ENTROPY AND SOFTWARE SYSTEMS: TOWARDS AN INFORMATION-THEORETIC FOUNDATION OF SOFTWARE TESTING

By

LINMIN YANG

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2011

To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of LINMIN YANG find it satisfactory and recommend that it be accepted.

Zhe Dang, Ph.D., Co-Chair

Thomas R. Fischer, Ph.D., Co-Chair

Min Sik Kim, Ph.D.

ACKNOWLEDGEMENT

I would like to thank Dr. Zhe Dang and Dr. Thomas R. Fisher for being such wonderful advisors. They have provided me with consistent guidance and strong support throughout my Ph.D studies. They are knowledgeable researchers who are always enthusiastic on the research work. Working with the two exceptional professors is inspiring, encouraging, and of so much fun. I have learned not only the skills, but also the right attitude to do research from them.

I am grateful to Dr. Min Sik Kim for participating in my committee. I thank him for the invaluable comments and suggestions on my dissertation work. Thanks to Dr. Li Tan for his kind help on our research work. I would also like to thank Dr. Oscar Ibarra for all the discussions and communications. It is my great honor to work with such an outstanding scholar. I am also deeply thankful to Dr. Ali Saberi who encouraged me a lot when I just became a graduate student five years ago. I also thank Dr. Sandip Roy for his support and encouragement during the past five years: you are not only a great professor, but also a good friend. I must also thank all the faculty and staff in the School of EECS at Washington State University; they have provided tremendous help to students, and I am so proud to be part of our school.

I am very fortunate to have so many good friends. They are Yong Wang, Yan Wan, Haibo Zhu, Zheng Wen, Tao Yang, Xu Wang, Kevin Chang, Mengran Xue, Cewei Cui and many many more. Thank you for having fun together.

I am greatly indebted to my parents for their unconditional love. They have always been supportive no matter what happens. No words can express my thanks to them.

Finally, I would like to thank my dear husband Xiangjun for always being there for me, and thank our baby-on-the-way for the laughs and happiness you bring to us: we love you before we meet you.

ENTROPY AND SOFTWARE SYSTEMS: TOWARDS AN INFORMATION-THEORETIC FOUNDATION OF SOFTWARE TESTING

Abstract

by Linmin Yang, Ph.D.
Washington State University
May 2011

Chair: Zhe Dang and Thomas R. Fischer

In this dissertation, we introduce an information-theoretic framework for software testing and integrate information theory into it. We first propose a syntax-independent coverage criterion for software testing. We use entropy in information theory to measure the amount of uncertainty in a software system, and we show how the amount of uncertainty decreases when we test the system. We model the system under test as a random variable, whose sample space consists of all possible behavior sets over a known interface. The entropy of the system is measured as the Shannon entropy of the random variable. In our criterion, the coverage of a test set is measured as the expected amount of entropy decrease, or the expected amount of information gained. Since our criterion is syntax-independent, we study the notion of information-optimal software testing where a test set is selected to gain the most information. Furthermore, we introduce the behaviorial complexity as a novel complexity metric for labeled graphs (which can be interpreted as control flow graphs, design specifications, etc.), which is also based on information theory. We also study how the behavioral complexity changes when graphs are composed together, and we show that behavioral complexity can increase when units are sequentially composed through loops, or composed in parallel through synchronization. Our results can be very helpful for software developers: they can use the metric to predict the complexity of the semantics of a software system to be built from their designs. Our results also suggest that integration testing is necessary after units are tested.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

Dedication

To my dear husband Xiangjun Fan,

and our baby-on-the-way.

CHAPTER 1

INTRODUCTION

Software testing (roughly speaking, evaluating software by running it) is still the most widely accepted approach for quality assurance of software systems with nontrivial complexity. Why do we need to test software systems? Essentially, we test a software system since there is uncertainty in its actual behaviors. The uncertainty comes from the fact that behaviors of the software system are too hard to be analytically analyzed (e.g., the software system is Turing-complete), or even not available to analyze (e.g., the software system under test is a black-box). In other words, the actual behaviors (i.e., semantics) of the software system are (at least partially) unknown. In our opinion, software testing is an approach to resolve the uncertainty, and it gains knowledge of a software system by running it, which resembles opening the box to learn the situation of the famous Schrödinger's cat [58], or more intuitively, the fact that opening a box of chocolates resolves the uncertainty of what kinds of chocolates are in the box.

How is "uncertainty" defined in mathematics? Shannon entropy, or simply entropy, is specifically used to measure the amount of uncertainty in an object in information theory [59, 19], which is a well-established mathematical theory underpinning all modern digital communications. Can entropy in information theory be used to characterize the uncertainty in a software system? Our answer is yes, but we have to deal with the following challenges first:

1. Information theory is a probability-based theory. People may argue that there are no probabilities existing in software systems. In reality, people use probabilities to measure the distribution of an object over a measurable space, and it is a useful way to handle the uncertainty. In the aforementioned example Schrödinger's cat, at any moment, the cat could be either dead or alive, and the answer is deterministic. However, since we cannot open the box and see the cat, we cannot give the answer directly. In this case, probabilities can be used to model the state of the cat. The probability that the cat is dead or alive might be meaningless

for one Schrödinger's cat; however, it is statistically meaningful for a large number of cats. Similarly, for software systems, though probabilities might not be meaningful for one particular system, they are useful for a class of software systems. Additionally, we do not need any pre-assigned probabilities to calculate the entropy of a software system; instead, we always consider the *worst-case* entropy, where we do not know any additional information about the system except its specification, and we calculate probabilities that achieve the maximal entropy.

2. In information theory, entropy is defined on a random variable with no internal structures and also generalized to a sequence of random variables (i.e., a random process). However, in computer science, the subjects of testing are software systems which are structural, e.g., software systems modeled as labeled graphs. Therefore, there is need to develop an information theory on structural random variables and the procedure of how the uncertainty of the structured random variables is resolved. Basically, in our approach, a software system is modeled as either a structured random variable or a random process, which will be illustrated in the following chapters.

What merits can this information-theoretic approach bring to software testing? The most desirable property of Shannon entropy is that the Shannon entropy of a discrete random variable remains unchanged after a one-to-one function is applied [19]. Such a characterization is of great importance, since we find a way to describe a software system based on its internal meanings (i.e., semantics), instead of its appearance (i.e., syntax). We explore the usefulness of this information-theoretic approach in two applications:

1. This information-theoretic approach provides a syntax-independent coverage criterion for software testing. For instance, consider a component-based system which is a nondeterministic choice $C_1 \Box C_2$ over two components C_1 and C_2. C_1 is modeled using statecharts [31] in standardized modeling language UML [1], while C_2 is modeled using logical expressions

such as LTL formulas [18]. Suppose that we use the branch-coverage [4] criterion and the property-coverage criterion [64] as testing criteria for C_1 and C_2, respectively. We also have a test set t which consists of two subsets t_1 and t_2, which are test sets for components C_1 and C_2, respectively. It would be impossible to obtain a coverage that the test set t achieves on the whole system, even if we already have the branch coverage that t_1 achieves on C_1 and the predicate coverage that t_2 achieves on C_2. On the contrary, our information-theoretic approach can overcome this problem, since our approach is syntactic independent, and it does not care whether a system is modeled as a graph or a formula, as long as its semantics remains the same. Moreover, this information-theoretic approach can help us develop optimal testing strategies in choosing test cases. For a syntax-based test adequacy criterion, if it returns the same adequacy degree for two test sets, then the two test sets are indistinguishable. For instance, every branch is born equal in branch coverage criterion. However, this is not intuitively true. In our information-theoretic approach, we choose the branch that can reduce the entropy most.

2. This information-theoretic approach provides a novel software complexity metric that is semantics-based, independent of the syntactic appearance of the software system. We have noticed that almost all of the existing software complexity metrics are *syntax-dependent*. In other words, the metrics are measured on the syntactic appearance of the software system instead of its semantics (i.e., meanings). For instance, two control graphs with the same topology but with different initial nodes might have dramatically different behaviors. A syntax-dependent metric (e.g., the McCabe Metric [45]) would give the same complexity measure to the two control graphs, which is obviously not sufficient. Our syntax-independent complexity metric would provide a better way to interpret software systems. Furthermore, our complexity metric measures a system from the perspective of software testing: it intends to asymptotically measure the cost of exhaustive testing of the system. Though exhaustive

testing usually is not possible, the asymptotical cost is naturally a good indicator of the complexity of software systems. We also show that how the complexity of a component-based software system changes when units are composed together. This approach provides a guidance on grading the importance of units in a component-based system – it is natural that we consider units containing more information of greater importance. Such a criterion can be very useful, for example, when we distribute cost in testing units within a component-based system.

There are two totally different views in modeling a software system: the branching time view and the linear time view. The two views are first introduced in [54], and then widely used in model checking [18]. In the branching time view, a system is modeled as a tree, where at one time, there are many possible futures. This branching time view is adopted in the computation-tree logic (CTL) [18] in model checking. In the linear time view, a system is modeled as a set of sequences, where at one time, there is only one future. This linear time view is adopted in the linear-time logic (LTL) [18] in model checking. Notice that the two views are incomparable with each other: in the branching time view, the system is modeled as a tree automaton, while in the linear time view, the system is actually modeled as a Büchi automaton. In our work, we model software systems from the above two views, which correspond to the two aforementioned applications, respectively. Our work is outlined as follows.

In Chapter 2, we propose a syntax-independent coverage criterion for software testing. We treat the system under test in as a black-box [8, 48, 4], about which we only know its input-output interface. We model the specification of the system as a tree, and the actual behaviors (with respect to the specification) of the system as a subtree of the specification. Before testing, we do not know the actual behaviors of the system. Therefore, we model the system under test as a random variable, whose sample space consists of all possible behavior sets with respect to the specification over the known interface. The entropy of the system is measured as the entropy of the random variable.

The purpose of testing is to find out the set of actual behaviors of the system, which is a sample in that sample space. By running more test cases, we know more about the actual behaviors, which means that we gain more information about the system. In our criterion, the coverage of a test set is measured as the expected amount of entropy decrease (i.e., the expected amount of information gained) once the test set is run. Since our criterion is syntax-independent, we study the notion of information-optimal software testing where, within a given constraint, a test set is selected to gain the most information. We also develop efficient algorithms to select optimal test cases under different circumstances.

In Chapter 3, we propose a syntax-independent complexity metric for labeled graphs, which serve as specifications for software systems. The expected behaviors of a software system is then the set of labeled sequences collected from the graph that models the specification of the system. We introduce behavioral complexity as a novel complexity metric for labeled graphs. We define the behavioral complexity as

$$\lim_{n\to\infty} \frac{\log N(n)}{n},$$

where $N(n)$ is the number of sequences of length n in the graph. For every labeled graph, there is a Markov chain corresponding to it. We mathematically prove that the behavioral complexity of a graph is actually the upper limit of the entropy rate that the corresponding Markov chain can achieve. Notice that some classic work, e.g., [17], [19], also mention that there is some relationship between $\lim_{n\to\infty} \frac{\log N(n)}{n}$ and the limit-form entropy rate of a Markov chain. We are almost confident that the limit-form entropy rate does not work for all Markov chains. The work [11] mentions the upper-limit form, but only with a very intuitive explanation and does not dig further into it. Our work provides a solid mathematical foundation for it, which is a contribution to information theory. We also study how the behavioral complexity changes when graphs are composed together, and we show that behavioral complexity can increase when units are sequentially composed through loops, or composed in parallel through synchronization. Our results can be very helpful for software

developers: they can use the metric to predict the complexity of the semantics of a software system to be built from their designs. Our results also suggest that integration testing is necessary (after units are tested).

Chapter 4 concludes this dissertation and also proposes some future work.

CHAPTER 2

INFORMATION GAIN OF BLACK-BOX TESTING

2.1 Overview

Software testing is critical to ensure the quality of a software system. A textbook testing procedure is shown in Figure 2.1 [4]. A test case is the input data fed to the system under test. When a test case is selected, the system can then be executed with the test case. In consequence, the tester decides whether the result of the execution is as expected or not (e.g., comparing the result with the system's specification). After a set of test cases are run, an error (the system does not meet its specification) is possibly identified. However, when there is no error found, one usually cannot conclude that the system does meet its specification. This is because, for a nontrivial system, there are potentially infinitely many test cases and a tester can only run finitely many of them. On the other hand, a test case is selected before the test case is run and an error can only be identified after a test case is run. This raises a great challenge in software testing: How should test cases be selected?

Test cases are typically generated according to a pre-given test data adequacy criterion [28], which associates a degree of adequacy with the test set (i.e., the set of generated test cases) to indicate the coverage of the test set with respect to a system specification [81]. Formally, a test data adequacy criterion C is a function that maps a triple of a system under test, a specification and a test set to an (adequacy) degree in $[0, 1]$ [80]. In particular,

$$C(Sys, Spec, t) = r \tag{2.1}$$

indicates that, under the criterion C, the adequacy of the test set t on the system Sys with respect to the specification $Spec$ is of degree r. Naturally, as we mentioned earlier, a test data adequacy criterion, besides judging the test quality of an existing test set, is also a guideline of a test case

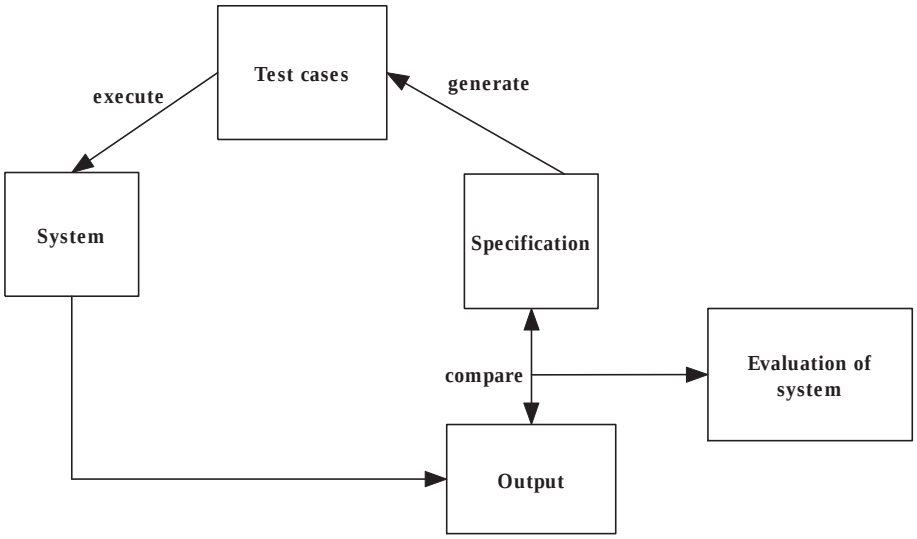


Figure 2.1: A textbook testing procedure.

generator [81]. In this chapter, we treat the system Sys in (2.1) as a black-box [8, 48, 4]. This is typical when the source code of the system is usually unavailable (e.g., commercial off-the-shelf (COTS) systems) or is too complex to analyze. In this case, even though the implementation details of the Sys are not clear, some attributes $Attr$ of it could be known; e.g., whether Sys is deterministic or nondeterministic, how many states the Sys has, what constitutes the input-output interface of the Sys, etc. Consequently, we can rewrite (2.1) into

$$C(Attr, Spec, t) = r. \tag{2.2}$$

Sometimes, we just omit the parameter $Attr$ when it is clearly given in the context. We shall emphasize that, since the system under test is a black-box and the test set is selected before the testing, the degree r in (2.2) is independent of the specific system Sys under test and also independent of the results of executing the test set t.

Note that in (2.2), when the criterion C and the system attributes $Attr$ are given, the test set t is essentially generated from the system specification $Spec$ for each given r. Examples of formalisms

used for the system specification are logical expressions (such as Boolean formulas and temporal logic formulas [54]) which describe the system's behaviors as a mathematical statement, C-like code (such as PROMELA [36]), tables (such as SCR [35]), and graphs (such as data flow graphs, control flow graphs and statecharts [31]) which describe how a system is intended to operate. One might have already noticed that a system can be described using different formalisms. Even within one formalism, one can specify the same system in different ways such that the resulting specifications share the same semantics. For instance, suppose that Boolean formulas $A \vee B$ and $\overline{(\overline{A} \wedge \overline{B})}$ are specifications for a Boolean circuit. These two specifications have the same semantics, though their syntactic appearances are different.

Common testing criteria give the adequacy degree of a test set based on the syntactic appearance of the system specification. Hence, those criteria are *syntax-based*. This causes problems. For instance, a slight change to a specification's syntactic appearance, even if the specification still keeps its semantics, might result in a dramatically different adequacy degree for the same test set. Additionally, if a syntax-based test data adequacy criterion returns the same adequacy degree for two test sets, then the two test sets are indistinguishable with respect to the criterion. For instance, suppose that two distinct test sets are selected from a graph modeling the system's control flow, both with 75%, say, branch coverage. The adequacy degree does not differentiate the two sets. Or, in other words, each branch is born equal. This is not intuitively true. A similar problem exists for Clause Coverage [5] for ground formulas (i.e., without quantifiers) in first order logic and testing criteria for temporal logic formulas [64]; the criteria could not tell the difference between two test sets that achieve the same coverage.

An ideal test set will always identify an error whenever the system under test has an error. It is widely agreed that one direct measurement of the effectiveness of a test set is its fault-detecting ability [25, 68, 81]. However, it is also understood that there are simply no such computable and ideal criteria to generate effective test sets [81]. In our opinion, fault-detecting in a black-box system is closely related to our knowledge about the system. This is particularly true considering

the fact that faults are often not easy to find. We summarize our opinion into the following two intuitive statements:

- the more we test, the more we know about the system under test;
- the more we know about the system under test, the more likely faults can be found.

From these statements, it is desirable to have a way to measure the amount of information of the black-box system Sys (about which we only know its attributes $Attr$) we gain with respect to the specification $Spec$ once the tests t (selected according to (2.2) using a given adequacy degree r) are run. Therefore, the measure concerns the system's semantics instead of its syntax. This naturally leads us to use Shannon entropy [59, 19] to measure the information gain and, because of its syntax-independence, we can now cross-compare the information gains of two test sets t_1 and t_2 of the same black-box system; even though t_1 and t_2 are generated from different criteria C, different specifications $Spec$, and/or different degrees r. As will be shown in the chapter, the information gain is calculated *before* tests are run. Therefore, the information gain also serves as a syntax-independent coverage measure once no faults are found after a test set is run (which is often the case). The rest of the chapter is outlined as follows.

We model the system under test as a reactive labeled transition system Sys whose observable behaviors are sequences of input-output pairs. We also assume that the system under test is deterministic. In some software testing literature [13], an observable behavior is called a *trace*. In this context, the objective of software testing is to test whether the observable behaviors of a black-box software system conform with a set of sequences of input-output pairs. The set is called a *trace-specification*, which specifies the observable behaviors that the system under test is intended to have. Let P be a set of sequences of input-output pairs, which is a trace-specification that the system under test is intended to conform with. This P is the whole or part of the system specification. Since in practice, we can only test finitely many test cases, here we assume that P is a finite (but could be huge) set. We use a tree T, called the *trace-specification tree*, to represent the

trace-specification, where each edge is labeled with an input-output pair. Clearly, it is possible that not every path (from the root of T to some node in T) is an observable behavior of the Sys; it is testing that tells which path is and which path is not. When, through testing, a path is indeed an observable behavior of the Sys, we mark each edge on the path as "connected". When, however, the path is not an observable behavior of the Sys, the test result shows the longest prefix of the path such that the prefix is an observable behavior of the Sys. In this case, we mark every edge (if any) in the prefix as "connected" and the remaining edges on the path as "disconnected". Hence, an edge is marked connected (resp. disconnected) when the path from the root of T to the edge itself (included) is (resp. is not) an observable behavior. Notice that observable behaviors of the Sys are prefix-closed and hence, when an edge is marked disconnected, all its offspring edges are also disconnected. Therefore, running tests is a procedure of marking the edges of T.

Before any tests are performed, we do not know whether, for each edge in T, it is connected or disconnected. (It is the tests that tell which is the case for each edge.) After testing a sufficient number of test cases, every edge in T is marked either connected or disconnected. At this moment, the *system tree* is the maximal subtree of T such that every edge of the system tree is marked connected. The system tree represents all the observable behaviors of the Sys with respect to the trace-specification P. Hence, before any tests are run, there is uncertainty in what the system tree would be. Adopting the idea of entropy in information theory, we model the system tree (which we do not clearly know before the testing) as a random variable X_T, whose sample space is the set of all subtrees that share the same root with the tree T. The entropy , written $H(T)$, of the system is measured as the entropy of the random variable X_T. In order to calculate $H(T)$, we need probabilities of edges being connected or not. Those probabilities could be pre-assigned. However, usually probabilities of edges are simply unknown. In that case, we can calculate the probabilities of edges such that $H(T)$ reaches the maximum (i.e., we do not have any additional information).

After a set t of test cases is executed, we know a little more about the system tree from the execution results. In consequence, the entropy of the system decreases from $H(T)$ to

$H(T|\text{after testing } t)$, the conditional entropy given the tests. That is, the information *gain* of running tests t is

$$G(t) = H(T) - H(T|\text{after testing } t).$$

This gain is calculated before the testing begins, and hence we can use the gain as a guideline to develop information-optimal testing strategies that achieve the most gain. In other words, we can pick the test set t that can achieve $\max G(t)$ subject to some constraint (e.g., the size of t is bounded by a certain number).

Notice that the aforementioned information gain $G(t)$ is syntax-independent; i.e., it is independent of the formalism that is used to describe the behaviors in P. This is because the entropy of a discrete random variable remains unchanged after a one-to-one function is applied [19]. More intuitively, the amount of Shannon information in an object is the same no matter whether one describes it in English or in French.

Shannon entropy has been used in various areas in Computer Science. For instance, in data mining, a neural network-like classification network with hidden layers is constructed in analyzing a software system's input-output relation through a training set [42]. The algorithm in constructing the network stops when further adding new layers would not make the entropy of the network significantly decrease. The resulting classification network is then used to help select "non-redundant" test cases. Note that our work is completely different from [42], as we study "information-optimal" testing processes, and our information-optimal test strategy is pre-computed *without* running any training set. Also, our work emphasizes the semantic dependency between test cases, whereas in many cases, researchers in testing treat test cases as a set (all members are born equal) [26]. Reference [49] studies optimal testing strategies for nondeterministic systems, while using a game theory approach.

The concept of Shannon entropy, in some historical literature [41, 40], is closely related to the second law of thermodynamics in physics. This law requires that the process that brings down

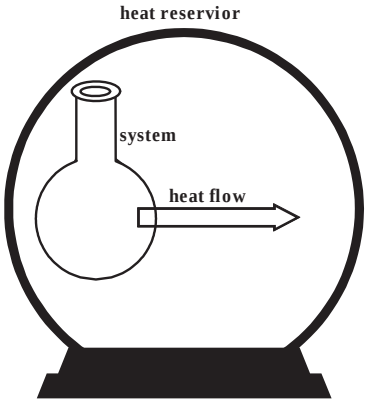


Figure 2.2: A system put in a heat reservoir.

the entropy of a thermal system causes a positive heat flow from the system to its environment (Figure 2.2 [40]). Landauer's principle [41] states a similar principle in the digital world. That is, in a computational device, the erasure of the Shannon information is accompanied by a generation of heat. In our approach, while running the test cases, the Shannon entropy of the system under test is decreasing. From Landauer's principle, there should be a heat flow from the system under test to the environment during the software testing process. Hence, the software system is *cooling down* during a software testing process. From this point of view, software testing is a cooling-down process, and software testing using our criterion is called cooling-down software testing and, intuitively, our optimal software testing approach cools down the system under test fastest.

The rest of the chapter is organized as follows. We formally give our definitions and terminologies on languages, trees, transition systems, and finite automata in Section 2.2, together with an array selection algorithm that will be used throughout this chapter. In Section 2.3, we formally define the entropy of a trace-specification represented as a tree and the information-optimality of a testing strategy, and develop algorithms to calculate information-optimal testing strategies of the system under test. In section 2.4, we study information-optimal testing strategies when the trace-specification is represented as a finite automaton. We summarize our study in Section 2.5.

The main part of this chapter is summarized in paper [71].

