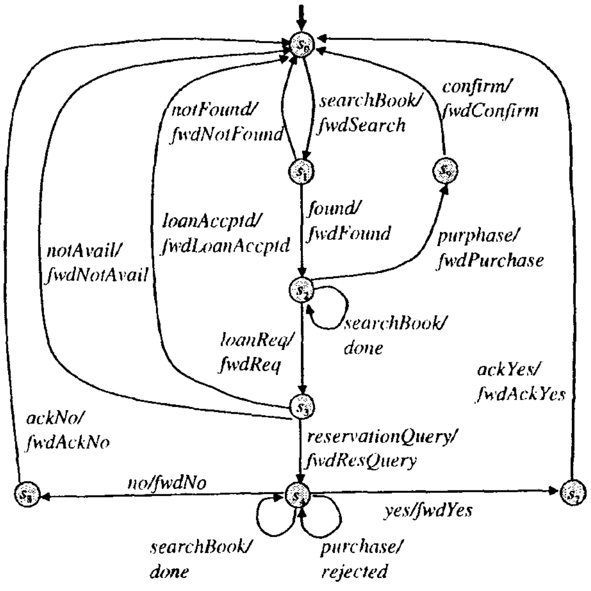


Figure 18: Specification FSM of the borrowing library

Now we show how to find $\mathcal{R}$. Let $\rho = loanReq/\langle -, fwdReq\rangle \circ notAvail/\langle fwdNotAvail, -\rangle$. Clearly, ρ is a composable path in $\mathcal{S}$. In order to use SPIN to check whether ρ is executable, we need an LTL formula to express the negation of the existence of a transition in $\mathcal{S} \times \mathcal{C}$ derived from ρ.

Since the sending actions are always executable, we focus on finding a path to enable the receiving actions in ρ. Let the temporal logic variables be defined as follows:

$r = Borrower@s_2$

$p = fromUser?[loanReq]$

$q = fromLender?[notAvail]$

Here, r represents that process $Borrower$ is in state s_2; p represents that message $loanReq$ is received from channel $fromUser$; and q represents that message $notAvail$ is received from channel $fromLender$. Then the desired trap LTL formula can be expressed as

$\phi = !(<> (rUp)Uq)$.

When verifying the ILS Promela model against ϕ, we obtain the following result from

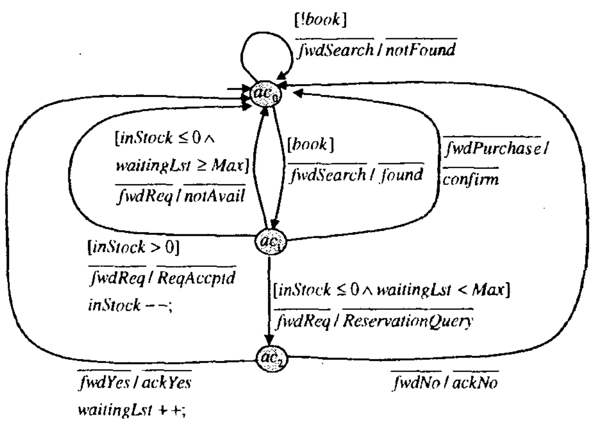


Figure 19: Specification extended FSM of the lending library

the returned counter-example:

$\sigma = searchBook \circ loanReq \circ searchBook \circ loanReq \circ searchBook \circ loanReq \circ searchBook \circ loanReq \circ yes \circ searchBook \circ loanReq \circ yes \circ searchBook \circ loanReq \circ yes \circ searchBook$

$t = ((s_2, c_{1,2}), (s_0, c_{0,4}), loanReq/\langle -, fwdReq\rangle \circ notAvail/\langle fwdNotAvail, -\rangle)$,

where $c_{0,4}$ and $c_{1,2}$ are concrete states split from abstract state ac_0 and ac_1 in the situation when $inStock = 0$ and $waitingLst = 3$, respectively.

This result actually describes a possible scenario of having a transition in $\mathcal{S} \times \mathcal{C}$ derived from ρ when all the books in the lending library are checked out and the waiting list is full.