2.2 Preparations

In this section, we provide definitions and terminologies on languages, finite automata, trees, labeled transition systems, and an algorithm, called `MAX-SELECT`, on selecting certain numbers from arrays, which will be used later in the chapter.

2.2.1 Languages and Finite Automata

Let Σ be an alphabet. A language L is a set of words on the alphabet; i.e., $L \subseteq \Sigma^*$. For two words ω and ω', we use $\omega' \prec \omega$ to denote the fact that ω' is a (not necessarily proper) prefix of ω. L is *prefix-free* if, for any $\omega \in L$ and any $\omega' \prec \omega$, we have

$$\omega' \in L \text{ implies } \omega = \omega'.$$

L is *prefix-closed* if, for any ω and any $\omega' \prec \omega$, we have

$$\omega \in L \text{ implies } \omega' \in L.$$

Let $A = \langle S, s_{init}, F, \Sigma, R\rangle$ be a *deterministic finite automaton* (DFA), where S is the set of states with s_{init} being the initial state, F is the set of accepting states, $\Sigma = \{a_1, \cdots, a_k\}$ is the alphabet, and $R \subseteq S \times \Sigma \times S$ is the set of state transitions, satisfying that from a state and a symbol, one can at most reach one state (formally, $\forall s \in S, a \in \Sigma$, there is at most one s' with $(s, a, s') \in R$). A word $\omega = x_1 \cdots x_i$ in Σ^* is *accepted* by A if there is a sequence of states $s_0 s_1 \cdots s_i$, such that $s_0 = s_{init}$, $s_i \in F$, and $(s_{j-1}, x_j, s_j) \in R$ for $1 \leq j \leq i$. The language $L(A)$ that A accepts is the set of words accepted by A. Without loss of generality, we assume that A is cleaned up. That is, every state in A is reachable from the initial state, and can reach an accepting state.

When a language is finite, a tree can also be employed to represent it, as in below.

2.2.2 *Trees*

Let $L \subseteq \Sigma^*$ be a finite language and $\widehat{L}$ be its maximal prefix-free subset. Naturally, one can use a tree T to *represent* L. Every edge in T is labeled with a symbol in Σ, and any two distinct child edges (a child edge of a node N is the edge from N to a child node of N) of the same node cannot have the same label. Furthermore, T has $|\widehat{L}|$ leaves, and for each leaf, the sequence of the labels on the path from the root to the leaf is a word in $\widehat{L}$. We use the following terminologies for the tree T:

- an *edge* from a node N to its child node N' is denoted by $\langle N, N' \rangle$ where N is the *source* of $\langle N, N' \rangle$ and N' is the *a-child* of N when the edge is labeled with $a \in \Sigma$;
- the *parent edge* of an edge $\langle N, N' \rangle$ (N is not the root) is the edge from the parent node of N to N itself;
- a *sibling edge* of edge e is an edge that shares the same source with e;
- t is a *subtree* of T if t is a tree, and every node and every edge in t is a node and an edge, respectively, in T;
- the *root path of node* N is the path (of edges) from the root of T all the way down to N itself. When we collect the labels of the edges on the path, a word in Σ^* is obtained. Sometimes, we simply use the word to (uniquely) identify the path;
- the *root path of subtree* t is the root path of the root of the subtree t in T;
- an *empty tree* is one that contains exactly one node.We use $\emptyset$ to denote an empty tree when the node in the tree is clear from the context;
- we use $t \prec T$ to denote that t is a subtree of T and shares the same root with T. Intuitively, when $t \prec T$, then t can be obtained by dropping some leaves repeatedly from T;

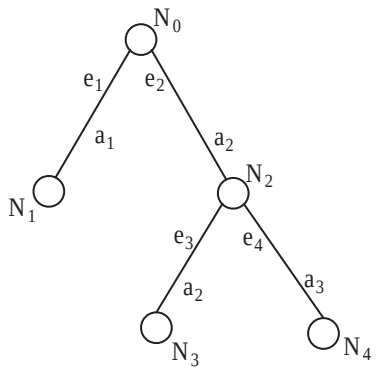


Figure 2.3: A tree representation of a language.

- a *subtree at node* N is a subtree of T with root N. We use the *maximal subtree at* N to denote the maximal such subtree at N;
- a *subtree under edge* $\langle N, N' \rangle$ is a subtree at node N';
- the *child-tree under edge* $\langle N, N' \rangle$ is the maximal subtree at node N';
- a *child-tree of node* N is the child-tree under edge $\langle N, N' \rangle$ for some N'. In this case, the child-tree is also called the a*-child-tree* of N, when N' is the a-child of N, for some $a \in \Sigma$;

Example 1. Figure 2.3 gives a tree T representing $L = \{a_1, a_2, a_2a_2, a_2a_3\}$ over the alphabet $\Sigma = \{a_1, a_2, a_3\}$. The subtree with the set of edges $\{e_1, e_2\}$ is a subtree $t \prec T$. The a_2-child-tree of the root of T is the one with the set of edges $\{e_3, e_4\}$; the a_1-child-tree with the set of node(s) $\{N_1\}$ is simply an empty tree. □

2.2.3 Transition Systems

Let Sys be the software system under test. Here we modify the formal model in [70]. A similar model can be found in a later paper [65]. Sys is a transition system that changes from one state to another while executing a transition labeled by a symbol. Formally,

$$Sys = \langle S, s_{init}, \nabla, R \rangle$$

is a *(labeled) transition system*, where S is the set of states with $s_{init} \in S$ being the initial state, ∇ is a finite set of symbols, and R will be explained in a moment. More specifically, $\nabla = \Pi \cup \Gamma$, where Π and Γ are two disjoint subsets, and Π is the set of input symbols, while Γ is the set of output symbols. In particular, we call $\Pi \times \Gamma$ the set of *observable (input-output) pairs* (sometimes, we also call such pairs as observable events), and (Π, Γ) is the *interface* of Sys. $R \subseteq S \times (\Pi \times \Gamma) \times S$ is the set of state transitions. An *executable sequence* of Sys, $\omega = x_1 \cdots x_i$ for some i, is a word on alphabet $(\Pi \times \Gamma)$, such that there is a sequence of states, say, $s_0 \cdots s_i$, with $(s_{j-1}, x_j, s_j) \in R$ for $1 \leq j \leq i$, and s_0 is the initial state s_{init}. We call such a word ω an *(observable input-output) behavior* of the system. Note that a general form of executable sequence (i.e., a sequence of input, output and internal symbols) can be modeled as ours, if one has at most a fixed number of inputs, followed by an output. For instance, we can encode the sequence, say, $input_1, \cdots, input_i, output$, as one pair $\langle (input_1, \cdots, input_i), output \rangle$ in an expanded input alphabet. Our system here is actually a reactive system [32], and the theoretical root of our model is Mealy machines [37] and I/O automata [43]. Note that the labeled transition system is universal since the number of states could be infinite. Also note that behaviors of Sys are prefix-closed. In our work, Sys is a *black-box* system (i.e., we assume that we only know its input-output interface).

In our context, software testing is to test whether the (black-box) software system Sys conforms with a trace-specification. No matter what formalism is used, the trace-specification, semantically, is commonly a set of sequences in $(\Pi \times \Gamma)^*$. This set is a language that specifies the (observable) behaviors that the system under test is intended to have. The trace-specification, denoted as $P_{original} \subseteq (\Pi \times \Gamma)^*$, could be an infinite language. For instance, when testing a TV remote control, theoretically, there are an infinite number of button combination sequences to test. However, in the real world, we can only test up to a given bound d on the length of the input sequence. The trace-specification, denoted as P, that in practice we plan to test against, is a "truncation" of the original trace-specification $P_{original}$. That is, $P = \{\omega : \exists \omega' \in P_{original},\ \ \omega \prec \omega' \text{ and } |\omega| \leq d\}$. In other words, P is the set of all prefixes, up to the given length d, of words in $P_{original}$. Hence, P

is simply a prefix-closed finite language.

In practice, this $P_{original}$ (as well as P) can be the whole or part of the *system specification* that describes the expected behaviors of the system under test. Such a specification can be drawn from, for instance, the design documents and requirement documents of the system Sys under test. How to derive such a $P_{original}$ and P is out of the scope of this work; the reader is referred to [13] for, e.g., model-based software testing. In our work, we simply assume that the $P_{original}$ and the P are given.

The transition system Sys defined earlier is in general *output-nondeterministic*. That is, it is possible that, for some observable behavior $\omega \in (\Pi \times \Gamma)^*$ and some input symbol $b \in \Pi$, there are more than one output symbol $c \in \Gamma$ such that $\omega(b, c)$ (i.e., the concatenation of the string ω and the symbol (b, c)) is also an observable behavior of the Sys. In other words, one can possibly observe more than one output from an input symbol. The source of output-nondeterminism comes from such things as a highly nondeterministic implementation (such as a concurrent program) of the Sys under test, or a partial specification of the interface. It is still an on-going research issue how to test an output-nondeterministic system Sys [49, 66, 53].

On the other hand, *output-deterministic* systems constitute a most important and most common class of software systems in practice; this is particularly true when such a system is used in a safety-critical application in which nondeterministic outputs are intended to be avoided. Formally, the transition system Sys is *output-deterministic* if for each $\omega \in (\Pi \times \Gamma)^*$ and input symbol $b \in \Pi$, there is at most one output symbol $c \in \Gamma$ such that $\omega(b, c)$ is an observable behavior of Sys. Implicitly, we have an option of "crash" after applying an input, which makes our approach more general than other models. For instance, suppose that, for a traffic light system with sensors, when a car is approaching at midnight (so no other cars are around), it is desirable that the light turns yellow or turns red. In here, we use a_{yellow} and a_{red} to represent the input-output pairs $(approaching, yellow)$ and $(approaching, red)$, respectively. When the system is not assumed output-deterministic, a test shows that a_{yellow} is actually observable does not necessarily

conclude that a_{red} is not actually observable. However, when we assume that the system is output-deterministic, this additional knowledge will conclude exactly one of the following three scenarios: when the car is approaching,

(i) the light turns yellow;

(ii) the light turns red;

(iii) neither (i) or (ii); e.g. the light system crashes.

Hence, the positive test result of, say, (i), immediately implies that outcomes like (ii) and (iii) are not possible.

In our work, we focus on output-deterministic systems. Actually, if a test execution engine can be built for output-nondeterministic systems, we can see that testing output-nondeterministic systems is a special case of testing output-deterministic systems, which will be discussed later in this chapter.

2.2.4 A Technical Algorithm MAX-SELECT

We now present an algorithm to solve the following selection problem, which will be used in several algorithms later in the chapter.

Let k be a number. Suppose that we are given q arrays of numbers, $Y_1, \cdots, Y_q$, each of which has $k+1$ entries. Each array Y_j is nondecreasing, i.e., $Y_j[\texttt{index}] \leq Y_j[\texttt{index}+1]$, $0 \leq \texttt{index} \leq k-1$, and the first entry $Y_j[0]$ is 0. Let $I \leq k$ be a number. We would like to select indices $\texttt{index}_1, \cdots, \texttt{index}_q$, satisfying $\sum_{1 \leq j \leq q} \texttt{index}_j = I$, of the arrays $Y_1, \cdots, Y_q$, respectively, such that the sum

$$\sum_{1 \leq j \leq q} Y_j[\texttt{index}_j] \tag{2.3}$$

is maximal. The instance of the problem is written as MAX-SELECT over $(\{Y_1, \cdots, Y_q\}, I)$. The result includes the desired indices and the sum in (2.3).

We use $SUM(\{Y_1, \cdots, Y_q\}, I)$ to denote the maximal sum reached in (2.3) for the problem instance. Suppose that the set $\mathcal{Y} = \{Y_1, \cdots, Y_q\}$ is partitioned into two nonempty subsets $\mathcal{Y}'$ and

$\mathcal{Y}''$. One can show

$$SUM(\mathcal{Y}, I) = \max_{0 \le i \le I}(SUM(\mathcal{Y}', i) + SUM(\mathcal{Y}'', I - i)). \tag{2.4}$$

The algorithm `MAX-SELECT` that solves the problem is as follows. We first build, in linear time, a balanced binary tree t with q leaves, $\text{leaf}_1, \cdots, \text{leaf}_q$, and with roughly $2q$ nodes. Each leaf_j corresponds to the array Y_j, $1 \le j \le q$. In the sequel, we simply use a set of arrays to denote a set of leaves. Let N be a node in t. We associate it with a table of $k+1$ entries. The i^{th} entry, $0 \le i \le k$, contains a number $\texttt{SUM}_N[i]$ and a set $\texttt{IND}_N[i]$ of pairs: each pair is a number $1 \le j \le q$ and an index to the array Y_j. Initially, $\texttt{SUM}_N[i] = 0$ and $\texttt{IND}_N[i] = \emptyset$, for all i. When N is a leaf Y_j, we further initialize $\texttt{SUM}_N[i] = Y_j[i]$ and $\texttt{IND}_N[i] = \{(j, i)\}$, for all $0 \le i \le k$. We now explain the meaning of the table. Let N be a nonleaf node in t. We use $\mathcal{Y}_N$ to denote the set of leaf nodes of which N is an ancestor. After the algorithm is run, $\texttt{SUM}_N[i]$ is the value $SUM(\mathcal{Y}_N, i)$, and $\texttt{IND}_N[i]$ records the desired indices for the MAX-SELECT instance over $(\mathcal{Y}_N, i)$. That is,

$$\texttt{SUM}_N[i] = \sum_{(j, \texttt{index}_j) \in \texttt{IND}_N[i]} Y_j[\texttt{index}_j].$$

Let N_1 and N_2 be the two child nodes of N (every nonleaf node in a balanced binary tree has exactly two children). From (2.4), $\texttt{SUM}_N[i] = \max_{0 \le l \le i}(\texttt{SUM}_{N_1}[l] + \texttt{SUM}_{N_2}[i-l])$, which provides a way to calculate $\texttt{SUM}_N[\cdot]$ and $\texttt{IND}_N[\cdot]$ presented in the following algorithm:

```
MAX-SELECT({Y_1, ..., Y_q}, I):
//To find solutions to MAX-SELECT problem instances over
//({Y_1, ..., Y_q}, i), for all i ≤ I, and return the solution for i = I.
//Suppose that the balanced binary tree t is already built with
//the arrays Y_1, ... Y_q being the leaves.  Each node N is
```

```
//associated with SUM_N[·] and IND_N[·] that are already
//initialized as described earlier in the subsection.
1. For level := 1 to (height of t)
   //a node of level (height of t) is the root
2.     For each nonleaf node N of level level
       //suppose that N_1 and N_2 are the two child nodes of N
3.         For i := 0 to I
4.             SUM_N[i] := max_{0≤l≤I} (SUM_{N_1}[l] + SUM_{N_2}[i − l]);
5.             Assume that 1 ≤ l* ≤ I reaches maximum in line 4;
6.             IND_N[i] := IND_N[l*] ∪ IND_N[i − l*];
7. Return SUM_root[I] and index_1, ··· , index_q.
   //root is the root of T
   //SUM_root[I] is the value SUM({Y_1, ··· , Y_q}, I)
   //IND_root[I] is a set of pairs (j, index_j), for each 1 ≤ j ≤ q
```

Clearly, the algorithm runs in worst-case time $O(k^2q)$, recalling that $k+1$ is the size of each of the arrays $Y_1, \cdots, Y_q$.

2.3 Information Gain of Tests and Information-Optimal Testing Strategies

In this section, we assume that the system Sys under test is a black-box transition system with its interface known. Recall that Π and Γ are the input symbols and output symbols of Sys, respectively. Throughout this section, the system Sys under test is assumed to be output-deterministic. Before we develop algorithms for information-optimal testing of such systems, we need definitions on the test oracle.

As explained before, we use $P_{original} \subseteq (\Pi \times \Gamma)^*$ to denote a set of intended input-output behaviors of the Sys under test. The trace-specification $P \subseteq P_{original}$ is a finite set that we will

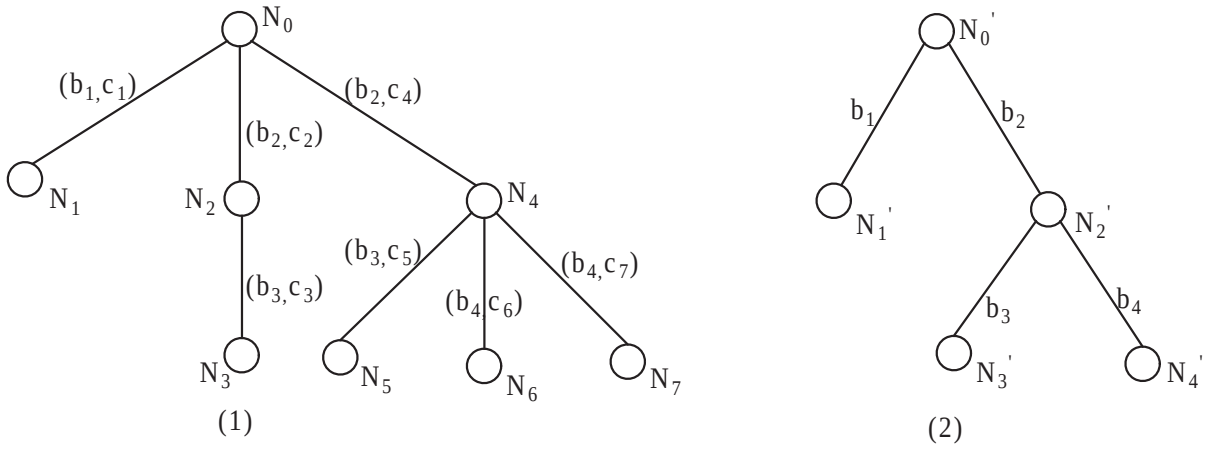


Figure 2.4: A trace-specification tree and the corresponding input testing tree.

actually test against, which is the set of all prefixes (up to length of a given number d) of sequences in $P_{original}$. In the sequel and without loss of generality, we simply assume that P is a finite set $\subseteq (\Pi \times \Gamma)^*$. We use a tree T, i.e., the trace-specification tree, to represent P, where every edge on T is labeled with a pair of input and output symbols. When we drop output symbols from every sequence in P, we obtain a set $P_\Pi \subseteq \Pi^*$ of input symbol sequences; i.e., $P_\Pi = \{b_1 \cdots b_n : (b_1, c_1) \cdots (b_n, c_n) \in P, \text{ for some } n \text{ and each } c_i \in \Gamma\}$. The tree T_Π that represents P_Π is called the *input testing tree*.

Example 2. In Figure 2.4 (1), we show the trace-specification tree T representing the trace-specification $P = \{(b_1, c_1), (b_2, c_2), (b_2, c_4), (b_2, c_2)(b_3, c_3), (b_2, c_4)(b_3, c_5), (b_2, c_4)(b_4, c_6), (b_2, c_4)(b_4, c_7)\}$. The trace-specification P specifies that the system under test (which is output-deterministic) is expected to have the following behaviors:

- Initially, when button b_1 is pressed, color c_1 is shown.
- Initially, when button b_2 is pressed, either color c_2 or color c_4 is shown.
  - when color c_2 is shown, further pressing button b_3 will show color c_3.
  - when color c_4 is shown, we can further press button b_3 or b_4.

- when pressing button b_3, color c_5 is shown.
- when pressing button b_4, either color c_6 or c_7 is shown.

From P, we can obtain the set $P_\Pi = \{b_1, b_2, b_2b_3, b_2b_4\}$, and the corresponding input testing tree is shown in Figure 2.4 (2). □

We further assume that Sys is *sequentially testable*; that is, there is a test execution engine *oracle* such that, if we send an input sequence $\pi = b_1 \cdots b_n$ on Π to the oracle, it will "read" symbols in π, one by one, from b_1 up to b_n while running the black-box Sys (this is a common assumption for black-box testing [52]). As a result, the oracle returns an output symbol right after each input symbol read. The run stops when the Sys crashes on some input, or, the last symbol in the sequence is read and the corresponding output symbol is returned. Formally, a *test case* π is a word in Π^*. The oracle is equipped with a deterministic program $Test$ that runs on Sys and π and always halts with an output $Test(Sys, \pi)$. The run is said to be testing π. The oracle, as usual, honestly relays the outputs from the Sys. That is, for all n and $b_1 \cdots b_n \in \Pi^*$,

- $Test(Sys, b_1 \cdots b_n) = c_1 \cdots c_n$ if $(b_1, c_1) \cdots (b_n, c_n)$ is an observable behavior of Sys;
- $Test(Sys, b_1 \cdots b_n) = c_1 \cdots c_i \perp$, for some $i < n$, if $(b_1, c_1) \cdots (b_i, c_i)$ is an observable behavior of Sys, and $(b_1, c_1) \cdots (b_i, c_i)(b_{i+1}, c)$ is not an observable behavior of Sys, for any $c \in \Gamma$. The symbol $\perp \notin \Gamma$ indicates "Sys crashes".

Notice that, for empty string ϵ, $Test(Sys, \epsilon) = \epsilon$ by definition. Since Sys is output-deterministic, $Test(Sys, \pi)$ is unique.

We say that the Sys *conforms with* P if, for every input symbol sequence $b_1 \cdots b_n \in P_\Pi$, for some n, we have $Test(Sys, b_1 \cdots b_n) = c_1 \cdots c_n \in \Gamma^*$ and $(b_1, c_1) \cdots (b_n, c_n) \in P$ (in some literature, this is called $\leq_{iot}$*-correct* with respect to P [13]). That is, in terms of the trace-specification tree, for each path from the root in the input testing tree T_Π, suppose that the sequence of labels on the path is $b_1 \cdots b_n$, one can find a path in the trace-specification tree T such that the label sequence

on the latter path is $(b_1, c_1) \cdots (b_n, c_n)$ for some $c_1, \cdots, c_n$ satisfying that $(b_1, c_1) \cdots (b_n, c_n)$ is an observable behavior of Sys.

Running a test case can be considered as a process of marking on the trace-specification tree T as follows. An edge in T is marked with "connected" if the sequence of the input-output labels on the path from the root to the edge (included) is an observable behavior of Sys; it is marked with "disconnected" if otherwise. Let the edge e be labeled with $(b, c) \in \Pi \times \Gamma$. One can observe that:

(1) once e is marked disconnected, every edge in the child-tree under edge e must be also marked disconnected (this is because the set of observable behaviors of Sys is prefix-closed);

(2) once e is marked connected, then every sibling edge e that is labeled with (b, c') for some $c' \in \Gamma$ must be marked disconnected (this is because Sys is output-deterministic), and hence edges in the child-trees under such sibling edges must also be marked disconnected.

Therefore, once e is marked, we implicitly assume that markings are already propagated further to its siblings (that share the same input symbol with e) and the offspring edges of itself and the siblings, using the above observation. Now, suppose that we run $Test(Sys, b_1 \cdots b_n)$ and obtain $c_1 \cdots c_n \in \Gamma^*$ as the result. Clearly by definition, the edges on the path (from the root of T) labeled with $(b_1, c_1) \cdots (b_n, c_n)$ are all marked connected. When $Test(Sys, b_1 \cdots b_n)$ results in $c_1 \cdots c_i \perp$ for some $i < n$, the edges on the path (from the root of T) labeled with $(b_1, c_1) \cdots (b_i, c_i)$ are all marked connected but all the edges in the child-tree under the last edge on the path are marked disconnected.

Initially, none of the edges in T is marked. As we test more and more test cases in P_Π, more and more edges in T are marked. Clearly, when all test cases in P_Π are tested, every edge in T is marked. Notice that, in this case, the *system tree* is the maximal subtree t of T such that $t \prec T$ and every edge in t is marked connected. The system tree exactly characterizes all the observable

behaviors of Sys for all input sequences in P_Π. Consider a subtree t of T. We say that t is *output-deterministic* if, for any edge e with some label $(b, c) \in \Pi \times \Gamma$, e does not have a sibling edge with label (b, c') for any $c' \in \Gamma$. Observe that the system tree must be an output-deterministic subtree $t \prec T$.

2.3.1 Entropy of a Trace-specification Tree

Before any testing is performed, we do not know exactly what the system tree is except that it is an output-deterministic subtree $t \prec T$. We now treat the system tree as a random variable X_T and first study the algorithm in calculating its entropy.