As shown in [56], the role of distinguishing sequences can actually be replaced by their prefixes, one for each state. This very often helps us achieve shorter test sequences. The definition of a distinguishing sequence over $\mathcal{W}$ can be extended to prefix distinguishing sequences D_i (for state s_i) straightforwardly. Following Algorithm 4, prefix distinguishing sequence D_i over $states(\mathcal{R})$ can be found with SPIN. For example, we have $D_0 = searchBook$ and $D_2 = D_4 = searchBook \circ purchase$. Thus, test sequences for t are $\sigma \circ D_2$ and $\sigma \circ loanReq \circ D_0$.

```
proctype Lender() {
bool book; /*initialization*/
int inStock = 3; /*No. of available books*/
int waitingLst = 0;
int Max = 3; /*the maximum length of waiting list*/
if
:: book = true;
:: book = false;
fi;

ac0: /*label ac0 is associated with abstract state $ac_0$*/
if
:: book == true → toLender ? fwdSearch → fromLender ! found;
:: book == false → toLender ? fwdSearch → fromLender ! notFound → goto ac0
fi;

ac1: /*label ac1 is associated with abstract state $ac_1$*/
if
:: toLender ? fwdReq;
        if
        :: inStock > 0 → fromLender ! loanAccptd → inStock-- → goto ac0
        :: (inStock <= 0) ∧ (waitingLst >= Max) → fromLender ! notAvail → goto ac0
        :: (inStock <= 0) ∧ (waitingLst < Max) → fromLender ! reservationQuery
        fi;
:: toLender ? fwdPurchase → fromLender ! confirm → goto ac0
fi;

ac2: /*label ac2 is associated with abstract state $ac_2$*/
if
:: toLender ? fwdYes → fromLender ! ackYes → waitingLst++ → goto ac0
:: toLender ? fwdNo → fromLender ! ackNo → goto ac0;
fi;
}
```

Figure 20: Promela model of the lending library

8 Conclusion

In this dissertation, we presented techniques to generate a minimal-length test sequence from the given specifications of a stateful IUT and its embedded context, either stateless or stateful, while avoiding the state explosion problem during test generation and avoiding the test executability problem during testing in context.

In particular, when the context is stateful, we provided a way of implementing our method by making use of model checking tools. As an initial piece of work on testing in context with model checkers, our focus has been put on the general method. Further improvements can be made in terms of the size of the constructed test suite. For example, we can adopt those model checkers that can always find *shortest* counter-examples in terms of the lengths so that shorter test sequences can be derived. Apart from the optimization issue, there are many other directions to extend our current work.

- It remains interesting to discuss our test generation technique in more general situations where both the IUT and its context have communications with the environment.
- IUT may be nondeterministic: we would like to study how to extend our results to nondeterministic testing in context.
- When the IUT is completely specified, it is possible to achieve (global) trace equivalence. However, this is not trivial due to the interoperability of the IUT and its context. For example, when some transitions in $\mathcal{S}$ are intrinsically non-executable w.r.t. context $\mathcal{C}$, we cannot generate executable tests to verify these transitions. On the other hand, the corresponding transitions in a faulty IUT may be executable, and thus there are more traces in $I_{\mathcal{S}} \times \mathcal{C}$ than those in $\mathcal{S} \times \mathcal{C}$. To reach trace equivalence in such cases, adapting failure semantics could be a promising solution.
- We have used distinguishing sequence for state identification. At expense of its convenience for testing, distinguishing sequence does not always exist. Although the use of characterization set usually results in much bigger test suites, a characterization set is more likely to exist in an FSM with context. Therefore, we would like to study on how to use model checking tools to generate characterization set in our setting.

References

[1] A. Aho, A. Dahbura, D. Lee, and M. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. *IEEE Trans Comm.*, 39(11):1604–1615, 1991.

[2] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate test from specifications. In *Proc. of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, 1998.

[3] J. H. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[4] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Proc. of the DIMACS/SYCON workshop on Hybrid systems III : verification and control: verification and control*, pages 232–243, 1995.

[5] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer, 1999.

[6] G. Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga. Automating the process of test derivation from SDL specifications. In *Proc. of 8th SDL Forum*, 1997.

[7] D. Brand and P. Zafiropulo. On communicating finite state machines. *Journal of ACM*, 30(2):323–342, 1983.

[8] J. Chen, R. M. Hierons, H. Ural, and H. Yenigun. Eliminating redundant tests in a checking sequence. In *Proc. of the 18th IFIP International Conference on Testing of Communicating Systems (TestCom 2005), LNCS 3502*, pages 146–158, 2005.

[9] M. S. Chen, Y. Choi, and A. Kershenbaum. Approaches utilizing segment overlap to minimize test sequences. In *Proc. of the IFIP WG6.1 Tenth International Symposium on Protocol Specification, Testing and Verification*, pages 85 – 98, 1990.

[10] W.-H. Chen and H. Ural. Minimum-cost synchronizable test sequences based on multiple uios. *IEEE/ACM Transactions on Networking*, 3(2):152–157, 1995.

[11] T. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, SE-4(3):178–187, 1978.

[12] W. Chun and P. D. Amer. Improvements on UIO sequence generation and partial UIO sequences. In *Proc. of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification*, pages 245 – 260, 1992.

[13] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM transactions on programming languages and systems*, 8(2):244–263, 1986.

[14] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order reduction. *Intl Journal of Software Tools for Technology Transfer*, 2:279–287, 1999.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.

[16] R. De Nicola and R. Segala. A process algebraic view of Input/Output Automata. *Theoretical Computer Science*, 138:391–423, 1995.

[17] R. de Vries and J. Tretmans. On-the-fly conformance testing using Spin. *International Journal on Software Tools for Technology Transfer*, 2(4):382–393, 2000.

[18] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. Automated unique input output sequence generation for conformance testing of FSMs. *The Computer Journal*, 49(3):331–344, 2006.

[19] L. Duan and J. Chen. Alternative beta-sequences. In *Proc. of the Seventh International Conference on Quality Software (QSIC'07)*, pages 127–136, 2007.

[20] L. Duan and J. Chen. Reducing test sequence length using invertible sequences. In *Proc. of 9th International Conference on Formal Engineering Methods (ICFEM'07), LNCS 4789*, pages 171–190, 2007.

[21] L. Duan and J. Chen. An approach to testing with embedded context using model checker. In *Proc. of 10th International Conference on Formal Engineering Methods (ICFEM'08). To appear*, 2008.

[22] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc routing problems, part II: the Rural Postman Problem. *Operations Research*, 43:399–414, 1995.

[23] K. El-Fakih, A. Petrenko, and N. Yevtushenko. FSM test translation through context. In *Proc. of TestCom 2006, LNCS 3964*, pages 245–258, 2006.

[24] K. El-Fakih and N. Yevtushenko. Fault propagation by equation solving. In *Proc. of IFIP 24th International Conference on Formal Techniques for Networked and Distributed Systems, LNCS 3235*, pages 185–198, 2004.

[25] S. Fujiwara and G. von Bochmann. Testing non-deterministic state machines with fault coverage. In *Proc. of IFIP TC6/WG6.1 International Workshop on Protocol Test Systems IV*, pages 267–280, 1991.

[26] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *ACM SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.

[27] A. Gill. *Introduction to the theory of finite-state machines.* McGraw-Hill, 1962.

[28] S. M. Gobershtein. Check words for the states of a finite automaton. *Kebernetika*, 1:46–49, 1974.

[29] G. Gonenc. A method for the design of fault detection experiments. *IEEE Trans. Computers*, 19(6):551–558, 1970.

[30] R. Gotzhein and F. Khendek. Compositional testing of communication systems. In *Proc. of TestCom 2006, LNCS 3964*, pages 227–244, 2006.

[31] O. Grumberg and H. Veith, editors. *25 Years of Model Checking: History, Achievements, Perspectives, LNCS 5000.* Springer, 2008.

[32] F. Hennie. Fault detecting experiments for sequential circuits. In *Proc. of 5th Ann. Symp. Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

[33] R. M. Hierons. Extending test sequence overlap by invertibility. *The Computer Journal*, 39(4):325–330, 1996.

[34] R. M. Hierons. Testing from a finite state machine: extending invertibility to sequences. *The Computer Journal*, 40(4):220–230, 1997.

[35] R. M. Hierons. Testing from a non-deterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers*, 53(10):1330–1342, 2004.

[36] R. M. Hierons. Applying adaptive test cases to nondeterministic implementations. *Information Processing Letters*, 98(2):56–60, 2006.

[37] R. M. Hierons. Separating sequence overlap for automated test sequence generation. *Automated Software Engineering*, 13(2):283–302, 2006.