Consider a path (labeled by) $(b_1, c_1) \cdots (b_n, c_n)$ in the trace-specification tree T, and the set E of the child edges of the last edge (b_n, c_n) on the path. Suppose that $e_1, \cdots, e_l$ for some l, are all the edges in E that are labeled by $(b, c^1), \cdots, (b, c^l)$, respectively, for some $b \in \Pi$ and some $c^1, \cdots c^l \in \Gamma$. We use E_b to denote $\{e_1, \cdots, e_l\}$. Let $p(e_i)$ be the probability that edge e_i is marked connected when all its ancestor edges are marked connected, and all other edges in E_b are marked disconnected. That is, $p(e_i)$ is the probability that $(b_1, c_1) \cdots (b_n, c_n)(b, c^i)$ is the only observable input-output behavior of the Sys (with $(b_1, c_1) \cdots (b_n, c_n)$ as prefix and with length $n + 1$) given that $(b_1, c_1) \cdots (b_n, c_n)$ is an observable behavior of the Sys. Probabilities $p(e_i)$ could be pre-assigned (e.g., obtained from usage study [6]). However, usually probabilities of edges are simply unknown. In that case, we can calculate the probabilities of edges such that $H(T)$ reaches the maximum (i.e., we do not have any additional information), which will be discussed later. Since Sys is output-deterministic, the $p(\cdot)$ must obviously satisfy the following additional constraint, for each b,

$$\sum_{e_i \in E_b} p(e_i) \leq 1.$$

We use $p(t, T)$ to denote the probability of t being the system tree (that shares the same root

with T; i.e., $t \prec T$). Clearly, $p(t, T) = 0$ when t is not output-deterministic. Observe that

$$\sum_{\substack{t \prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} p(t, T) = 1$$

and hence $p(\cdot, T)$ is a distribution. Now, the entropy of X_T, simply written $H(T)$, is

$$H(T) = - \sum_{\substack{t \prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} p(t, T) \log p(t, T).$$

Similarly, we can also define $H(t)$ for a subtree t (while keeping the probability assignments of the edges) of T as:

$$H(t) = - \sum_{\substack{t' \prec t \text{ and} \\ t' \text{ is output}-\text{deterministic}}} p(t', t) \log p(t', t). \tag{2.5}$$

Note that throughout this chapter, the base of the logarithm is 2. In other words, we measure entropy in *bits*. By definition, the entropy of a empty tree $\emptyset$ is $H(\emptyset) = 0$, since for the empty tree $\emptyset$, the system tree has only one choice that is the empty tree itself. Before we show how to calculate $H(T)$, some more definitions are needed. Let $b \in \Pi$ and E be the set of child edges of T's root. The $(b, \cdot)$-component tree T_b of T exactly consists of each edge e in E that is labeled with (b, c) for some c and the child-tree under the edge e. We use $C < T$ to denote that C is a $(b, \cdot)$-component tree of T for some $b \in \Pi$.

Example 3. For the trace-specification tree T shown in Figure 2.4, the edge (b_1, c_1) forms the $(b_1, \cdot)$-component tree of T; edges (b_2, c_2) and (b_2, c_4), together with the child-trees under them, form the $(b_2.\cdot)$-component tree of T. □

One can show the following proposition,

Proposition 1.

$$H(T) = \sum_{C < T} H(C).$$

That is, the entropy of T is the summation of the entropy of each $(b,\cdot)$-component trees, $b \in \Pi$.

Proof. we use $\{C_1, \cdots, C_z\}$, for some z, to denote all $C < T$. For a output-deterministic subtree $t \prec T$, we use t_{C_i} to denote the subtree obtained from intersecting C_i with t. Since $p(t,T) = \prod_{1\le i\le z} p(t_{C_i}, C_i)$, we have,

$$\begin{aligned}
H(T) &= -\sum_{\substack{t\prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} p(t,T)\log p(t,T)) \\
&= -\sum_{\substack{t\prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} (\prod_{1\le i\le z} p(t_{C_i}, C_i))\log(\prod_{1\le i\le z} p(t_{C_i}, C_i)) \\
&= -\sum_{\substack{t\prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} \sum_{1\le i\le z} (\prod_{1\le j\le z} p(t_{C_j}, C_j))\log p(t_{C_i}, C_i) \\
&= -\sum_{\substack{t\prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} \sum_{1\le i\le z} p(t_{C_i}, C_i)\log p(t_{C_i}, C_i)(\prod_{j\neq i, 1\le j\le z} p(t_{C_j}, C_j)) \\
&= -\sum_{1\le i\le z} \sum_{\substack{t\prec T \text{ and} \\ t \text{ is output}-\text{deterministic}}} p(t_{C_i}, C_i)\log p(t_{C_i}, C_i)(\prod_{j\neq i, 1\le j\le z} p(t_{C_j}, C_j)) \\
&= -\sum_{1\le i\le z} \sum_{\substack{t\prec C_i \text{ and} \\ t \text{ is output}-\text{deterministic}}} p(t, C_i)\log p(t, C_i) \\
&= \sum_{C<T} H(C).
\end{aligned}$$

□

For a $(b,\cdot)$-component tree $C = T_b$, suppose that it consists of child trees $T_1, \cdots, T_l$ for some l, and edges $e_1, \cdots e_l$ with labels $(b, c^1), \cdots, (b, c^l)$ from the root of T to the root of $T_1, \cdots, T_l$, respectively (see Figure 2.5). We use $p_1, \cdots, p_l$ to denote the probability assignments $p(e_1), \cdots, p(e_l)$, respectively. One can show,

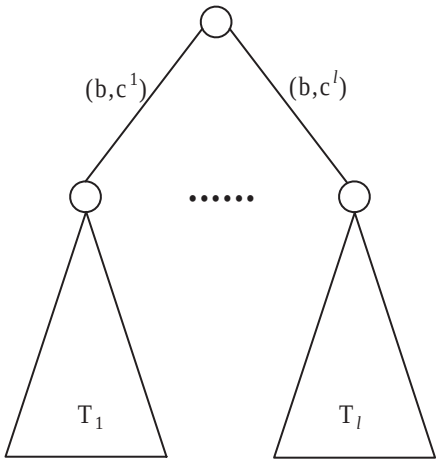


Figure 2.5: A $(b, \cdot)$-component tree T_b of T.

Proposition 2. *For the $(b, \cdot)$-component tree $C = T_b$ shown in Figure 2.5,*

$$H(C) = H(T_b) = \sum_{1 \le i \le l} p_i H(T_i) - \sum_{1 \le i \le l} p_i \log p_i - (1 - \sum_{1 \le i \le l} p_i) \log(1 - \sum_{1 \le i \le l} p_i).$$

Proof. For the notational convenience, in the following proof, subtrees t and t' are already assumed

to be output-deterministic. By definition,

$$
\begin{aligned}
H(T_b) &= -\sum_{t \prec T_b} p(t, T_b) \log p(t, T_b) \\
&= -\Bigg(\sum_{t \prec T_b, t=\emptyset} p(t, T_b) \log p(t, T_b) + \sum_{t \prec T_b, t \neq \emptyset} p(t, T_b) \log p(t, T_b) \Bigg) \\
&= -\Bigg((1 - \sum_{1 \leq i \leq l} p_i) \log (1 - \sum_{1 \leq i \leq l} p_i) + \sum_{1 \leq i \leq l} \sum_{t' \prec T_i} p_i \cdot p(t', T_i) \log (p_i \cdot p(t', T_i)) \Bigg) \\
&= -\Bigg((1 - \sum_{1 \leq i \leq l} p_i) \log (1 - \sum_{1 \leq i \leq l} p_i) \\
&\quad + \sum_{1 \leq i \leq l} (p_i \log p_i \sum_{t' \prec T_i} p(t', T_i) + p_i \sum_{t' \prec T_i} p(t', T_i) \log p(t', T_i)) \Bigg) \\
&= -\Bigg((1 - \sum_{1 \leq i \leq l} p_i) \log (1 - \sum_{1 \leq i \leq l} p_i) \\
&\quad + \sum_{1 \leq i \leq l} (p_i \log p_i + p_i \sum_{t' \prec T_i} p(t', T_i) \log (p(t', T_i))) \Bigg) \\
&= \sum_{1 \leq i \leq l} p_i H(T_i) - \sum_{1 \leq i \leq l} p_i \log p_i - (1 - \sum_{1 \leq i \leq l} p_i) \log (1 - \sum_{1 \leq i \leq l} p_i).
\end{aligned}
$$

□

The entropy $H(T_b)$, according to Proposition 2, is the "average" entropy of $H(T_i)$'s, together with the uncertainty introduced by output-determinism: among edges $(b, c^1), \cdots, (b, c^l)$, at most one of them is contained in the system tree since at most one of $c^1, \cdots, c^l$ can be the output from the input b. This additional entropy,

$$
- \sum_{1 \leq i \leq l} p_i \log p_i - (1 - \sum_{1 \leq i \leq l} p_i) \log (1 - \sum_{1 \leq i \leq l} p_i),
$$

is exactly the entropy of a random variable with $(l+1)$ outcomes, and each outcome with probability assignments $p_1, \cdots, p_l, 1 - \sum_{1 \leq i \leq l} p_i$, respectively. From the two propositions above, an algorithm

that calculates the entropy of an output-deterministic trace-specification tree T is immediately given as follows, with time complexity $O(n)$, where n is the size of T (i.e., the number of edges in T).

ALG-entropy-tree(T):
//To calculate the entropy $H(T)$ of the trace-specification
//tree T with given probability assignments $p(\cdot)$ for
//an output-deterministic system.
//The return value of this algorithm is the entropy $H(T)$.
1. If $T = \emptyset$
2. $H(T) := 0$;
3. Return $H(T)$;
4. If T only has one $(b, \cdot)$-component tree
//T now is in the form of a $(b, \cdot)$-component tree for some b,
//shown in Figure 2.5, consists of l edges $(b, c^1), \cdots, (b, c^l)$
//along with child-trees T_i under edges (b, c^i) $(1 \leq i \leq l)$;
//each edge (b, c^i) with probability assignment p_i
5. $H(T_i) :=$ ALG-entropy-tree(T_i);
6. $H(T) := \sum_{1 \leq i \leq l} p_i H(T_i) - \sum_{1 \leq i \leq l} p_i \log p_i - (1 - \sum_{1 \leq i \leq l} p_i) \log(1 - \sum_{1 \leq i \leq l} p_i)$;
7. Return $H(T)$;
8. $H(T) := \sum_{C < T}$ ALG-entropy-tree(C);
9. Return $H(T)$.

Before we proceed further, we need more notations. Given a trace-specification tree T and its corresponding input testing tree T_Π, for a path (labeled by) $\omega = (b_1, c_1) \cdots (b_i, c_i)$ in T, by definition, we have a path $\omega_\Pi = b_1 \cdots b_i$ in T_Π; in this case, we write $\omega \sim \omega_\Pi$. Similarly, for a node

N in T, suppose that the path from the root to N is ω. Correspondingly, in T_Π, we have a path ω_Π from the root to some node N_Π, such that $\omega \sim \omega_\Pi$; in this case, we say $N \sim N_\Pi$. For a subtree t of T, we can find a minimal subtree t_Π of T_Π, such that for each node N in t, there is some node N_Π in t_Π such that $N \sim N_\Pi$; in this case, we say $t \sim t_\Pi$. For a subtree t_Π of T_Π, let t be a maximal subtree t of T such that $t \sim t_\Pi$; in this case, we say $t \simeq t_\Pi$ (note that there could be more than one such t, depending on the location of the root of t in T). Intuitively, a subtree t that satisfies $t \simeq t_\Pi$ is a maximal subtree of T, such that once every edge in t_Π is tested, every edge in t will be marked.

Example 4. For the trace-specification tree T and its corresponding input testing tree T_Π shown in Figure 2.4, considering the subtree t of T that consists of nodes N_4, N_5, N_6 and N_7, and the subtree t' of T_Π that consists of nodes N'_2, N'_3 and N'_4, we have $t \simeq t'$. For the subtree t'' of T that consists of nodes N_2 and N_3, we also have $t'' \simeq t'$. □

Now, consider an edge e in T_Π. In the following, we will define the entropy "gain" $G(e)$ after the edge e is tested. To do this, consider all those subtrees t of T that satisfy $t \simeq e$ (here e is treated as the subtree of T_Π that only has the edge e). Recall that the t's are those t's that are marked after e is tested.

Example 5. Consider the trace-specification tree T and its corresponding input testing tree T_Π shown in Figure 2.4. For the edge $e = \langle N'_2, N'_3 \rangle$ in T_Π, we have $t_1 \simeq e$ and $t_2 \simeq e$, where t_1 consists of nodes N_2 and N_3, and t_2 consists of nodes N_4 and N_5 in T. Similarly, for the edge $e' = \langle N'_2, N'_4 \rangle$, the only subtree t' of T such that $t' \simeq e'$ is the one consisting of nodes N_4, N_6 and N_7. □

For a subtree t of T with $t \simeq e$, we use $\omega_t = e^1 \cdots e^j$, for some j, to denote the root path of t in the trace-specification tree T. Also, we define $p(\omega_t) = p(e^1) \cdots p(e^j)$ when ω_t is not empty, and when ω_t is an empty path, $p(\omega_t) = 1$. Since $t \simeq e$, it is clear that t must be in the form of a subtree only containing a number edges, say, $e_1, \cdots, e_l$, for some l, that share the same source (the root of t), shown in Figure 2.6. By definition in (2.5), we have $H(t) = -(\sum_{1 \le i \le l} p_i \log p_i + (1 -$

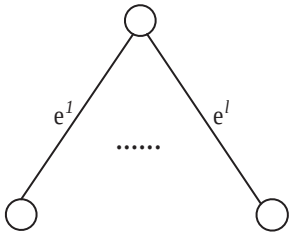


Figure 2.6: A subtree t of T such that $t \simeq e$ for some edge e in T_Π.

$\sum_{1\le i\le l} p_i) \log(1 - \sum_{1\le i\le l} p_i))$. Now, we define the entropy gain of the edge e in T_Π as

$$G(e) = \sum_{t\simeq e} p(\omega_t) H(t). \tag{2.6}$$

One can show, using Propositions 1 and 2, that the entropy $H(T)$ of the trace-specification tree T can be expressed as the summation of the entropy gains of the edges in the input testing tree T_Π:

Proposition 3.

$$H(T) = \sum_{e \text{ in } T_\Pi} G(e).$$

That is, once all the edges in the input testing tree are tested, the total gain is exactly the uncertainty $H(T)$ of the trace-specification tree; i.e., no uncertainty is left.

Example 6. Consider the trace-specification tree T and its corresponding input testing tree T_Π shown in Figure 2.4. Suppose that the probability assignments of edges in T are as follows: $p(\langle N_0, N_1\rangle) = p(\langle N_2, N_3\rangle) = p(\langle N_4, N_5\rangle) = 1/2$, $p(\langle N_0, N_2\rangle) = 2/9$, $p(\langle N_0, N_4\rangle) = 2/3$, and $p(\langle N_4, N_6\rangle) = p(\langle N_4, N_7\rangle) = 1/3$. We have $H(T) = \log 18$ bits by applying the algorithm `ALG-entropy-tree(`T`)`. We can also calculate $H(T)$ using Proposition 3. For the edge $e = \langle N_2', N_3'\rangle$ in T_Π, we have $t_1 \simeq e$ and $t_2 \simeq e$, where t_1 consists of nodes N_2 and N_3, and t_2 consists of nodes N_4 and N_5 in T. By definition, $H(t_1) = H(t_2) = 1$. By (2.6), we have $G(e) = p(\langle N_0, N_2\rangle)H(t_1) + p(\langle N_0, N_4\rangle)H(t_2) = 2/9 + 2/3 = 8/9$. Similarly, for all other edges in T_Π, we have $G(\langle N_0', N_1'\rangle) = 1$, $G(\langle N_0', N_2'\rangle) = 4/3 \log 3 - 8/9$, and $G(\langle N_2', N_4'\rangle) = 2/3 \log 3$.

From Proposition 3, we have $H(T) = 1 + 2\log 3 = \log 18$, which coincides with the results obtained from `ALG-entropy-tree(`T`)`. □

2.3.2 Testing Strategies and Gain

By running test cases in P_Π, one can check whether a software system Sys conforms with the given trace-specification P, which is a set of intended observable behaviors that Sys is supposed to have. Let T be the trace-specification tree that represents P. Recall that, running test cases using the oracle resembles the process of marking some edges in T. Since the gain of information on the initially unknown system tree of T after running a number of test cases in P_Π corresponds to the reduction of entropy of the testing tree by marking edges in T, a strategy that specifies the ordering of marking edges also represents the process of gaining the information while running tests.

A testing strategy $\mathcal{C}$ is a sequence of edges in the input testing tree T_Π, in the form of

$$e_{\mathcal{C}(1)}, \cdots, e_{\mathcal{C}(g)},$$

for some $g \leq m$ (m is the number of edges in T_Π), satisfying the constraint that parent edges should precede their child edges in $\mathcal{C}$. That is, for any $1 \leq i \leq g$, if e is the parent edge of $e_{\mathcal{C}(i)}$, then there is a j with $1 \leq j < i \leq g$ such that $e = e_{\mathcal{C}(j)}$. Naturally, the strategy gives the ordering that test cases (i.e., input symbol sequences) should be run.

Let α be a prefix of $\mathcal{C}$, which corresponds to a subtree t_α (i.e., t_α exactly contains all the edges in α) of the input testing tree T_Π with $t_\alpha \prec T_\Pi$. Let $T_\alpha \prec T$ be the maximal subtree of the trace-specification tree T such that, for each path $(b_1, c_1) \cdots (b_i, c_i)$, for some i, in T_α, we have that $b_1 \cdots b_i$ is a path in the aforementioned subtree t_α of T_Π that represents α; i.e., $T_\alpha \simeq t_\alpha$. That is, after α is tested (i.e., every test case represented in t_α is tested), the uncertainty in T_α is gone completely, since, in this case, every edge in T_α is marked already using the test results. We use $G(\alpha)$ to denote the expected entropy reduction of T after α is tested, which is also the information gained on the system tree of T. The gain $G(\alpha)$ is defined as $H(T_\alpha)$. From Proposition 3 (taking

T_α as T and t_α as T_Π), we have

$$G(\alpha) = \sum_{e \text{ in } t_\alpha} G(e), \tag{2.7}$$

and naturally, $G(\alpha) = H(T)$ when α is the entire strategy $\mathcal{C}$ when $g = m$ (that is, $\mathcal{C}$ covers all the edges in T_Π). When all the edges in T_Π are tested, the entropy gain is $H(T)$; i.e., no uncertainty is left. Let $Pre_{\mathcal{C}}(k)$ be the prefix of $\mathcal{C}$ of length k. One can easily show that $G(Pre_{\mathcal{C}}(k_1)) \leq G(Pre_{\mathcal{C}}(k_2))$, for any $k_1 \leq k_2$. That is, as we test more edges in the input testing tree, more information is gained. Sometimes, we abuse the notation G and use

$$G(t_\alpha) = G(\alpha) \tag{2.8}$$

to denote the gain of the subtree t_α of T_Π. In fact, (2.8) already gives the information gain of a test set $tests$ as follows. Recall that the set can be represented as a subtree t of T_Π; the gain of $tests$ is defined as

$$G(tests) = G(t). \tag{2.9}$$

In later sections, we will show how to select test cases that achieve the maximal amount of information gain under certain constraints.

2.3.3 Information-Optimal Testing Strategies with Pre-given Probability Assignments

In this subsection, unless stated otherwise, we assume that a testing strategy $\mathcal{C}$ covers all the edges in T_Π. An *information-optimal* testing strategy, as explained before, tries to make a maximal reduction of entropy in the testing tree. Formally,

Definition 1. Let T be a trace-specification tree and T_Π be the corresponding input testing tree with m edges. $\mathcal{C}^*$ is a `global information-optimal testing strategy` if, for any testing strategy $\mathcal{C}$, $G(Pre_{\mathcal{C}^*}(k)) \geq G(Pre_{\mathcal{C}}(k))$, for each $0 \leq k \leq m$.

That is, for any given length of prefix, a global information-optimal testing strategy reduces more uncertainty than any other strategy. Note that a global information-optimal testing strategy may

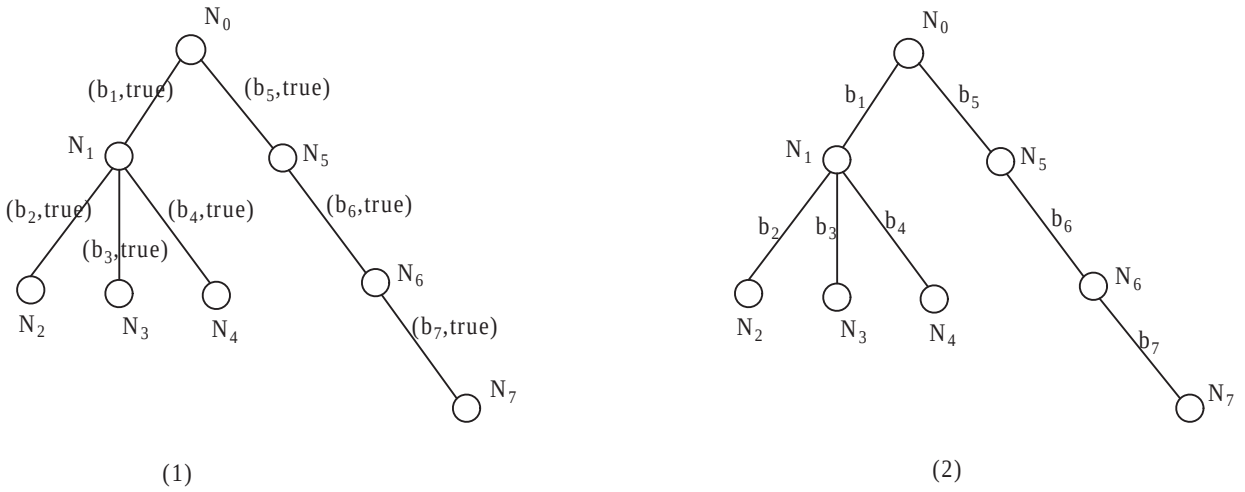


Figure 2.7: A trace-specification tree and its corresponding input testing tree.

not necessarily exist.

Example 7. Consider the trace-specification tree T in Figure 2.7 (1) with $\Pi = \{b_1, \cdots, b_7\}$ and $\Gamma = \{true\}$. Its input testing tree is shown in Figure 2.7 (2). Let e_i be the edge in T labeled with $(b_i, true)$, and e'_i be the edge in T_Π labeled with b_i, $1 \leq i \leq 7$. The probability assignments of edges are as follows: $p(e_1) = 8/9$, $p(e_2) = p(e_3) = p(e_4) = p(e_7) = 1/2$, $p(e_5) = 3/4$ and $p(e_6) = 2/3$. The input testing tree T_Π does not have a global information-optimal testing strategy, since one can easily check that, assuming that $\mathcal{C}^*$ were a global information-optimal testing strategy, $Pre_{\mathcal{C}^*}(1) = e'_5$, while $Pre_{\mathcal{C}^*}(3) = e'_1, e'_2, e'_3$, which leads to a contradiction. □

Now, we define a weaker form of information-optimality, when we are only allowed to test for k edges, and we try to maximize the entropy reduction after testing the k test cases.

Definition 2. Let T be a trace-specification tree, T_Π be the corresponding input testing tree with m edges and k be a number $\leq m$. $\mathcal{C}^*$ is a k-`information-optimal testing strategy` if, for any testing strategy $\mathcal{C}$, $G(Pre_{\mathcal{C}^*}(k)) \geq G(Pre_{\mathcal{C}}(k))$.

Note that the testing ordering within the first k edges would not matter (of course, parents should be tested before their children) when the k edges are given, since the gain is always the entropy of the subtree consisting of those k edges. Hence, finding a k-information-optimal testing

strategy is equivalent to picking the first k edges to test in T_Π. Those k edges form a *k-information-optimal* subtree of T_Π, which is a subtree sharing the same root with T_Π that has the maximal gain among all the subtrees (sharing the same root with T_Π) which have k edges. Clearly, enumerating all possible subtrees $t \prec T_\Pi$ with k edges and picking the one with the maximal gain will result in an exponential time algorithm in k. In below, we will present an efficient algorithm that finds a k-information-optimal testing strategy $\mathcal{C}^*$ for a given number k.