[38] R. M. Hierons and M. Harman. Testing conformance to a quasi-non-deterministic stream X-machine. *Formal Aspects of Computing*, 12(6):423–442, 2000.

[39] R. M. Hierons and H. Ural. Reduced length checking sequences. *IEEE Transactions on Computers*, 51(9):1111–1117, 2002.

[40] R. M. Hierons and H. Ural. UIO sequence based checking sequences for distributed test architectures. *Information and Software Technology*, 45(12):793–803, 2003.

[41] R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE Trans. Computers*, 55(5):618–629, 2006.

[42] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 2004.

[43] G. Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[44] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[45] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proc. of IEEE ICSE'03*, pages 232–242, 2003.

[46] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2001.

[47] E. P. Hsieh. Checking experiments for sequential machines. *IEEE Trans. Computer*, C-20:1152–1166, 1971.

[48] ISO/IEC 9646. *Information technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 1-7.* ISO, June 1996.

[49] ISO/TC97/SC21/WG1/DIS9074, editor. *Estelle - A formal description technique based on an extended state transition model.* ISO, 1987.

[50] ITU-T, editor. *Recommendation Z.100.* International Telecommunication Union, 1992.

[51] W. Johnson, J. Porter, S. Ackley, and D. Ross. Automatic generation of efficient lexical processors using finite state techniques. *Communications of the ACM*, 11(12):805–813, 1968.

[52] Z. Kohavi. *Switching and finite automata theory.* New York: McGraw-Hill, 2nd edition, 1978.

[53] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62:21–46, 2002.

[54] D. Lee, K. K. Sabnani, D. M. Kristol, and S. Paul. Conformance testing of protocols specified as communicating finite state machines-a guided random walk based approach. *IEEE Transactions on Communications*, 44(5):631–640, 1996.

[55] D. Lee and M. Yannakakis. Testing finite state machines: state identification and verification. *IEEE Tran. Computers*, 43:306–320, 1994.

[56] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of The IEEE*, 84(8):1090–1123, 1996.

[57] L. P. Lima and A. R. Cavalli. A progmatic approach to generating test sequences for embedded systems. In *Proc. of 10th International Workshop on Testing of Communicating Systems*, pages 125–140, 1997.

[58] G. Luo, G. Bochmann, and A. Petrenko. Test selection based on communicating nondeterministic finite state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20:149–162, 1994.

[59] G. Luo, A. Das, and G. von Bochmann. Generating tests for control portion of SDL specification. In *Proc. of Protocol Test Systems VI*, pages 51–66, 1994.

[60] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[61] G. Mealy. A method for synthesizing sequential circuits. *Journal of Bell System Tech*, 34:1045–1079, 1955.

[62] P. Merlin and G. von Bochmann. On the construction of submodule specifications and communications protocols. *ACM Trans. Programming Languages and Systems*, 5(1):1–25, 1983.

[63] R. Miller and S. Paul. On the generation of minimal length conformance tests for communications protocols. *IEEE/ACM Transactions on Networking*, 1(1):116–129, 1993.

[64] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[65] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[66] E. Moore. Gedanken-experiments on sequenctial machines. *Automata Studies*, 34:129–153, 1956. Princeton Univ. Press.

[67] H. Motteler, A. Chung, and D. Sidhu. Fault coverage of UIO-based methods for protocol testing. In *Proc. of IFIP TC6/WG6.1 6th International Workshop on Protocol Test Systems*, pages 21–33, 1994.

[68] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. In *Proc. of 11th. IEEE Fault Tolerant Computing Symposium*, pages 238–243, 1981.

[69] Object Management Group. *Unified Modeling Language 2.1.2.* OMG, 2007.

[70] A. Petrenko and N. Yevtushenko. Testing faults in embedded components. In *Proc. of 10th International Workshop on Testing of Communicating Systems*, pages 272–287, 1997.

[71] A. Petrenko, N. Yevtushenko, A. Lebedev, and A. Das. Nondeterministic state machines in protocol conformance testing. In *Proc. of IFIP TC6/WG6.1 International Workshop on Protocol Test systems VI*, pages 363–378, 1993.