We associate each node N in T_Π with a table of $k+1$ entries. The i^{th} entry, $0 \leq i \leq k$, contains a number $G_N[i]$ and a set $OPT_N[i]$ of edges, where $G_N[i]$ is the gain of the i-information-optimal subtree at node N, and $OPT_N[i]$ records the set of edges in that i-information-optimal subtree at node N. Suppose that N has q child nodes $N'_1, \cdots, N'_q$, for some $q \geq 1$. $G_N(i)$ can be calculated from $G_{N'_1}[\mathtt{index}_1], \cdots, G_{N'_q}[\mathtt{index}_q]$, for some $\mathtt{index}_1, \cdots, \mathtt{index}_q < i$. We define another array $Y_j[\cdot]$ for each N'_j. Each $Y_j[i+1]$ records the gain of the $(i+1)$-information-optimal subtree of the component tree $T_{N'_j}$, which consists of the edge $\langle N, N'_j \rangle$ together with the child-tree under the edge. Clearly, the $(i+1)$-information-optimal subtree of $T_{N'_j}$ exactly contains the edge $\langle N, N'_j \rangle$, together with the i-information-optimal subtree at node N'_j. From (2.7), we have $Y_j[i+1] = G(e) + G_{N'_j}[i]$, where $e = \langle N, N'_j \rangle$. One can observe that the i-information-optimal subtree at node N must be, for some $\mathtt{index}_1, \cdots, \mathtt{index}_q$, the union of $\mathtt{index}_j$-information-optimal subtrees of $T_{N'_j}$, for all $1 \leq j \leq q$. Therefore, we have $G_N[i] = \sum_{1 \leq j \leq q} Y_j[\mathtt{index}_j]$. The selection of $\mathtt{index}_1, \cdots, \mathtt{index}_q$ is left to the algorithm `MAX-SELECT`. The algorithm `ALG-opt` (T, T_Π, k) that calculates the k-information-optimal subtree of T_Π is given as follows, where $G_N[\cdot]$ and $OPT_N[\cdot]$ are global variables. If N is a leaf node, then we initialize $G_N[i] = 0$ and $OPT_N[i] = \emptyset$ for each $0 \leq i \leq k$.

```
ALG-opt (T, T_Π, k) :
//To find a k-information-optimal subtree of a given input
//testing tree T_Π with pre-given probability assignment p(e)
//for every edge e in the trace-specification tree T.
```

//The return value has two parts: the entropy $H_{root}[k]$ and the
//set of edges $OPT_{root}[k]$ of the k-information-optimal subtree,
//where *root* is the root of T_Π. $G_N[\cdot]$ and $OPT_N[\cdot]$ in below
//are global variables and are initialized as described above.
1. If $k = 0$
2. For each node N in T_Π
3. $G_N[k] := 0$ and $OPT_N[k] := \emptyset$;
4. Return;
5. Run ALG-opt-tree$(T, T_\Pi, k-1)$;
6. For $level := 1$ to (height of T_Π)
//a node of level (height of T_Π) is the root
7. For each nonleaf node N of level $level$
//suppose that $N'_1, \cdots, N'_q$, for some $q \geq 1$,
//are all the child nodes of N
8. For each $1 \leq j \leq q$
9. $Y_j[0] := 0$;
10. $G(e) := \sum_{t \simeq e} p(\omega_t)(-(\sum_{1 \leq i \leq l} p_i \log p_i + (1 - \sum_{1 \leq i \leq l} p_i) \log(1 - \sum_{1 \leq i \leq l} p_i)))$;
//$e = \langle N, N'_j \rangle$; in the RHS of line 10, e is treated
//as the subtree of T_Π that only has the edge e;
//$\omega_t = a_1 \cdots a_i$ is the root path of t,
//$p(\omega_t) = p(a_1) \cdots p(a_i)$; $p(\omega_t) = 1$ if $\omega_t = \emptyset$;
//suppose that t exactly contains l edges,
//say $e_1, \cdots, e_l$ that share the same source,
//and each edge e_l has probability assignment p_i
11. For $0 \leq i \leq k-1$
12. $Y_j[i+1] := G(e) + G_{N'_j}(i)$;

//$Y_j[i+1]$ stores the entropy of the

//$(i+1)$-information-optimal subtree of the

//component tree $T_{N'_j}$ mentioned earlier

13. Run MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$);

//MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$) returns

//a sequence $\texttt{index}_1, \cdots, \texttt{index}_q$

14. $G_N[k] := \sum_{j:\texttt{index}_j \neq 0} Y_j[\texttt{index}_j]$;

15. $OPT_N[k] := \bigcup_{j:\texttt{index}_j \neq 0} OPT_{N'_j}[\texttt{index}_j - 1] \cup \{\langle N, N'_j \rangle\}$;

16. Return $G_{root}[k]$ and $OPT_{root}[k]$.

In line 13, the (worst-case) time of running the algorithm MAX-SELECT($\{Y_1, \cdots, Y_q\}, k$) is $O(k^2 q)$, as pointed out in Section 2.2.4, MAX-SELECT is called every time when calculating an entry of $H_N[\cdot]$ and an entry $OPT_N[\cdot]$ of node N. Since there are $k+1$ entries for N, the time of finishing calculating all the entries of N is $O(k^3 q)$. Notice that q is the *branching factor* of N (see line 7), that is the number of child nodes of N, and the summation of all such branching factors for all nodes is m, which is the size of T. Hence the (worst-case) time complexity of ALG-opt (T, T_Π, k) is $O(mk^3)$, given that in line 10, all the t's satisfying $t \simeq e$ are preprocessed and already stored in a data structure at e. The preprocessing can be done in time $O(n+m)$, where n is the size of T. Therefore, the (worst-case) time complexity of ALG-opt (T, T_Π, k) is $O(n + mk^3)$.

As mentioned earlier, a global information-optimal testing strategy might not exist for certain trace-specification trees. We can use the algorithm ALG-opt (T, T_Π, k) for computing k-information-optimal testing strategies to determine the existence of a global information-optimal testing strategy by setting k all the way from 1 to m. However, this is not practically efficient when m is large. It is interesting to see whether there are efficient algorithms in deciding the existence of a global information-optimal testing strategy by looking at the tree's structure.

We now consider the following *greedy information-optimal* testing strategy, which aims to reduce the entropy most at each step.

Definition 3. Let T be a trace-specification tree and T_{Π} be the corresponding input testing tree with m edges. $\mathcal{C}^*$ is a `greedy information-optimal testing strategy` if, for any testing strategy $\mathcal{C}$ with $k-1$, for some k, being the length of the longest common prefix between $\mathcal{C}^*$ and $\mathcal{C}$, $G(Pre_{\mathcal{C}^*}(k)) \geq G(Pre_{\mathcal{C}}(k))$.

From the definition, we can have the following observation. Consider two testing strategies $\mathcal{C}$ and $\mathcal{C}'$ of T_{Π} with the longest common prefix between them of length $k-1$. Suppose that $e_{\mathcal{C}(k)}$ is e and $e_{\mathcal{C}'(k)}$ is e'. We have

$$G(Pre_{\mathcal{C}}(k)) \geq G(Pre_{\mathcal{C}'}(k)), \text{ if and only if, } G(e) \geq G(e'). \tag{2.10}$$

To construct the greedy information-optimal testing strategy, we first introduce the concept of *available set*. Let E be a set of edges in T_{Π}. We define the *available set* of E, written $\texttt{AS}(E)$, to be the set of edges e in T_{Π} such that e is either a child edge or a sibling edge of some edge in E. Intuitively, if edges in E form a subtree $t \prec T_{\Pi}$, when adding one or more edges in $\texttt{AS}(E)$ to t, we can obtain a new subtree t', such that $t' \prec T_{\Pi}$ and $t \prec t'$. By definition, $e_{\mathcal{C}(k)}$ must be selected from $\texttt{AS}(\{e_{\mathcal{C}(1)}, \cdots, e_{\mathcal{C}(k-1)}\})$. By (2.10), at each step we pick the edge (from the available set) that can achieve the maximal gain. The gain of an edge e, $G(e)$, is calculated in the same way as in `ALG-opt` (T, T_{Π}, k).

`ALG-greedy-opt` (T, T_{Π})`:`
`//To find a greedy information-optimal testing strategy of the`
`//input testing tree` T_{Π}`, with pre-given probability assignment`
`//`$p(e)$ `for each edge` e `in the trace-specification tree` T`.`
`//The return value is a greedy information-optimal testing`

//strategy $\mathcal{C}^*$ of T_Π.

1. $\mathtt{AS}[1] := \{e : e$ is an edge originating from T_Π's root$\}$.

2. For each $1 \leq i \leq m$

3. For each $e \in \mathtt{AS}[i]$

4. $G(e) := \sum_{t \simeq e} p(\omega_t)(-(\sum_{1 \leq i \leq l} p_i \log p_i + (1 - \sum_{1 \leq i \leq l} p_i) \log(1 - \sum_{1 \leq i \leq l} p_i)))$;
//in the RHS of line 4, e is treated as a subtree
//of T_Π that only has the edge e; $\omega_t = a_1 \cdots a_i$ is the
//root path of subtree t in T, $p(\omega_t) = p(a_1) \cdots p(a_i)$;
//$p(\omega_t) = 1$ if $\omega_t = \emptyset$; suppose that
//t exactly contains l edges, say $e_1, \cdots, e_l$
//that share the same source, and each edge e_l has
//probability assignment p_i in T

5. Suppose that $e^* \in \mathtt{AS}[i]$ achieves $\max_{e \in \mathtt{AS}[i]} G(e)$;

6. $e_{\mathcal{C}(i)} := e^*$;

7. $\mathtt{AS}[i+1] := (\mathtt{AS}[i] - \{e_{\mathcal{C}(i)}\}) \cup \mathtt{CHILDREN}(e_{\mathcal{C}(i)})$;
//$\mathtt{AS}[i+1]$ now records $\mathtt{AS}(\{e_{\mathcal{C}(1)}, \cdots, e_{\mathcal{C}(i)}\})$

8. Return $\mathcal{C}^* = e_{\mathcal{C}(1)}, \cdots, e_{\mathcal{C}(i)}, \cdots, e_{\mathcal{C}(m)}$.

The (worst-case) time complexity of ALG-greedy-opt (T, T_Π) is $O(n + m^2)$, with n being the size of T and m being the size of T_Π.

By definition, a greedy information-optimal testing strategy always selects the edge that can reduce the entropy most at each step. Of course, this greedy strategy does not necessarily lead to the information-optimal testing strategies as given in Definition 1 and 2. For example, for the trace-specification tree and the input testing tree in Figure 2.7 with probability assignments given in Example 7., the prefix of length 3 of a 3-information-optimal testing strategy is e'_1, e'_2, e'_3, while in a greedy information-optimal testing strategy, the prefix of length 3 is e'_5, e'_6, e'_1. However, the

greedy information-optimal testing strategy fits the situation that the testing procedure may be stopped at any time, and hence each time, we only aim to maximally reduce the entropy for each individual step.

2.3.4 Entropy and Information-Optimal Test Strategies of a Trace-specification Tree with Probability Assignments in the Worst Case

The aforementioned algorithms for information-optimal testing strategies rely on the probability assignments of edges, $p(\cdot)$, in the trace-specification tree T. When the assignment $p(\cdot)$ is explicitly given, we write $H(T, p(\cdot))$ to denote the entropy of T under $p(\cdot)$, which is calculated using algorithm `ALG-entropy-tree(`T`)`. However, in practice, we usually do not know what the $p(\cdot)$ is. That is, we do not know the probability of whether an edge will be connected or disconnected with respect to the system under test. We now consider the worst case that we know the least amount of information on the system (i.e., the system under test is a truly black-box). We will give the result in the worst case, where the uncertainty $H(T, p(\cdot))$ achieves the maximum for some $p(\cdot)$. Unless stated otherwise, from now on in this section, we use $H(T)$ to denote the maximal entropy of the testing tree T in the worst case; i,e., $H(T) = \sup_{p(\cdot)}\{H(T, p(\cdot))\}$. We use $p^*(\cdot)$ to denote the worst-case probability assignments on which the maximum is achieved.

Because of Proposition 1, it suffices for us to consider the case when T is a $(b, \cdot)$-component tree shown in Figure 2.5. In this case,

$$H(T) = \sum_{1\le i\le l} p_i H(T_i) - \sum_{1\le i\le l} p_i \log p_i - (1 - \sum_{1\le i\le l} p_i) \log(1 - \sum_{1\le i\le l} p_i). \tag{2.11}$$

Suppose that, for the child-trees $T_1, \cdots, T_l$, we have already obtained the worst-case probability assignments and each $H(T_i)$ in (2.11) is already the worst-case entropy. We now calculate the $p_i = p_i^*$, $i = 1, \cdots, l$, in (2.11) that make the RHS maximal. Notice that $H(T)$ in (2.11) is a

concave function over $p_1, \cdots, p_l$, since the second-order partial derivatives

$$\frac{\partial^2 H(T)}{\partial p_i \partial p_j} < 0, \text{ for all } 1 \leq i, j \leq l.$$

Therefore, let partial derivatives

$$\frac{\partial H(T)}{\partial p_i} = 0, \ i = 1, \cdots l.$$

We have,

$$H(T_i) - \log p_i^* + \log(1 - \sum_{1 \leq j \leq l} p_j^*) = 0, \ i = 1, \cdots l.$$

Solving the above equations for the $p_1^*, \cdots, p_l^*$, we obtain the solutions

$$p_i^* = \frac{2^{H(T_i)}}{1 + \sum\limits_{1 \leq j \leq l} 2^{H(T_j)}}, \ i = 1, \cdots l. \tag{2.12}$$

This already gives an algorithm to calculate the $p^*(\cdot)$ as follows.

ALG-$p^*(T)$:
//To calculate the probability assignments $p^*(\cdot)$ of edges
//in the trace-specification tree T in the worst case.
//The return value is the probability assignments $p^*(\cdot)$.
1. For each leaf node N in T
2. $H(t_N) := 0$;
//t_N is the child-tree under edge $e = \langle N', N \rangle$ for some N'
3. For $level := 1$ to (height of T)
//a node of level (height of T) is the root
4. For each nonleaf node N of level $level$

//suppose that the component trees of N are

//$(b_1,\cdot)$-component tree, $\cdots$, $(b_q,\cdot)$-component tree for some q

5. For $1 \le h \le q$

//let t_h be the $(b_h,\cdot)$-component tree that consists

//of edges $e_1,\cdots,e_l$ labeled with $(b_h,c^1),\cdots,(b_h,c^l)$

//respectively, together with the child-trees T_i's

//under the edges, and the entropy $H(T_i)$ is

//already calculated in the previous level;

//suppose the probability assignment of e_i

//labeled with (b_j,c^i) is p_i^*

6. $p_i^* := \frac{2^{H(T_i)}}{1+\sum_{1\le j\le l} 2^{H(T_j)}}$, $i = 1,\cdots,l$;

7. $H(t_h) := \sum_{1\le i\le l} p_i^* H(T_i) - \sum_{1\le i\le l} p_i^* \log p_i^* - (1-\sum_{1\le i\le l} p_i^*)\log(1-\sum_{1\le i\le l} p_i^*)$;

8. $H(t_N) := \sum_{1\le h\le q} H(t_h)$;

//t_N is the child-tree under the edge $e = \langle N', N\rangle$

//for some N'

9. Return $p^*(\cdot)$.

The time complexity of ALG-$p^*(T)$ is $O(n)$, where n is the size of T.

Example 8. Consider the trace-specification tree T shown in Figure 2.4 (1). Now we calculate the probability assignments of edges in the worst case. For the node N_4, it has a $(b_3,\cdot)$-component tree and a $(b_4,\cdot)$-component tree. For the $(b_3,\cdot)$-component tree, the child-tree under the edge $\langle N_4, N_5\rangle$ is empty and with entropy 0; hence $p^*(\langle N_4, N_5\rangle) = \frac{2^0}{1+2^0} = \frac{1}{2}$. For the $(b_4,\cdot)$-component tree, the child-trees T_1 and T_2 under the edges $\langle N_4, N_6\rangle$ and $\langle N_4, N_7\rangle$, respectively, are also empty; hence $p^*(\langle N_4, N_6\rangle) = p^*(\langle N_4, N_7\rangle) = \frac{2^0}{1+(2^0+2^0)} = \frac{1}{3}$. Similarly, we have $p^*(\langle N_2, N_3\rangle) = \frac{1}{2}$, $p^*(\langle N_0, N_1\rangle) = \frac{1}{2}$, $p^*(\langle N_0, N_2\rangle) = \frac{2}{9}$, and $p^*(\langle N_0, N_4\rangle) = \frac{6}{9}$. Also, the entropy of T in the worst case is $H(T) = \log 18$ bits. □

Now, one can find the k-information-optimal testing strategy and the greedy information-optimal testing strategy of an input testing tree in the worst case using the algorithms `ALG-opt` (T, T_Π, k) and `ALG-greedy-opt` (T, T_Π), respectively, by running the algorithm `ALG-`p^* (T) first to give the probability assignments of edges in the worst case. Note that, as before, the global information-optimal testing strategy may or may not exist.

Example 9. Consider the trace-specification tree T in Figure 2.4 (1) and its corresponding input testing tree T_Π in Figure 2.4 (2). In the worst case that the entropy $H(T)$ reaches the maximum, let $k = 3$. Then a k-information-optimal testing strategy of T_Π is $\langle N'_0, N'_2\rangle, \langle N'_2, N'_4\rangle, \langle N'_0, N'_1\rangle$. The greedy information-optimal strategy is $\langle N'_0, N'_2\rangle, \langle N'_2, N'_4\rangle, \langle N'_0, N'_1\rangle, \langle N'_2, N'_3\rangle$. Note that in this example, the global information-optimal testing strategy exists, which is (in this order) $\langle N'_0, N'_2\rangle, \langle N'_2, N'_4\rangle, \langle N'_0, N'_1\rangle, \langle N'_2, N'_3\rangle$. □

2.3.5 Entropy and Information-Optimal Test Strategies of a Trace-specification Tree with Probability Assignments in the Worst Case with Lower Bounds

In the previous section, we develop optimal testing strategies on input testing trees with the probability assignments in the worst case, where the entropy of the trace-specification tree reaches the maximum. That is, the system has the maximal amount of uncertainty. In practice, we could have additional information about the system under test. For instance, when testing a software system that has already been well tested, or when testing a mature software system, we could assume that an edge e in the trace-specification tree T is connected with a probability that has a *lower bound*; i.e., for each edge e, we assume that $p(e) \geq \delta(e)$ for some given number $\delta(e)$, which is the lower bound of $p(e)$.

We now consider probability assignments $p^*_\delta(\cdot)$ on the trace-specification tree T in the worst case with lower bounds $\delta(\cdot)$ such that the entropy $H(T)$ reaches the maximum. In this section, we require that, when e is labeled with some $(b, c) \in \Pi \times \Gamma$, there is at most one $e' \in E_b$ with $\delta(e') > 0$ (all other $e'' \in E_b$ are with $\delta(e'') = 0$), where E_b is the set containing e and all its siblings with

label (b, c') for some $c' \in \Gamma$. That is, the lower bound is only applied to at most one output symbol per input symbol, at each node of T. Because of Proposition 1, it suffices for us to consider the case when T is a $(b, \cdot)$-component tree shown in Figure 2.5, where

$$H(T) = \sum_{1 \le i \le l} p_i H(T_i) - \sum_{1 \le i \le l} p_i \log p_i - (1 - \sum_{1 \le i \le l} p_i) \log(1 - \sum_{1 \le i \le l} p_i). \tag{2.13}$$

Suppose that each $H(T_i)$ is already the worst-case entropy of T_i with lower bound $\delta(\cdot)$. Now we compute the worst-case entropy $H(T)$ in (2.13) subject to the lower bound constraint. Let $e_1, \cdots e_l$ be the edges (with labels $(b, c^1), \cdots, (b, c^l)$ and probability assignments $p_1 = p(e_1), \cdots, p_l = p(e_l)$, respectively) originating from the root of T, as shown in Figure 2.5. We use $p_1^* = p_\delta^*(e_1), \cdots, p_l^* = p_\delta^*(e_l)$ to denote the probabilities $p_1, \cdots, p_l$, respectively, that make $H(T)$ in (2.13) maximal subject to the lower bound constraint. Suppose that edge e_{j_0}, for some $1 \le j_0 \le l$, is with $\delta_{j_0} = \delta(e_{j_0})$, $0 < \delta_{j_0} \le 1$. By definition, the probability p_{j_0} in (2.13) must satisfy $p_{j_0} \ge \delta_{j_0}$ and there is no constraint for other probabilities p_i with $i \ne j_0$, except $\sum_{1 \le i \le l} p_i \le 1$. That is, we are to maximize $H(T)$ in (2.13) subject to $p_{j_0} \ge \delta_{j_0}$ and $\sum_{1 \le i \le l} p_i \le 1$, for some given j_0 and δ_{j_0}. For instance, in the traffic light example in Section 2.2.3, we can designate the case (ii) that the light turns red with probability ≥ 0.9, which is meaningful in practice. Recall that $H(T)$ in (2.13) is a concave function over $p_1, \cdots, p_l$. We can show (using the concavity)

$$p_{j_0}^* = \max\{\frac{2^{H(T_{j_0})}}{1 + \sum_{1 \le i \le l} 2^{H(T_i)}}, \delta_{j_0}\}.$$

If $\delta_{j_0} \le \frac{2^{H(T_{j_0})}}{1 + \sum_{1 \le i \le l} 2^{H(T_i)}}$, then $p_{j_0}^* = \frac{2^{H(T_{j_0})}}{1 + \sum_{1 \le i \le l} 2^{H(T_i)}}$, and therefore other p_i^*'s ($i \ne j_0$) still take the probability assignments in the form of (2.12), which make $H(T)$ achieve the maximum. If $\delta_{j_0} >$

$\frac{2^{H(T_{j_0})}}{1+\sum_{1\le i\le l} 2^{H(T_i)}}$, then $p^*_{j_0} = \delta_{j_0}$. In the latter case, let

$$\frac{\partial H(T)}{\partial p_i} = 0,\ i = 1, \cdots l,\ i \neq j_0.$$

We have,

$$H(T_i) - \log p^*_i + \log(1 - \delta_{j_0} - \sum_{1\le j\le l, j\neq j_0} p^*_j) = 0,\ i = 1, \cdots l,\ i \neq j_0.$$

Solving the equations for the p^*_i's, $i \neq j_0$, we obtain the solutions

$$p^*_i = \frac{(1 - \delta_{j_0})2^{H(T_i)}}{1 + \sum_{1\le j\le l, j\neq j_0} 2^{H(T_j)}},\ i = 1, \cdots l,\ i \neq j_0. \tag{2.14}$$

Now, we have the following algorithm to calculate the probability $p^*_\delta(\cdot)$ using (2.14).

```
ALG-p*_δ-low(T, δ(·)):
//To calculate the probability assignments p*_δ(·) of edges
//in the trace-specification tree T in the worst case with
//lower bounds δ(·).  The return value is the
//probability assignments p*_δ(·).
1. For each leaf node N in T
2.     H(t_N) := 0;
       //t_N is the child-tree under edge e = ⟨N', N⟩ for some N'
3. For level := 1 to (height of T)
   //a node of level (height of T) is the root
4.     For each nonleaf node N of level level
       //suppose that the component trees of N are
```

//$(b_1,\cdot)$-component tree, $\cdots$, $(b_q,\cdot)$-component tree for some q

5. For $1 \le h \le q$

//let t_h be the $(b_h,\cdot)$-component tree that consists of

//edges $e_1,\cdots,e_l$ labeled with $(b_h,c^1),\cdots(b_h,c^d)$

//respectively, together with the child-trees T_i's

//under those edges, and the entropy $H(T_i)$ is already

//calculated in the previous level; suppose that the

//probability assignment of e_i is $p^*_\delta(e_i)$; we require

//that $p^*_\delta(e_{j_0}) \ge \delta(e_{j_0})$ for a designated j_0 and $\delta(e_{j_0}) > 0$

6. If $\delta(e_{j_0}) \le \frac{2^{H(T_{j_0})}}{1+\sum_{1\le i\le l} 2^{H(T_i)}}$

7. $p^*_\delta(e_i) := \frac{2^{H(T_i)}}{1+\sum_{1\le j\le l} 2^{H(T_j)}}$, for all $1 \le i \le l$;

8. Else

9. $p^*_\delta(e_{j_0}) := \delta(e_{j_0})$;

10. $p^*_\delta(e_i) := \frac{(1-\delta(e_{j_0}))2^{H(T_i)}}{1+\sum_{1\le j\le l, j\ne j_0} 2^{H(T_j)}}$, $i = 1,\cdots l,\ \ i \ne j_0$;

11. $H(t_h) := \sum_{1\le i\le l} p^*_\delta(e_i)H(T_i) - \sum_{1\le i\le l} p^*_\delta(e_i)\log p^*_\delta(e_i)$
$-(1-\sum_{1\le i\le l} p^*_\delta(e_i))\log(1-\sum_{1\le i\le l} p^*_\delta(e_i))$;

12. $H(t_N) := \sum_{1\le h\le q} H(t_h)$;

//t_N is the child-tree under edge $e = \langle N', N\rangle$ for some N'

13. Return $p^*_\delta(\cdot)$.

The time complexity of ALG-p^*_δ-low$(T,\delta(\cdot))$ is $O(n)$, where n is the size of T. Similarly as in Section 2.3.4, one can find the k-optimal testing strategy and the greedy optimal testing strategy of an input testing tree in the worst case with lower bounds still using the algorithms ALG-opt(T,T_Π,k) and ALG-greedy-opt(T,T_Π), respectively, by running the algorithm ALG-p^*_δ-low$(T,\delta(\cdot))$ first to obtain probability assignments.

Note that lower bounds can be given in various forms. For instance, we can require that for some edges $(b, c_1), \cdots, (b, c_i)$ in a $(b, \cdot)$-component tree shown in Figure 2.5 , the summation of their probability assignments must be larger than some given number. Currently we have not found polynomial-time algorithms to solve the above problem (which turns out to be a quite complex nonlinear optimization problem).

2.4 Information-Optimal Testing Strategies on Automata Used as Tree Representations

Throughout this section, the system Sys under test is assumed to be output-deterministic. Additionally, we assume that a trace-specification tree is *tight*; i.e., at each node, each input symbol has at most one output symbol. Formally, T is tight if, for each node N in T and each input symbol $b \in \Pi$, there is at most one edge from node N with label (b, c), for some $c \in \Gamma$. In this case, a trace-specification tree and its input testing tree have exactly the same topological appearance (see Figure 2.7 for an example). Since T is tight, for a test case $\omega = b_1 \cdots b_l$ (for some l) in the input testing tree T_Π, there is a unique path in T with labels $(b_1, c_1) \cdots (b_l, c_l)$, for some $c_1, \cdots, c_l \in \Gamma$. That is, the expected output $c_1 \cdots c_l$ is unique and already specified in T for the test case $\omega = b_1 \cdots b_l$. Therefore, we do not distinguish the two trees, and simply treat the trace-specification tree T as the *testing tree*. Because of this, in this section, we also call the input-output sequence $(b_1, c_1) \cdots (b_l, c_l)$ as a test case. Let $\Sigma = \Pi \times \Gamma$. Hence, the testing tree T is simply a tree with labels in Σ.

Recall that, the original trace-specification, $P_{original}$, could be an infinite set of words over the interface Σ; the trace-specification P we actually plan to test is the set of words where each word is a prefix (of length $\leq d$) of some word in $P_{original}$. In other words, P can be specified by $P_{original}$ together with d. One can often use a (deterministic finite) automaton A to represent $P_{original}$ (when it is regular), since it is practically a more succinct model compared with the tree representation of the trace-specification P. For instance, for the original trace-specification $P_{original} = (a_1 + a_2)^*$,

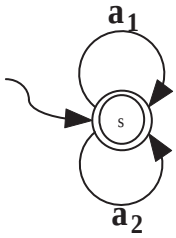


Figure 2.8: A DFA A with the accepting language $(a_1 + a_2)^*$.

the trace-specification we actually test is $P = (a_1 + a_2)^{\leq d}$. The testing tree T representing P is a complete binary tree, whose size is exponential in d. On the other hand, if we use an automaton A in Figure 2.8 to accept the language $(a_1 + a_2)^*$ (i.e., $P_{original}$), the automaton only has one state and two transitions. Therefore, the trace-specification P we actually test can be specified by two parameters: an automaton A representing $P_{original}$, and the longest length d we could test for each sequence in $P_{original}$. Suppose that P is given as an automaton A and a length d. If we calculate the entropy of the testing tree T representing P by building T (using `ALG-entropy-tree(`T`)`), the time complexity would be exponential in the size of the automaton A and the length d. In the following, we discuss how to efficiently calculate the worst-case entropy of the testing tree T, and develop information-optimal testing strategies directly on the automaton A, without building T.

Let $A = \langle S, s_{init}, F, \Sigma, R\rangle$ be a DFA specified in Section 2.2.1. We further define $A(s) = \langle S, s, F, \Sigma, R\rangle$ as the finite automaton that keeps all the parameters in A except that it changes the initial state s_{init} to $s \in S$. Let $L(A, s, d)$ be the set of words such that each word is a prefix (of length $\leq d$) of some word in $L(A(s))$.

Let T be the testing tree that represents P. Recall that P is truncated from $P_{original} = L(A)$ up to length d; i.e., $P = L(A, s_{init}, d)$. Now we develop algorithms to calculate the entropy of T in the worst case (i.e., the entropy $H(T)$ reaches the maximum). The algorithms are based on Proposition 1 and 2. Note that since T is tight, a $(b, \cdot)$-component tree T_b, with $b \in \Pi$, in Figure 2.5 now exactly consists of one edge e originating from the root and a child-tree t' under e, as shown

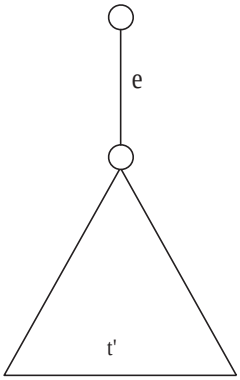


Figure 2.9: The tree T_b consists of a child-tree t' and an edge e.

in Figure 2.9; hence the formula in Proposition 2,

$$H(T_b) = \sum_{1 \le i \le l} p_i H(T_i) - \sum_{1 \le i \le l} p_i \log p_i - (1 - \sum_{1 \le i \le l} p_i) \log(1 - \sum_{1 \le i \le l} p_i),$$

now can be written as

$$H(T_b) = H_{\texttt{Binary}}(p(e)) + p(e)H(t'),$$

where $H_{\texttt{Binary}}(\cdot)$ is the binary entropy function, $H_{\texttt{Binary}}(p(e)) = -p(e)\log p(e) - (1-p(e))\log(1-p(e))$. Therefore,

$$H(T) = \sum_i (H_{\texttt{Binary}}(p(e_i)) + p(e_i)H(T_i)), \tag{2.15}$$

where e_i is an edge that directly originates from the root of T, and T_i is the child-tree under e_i. In the worst case, from (2.12), we have $p^*(e_i) = \frac{2^{H(T_i)}}{1+2^{H(T_i)}}$, and therefore (2.15) can be written as

$$H(T) = \sum_i \log(1 + 2^{H(T_i)}). \tag{2.16}$$

Recall that the tree T is represented by a given automaton A. To use (2.16) to calculate the worst-case entropy $H(T)$ directly on the automaton A, we build an array $h_s[0 \cdots d]$ with $d+1$ entries for each state s in the automaton A, and the $level^{th}$ ($0 \le level \le d$) entry $h_s[level]$ equals

$H(A, s, level)$, where $H(A, s, level)$ is the (worst case) entropy of the testing tree that represents $L(A, s, level)$. Clearly, for the testing tree that represents $L(A, s, level)$, each of its child-trees represents $L(A, s', level-1)$, for each direct successor s' of s (i.e., $(s, a, s') \in R$ for some $a \in \Sigma$), respectively. From (2.16), we have

$$h_s[level] = \sum_{s' \in \mathtt{SUCC}(s)} \log(1 + 2^{h_{s'}[level-1]}),$$

where $\mathtt{SUCC}(s)$ is the set of direct successors of s. Note that $\mathtt{SUCC}(s)$ could be a *multiset*; i.e., if there are multiple transitions from s to s', then s' should be counted multiple times in $\mathtt{SUCC}(s)$. In particular, $h_s[0] = 0$ for each state s, which means that, the testing tree representing $L(A, s, 0)$ is an empty tree, therefore, $H(A, s, 0) = 0$. The algorithm `ALG-entropy-DFA-worst` (A, d) is given as follows.

`ALG-entropy-DFA-worst` (A, d) `:`

`//To calculate the entropy of the testing tree representing`

`//` $P = L(A, s_{init}, d)$ `in the worst case. The return value is the`

`//entropy` $h_s[level]$ `of the testing tree representing` $L(A, s, level)$`,`

`//for each state` s `in` A`, and` $0 \leq level \leq d$`.`

`1. For each` $s \in S$ `and for` $0 < level \leq d$

`2.` $h_s[level] :=$ `null;`

`3.` $h_s[0] := 0$`;`

`4.` $level := 1$`;`

`5. Repeat`

`6. For each state` s

`7.` $h_s[level] := \sum_{s' \in \mathtt{SUCC}(s)} \log(1 + 2^{h_{s'}[level-1]})$`;`

`8.` $level := level + 1$`;`

9. Until $level \geq d+1$;

10.Return $h_s[0 \cdots d]$ for each state s.

The time complexity of the algorithm ALG-entropy-DFA-worst(A, d) is $O(dn)$, where n is the number of transitions in A (also called the size of A). When we finish running the above algorithm on A, we will get an array for each state s, and the value $h_s[level]$ equals $H(A, s, level)$. Therefore, $h_{s_{init}}[d]$ is the entropy of the testing tree T representing $L(A, s_{init}, d)$, which is the trace-specification P that we actually test, as mentioned earlier.

In the following, we show an example of running the algorithm ALG-entropy-DFA-worst(A, d) on an automaton A with $d = 3$.

Example 10. An automaton A is given in Figure 2.10, with s_1 being the initial state, and s_5 being the accepting state. Recall that a, b, c, d, e, f in Figure 2.10 are in $\Sigma = \Pi \times \Gamma$. Assume that we can only test up to $d = 3$ steps for each string in $L(A)$; i.e., the trace-specification P we actually test is $L(A, s_1, 3)$. Figure 2.10 $(1) \sim (4)$ shows how the arrays of states evolve while ALG-entropy-DFA-worst(A, d) is running on A (blank entries denotes for null). At the beginning, lines $1 \sim 3$ initialize the entries as in Figure 2.10 (1), which corresponds to $H(A, s, 0) = 0$ for every $s \in S$, since the set of strings originating from s of length 0 is empty. For each state $s \in S$, we gradually update its entries according to lines $6 \sim 8$ as shown in Figure 2.10 $(2) \sim (4)$. Finally, $h_{s_1}[3] = \log 16$ implies that $H(A, s_1, 3) = \log 16 = 4$ bits. That is, the entropy of the testing tree T representing $P = L(A, s_1, d)$ is 4 bits. We can check the result with the tree T in Figure 2.11. In the worst case, $H(T) = 4$ bits, which coincides with $h_{s_1}[3]$. □

Note that, the number d is usually pre-given. However, if A is a directed acyclic graph (DAG), we could set d to the length of the longest string in $L(A)$ (which could be found using depth-first-search) such that the trace-specification we actually test is $L(A)$; i.e., $P = P_{original}$.

Example 11. In the DFA A in Figure 2.10, the longest string in $L(A)$ is of length 4. There-

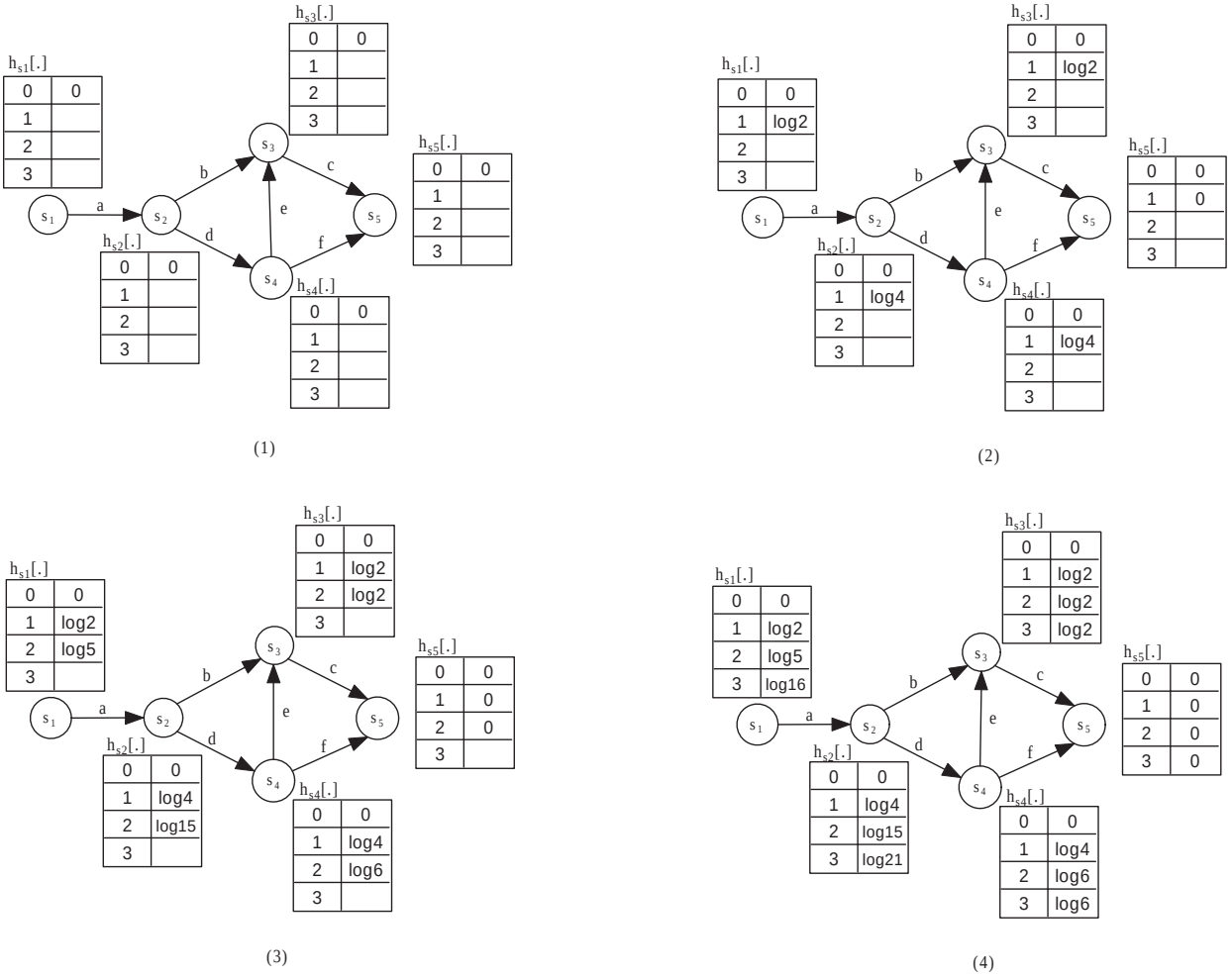


Figure 2.10: Run `ALG-entropy-DFA-worst`(A, d).

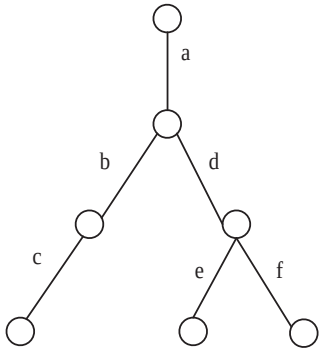


Figure 2.11: The testing tree representing $P = L(A, s_1, 3)$.

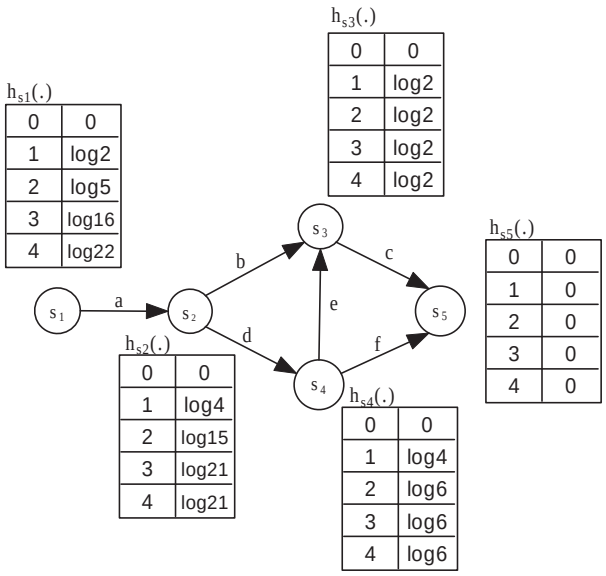


Figure 2.12: Run `ALG-entropy-DFA-worst` (A, d).

fore, we set d to 4, and run `ALG-entropy-DFA-worst` (A, d) on A. In this case, the trace-specification P is $L(A)$. The result is shown in Figure 2.12. Since $h_{s_1}[4] = \log 22$, the entropy of the testing tree T that represents $P = L(A)$ is $\log 22$. □

Next, we study information-optimal testing strategies of the trace-specification P in the worst case directly on the finite automaton A. Recall that in the testing tree T that represents P, an edge e is labeled with some symbol $a \in \Sigma = \Pi \times \Gamma$. $p(e)$ denotes the probability that e is connected; i.e., the symbol a *succeeds* (in fact, $p(e)$ is the probability that a succeeds, given that the symbols labeling the parent of e succeeds. In the sequel, we simply say that $p(e)$ is the probability that a succeeds, when the context is clear). Correspondingly, we also have the probability that symbol a succeeds in the automata representation. Let $r = (s, a, s')$ be a transition in the finite automaton A. We define $p(r(level))$ as the probability that, in the testing tree representing $L(A, s, level)$, the symbol a (that is labeled on an edge originating from the root of the testing tree) succeeds, for each $1 \leq level \leq d$. Recall that in the worst case, the probability of an edge e is $p^*(e) = \frac{2^{H(t')}}{1+2^{H(t')}}$, where t' is the child-tree under e. In the DFA A, after running `ALG-entropy-DFA-worst` (A, d) on

A, $h_s[level]$ is the entropy of the testing tree t that represents $L(A, s, level)$, and $h_{s'}[level-1]$ is the entropy of the testing tree t' that represents $L(A, s', level-1)$. Clearly, t' is a child-tree of t. Therefore, in the worst case, $p(r(level)) = p^*(r(level)) = \frac{2^{h_{s'}[level-1]}}{1+2^{h_{s'}[level-1]}}$. The algorithm that calculates probability assignments in the worst case for transitions in the DFA A, $p^*(\cdot)$, is given as follows.

ALG-p^*-DFA-worst(A, d):
//To calculate the probability assignments $p^*(\cdot)$ in the worst
//case in a DFA A when only testing sequences of length
//up to d. The return value is the probability $p^*(r(level))$
//for each transition r in A, $1 \leq level \leq d$.
1. Run ALG-entropy-DFA-worst(A, d) on A;
   //This will return $h_s[0 \cdots d]$ for each state s.
2. For each transition $r = (s, a, s')$ in A
3. For each $1 \leq level \leq d$
4. $p^*(r(level)) := \frac{2^{h_{s'}[level-1]}}{1+2^{h_{s'}[level-1]}}$;
5. Return $p^*(\cdot)$.

The time complexity of ALG-p^*-DFA-worst(A, d) is actually the time complexity of ALG-entropy-DFA-worst(A, d), which is $O(dn)$. (From now on, when the context is clear, we simply use a to denote a transition (s, a, s')).

Example 12. Now we calculate the probability assignments $p^*(\cdot)$ in the DFA A shown in Figure 2.12. After running ALG-p^*-DFA-worst(A, d), $p^*(a(4)) = 21/22$, $p^*(b(4)) = 2/3$, $p^*(d(4)) = 6/7$, $p^*(e(4)) = 2/3$, $p^*(c(4)) = 1/2$, and $p^*(f(4)) = 1/2$. The complete probability assignments $p^*(r(level))$ for each transition r and $1 \leq level \leq 4$ are illustrated in Figure 2.13. □

Remember that a finite automaton is just another form of a testing tree. A testing strategy $\mathcal{C}$

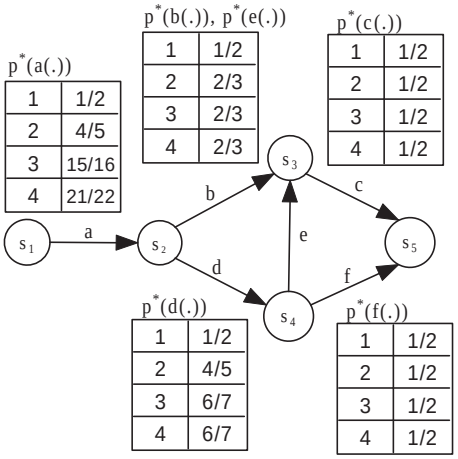


Figure 2.13: Run ALG-p^*-DFA-worst(A, d).

of a finite automaton A, which resembles a testing strategy of a testing tree, specifies an ordering in which we test sequences in the trace-specification $P = L(A, s_{init}, d)$. Now that we have ways to calculate probability assignments on DFA A, we can develop algorithms to calculate testing strategies of A. Before giving the formal definition of a testing strategy on a DFA, we first introduce some notation. Let α be a string and L be a language. We say $\alpha \prec L$, if α is a prefix of some word in L. We use $\alpha = \beta \circ \kappa$ to denote the string α that is a concatenation of β and κ, where each of β and κ can either be a symbol or a string. Sometimes, we simply write $\alpha = \beta\kappa$. Suppose that $\alpha = \omega a \prec L(A, s_{init}, d)$. When A runs on α, we use $s(\alpha)$ to denote the state of A right after the last symbol a in $\alpha = \omega a$ is read. In particular, for an empty string ϵ, we define $s(\epsilon) = s_{init}$. A testing strategy $\mathcal{C}$ of a DFA A with respect to a given length d is in the form of

$$\mathcal{C} = \alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i)}, \alpha_{\mathcal{C}(i+1)}, \cdots, \alpha_{\mathcal{C}(g)},$$

for some $g > 0$, where each string $\alpha_{\mathcal{C}(i)} = \omega a \prec L(A, s_{init}, d)$ for some ω and a. Sometimes, we just call $\mathcal{C}$ a testing strategy of $L(A, s_{init}, d)$. By testing $\alpha_{\mathcal{C}(i)} = \omega a$ in the testing strategy $\mathcal{C}$, we mean to test the rightmost symbol a in $\alpha_{\mathcal{C}(i)}$, and symbols in ω have already been tested. In this way, the testing strategy $\mathcal{C}$ actually specifies a testing strategy in the corresponding testing

tree T that represents $L(A, s_{init}, d)$. Therefore, a prefix of $\mathcal{C}$ corresponds to a subtree $t \prec T$ that represents the strings in the prefix. We simply call the entropy of that subtree t as the gain of that prefix of the testing strategy $\mathcal{C}$. The number g is pre-given. In fact, in order to make the strategy $\mathcal{C}$ be an exhaustive testing strategy of $L(A, s_{init}, d)$, the g, in worst case, can be exponential in n (the number of transitions in A). In practice, such a long strategy may not be exhaustively tested anyway. Therefore, one can expect that the given length g of the strategy is not unreasonably large.