[72] A. Petrenko, N. Yevtushenko, and G. von Bochmann. Fault models for testing in context. In *Proc. of Internation Conference on Formal Techniques for Networked and Distributed Systems*, pages 125–140, 1996.

[73] A. Petrenko, N. Yevtushenko, G. von Bochmann, and R. Dssouli. Testing in context: framework and test derivation. *Computer Communications*, 19(14):1236–1249, 1996.

[74] I. Pomeranz and S. M. Reddy. Test generation for multiple state-table faults in finite-state machines. *IEEE Transactions on Computers*, 46:783–794, 1997.

[75] C. Robinson-Mallett, P. Liggesmeyer, T. Mcke, and U. Goltz. Generating optimal distinguishing sequences with a model checker. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[76] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 4(15):285–297, 1988.

[77] K. Saleh, H. Ural, and A. Williams. Test generation based on control and data dependencies within system specifications in SDL. *Computer Communications*, 23(7):609–627, 2000.

[78] Y. N. Shen and F. Lombardi. Protocol conformance testing using multiple UIO sequences. *IEEE Transactions on Communications*, 40(8):1282–1287, 1992.

[79] D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, 1989.

[80] Q. M. Tan, A. Petrenko, and G. Bochmann. Modeling basic LOTOS by FSMs for conformance testing. In *Proc. of 15th International Symposium on Protocol Specification, Testing and Verification (PSTV 15)*, pages 137-152, 1995.

[81] K. T. Tekle, H. Ural, M. C. Yalcin, and H. Yenigun. Generalizing redundancy elimination in checking sequences. In *Proc. of ISCIS'05, LNCS 3733*, pages 915–926, 2005.

[82] J. Tretmans. Conformance testing with labelled transition systems: Implementation relation and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[83] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software Concepts and Tools*, 17(3):103–120, 1996.

[84] J. Tretmans. Testing concurrent systems: a formal approach. In *Proc. of CONCUR'99, LNCS 1664*, pages 46-65, 1999.

[85] H. Ural, X. Wu, and F. Zhang. On minimizing the lengths of checking sequences. *IEEE Transactions on Computers*, 46(1):93–99, 1997.

[86] H. Ural and F. Zhang. Reducing the lengths of checking sequences by overlapping. In *Proc. of IFIP TestCom'06, LNCS 3964*, pages 274–288, 2006.

[87] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *Proc. of FATES 2003*, pages 86–100, 2003.

[88] M. P. Vasilevskii. Failure diagnosis of automata. *Kibernetika*, 4:98–108, 1973.

[89] G. von Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *Proc. of Protocol Test Systems 1991*, pages 17–30, 1991.

[90] G. von Bochmann and C. A. Sunshine. Formal methods in communication protocol design. *IEEE Trans. Comm.*, 28:624–631, 1980.

[91] W3C Recommendation. *SOAP specification.* World Wide Web Consortium, 2007.

[92] B. Wang and D. Huthinson. Protocol testing techniques. *Computer Communications*, 10:79–87, 1987.

[93] C. West. General technique for communications protocol validation. *IBM J. Res. Develop.*, 22:393–404, July 1978.

[94] M. C. Yalcin and H. Yenigun. Using distinguishing and UIO sequences together in a checking sequence. In *Proc. of TestCom 2006, LNCS 3964*, pages 259–273, 2006.

[95] B. Yang and H. Ural. Protocol conformance test generation using multiple UIO sequence with overlapping. *ACM SIGCOMM Computer Communication Review*, 20(4):118–125, 1990.

[96] N. V. Yevtushenko, A. V. Lebedev, and A. F. Petrenko. On checking experiments with nondeterministic automata. *Automatic Control and Computer Sciences*, 16:81–85, 1991.

[97] P. Zafiropulo, C. West, H. Rudin, D. Cowan, and D. Brand. Towards analyzing and synthesizing protocols. *IEEE Trans. Commun.*, COM-28:651–661, 1980.

Vita Auctoris

Lihua Duan was born in 1976 in Taiyuan, China. She graduated from Beijing University of Posts and Telecommunications (Beijing, China), where she received a Bachelor's degree in Electrical and Electronic Engineering in 1999. In 2005, she obtained her Master's degree in Computer Science from the University of Windsor, Canada. She is currently a Ph.D's candidate in the School of Computer Science at the University of Windsor and expects to graduate in summer, 2009.