For the k-information-optimal testing strategy (with worst-case entropy) of a DFA A, the idea to calculate it is similar to `ALG-opt(`T, k`)`. We first run `ALG-`p^*`-DFA-worst(`A, d`)` to obtain the worst-case probability assignments $p^*(\cdot)$. We associate each state s in A with two arrays $H_s[\cdot, \cdot]$ and $OPT_s[\cdot, \cdot]$. The meanings of the two arrays are explained as follows. Let T be the testing tree representing $P = L(A, s_{init}, d)$. For a state s in A and a number $0 \leq level \leq d$, consider the subtree $T(s, level)$ of T that represents $L(A, s, level)$, which keeps the probability assignments of edges in T. $H_s[level, i]$ and $OPT_s[level, i]$ record the entropy and the set of root paths (i.e., the k-information-optimal testing strategy) of the i-information-optimal subtree of $T(s, level)$, for each $0 \leq level \leq d$ and $0 \leq i \leq k$, respectively. We initialize $H_s[level, i] = 0$ and $OPT_s[level, i] = \emptyset$ for each $0 \leq level \leq d$ and $0 \leq i \leq k$. For each state s in A, suppose that $r_1, \cdots, r_q$ are all the transitions from s, and $r_j = (s, a_j, s'_j)$ for some a_j and s'_j, for all $1 \leq j \leq q$. Note that the $s'_1, \cdots, s'_q$ are not necessarily distinct. From the i-information-optimal subtree of $T(s'_j, level)$ (whose entropy is stored in $H_{s'_j}[level, i]$), we can calculate the $(i+1)$-information-optimal subtree of the component tree $T_{s'_j}$ which consists of an edge labeled with a_j together with $T(s'_j, level)$ under the edge. Similar to the algorithm `ALG-opt(`T, k`)`, we define another array $Y_j[\cdot]$ to record the entropy of that $(i+1)$-information-optimal subtree of $T_{s'_j}$ in $Y_j[i+1]$. In this way, we obtain arrays $Y_1[\cdot], \cdots, Y_q[\cdot]$. The problem of calculating the entropy of the i-information-optimal subtree of $T(s, level+1)$ (i.e., $H_s(level+1, i)$) now becomes selecting indices $\texttt{index}_1, \cdots, \texttt{index}_q$, satisfying $\sum_{1 \leq j \leq q} \texttt{index}_j = i$, for $Y_1, \cdots, Y_q$, such that $\sum_{1 \leq j \leq q} Y_j[\texttt{index}_j]$

achieves the maximum, which, again, can be solved by the algorithm MAX-SELECT. The algorithm ALG-opt-DFA-worst(A, d, k) that calculates the k-information-optimal subtree of the testing tree representing $L(A, s_{init}, d)$ is given as follows, where $H_s[\cdot,\cdot]$ and $OPT_s[\cdot,\cdot]$ are global variables, which are already initialized in above.

ALG-opt-DFA-worst(A, d, k):
//To calculate the k-information-optimal subtree of the tree
//representing $L(A, s_{init}, d)$. Assume that probability assignments
//in the worst case $p^*(\cdot)$ are pre-calculated using
//ALG-p^*-DFA-worst(A, d). The return values has two parts:
//the entropy $H_{s_{init}}[d, k]$ and the set of root paths (i.e.,
//the k-information-optimal testing strategy) $OPT_{s_{init}}[d, k]$.
//$H_s[\cdot,\cdot]$ and $OPT_s[\cdot,\cdot]$ are global variables.
1. If $k = 0$
2. For each state s and each $0 \le level \le d$
3. $H_s[level, k] := 0$ and $OPT_s[level, k] := \emptyset$;
4. Return;
5. Run ALG-opt-DFA-worst$(A, d, k-1)$;
6. For each state s that has at least one successor
7. For $level := 1$ to d
//suppose that $r_1, \cdots, r_q$, for some $q \ge 1$, are all the
//transitions from state s. We use $s'_1, \cdots, s'_q$ to denote
//the target-states in the transitions
//($s'_1, \cdots, s'_q$ are not necessarily distinct).
8. For each $1 \le j \le q$
9. $Y_j[0] := 0$;

10. For $0 \leq i \leq k-1$

11. $Y_j[i+1] := p^*(r_j(level)) \cdot H_{s'_j}[level-1, i] + H(p^*(r_j(level)))$;

12. Run MAX-SELECT$(\{Y_1, \cdots, Y_q\}, k)$;

13. $H_s[level, k] := \sum_{j:\mathtt{index}_j \neq 0} Y_j[\mathtt{index}_j]$;

14. $OPT_s[level, k] := \bigcup_{j:\mathtt{index}_j \neq 0} \{a_j\} \cup \{a_j \circ \omega : \omega \in OPT_{s'_j}[\mathtt{index}_j - 1]\}$;

//a_j is the symbol on transition r_j

15. Return $H_{s_{init}}[d, k]$ and $OPT_{s_{init}}[d, k]$.

Following the analysis of algorithm ALG-opt(T, k), we can obtain that the (worst-case) time complexity of ALG-opt-DFA-worst(A, d, k) is $O(ndk^3)$, where n is the number of transitions in A.

For the greedy information-optimal testing strategy, similarly as in the case for the testing tree, we also have the *available set* of $\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\}$, denoted as $\mathtt{AS}(\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\})$, to represent the set of strings that are qualified to be $\alpha_{\mathcal{C}(i)}$. Formally, for a set of strings L, we define $\mathtt{AS}(L) = \{\omega a : \omega \in L \text{ and } s(\omega, a, s') \in R \text{ for some state } s'\}$. That is, the $\mathtt{AS}(L)$ is the set of string ωa that are extended from a string ω in L, with an additional symbol a such that the automaton A will not crash after reading through ωa. Notice that, when strings in $\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\}$ form a subtree $t \prec T$ (in here, each string in t is a path), we can show that, strings in $\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i)}\}$ also form a subtree $t' \prec t'$ with $t \prec t' \prec T$, whenever $\alpha_{\mathcal{C}(i)} \in \mathtt{AS}(\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}\})$. Now consider $\alpha_{\mathcal{C}(i)} = \omega a$, for some ω and a. Suppose that, when running A on the word ωa, the last transition fired is $r = (s, a, s')$ for some s, s'. When one tests the last symbol a in the ωa (so, in this case, all symbols in ω are already tested), the probability that a succeeds in the worst case, as show in ALG-p^*-DFA-worst(A, d), is $p^*(r(d - |\omega|)) = p^*(r(d - |\alpha_{\mathcal{C}(i)}| + 1))$. Modifying the algorithm ALG-greedy-opt(T) that calculates the greedy information-optimal testing strategy of a testing tree, we have the following algorithm, ALG-greedy-DFA-worst(A, d), that calculates the greedy information-optimal testing strategy of a DFA A in the worst case.

ALG-greedy-DFA-worst(A, d):

//To calculate the greedy information-optimal testing strategy

//of a DFA A, when only testing sequences of length up to d.

//The return value is the greedy information-optimal testing

//strategy $\mathcal{C}^*$ of A.

1. run ALG-p^*-DFA-worst(A, d) on A;

   //This gives probability assignment $p^*(r(level))$ for all

   //transitions $r \in R$ in A and $1 \leq level \leq d$.

2. $\mathbf{AS}[1] := \{a : r = (s_{init}, a, s') \in R \text{ for some s}'\}$;

3. For each $1 \leq i \leq g$

4. If $\mathbf{AS}[i] = \emptyset$

5. Return $\mathcal{C}^* = \alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i-1)}$;

6. Else

7. For each $\alpha \in \mathbf{AS}[i]$

8. If $|\alpha| = 1$

9. $\Delta(\alpha) := H(p^*(r(d)))$;

    //$\alpha = a$, and $r = (s_{init}, a, s')$ for some s'.

10. Else

11. $\Delta(\alpha) := (\prod_{1 \leq j \leq |\alpha|-1} p(r_j(d-j+1)))H(p^*(r_{|\alpha|}(d-|\alpha|+1)))$;

    //$\alpha = a_1 \cdots a_j \cdots a_{|\alpha|-1} a_\alpha$, and $r_j = (s, a_j, s')$

    //for some s and s'.

12. Suppose that $\alpha^* \in \mathbf{AS}[i]$ achieves $\max_{\alpha \in \mathbf{AS}(\alpha_{\mathcal{C}(i)})} \Delta(\alpha)$;

13. $\alpha_{\mathcal{C}(i)} := \alpha^*$;

14. $\mathbf{AS}[i+1] := (\mathbf{AS}[i] - \{\alpha_{\mathcal{C}(i)}\}) \cup \{\alpha_{\mathcal{C}(i)} \circ a : (s(\alpha_{\mathcal{C}(i)}), a, s') \in R \text{ for some s}'\}$;

    //Now, $\mathbf{AS}[i+1] = \mathbf{AS}(\{\alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(i)}\})$

15.Return $\mathcal{C}^* = \alpha_{\mathcal{C}(1)}, \cdots, \alpha_{\mathcal{C}(g)}$.

Let b_s be the number of transitions starting from a state s, and in worst case, $b_s = O(n)$, where n is the number of transitions in A. We assume that $g > d$. The time complexity of ALG-greedy-DFA-worst(A, d) is $O(dn) + O(gb_s \log b_s) = O(gn \log n)$, where $O(dn)$ is time of running ALG-p^*-DFA-worst(A, d), and $O(gn \log n)$ is time of running lines $7 \sim 14$ for maximally g times.

Example 13. Following ALG-greedy-DFA-worst(A, d), we calculate the greedy information-optimal testing strategy $\mathcal{C}^*$ for the DFA A in Figure 2.12, where $d = 4$. We set g to a relatively large number such that the testing strategy obtained is the exhaustive testing of $P = L(A, s_{init}, d)$. At the beginning, from line 2 in ALG-greedy-DFA-worst(A, d), $\mathtt{AS}[1] = \{a\}$; therefore, $\alpha_{\mathcal{C}(1)} = a$, and $\mathtt{AS}[2] = \{ab, ad\}$ according to line 9. Since the probability that the symbol b in ab succeeds is $p^*(b(4 - |ab| + 1)) = p(b(3)) = 2/3$, and the probability that the symbol d in ad succeeds is $p^*(d(3)) = 6/7$. We have $\Delta(ab) = p^*(a(4))H(p^*(b(3))) = 0.88$, and $\Delta(ad) = p^*(a(4))H(p^*(d(3))) = 0.56$; from lines 12 and 13, we have $\alpha_{\mathcal{C}(2)} = ab$ and $\mathtt{AS}[3] = \{ad, abc\}$. Keep doing so, until $\mathtt{AS}[i] = \emptyset$ for some i, and finally we obtain the greedy information-optimal testing strategy $\mathcal{C}^* = a, ab, abc, ad, adf, ade, adec$ (in this order). □

Now, we finish developing algorithms to calculate greedy optimal strategies and k-optimal strategies on a DFA in the worst case where the entropy reaches the maximum. The algorithms can be easily generalized to the worst case with lower bounds δ (here we only discuss the case where each transition is assigned with the same lower bound δ), by noticing that in the worst case with lower bounds, $p(e) = p^*_\delta(e) = \max\{\frac{2^{H(t')}}{1+2^{H(t')}}, \delta\}$, instead of $p(e) = p^*(e) = \frac{2^{H(t')}}{1+2^{H(t')}}$ in the worst case, where t' is the child-tree under the edge e in the testing tree. Hence, the algorithm ALG-p^*_δ-DFA-worst(A, d) that calculates probability assignments in the worst case with lower bounds δ can be obtained from ALG-p^*-DFA-worst(A, d) by adopting slight modifications. ALG-p^*_δ-DFA-worst(A, d) is given as follows:

ALG-p^*_δ-DFA-worst(A, d) :

//To calculate the probability assignment $p^*_\delta(\cdot)$ in the worst

//case with lower bounds δ in a DFA A, when only

//testing sequences of length up to d. The return value is

//the probability $p^*_\delta(r(i))$ for each r in A, $1 \leq i \leq d$.

1. Run ALG-entropy-DFA-worst-$\delta(A, d)$ on A;

2. For each transition $r = (s, a, s')$ in A

3. For each $1 \leq i \leq d$

4. $p^*_\delta(r(i)) = \max\{\frac{2^{s'(i-1)}}{1+2^{s'(i-1)}}, \delta\}$;

5. Return $p^*_\delta(\cdot)$.

Correspondingly, it is easy to obtain the algorithm ALG-opt-DFA-worst-$\delta(A, d, k)$ (resp. ALG-greedy-DFA-worst-$\delta(A, d)$) that calculates the k-optimal (resp. greedy optimal) testing strategy in the worst case with lower bounds δ from ALG-opt-DFA-worst(A, d, k)) (resp. ALG-greedy-DFA-worst(A, d)) by first running the algorithm ALG-p^*_δ-DFA-worst(A, d) instead of running ALG-p^*-DFA-worst(A, d). Currently, we are unable to obtain polynomial time algorithms for calculating k-optimal and greedy optimal testing strategies in the worst case with non-uniform lower bounds $\delta(\cdot)$ (i.e., different edges have different designated lower bounds). Part of the problem is the difficulty in specifying the nonuniform bounds $\delta(\cdot)$ on the automaton A.

Finally, we show that a test set that achieves 100% branch coverage could still reveal very little information of a software system. Consider the finite automaton A (which can be interpreted as a software design) shown in Figure 2.14. Let a test set t consist of $a_1 \cdots a_n$, $b_1 \cdots b_n$, $c_1 \cdots c_n$, $d_1 \cdots d_n$ and together with all their prefixes. Clearly, t achieves 100% branch coverage in A. However, one can show that, under worst-case probability assignments of edges, the ratio of t's

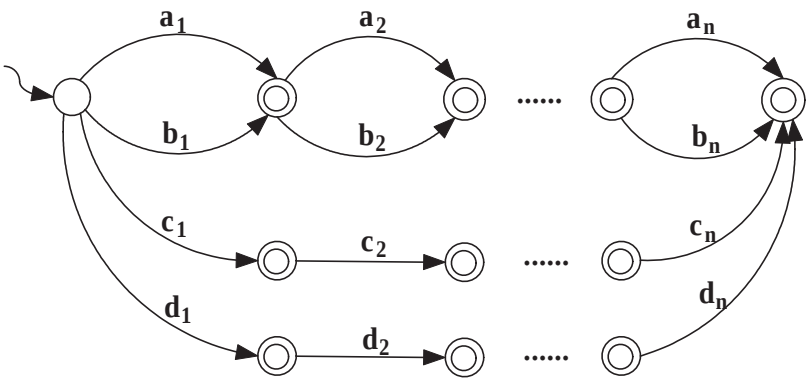


Figure 2.14: An example finite automaton A.

information gain to the entropy of the automaton approaches 0 as $n \to \infty$. On the other hand, two test sets sharing the same branch coverage may gain dramatically different amount of information. For instance, consider the two test sets

- t_1 consisting of $a_1 \cdots a_n$, $b_1 \cdots b_n$, and together with all their prefixes,
- t_2 consisting of $c_1 \cdots c_n$, $d_1 \cdots d_n$, and together with all their prefixes.

Clearly, t_1 and t_2 both achieve 50% branch coverage. One can show that under worst-case probability assignments of edges, the ratio of the information gain of t_1 to the information gain of t_2 approaches to ∞ as $n \to \infty$. Similarly, our conclusions remain for path coverage. This example implies that the path $a_1 \cdots a_n$ is "more important" (i.e., contains much more information) than the path $c_1 \cdots c_n$ as n becomes large. Notice that, under path coverage, these two paths are not distinguishable since they have the same path coverage. As the information gain criterion is syntax-independent, it provides a way to compare test sets that are not differentiable under other traditional testing criteria and helps us select a set that achieves maximal information gain using algorithms presented earlier.

2.5 Summary

In our understanding, software testing is a cooling-down process, during which the entropy (or uncertainty) of the system under test decreases. In this chapter, we have studied information-optimal software testing where test cases are selected to gain the most information (i.e., cools down the system under test fastest). More specifically, we represent a trace-specification (a finite set of input-output sequences) as a tree, that the black-box transition system under test is intended to conform with. When the tree is associated with pre-given probability assignments on the edges, we have developed polynomial-time algorithms calculating the entropy of the trace-specification and computing k-information-optimal and greedy information-optimal testing strategies. When the tree is not pre-given with probability assignments on the edges, we have studied efficient algorithms calculating the assignments that make the trace-specification tree be with maximal entropy (i.e., the worst case that we know the least amount of information about the system under test). In this latter case, we have also provided polynomial-time algorithms computing k-information-optimal and greedy information-optimal testing strategies. We also study the case where the entropy of the trace-specification tree reaches the maximum while lower bounds of probability assignments are imposed on the tree. Finally, we have generalized our algorithms to find information-optimal testing strategies to the case when the trace-specification is tight and when the trace-specification is, often succinctly, represented as a finite automaton. We shall emphasize that the information-optimal testing strategies are calculated *before* any testing is performed.

We now briefly discuss testing an output-nondeterministic system Sys. We assume that there is an oracle $\mathcal{O}$ (i.e., a test engine) to consume a test case (which is a sequence of input-output pairs) and provide a sequence of Boolean values to tell whether the prefixes of a test case are the observable behaviors of Sys. Logically, we can treat the Sys as an output-deterministic system Sys' where an input-output pair in Sys is an input symbol in Sys' and the output symbols in Sys'

are simply Booleans. However, this does not confirm the following statement: testing output-nondeterministic systems is special case of testing output-deterministic ones. A precise conclusion should be, once the oracle $\mathcal{O}$ for testing output-nondeterministic system is built, the statement is true. However, as we have mentioned earlier, the oracle is difficult to build in practice, which makes testing nondeterministic systems hard.

CHAPTER 3

AN INFORMATION-THEORETIC COMPLEXITY METRIC

3.1 Overview

Complexity of software systems is an indicator on how difficult it is to understand, test, and/or maintain the software systems. There are various metrics used to measure software complexity. The simplest and most straightforward way measures the physical size of a software system (e.g., the number of lines of code), which is obviously insufficient. Some of the more sophisticated metrics are graph-theoretic and measure the complexity of a graph specifying the software system. One such metric that has been widely used is the McCabe metric [45], which employs the *cyclomatic complexity* (that is $E + N - 2P$, with E, N and P being the numbers of edges, nodes and connected components, respectively) in traditional graph theory to measure the control flow graph of a software system. Basically, the McCabe metric counts the number of linearly independent paths (which constitute the minimal set of paths such that all paths can be generated as a combination of paths in that set) in the control flow graph [45]. Some other approaches are language-theoretic; i.e., treating a software system as a language expression written in a program language and measuring the complexity of the expression. One such metric is Halstead's metric [29] that measures a software system by counting the number of operators and operands in its language expression. A more recent approach, which is neither graph-theoretic nor language-theoretic, is the Dounce-Layzell-Buckley spatial complexity [24, 15, 16]. It analyzes the distance of components in a software system, where the distance is the number of lines of code between components (e.g., when components are functions, the distance is the number of lines of code between two function declarations). Wang's cognitive complexity metric [61] measures the complexity of a system from its units: if units are linearly composed, their complexities are added up; when units are composed as layers, their complexities are multiplied together. There are also other

studies, e.g., [30, 33, 39, 50, 55, 63, 69] on the complexity metrics in the literature.

There have been some experimental investigations [79] on comparing various metrics. Not surprisingly, some metrics work well on some software systems, but not all. Reference [44] also points out that some of the aforementioned metrics have inherent problems. Indeed, as been widely accepted, there is simply no perfect and ideal metric for software complexity. We have noticed that almost all of the existing software complexity metrics are *syntax-dependent*. In other words, the metrics are measured on the syntactic appearances of the software system instead of its semantics (i.e., meanings). For instance, although two control graphs with the same topology, but with different initial nodes, might have dramatically different behaviors, the semantic difference is not reflected in the McCabe metric. Halstead's metric and the spatial complexity can change when some dummy code is inserted into the program being measured. Reference [33] proposes a complexity metric that is entropy-based; however, it measures the entropy of a random variable whose sample space is the set of operators in the source code, with the frequency of each operator in the source code being its probability. Therefore, we consider it as a variant of Halstead's metric, and the entropy would change if the syntax of the system changes.

In summary, when one intends to measure the complexity of the semantics (instead of the syntactic appearance) of a software system, to our best knowledge, there is no existing syntax-independent metric studied in the literature. A goal of our research work is to propose such a metric, with the following two fundamental questions in mind:

- what is a mathematical interpretation of the complexity of the behaviors (i.e., semantics) of a software system? The traditional Turing computability theory has already answered the question. However, the answer does not lead to a useful complexity metric. This is because computing models for nontrivial software systems can easily be Turing-complete (e.g., programs with two integer variables are already Turing-complete (see Minsky machines[47])), and therefore, there is not much difference between the Turing computing powers of the

models. In this chapter, we seek an answer in Shannon's information theory, which is a well-established mathematical theory underpinning all modern digital communications. This is a natural choice since Shannon's entropy describes the amount of information in the meaning (instead of the appearance) of an object. This is because the entropy of a random variable remains unchanged after a one-to-one function is applied [19].

- what makes the behaviors (i.e., semantics) of a software system complex? Again, traditional automata theory has answered the question (e.g., a queue is much more powerful than a stack, etc.). However, the answer does not lead to a practical metric. In this chapter, we are particularly interested in answering questions like the following. Suppose that a software system is composed (such as using sequential composition) from a number of individual components. What is the relationship between the complexity of the component-based system and the complexities of its constituent components? Our intuition tells us that a component-based system could be strictly more complex than the components that are used in building the component-based system. The intuition is also consistent with numerous common facts in science: relatively simple cells can build a functionally more complex organ; individual ants can form a functionally more complex colony; etc. Turning back to software engineering, studies in model-checking [18] already confirm the intuition through the well-known state-explosion problem. However, the confirmation is syntax-dependent: the state number is only an indirect indicator of the complexity of the meaning of a finite-state transition diagram. In this chapter, we will seek a confirmation using Shannon's information theory.

Our work introduces a novel software complexity metric that is semantics-based, independent of the syntactic appearance of the software system. Basically, we measure the complexity of the behaviors of the software system. Our approach is outlined as follows. Inspired by the McCabe metric, we also model the software system as a labeled graph. Each edge in the graph is labeled with a symbol, where the symbol can be interpreted as a code statement (when, e.g., the graph is

a control flow graph) or an I/O event (when, e.g., the graph is a design specification of a reactive system). We further define the *behavior set* of the labeled graph as the set of label sequences collected on all paths (a path may contain loops) in the graph. Instead of measuring the complexity of the graph based on its syntactic appearance, we measure the complexity of the behavior set. In other words, we are measuring the behavioral complexity, since the behavior set can be interpreted as the set of behaviors of the software system. Here, we count the number of behaviors $N(n)$ of a given length n in the behavior set, and define the *behavioral complexity* as

$$\lim_{n\to\infty} \frac{\log N(n)}{n}.$$

Note that throughout this chapter, the base of the logarithm is 2. We use the behavioral complexity as the complexity of the labeled graph as well as the software system that the labeled graph specifies. The behavioral complexity is syntax-independent, since by definition, for two graphs that share the same behavior set, the complexity is the same. The mathematical foundation of this metric will be explained in Section 3.3.

Our complexity metric measures a system from the perspective of software testing (so now, the labeled graph serves as the system's specification): it intends to asymptotically measure the cost of exhaustive testing of the system [7, 4]. Though exhaustive testing usually is not possible, the asymptotical cost is naturally a good indicator of the complexity of software systems. (Hence, our complexity does not measure the structural complexity (such as the McCabe metric) of software systems; instead, it measures the behavioral complexity as it is named. We shall point out that a structurally complex system may have a low behavioral complexity and vice versa. Indeed, we have not found direct relations between the two complexities.)

McCabe also states that the total number of paths in a graph is a good indicator of complexity; however, McCabe said that counting the total number of paths was impractical [45]. We successfully and efficiently calculate the number of paths asymptotically by using the eigen decomposition

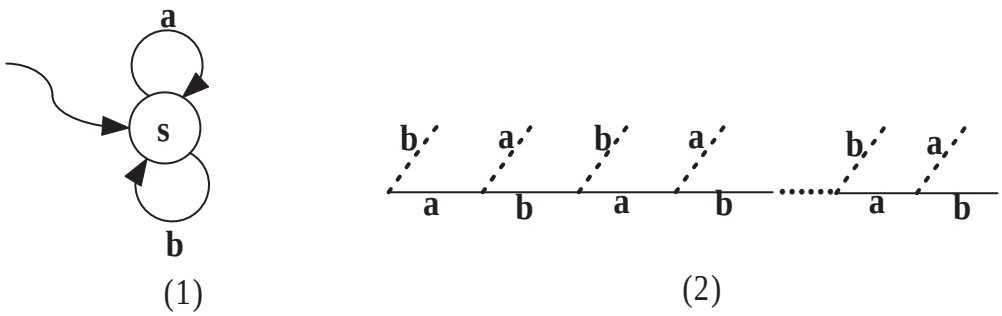


Figure 3.1: A labeled graph and a path on it.

of matrices. More importantly, we also show why the behavioral complexity of a labeled graph specifying a software system is a good indicator of software complexity: the behavioral complexity is exactly the maximal entropy rate of the Markov chain induced by the labeled graph. The entropy rate, widely used in information theory, is the number of bits that one needs to encode each sample in a stochastic process; the higher the rate is, the more resource needed to describe the process, and therefore, the behavioral complexity is a suitable metric of software complexity. For instance, Figure 3.1 (1) illustrates a labeled graph A whose behavior set is $L(A) = (a + b)^*$. Our results show that the behavioral complexity of the graph is 1 bit. A behavior $\omega = abab \cdots ab \in L(A)$ (which is also a path in A) is shown in Figure 3.1 (2). When A is generating ω, at each step, it has 2 choices: a and b. We need $\log 2 = 1$ bit (e.g, we use 0 and 1 to represent a and b, respectively) in order to encode the choice at each step. Notice that 1 bit is fairly large for entropy rate, considering that the alphabet is of size 2. The labeled graph A in Figure 3.1 seems to be simple from its appearance. However, our result shows that the semantics of A is relatively complex. Suppose that A is the design for a software system. The actual behaviors of the system may not conform with the behaviors specified in A. Hence, if we test the behaviors of the system, we have to test the paths (i.e., sequences of edges) in A, and the number of paths can exponentially blow up when the length of paths grows.

We also study how the complexity changes when individual components (units) are composed

together. We prove that the complexity can increase in the two following situations:

1. units are composed through loops. This reveals the fact that loops can make a graph strictly more complex;

2. units are composed through synchronization with zero pre-test knowledge (see section 3.4.2). This reveals the fact that (nontrivial) concurrency can also make a graph strictly more complex. [1]

The conclusion is quite meaningful for software designers when they try to predict the complexity of software systems built from a number of existing components. Finally, for software testers, we also point out a fundamental reason why integration testing is necessary even if unit testing is already done.

The main part of this chapter is summarized in paper [72].

3.2 Preparations

3.2.1 Labeled Graphs

Let

$$A = \langle S, s_{init}, \Sigma, E \rangle$$

be a *labeled graph*, where S is the set of nodes with s_{init} being the initial node, $\Sigma = \{a_1, \cdots, a_k\}$ is the set of symbols, and $E \subseteq S \times \Sigma \times S$ is the set of labeled edges, where $(s, a, s') \in E$ means there is an edge from node s to s', labeled with a. We further require that two edges originating from the same node cannot be labeled with the same symbol. A behavior $\omega = a_1 \cdots a_i \in \Sigma^*$ of A is a concatenation of labels on a path in A, i.e., there is a sequence of nodes $s_0 s_1 \cdots s_i$, such that $s_0 = s_{init}$, and $(s_{j-1}, a_j, s_j) \in E$ for $1 \leq j \leq i$. The behavior set $L(A)$ is the set of behaviors of

[1] Notice that the complexity's increment is not necessarily happening for all such loops and concurrency; the increment depends on the actual behavioral set of the composed graph.

A. Note that a labeled graph here is actually a deterministic finite automaton (where every node is an accepting state).

For notational convenience, given a labeled graph $A = \langle S, s_{init}, \Sigma, E \rangle$, we can uniquely convert it to an unlabeled graph

$$\hat{A} = \langle \hat{S}, \hat{s}_{init}, \hat{E} \rangle$$

(by building labels in A into nodes in $\hat{A}$), where $\hat{S}$ and $\hat{E}$ are the set of nodes and edges, respectively, and $\hat{s}_{init} \in \hat{S}$ is the initial node in $\hat{A}$ that every (node) path in $\hat{A}$ begins with. A and $\hat{A}$ are equivalent in the sense of path traversal, and there is a one-to-one mapping between a traversal of edges in A and a traversal of nodes in $\hat{A}$. A simple procedure that converts a labeled to an unlabeled graph is shown as follows. Note that throughout this chapter, we use $s \rightarrow s'$ to denote an edge from the node s to s' in an (unlabeled) graph.

convert(A):
//$A = \langle S, s_{init}, \Sigma, E \rangle$ is a labeled graph
1. Build a graph $\hat{A}$ with the set of nodes $\hat{S} = (S \times \Sigma) \cup \{(s_{init}, \Lambda)\}$ where $\Lambda \notin \Sigma$;
2. For each edge (s, a, s') in E
3. For each $a_i \in \Sigma$
4. Add an edge $(s, a_i) \rightarrow (s', a)$ to $\hat{A}$;
5. For each edge $(s_{init}, a, s) \in E$ for some $s \in S$
6. Add an edge $(s_{init}, \Lambda) \rightarrow (s, a)$ to $\hat{A}$;
7. Designate the node (s_{init}, Λ) as the initial node $\hat{s}_{init}$;
8. Return $\hat{A}$.

A path of $\hat{A}$ is a sequence of nodes $x_1 x_2 \cdots x_n$ for some $n > 0$, such that $x_1 = (s_{init}, \Lambda)$ ($\Lambda \notin \Sigma$)

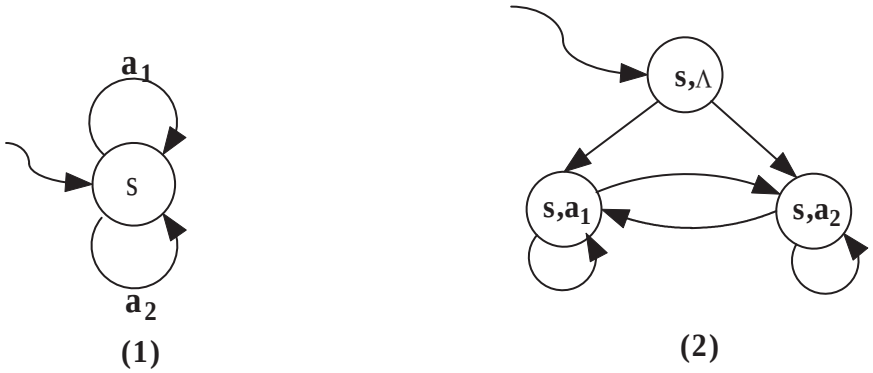


Figure 3.2: A labeled graph A and the graph $\hat{A}$ converted from A.

and $x_i \to x_{i+1}$ is an edge in $\hat{A}$ for $i = 1, 2, \cdots, n-1$. Given a path of $\hat{A}$, $u = x_1 x_2 \cdots x_n$, suppose that $x_i = (s_i, a_i)$, $s_i \in S$ and $a_i \in \Sigma$ for $i = 2, \cdots, n$, the projection of u on Σ is defined as a sequence $u_{\downarrow\Sigma}$ that only keeps symbols in Σ from u, while preserving their relative orderings (in this case, $u_{\downarrow\Sigma} = a_2 \cdots a_n$). We can easily check that for every path u in $\hat{A}$, there is a behavior $\omega \in L(A)$, such that $u_{\downarrow\Sigma} = \omega$, and vice versa.

Example 14. Figure 3.2 (1) gives a labeled graph $A = \langle S, s_{init}, \Sigma, E \rangle$, where $S = \{s\}$, $s_{init} = s$, $\Sigma = \{a_1, a_2\}$, and the set of edges is $E = \{(s, a_1, s), (s, a_2, s)\}$. The behavior set of A is $L(A) = (a_1 + a_2)^*$, and the corresponding graph $\hat{A}$ is shown in Figure 3.2 (2). $\hat{A}$ is built in the following way. The set of nodes is $\hat{S} = \{(s, a_1), (s, a_1), (s, \Lambda)\}$, where $\Lambda \notin \Sigma$. For the edge $(s, a_1, a) \in E$, we have edges $(s, a_1) \to (s, a_1)$ and $(s, a_2) \to (s, a_1)$ in $\hat{A}$; for the edge (s, a_2, a), we have edges $(s, a_1) \to (s, a_2)$ and $(s, a_2) \to (s, a_2)$ in $\hat{A}$; since s is the initial node of A, we also have $(s, \Lambda) \to (s, a_1)$ and $(s, \Lambda) \to (s, a_2)$. The initial node of $\hat{A}$ is (s, Λ). From Figure 3.2, we can see clearly that each traversal of edges in A corresponds to a traversal of nodes in $\hat{A}$, and vice versa. □

We now define the *branching factor* of a graph A with the set of nodes S. For each node $s \in S$, the branching factor $\rho(s)$ of s is the number of edges with s being the source, and the branching

factor $\rho(A)$ of the graph is defined as the maximal branching factor of the nodes, i.e.,

$$\rho(A) = \max_{s \in S} \rho(s).$$

3.2.2 Markov Chains

A *Markov chain* is a random process (i.e., a sequence of random variables) where each random variable only depends on the one that immediately precedes it [19]. Formally, let $\mathcal{X} = \{x_1, \cdots, x_m\}$ be a set of states. A Markov chain (or a Markov process) M is a discrete stochastic process $X_1, X_2, \cdots, X_n, \cdots$, where X_n is the state at time n, and we have conditional probability satisfying

$$\begin{aligned} & Pr(X_n = x_n | X_{n-1} = x_{n-1}, \cdots, X_1 = x_1) \\ & = Pr(X_n = x_n | X_{n-1} = x_{n-1}), \end{aligned}$$

for all $x_n, \cdots, x_1 \in \mathcal{X}$. Together with the *initial distribution* $Pr(X_1 = x)$, for all $x \in \mathcal{X}$, we can calculate the probability of a particular sequence $\pi = x_1 x_2 \cdots x_n \in \mathcal{X}^n$ for some $n > 1$,

$$\begin{aligned} Pr(\pi) \quad = \quad & Pr(X_1 = x_1) Pr(X_2 = x_2 | X_1 = x_1) \cdots \\ & Pr(X_n = x_n | X_{n-1} = x_{n-1}). \end{aligned}$$

A *valid* sequence is a sequence π with $Pr(\pi) > 0$. The set of valid sequences that are generated by M is defined as $S(M)$. Unless otherwise stated, all the sequences we mention in this chapter are valid. We can build a matrix $W = [W_{ij}]$, $1 \leq i, j \leq m$, called the *probability transition matrix*, where each entry W_{ij} denotes the *transition probability* $Pr(X_{n+1} = x_j | X_n = x_i)$ from state x_i to x_j for all $n \geq 1$ (hence, Markov chains in our work are time-invariant). A Markov chain M can be represented as a graph (also denoted by M), where each node represents a state (we also use

x_i to denote the node that represents the state x_i). For any two states x_i and x_j, if $W_{ij} > 0$, we have a directed edge from node x_i to node x_j (denoted by $x_i \rightarrow x_j$) in M, labeled with weight $w(x_i \rightarrow x_j) = W_{ij}$; otherwise there is no edge from x_i to x_j. Clearly, a sequence now is simply a path in the graph-represented Markov chain, and its probability is the product of the weights of the edges on the path.

In information theory, *entropy rate* is used to depict the growth rate of the entropy of sequences generated by a Markov chain. Formally, the entropy rate of a stochastic process $\{X_i\}$, where each sample in the process is drawn from the sample space $\mathcal{X}$, is defined as

$$H(\mathcal{X}) = \lim_{n\rightarrow\infty} \frac{H(X_1, X_2, \cdots X_n)}{n},$$

where $H(X_1, X_2, \cdots, X_n)$ is the joint entropy[2] of random variables $X_1, X_2, \cdots X_n$. In traditional information theory, usually the entropy rate is studied only if the above limit exists (e.g., when the stochastic process is stationary [19]. However, in general, the limit may not exist.). Here, when the above limit does not exist, we treat the entropy rate as the *upper limit*, i.e.,

$$H(\mathcal{X}) = \overline{\lim_{n\rightarrow\infty}} \frac{H(X_1, X_2, \cdots X_n)}{n}. \tag{3.1}$$

The upper limit always exists (since the size of the sample space of each X_i is finite), and indicates, asymptotically, a sufficient number of bits needed to encode a sample in the process.

3.2.3 Matrices

We first number the diagonals of an $m \times m$ matrix $K = [K_{i,j}]$. The k^{th} $(k = 1, \cdots, m)$ diagonal of K consisting of elements $K_{i,i+k-1}$, for $i = 1, \cdots, m-k+1$. We say K is a k-diagonal matrix if its k^{th} diagonal is filled with 1's, while other elements are all 0's. For instance, the following K

[2]The joint entropy of random variables $X_1, \cdots, X_n$ with joint distribution $Pr(X_1 = x_1, \cdots, X_n = x_n)$, $x_1 \cdots x_n \in \mathcal{X}^n$ is defined by Shannon as $\sum_{x_1 \cdots x_n \in \mathcal{X}^n} -Pr(X_1 = x_1, \cdots, X_n = x_n) \log Pr(X_1 = x_1, \cdots, X_n = x_n)$.

is a 3-diagonal matrix:

$$K = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

A matrix B of size $m \times m$ can be written into the following form [10]

$$B = VDV^{-1}$$

using Jordan decomposition, where V and D are $m \times m$ matrixes. Each column in V is an eigenvector or a generalized eigenvector, and D is a block diagonal matrix in the form of

$$D = \begin{bmatrix} J_1 & 0 & \cdots & \cdots \\ 0 & J_2 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix},$$

where each J_i is a Jordan block associated with a corresponding eigenvector V_i , which is a matrix composed of entries 0, except that its 1^{st} diagonal is filled with a fixed value (here it is the eigenvalue λ_i corresponding to V_i), and its superdiagonal (i.e., the 2^{nd} diagonal) is filled with 1's. There might be complex numbers involved in eigenvectors and eigenvalues. The norm of a complex number $z = a + bi$ is defined as $||z|| = \sqrt{a^2 + b^2}$, and the norm of a real number z is $||z|| = |z|$.

3.3 Efficient Computation of Behavioral Complexity

From a given graph $\hat{A}$ that is converted from a labeled graph A, we can calculate the number of paths of length n in $\hat{A}$. Suppose that there are m nodes $x_1, \cdots, x_m$ in $\hat{A}$. Let $Z_i(n)$ be the number

of paths of length n ending at the node x_i, and

$$Z(n) = [Z_1(n)\ Z_2(n)\ \cdots\ Z_m(n)]^T$$

be an m-dimensional vector. Let $B = [B_{ij}]$ be an $m \times m$ matrix, called the *counting matrix* of $\hat{A}$, defined in the following way: if there is an edge $x_j \to x_i$ in $\hat{A}$, then $B_{ij} = 1$, otherwise $B_{ij} = 0$. Let $\lambda_1, \cdots \lambda_m$ be the m (not necessarily distinct) eigenvalues of B. We have the following lemma:

Lemma 4. *$Z(n)$ can be written into the following form:*

$$Z(n) = g_1(n-1)\lambda_1^{n-1}Y_1 + \cdots + g_m(n-1)\lambda_m^{n-1}Y_m, \tag{3.2}$$

where Y_i's are m-dimensional vectors with all their elements being constant (not depending on n), and $g_i(n)$'s are polynomial functions on n and $g_i(n) > 0$ for almost all n.

Proof. Without loss of generality, we assume that x_1 is the initial node in $\hat{A}$. Since all paths begin with x_1, the number of paths of length 1 is 1, i.e., the path $u = x_1$. Therefore, $Z(1) = [1\ 0\ \cdots 0]^T$. Observe that $Z_i(n) = \sum_{x_j \to x_i} Z_j(n-1)$. From the above observation, we have $Z(n) = BZ(n-1)$, and therefore $Z(n) = B^{n-1}Z(1)$, with $Z(1)$ being the initial condition. Let $\lambda_1, \cdots, \lambda_m$ be the (not necessarily distinct) m eigenvalues of B. If an eigenvalue is repeated k times, the number of eigenvectors associated with this eigenvalue may be $j = 1, \cdots, k$. If $j \neq k$, we say this eigenvalue is *defective*. Without loss of generality, we assume that the eigenvalue λ_1 is repeated more than once. The following is a standard technique in Matrix Theory [56]. Now we are to solve the

following equations:

$$\begin{aligned} BV_1 &= \lambda_1 V_1 \\ BV_{11} &= \lambda_1 V_{11} + V_1 \\ BV_{12} &= \lambda_1 V_{11} + V_{11} \\ \cdots & \end{aligned}$$

until the above equations cannot be continued (i.e. $BV_{1i} = BV_{1(i-1)} + V_{1(i-1)}$ does not have a solution for V_{1i}). In these equations, V_1 is an eigenvector associated with λ_1, while V_{1i} $(i = 1, 2 \cdots)$ are generalized eigenvectors of V_1. Notice that if λ_1 is not defective, it only has eigenvector(s), and no generalized ones. Here, for our notational convenience, we simply assume that V_1 has two generalized eigenvectors, V_{11} and V_{12}. We have,

$$\begin{aligned} B\left[\begin{array}{ccc:c} V_1 & V_{11} & V_{12} & V_2 \cdots \end{array}\right] &= [\lambda_1 V_1 \quad \lambda_1 V_{11} + V_1 \quad \lambda_1 V_{12} + V_{11} \;\vdots\; \lambda_2 V_2 \quad \cdots] \\ &= \left[\begin{array}{ccc:c} V_1 & V_{11} & V_{12} & V_2 \cdots \end{array}\right] \cdot \left[\begin{array}{ccc:ccc} \multicolumn{3}{c:}{\multirow{3}{*}{$\begin{bmatrix} \lambda_1 & 1 & 0 \\ 0 & \lambda_1 & 1 \\ 0 & 0 & \lambda_1 \end{bmatrix}$}} & & & \\ \multicolumn{3}{c:}{} & 0 & \cdots & \cdots \\ \multicolumn{3}{c:}{} & & & \\ \hdashline & 0 & & \lambda_2 & 0 & \cdots \\ \hdashline & \vdots & & \vdots & \ddots & \cdots \end{array}\right], \end{aligned}$$

where V_2 is an eigenvector associated with λ_2. Thus, the matrix B can be written as $B = VDV^{-1}$, where V is an $m \times m$ matrix and each column is an eigenvector or generalized eigenvector, and D

is a block diagonal matrix in the form of

$$D = \begin{bmatrix} J_1 & 0 & \cdots & \cdots \\ 0 & J_2 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix},$$

where each J_i is a Jordan block associated with an eigenvalue λ_i , which is a matrix composed of zero-valued elements, except that its diagonal is filled with a fixed value (here it is the eigenvalue λ_i), and its superdiagonal is filled with 1's. Let V_i be an eigenvector of λ_i, and $V_{i1_i}, \cdots, V_{il_i}$ be the l_i generalized eigenvectors of V_i. The size of J_i is $(1 + l_i) \times (1 + l_i)$. For instance, a Jordan block of the aforementioned λ_1 is in the form of

$$\begin{bmatrix} \lambda_1 & 1 & 0 \\ 0 & \lambda_1 & 1 \\ 0 & 0 & \lambda_1 \end{bmatrix}.$$

Notice that there are j Jordan blocks associated with an eigenvalue that has j distinct eigenvectors.

Since

$$D^n = \begin{bmatrix} J_1^n & 0 & \cdots & \cdots \\ 0 & J_2^n & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix},$$

we have

$$\begin{aligned} Z(n) &= B^{n-1}Z(1) \\ &= (VDV^{-1})(VDV^{-1})\cdots(VDV^{-1})Z(1) \\ &= VD^{n-1}V^{-1}Z(1) \\ &= V\begin{bmatrix} J_1^{n-1} & 0 & \cdots & \cdots \\ 0 & J_2^{n-1} & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix} V^{-1}Z(1). \end{aligned}$$

For a Jordan block J, we have

$$J^k = \begin{bmatrix} \lambda & 1 & 0 & \cdots \\ 0 & \lambda & 1 & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda \end{bmatrix}^k = \begin{bmatrix} \lambda^k & k\lambda^{k-1} & \frac{k(k-1)}{2}\lambda^{k-2} & \cdots \\ 0 & \lambda^k & k\lambda^{k-1} & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda^k \end{bmatrix},$$

and therefore

$$
\begin{aligned}
Z(n) &= \lambda_1^{n-1} V \begin{bmatrix} 1 & 0 & 0 & \cdots \\ 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} V^{-1} Z(1) + (n-1)\lambda_1^{n-2} V \begin{bmatrix} 0 & 1 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} V^{-1} Z(1) \\
&\quad + \cdots + \begin{pmatrix} n-1 \\ l_1 \end{pmatrix} \lambda_1^{n-l_1} V \begin{bmatrix} 0 & 0 & \cdots 1 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} V^{-1} Z(1) \\
&\quad + \lambda_2^{n-1} V \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & 1 & \vdots \\ 0 & 0 & 0 & 0 \end{bmatrix} V^{-1} Z(1) + \cdots, \\
&= \lambda_1^{n-1} Y_1 + (n-1)\lambda_1^{n-1} Y_2 + \cdots + \begin{pmatrix} n-1 \\ l_1 \end{pmatrix} \lambda_1^{n-1} Y_{l_1} + \lambda_2^{n-1} Y_{l_1+1} + \cdots, \qquad (3.3)
\end{aligned}
$$

where $(l_1 + 1) \geq 1$ is the dimension of the Jordan block J_1, and does not depend on n; Y_i's are m-dimensional vectors and also do not depend on n. For instance, Y_2 in (3.3) is defined as $Y_2 = \frac{1}{\lambda_1} V K V^{-1} Z(1)$, where K is the matrix obtained after we replace J_1 with a 2-diagonal matrix in D, and all other elements in D are set to zero. Clearly, (3.3) is exactly in the form of (3.2). □

Let $N(n)$ be the number of behaviors of length n in the behavior set $L(A)$ of the labeled graph A.

As mentioned earlier, we define the *behavioral complexity* of A as

$$\mathcal{C}(A) = \lim_{n\to\infty} \frac{\log N(n)}{n}.$$

Notice that the $\mathcal{C}(A)$ is essentially defined on $L(A)$; hence it is syntax-independent (i.e., as we mentioned earlier, $\mathcal{C}(A) = \mathcal{C}(A')$ whenever $L(A) = L(A')$.). In (3.2), let λ_{max} be the eigenvalue λ_i with the largest norm and the elements in Y_i are not all 0's, called the *counting eigenvalue* of the counting matrix B. Formally,

$$||\lambda_{max}|| = \max\{||\lambda_i|| : Y_i \neq \mathbf{0}\},$$

where $\mathbf{0}$ is a vector with its elements all 0. We have,

Theorem 5. *Given a labeled graph A, let B be the counting matrix of $\hat{A}$. The behavioral complexity of A is*

$$\mathcal{C}(A) = \lim_{n\to\infty} \frac{\log N(n)}{n} = \log ||\lambda_{max}||,$$

where λ_{max} is the counting eigenvalue of B.

Proof. Let $Z = [z_1 \cdots z_m]$ be an m-dimensional vector. We define $\overline{Z} = \sum_{1\leq i\leq m} z_i$ as the sum of the elements in Z, and let $Z[j]$ be the j^{th} element in the vector Z. Note that a word of length n in $L(A)$ corresponds to a path of length $n+1$ in the graph $\hat{A}$. Therefore, $N(n) = \sum_i Z_i(n+1) = \overline{Z(n+1)}$. From (3.2), we have

$$\overline{Z(n+1)} = g_1(n)\lambda_1^n\overline{Y_1} + g_2(n)\lambda_2^n\overline{Y_2} + \cdots + g_m(n)\lambda_m^n\overline{Y_m}, \tag{3.4}$$

where $\overline{Y_i}$'s are constants. Without loss of generality, we assume that $\lambda_{max} = \lambda_1$ is the counting eigenvalue, and thus $Y_1 \neq \mathbf{0}$.

In the following, we have two cases to discuss: one case is that λ_1 is a real number, another

case is that λ_1 is a complex number.

Case 1: λ_1 is a real number.

Clearly, in this case, all elements in Y_1 are real numbers. We now prove that there is no negative element in Y_1. From the definition of the counting eigenvalue, we know that if there is some eigenvalue, say λ_i, such that $||\lambda_i|| > ||\lambda_1||$, then all the elements in Y_i should be zero. Therefore, $g_i(n)\lambda_i^n Y_i[j]$ is 0 for each $j = 1, \cdots, m$ and contributes nothing to $Z_j(n+1)$ (which is the number of paths of length $n+1$ ending at node x_j). Hence, if some element in Y_1, say $Y_1[j]$, is negative, then $Z_j(n)$ is dominated by the term $g_1(n)\lambda_1^n Y_1[j] < 0$ (when n is even), which is a contradiction to the fact that the number of paths is always a nonnegative number. Therefore, Y_1 does not have negative elements, and $\overline{Y_1} > 0$. Similarly, λ_1 should also be positive. Then the dominant term in (3.4) is $g_1(n)\lambda_1^n\overline{Y_1}$. We have,

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log N(n)}{n} &= \lim_{n\to\infty} \frac{\log \overline{Z(n+1)}}{n} \\
&= \lim_{n\to\infty} \frac{\log g_1(n)\lambda_1^n \overline{Y_1}}{n} \\
&= \lim_{n\to\infty} \frac{\log g_1(n) + \log \lambda_1^n + \log \overline{Y_1}}{n} \\
&= \log \lambda_1.
\end{aligned}$$

Case 2: λ_1 is a complex number.

Without loss of generality, we assume $\lambda_1 = a + bi$. There must be a conjugate of λ_1 which is also an eigenvalue of B, say, $\lambda_2 = a - bi$, and $g_1(n)$ and $g_2(n)$ are exactly the same functions (we simply say $g_1(n) = g_2(n) = g(n)$). Notice that λ_2 is also a counting eigenvalue of B, since $||\lambda_1|| = ||\lambda_2||$. Additionally, if there is a complex number in Y_1, say $Y_1[j] = c + di$, then $Y_2[j]$ must be a conjugate complex number of $Y_1[j]$, i.e., $Y_2[j] = c - di$ for any j. We can write the complex numbers in their polar forms. Let $\delta = \sqrt{a^2 + b^2}$ and $\tau = \sqrt{c^2 + d^2}$ be the norms of λ_1 and $Y_1[j]$, respectively. Then $\lambda_1 = \delta(\cos\theta + i\sin\theta)$ and $Y_1[j] = \tau(\cos\beta + i\sin\beta)$, where θ

and β are the phases of λ_1 and $Y_1[j]$, respectively. Since λ_2 (resp. $Y_2[j]$) is the conjugate of λ_1 (resp. $Y_1[j]$), we have $\lambda_2 = \delta(\cos\theta - i\sin\theta)$ (resp. $Y_1[j] = \tau(\cos\beta - i\sin\beta)$). Consider the term $t_j = g_1(n)\lambda_1^n Y_1[j] + g_2(n)\lambda_2^n Y_2[j]$. It is not hard to verify that

$$\begin{aligned} t_j &= g(n)\lambda_1^n Y_1[j] + g(n)\lambda_1^n Y_1[j] \\ &= g(n)\delta^n\tau(\cos n\theta + i\sin n\theta)(\cos\beta + i\sin\beta) \\ &\quad + g(n)\delta^n\tau(\cos n\theta + i\sin n\theta)(\cos\beta + i\sin\beta) \\ &= 2g(n)\delta^n\tau\cos(n\theta+\beta). \end{aligned}$$

As in *Case 1*, we can easily show that the coefficient $2g(n)\tau\cos(n\theta+\beta)$ of δ^n is not negative, since otherwise $t_j < 0$ and it would dominate $Z_j(n+1)$, which is a contradiction to the fact that $Z_j(n+1) \geq 0$. Therefore,

$$\begin{aligned} \lim_{n\to\infty}\frac{\log N(n)}{n} &= \lim_{n\to\infty}\frac{\log\overline{Z(n+1)}}{n} \\ &= \lim_{n\to\infty}\frac{\log(t_1+\cdots+t_m)}{n} \\ &= \lim_{n\to\infty}\frac{\log(Cg(n)\delta^n)}{n} \\ &= \lim_{n\to\infty}\frac{\log g(n)+\log\delta^n+\log C}{n} \\ &= \log\delta, \end{aligned}$$

where C is constant, and δ is the norm of λ_1.

This completes our proof. □

A similar result of Theorem 5 can actually be found in early work [17] in automata theory, which is the source of the well-known fact that the count of words with length n in a regular language grows either polynomially or exponentially. In here, we provide a more detailed proof than the one in [17] by using Jordan decomposition to provide an analytical form of the Y_i's in

(3.2) which are needed in the algorithm below.

Theorem 5 points out that computing the behavioral complexity of a labeled graph A is essentially calculating the counting eigenvector of the counting matrix of $\hat{A}$. How to compute the behavioral complexity of A is presented in the following algorithm:

complexity(A):

//To compute the behavioral complexity of a labeled graph A

1. $||\lambda_{max}|| := 0$;
2. Build a graph $\hat{A}$ = convert(A), and let m be the number of nodes in $\hat{A}$;
3. Build the counting matrix B of $\hat{A}$;
4. Let $Z(1) = [1\ 0\ \cdots\ 0]$ be an m-dimensional vector;
5. Find all the eigenvalues $\lambda_1, \cdots, \lambda_m$ of A;
6. Find all the k distinct eigenvectors $V_1, \cdots, V_k$ $(k \leq m)$;
7. For $i := 1$ to k
8. Find the $l_i \geq 0$ generalized eigenvectors of V_i;
9. Build the Jordan block J_i of size $(l_i + 1) \times (l_i + 1)$ for each V_i;
10. Do Jordan Decomposition of $B = VDV^{-1}$;

//the form of D is given in Section 3.2.3;

11. For $i := 1$ to k
12. For $j := 1$ to $(l_1 + 1)$
13. Replace the Jordan block J_i with the j-diagonal matrix in D, set all other elements in D to be 0, and name theobtained matrix K;

14. $h := \sum_{1 \le z \le i-1} (1 + l_z) + j$;

15. $Y_h := \frac{1}{\lambda_i^{j-1}} V K V^{-1} Z(1)$;

16. If $Y_h \neq \mathbf{0}$

17. If $||\lambda_h|| > ||\lambda_{max}||$

//λ_h is the eigenvalue corresponding to V_i

18. $||\lambda_{max}|| := ||\lambda_h||$;

19. Return $\log ||\lambda_{max}||$.

Since Jordan decomposition of a matrix can be done efficiently (both in theory [14] using existing numerical algorithms and in practice using some numerical computing tools, e.g., Mathematica [2] and Matlab [3]), the above algorithm complexity(A) is also efficient.

Next we discuss the information theory underlying the definition of the behavioral complexity. Notice that the graph $\hat{A}$ can be interpreted as a graph-represented Markov chain $M_{\hat{A}} = \{X_i\}$ $(i = 1, 2, \cdots)$. Suppose that the set of states of $M_{\hat{A}}$ is $\mathcal{X} = \{x_1, \cdots, x_m\}$, which is actually the set of nodes in $\hat{A}$. Without loss of generality, we assume that x_1 is the initial node in $\hat{A}$, and then we specify that the initial distribution of $M_{\hat{A}}$ is $Pr(X_1 = x_1) = 1$ and $Pr(X_1 = x) = 0$ for other $x \in \mathcal{X}$; i.e., x_1 serves as the *initial state* of $M_{\hat{A}}$. The weights on edges $w(\cdot)$ in $M_{\hat{A}}$ (i.e., transition probabilities) are unspecified. Since we designate the initial state, the probability of a sequence $\pi = x_1 x_2 \cdots x_n \in \mathcal{X}^n$ now is

$$Pr(\pi) = Pr(X_2 = x_2 | X_1 = x_1) \cdots Pr(X_n = x_n | X_{n-1} = x_{n-1}),$$

which only depends on the transition probabilities. Therefore, the entropy rate $H(\mathcal{X})$ in (3.1) is also a function only depending on $w(\cdot)$. We define $H(A)$ as the maximal achievable entropy rate, i.e.,

$$H(A) = \max_{w(\cdot)} H(\mathcal{X}) = \max_{w(\cdot)} \overline{\lim_{n \to \infty}} \frac{H(X_1, X_2, \cdots, X_n)}{n}.$$

The following theorem states that there is a Markov chain $X_1, \cdots, X_n, \cdots$ that achieves the rate.

Theorem 6. *The behavioral complexity of a labeled graph A is the maximal achievable entropy rate of the Markov chain $M_{\hat{A}}$. That is,*

$$\mathcal{C}(A) = H(A).$$

Proof. Our proof is hinted at by the solution of Exercise 4.16 in [19]. We first briefly introduce the concept *first order type* [20]. Let $\mathcal{X} = \{x_1, \cdots, x_m\}$ be a set of states. The type P_π of a sequence $\pi \in \mathcal{X}^n$ is an m-dimensional vector, and the i^{th} element in P_π is defined as $\frac{\#_\pi x_i}{n}$, where $\#_\pi x_i$ is the number of times that x_i occurs in π. Clearly, different sequences can share the same type. Let P_n be the set of types with denominator n. From [19], we know that the size of P_n, $|P_n| \leq (n+1)^m$, which is a polynomial function of n. For each type $P \in P_n$, we define the *type class* $T(P) = \{\pi \in \mathcal{X}^n : P_\pi = P\}$, which is a set of sequences that share the same type P. Clearly, all the type classes form a partition of all the possible sequences drawn from $\mathcal{X}^n$. Additionally, a type $P \in P_n$ naturally defines a probability distribution on $\mathcal{X}$, since the summation of all elements in P is 1.

A second order type [20] concerns the frequencies of pairs of symbols (instead of single symbols in the first order type) in a sequence generated by a Markov chain. Formally, the second order type $P_\pi^{(2)} = [P_\pi^{(2)}(i,j)]$ $(1 \leq i, j \leq m)$ of a sequence $\pi = a_1 a_2 \cdots a_n$ generated by a Markov chain $M = \{X_i\}$ with the set of states $\mathcal{X} = \{x_1, \cdots, x_m\}$ is an $m \times m$ matrix, where each entry in $P_\pi^{(2)}$ is defined as

$$P_\pi^{(2)}(i,j) = \frac{1}{n-1} |\{k : a_k = x_i, a_{k+1} = x_j\}|,$$

that is, the entry $P_\pi^{(2)}(i,j)$ records the frequency of the pair of two consecutive symbols $x_i x_j$ in π. Let $P_n^{(2)}$ be the set of second order types with denominator $(n-1)$ (which means that the sequences involved here are of length n) that satisfy state constraints of M (i.e., if there is no edge

$x_i \to x_j$, the entry $P_\pi^{(2)}(i,j)$ in the type $P_\pi^{(2)}$ should be 0). Similar to the case of the first order type, the size of $P_n^{(2)}$ is polynomial in n; without loss of generality, we say $|P_n^{(2)}| = g(n)$, where $g(n)$ is a polynomial function of n. For each second order type $F \in P_n^{(2)}$, we define the type class $T(F) = \{\pi \in \mathcal{X}^n : P_\pi^{(2)} = F\}$, and all the type classes form a partition of all the possible sequences drawn from $\mathcal{X}^n$. Each second order type $F \in P_n^{(2)}$ can be treated as a joint distribution over two random variables, and both variables are with the sample space $\mathcal{X}$; F also implicitly defines a probability transition matrix W, where

$$W_{ij} = \frac{F(i,j)}{\sum\limits_{1 \le k \le m} F(i,k)}. \tag{3.5}$$

It is known that sequences falling into the same type class and with the same initial state are of equiprobability. Since we are dealing with Markov chains that are converted from labeled graphs and sequences only have one possible initial state, we simply say that if sequences are in the same type class, they are of equiprobability.

Given a Markov chain $M = \{X_i\}$ with the set of states $\mathcal{X} = \{x_1, \cdots, x_m\}$, consider its generated sequences of length n. The stochastic process $X_1, \cdots, X_n$ can be treated as a joint random variable $(X_1, \cdots, X_n)$, and suppose that there are $R(n)$ instances in its sample space (i.e., there are $R(n)$ sequences of length n). It is well known [19] that the joint entropy

$$H(X_1, \cdots, X_n) \le \log R(n) \tag{3.6}$$

for any $n > 0$. Therefore, for the entropy rate $H(\mathcal{X})$ in (3.1), we have,

$$\begin{aligned} H(\mathcal{X}) &= \overline{\lim_{n\to\infty}} \frac{H(X_1, X_2, \cdots X_n)}{n} \\ &\le \overline{\lim_{n\to\infty}} \frac{\log R(n)}{n}, \end{aligned} \tag{3.7}$$

for any transition matrix W applied on M. On the other hand, we know that the number of second order types is $g(n)$, which is a polynomial function in n. Let $F_n^* \in P_n^{(2)}$ be the second order type that has the largest type class (i.e., $|T(F_n^*)| \geq |T(F_n)|$, for any second order type $F_n \in P_n^{(2)}$). Then, the size of the type class $T(F_n^*)$ should be no smaller than the average size of the types; that is, $|T(F_n^*)| \geq \frac{1}{g(n)} R(n)$.

Let F_n^1, F_n^2 be two second order types in $P_n^{(2)}$. Suppose that the Markov chain M has probability transition matrix W that is derived from F_n^1 using (3.5). We define $P_{F_n^1}(F_n^2)$ as the probability that a sequence generated by M of length n is in the type $T(F_n^2)$; that is,

$$P_{F_n^1}(F_n^2) = \sum_{\pi \in S(M),\ |\pi|=n,\ \pi \in T(F_n^2)} Pr(\pi).$$

(Recall that $S(M)$ is the set of valid sequences of the Markov chain M.)

We use W_n^* to denote the transition matrix deduced from F_n^*. Notice that the two sequences $\{F_n^*\}$ and $\{W_n^*\}$ are bounded. From the Bolzano-Weierstrass theorem, there is a convergent (converging pointwise) subsequence $\{F_{n_t}^*\}$ of $\{F_n^*\}$, and a convergent (converging pointwise) subsequence $\{W_{n_t}^*\}$ of $\{W_n^*\}$; let F_∞^* and W_∞^* be the limit of the subsequences $\{F_{n_t}^*\}$ and $\{W_{n_t}^*\}$, respectively. We use K to denote the set of n_t's. Note that F_n^* defines a joint distribution of two dummy random variables, say, X and Y. We use H_n^* to denote $H(Y|X)$. Similarly, we can define H_∞^* (from F_∞^*) and H_n (from F_n). Let $\epsilon > 0$. Clearly, for all $n \in K$ large enough, we have

$$|H_\infty^* - H_n^*| < \epsilon, \tag{3.8}$$

where H_∞^* and H_n^* are the conditional entropy deduced from the joint distribution F_∞^* and F_n^*, respectively. We further use the notation

$$D(P||W) = \sum_{x,y} P(x,y) \log \frac{P(y|x)}{W(y|x)}$$

introduced in [21], which denotes the Kullback-Leibler information divergence of a joint distribution P and a Markov transition matrix W. When $D(F_n||W_\infty^*) < \epsilon$, one can find a ϵ', such that

$$|H_n - H_\infty^*| < \epsilon' \tag{3.9}$$

for all n large enough. This $\epsilon' \to 0$ as $\epsilon \to 0$. Therefore, from (3.8) and (3.9), we can find a ϵ'', such that

$$|H_n - H_n^*| < \epsilon''$$

for all n large enough and satisfying $D(F_n||W_\infty^*) < \epsilon$. Notice that ϵ'' can be chosen such that $\epsilon'' \to 0$ as $\epsilon \to 0$. Therefore, using formula (VII.3) in [20], for large enough n which satisfies $D(F_n||W_\infty^*) < \epsilon$, we have

$$\big|\log|T(F_n)| - \log|T(F_n^*)|\big| < n\epsilon''. \tag{3.10}$$

We still need one more result before we continue with our proof. We use $F_\infty^{<\epsilon}$ to denote the set

$$\{F_n \in P_n^{(2)} : D(F_n||W_\infty^*) < \epsilon\}.$$

Similarly, we can also define $F_\infty^{\geq\epsilon} = \{F_n \in P_n^{(2)} : D(F_n||W_\infty^*) \geq \epsilon\}$. Using (VII.4) in [20] and Theorem 3.1 in [12], we have

$$\sum_{F_n \in F_\infty^{\geq\epsilon}} P_{F_\infty^*}(F_n) \to 0 \ \ as \ \ n \to \infty.$$

Therefore,

$$\sum_{F_n \in F_\infty^{<\epsilon}} P_{F_\infty^*}(F_n) \to 1 \ \ as \ \ n \to \infty. \tag{3.11}$$

Now, the entropy rate of $M = X_1, \cdots, X_n, \cdots$ with its transition matrix derived W_∞^* from

F^*_∞:

$$
\begin{aligned}
H(\mathcal{X}) &= \overline{\lim_{n\to\infty}} \frac{H(X_1, X_2, \cdots X_n)}{n} \qquad (3.12)\\
&\geq \overline{\lim_{n\to\infty}} \frac{\sum_{F_n \in F^{\leq\epsilon}_\infty} \sum_{\pi\in S(M), \pi\in T(F_n)} -Pr(\pi)\log Pr(\pi)}{n}\\
&= \overline{\lim_{n\to\infty}} \frac{\sum_{F_n \in F^{\leq\epsilon}_\infty} -P_{F^*_\infty}(F_n)\log\left(P_{F^*_\infty}(F_n)/|T(F_n)|\right)}{n}\\
&= \overline{\lim_{n\to\infty}} \frac{\sum_{F_n \in F^{\leq\epsilon}_\infty} (-P_{F^*_\infty}(F_n)\log P_{F^*_\infty}(F_n) + P_{F^*_\infty}(F_n)\log|T(F_n)|)}{n}\\
&\geq \overline{\lim_{n\to\infty}} \frac{\sum_{F_n \in F^{\leq\epsilon}_\infty} P_{F^*_\infty}(F_n)\log|T(F_n)|}{n}
\end{aligned}
$$

using(3.10),

$$
\begin{aligned}
&\geq \overline{\lim_{n\to\infty}} \frac{\sum_{F_n \in F^{\leq\epsilon}_\infty} P_{F^*_\infty}(F_n)(\log|T(F^*_n)| - n\epsilon'')}{n}
\end{aligned}
$$

using(3.11),

$$
\begin{aligned}
&\geq \overline{\lim_{n\to\infty}} \frac{\log|T(F^*_n)| - n\epsilon''}{n}.
\end{aligned}
$$

Recall that $|T(F^*_n)| \geq \frac{1}{g(n)}R(n)$, where $R(n)$ is the total number of sequences of length n, $g(n)$ is a polynomial function in n. (3.12) can be written as

$$
H(\mathcal{X}) \geq \overline{\lim_{n\to\infty}} \frac{\log(R(n)/g(n)) - n\epsilon''}{n} = \lim_{n\to\infty} \frac{\log R(n)}{n}, \qquad (3.13)
$$

by taking ϵ (and hence ϵ'') small. From (3.7) and (3.13), we known that the maximal entropy rate $\lim_{n\to\infty} \frac{\log R(n)}{n}$ is achievable, when the transition matrix of M is W^*_∞ derived from F^*_∞.

Given a labeled graph A, we can converted it to an (unlabeled) graph $\hat{A}$, which can be interpreted as a Markov chain $M_{\hat{A}}$. From the above analysis, we know that the maximal achievable

entropy rate of $M_{\hat{A}}$ is $H(A) = \lim_{n\to\infty} \frac{\log R(n)}{n}$, where $R(n)$ is the number of sequences of length n in $M_{\hat{A}}$. Let $N(n)$ be the number of behaviors of length n in A. Notice that a sequence of length n is actually a behavior of length $n-1$ in A, and thus $R(n) = N(n-1)$. We have

$$H(A) = \lim_{n\to\infty} \frac{\log R(n)}{n} = \lim_{n\to\infty} \frac{\log N(n-1)}{n} = \lim_{n\to\infty} \frac{\log N(n-1)}{n-1} = \mathcal{C}(A),$$

which completes our proof. □

In information theory, the entropy rate is used to indicate how many bits one needs to losslessly encode each sample in a stochastic process. Intuitively, a high entropy rate implies that the Markov chain has a high complexity, since one needs more resource (encoding rate) to faithfully describe the process. Since there is a one-to-one mapping between a behavior of A and a sequence generated by the Markov chain $M_{\hat{A}}$ (i.e., a path in $\hat{A}$), the entropy rate of $M_{\hat{A}}$, intuitively, is a good indicator of the complexity of the labeled graph A. Recall that a labeled graph can be interpreted as a software specification, and therefore it can be used to generate test cases for the software system. Since the actual behaviors of the software system may not conform with the specification, it is not sufficient for us to only test edges in the graph; instead, we need to test the sequences of edges (i.e., paths) in the graph. The behavior complexity can be used to indicate, asymptotically, the number of choices that a test case (i.e. a path) has in each step, and therefore it is also a measure of the difficulty of software testing and understanding.

Furthermore, the Markov chain $M_{\hat{A}}$ obtained in Theorem 6 has been used in a forthcoming paper [22] of our research group, which establishes an AEP (Asymptotic Equipartition Property) [19] for the paths of a graph. That is, using Theorem 6, one can effectively compute "typical" paths of the graph A and prove that

(1) those typical paths carry almost all the information of the Markov chain,

(2) those typical paths take probability 1, asymptotically.

Notice that the number of typical paths can be exponentially smaller than the number ($N(n)$) of all paths. This will help testers run tests only on typical paths if the goal is to reveal nontrivial amount of information from the black-box under test.

3.4 Component-based Graphs

In practice, almost all large software systems consist of multiple units. How the complexity of a component-based system changes is always of great interest for researchers (i.e., predicting the complexity of the system through the complexities of the constituent units). We are interested in the three common ways of composing a system [46, 9]:

- Sequential composition. In a component-based system, a sequencing operator specifies an ordering on executions of two units.
- Parallel composition. Two units in a component-based system can be executed in parallel.
- Nondeterministic choice.

We will study sequential composition and parallel composition in Section 3.4.1 and Section 3.4.2, respectively. We briefly discuss nondeterministic choice in Section 3.4.3.

3.4.1 Sequentially Composed Graphs

A sequentially composed graph

$$A = \langle A_c, \Phi, G \rangle$$

consists of a set of unit components $A_c = \{A_1, \cdots, A_k\}$ for some $k \geq 1$, where each unit

$$A_i = \langle S_i, s^i_{init}, \Sigma_i, E_i \rangle$$

is a labeled graph (without loss of generality, we assume that the alphabets Σ_i's and the sets of nodes S_i's are disjoint.); Φ is the set of *transition symbols* that is disjoint with Σ_i's, and will be

explained in a moment; the *component graph* G of A specifies the ordering on executions of units. Each unit A_i is represented as a node A_i in G; there is an edge from node A_i to A_j in G if and only if the unit A_j would be executed immediately after a unit A_i. The edge $A_i \rightarrow A_j$ is labeled with a transition symbol $t \in \Phi$, and the labeled edge is denoted by $A_i \stackrel{t}{\rightarrow} A_j$. Notice that there could be multiple edges from A_i to A_j, and they are labeled with different transition symbols. We also have the initial node in G to indicate the first unit to be executed in A. Without loss of generality, we assume the initial node is A_1. A behavior ω of A is a walk on the component graph G, i.e.,

$$\omega = \omega_1 t_{12} \omega_2 \cdots \omega_j t_{j(j+1)} \omega_{j+1} \cdots \omega_i$$

for some $j \geq 1$, where $\omega_j \in L(A_j)$ and $A_j \stackrel{t_{j(j+1)}}{\rightarrow} A_{j+1}$ is an edge in G for each j. The behavior set $L(A)$ of A is the set of all behaviors of A.

Recall that a unit (labeled graph)

$$A_i = \langle S_i, s^i_{init}, \Sigma_i, E_i \rangle$$

can be uniquely converted to an (unlabeled) graph $\hat{A}_i$, and there is a one-to-one mapping between a behavior of A_i and a path of $\hat{A}_i$. Similarly, we can also uniquely convert the sequentially composed graph A (which can be viewed as a labeled graph) to an (unlabeled) graph $\hat{A}$ by the following simple procedure:

```
convert-sequential(A):
//A = ⟨A_c, Φ, G⟩ is a sequentially composed graph
1. Build a (unlabeled) graph Â := NULL;
2. For each A_i = ⟨S_i, s^i_init, Σ_i, E_i⟩ ∈ A_c
3.     Â_i := convert(A_i);
```

4. Add $\hat{A}_i$ to $\hat{A}$;
5. For each edge $A_i \xrightarrow{t} A_j$ in G
6. Add a node (s^j_{init}, t) to $\hat{A}$;
7. Add an edge $(s^j_{init}, t) \rightarrow (s^j_{init}, \Lambda)$ to $\hat{A}$;
8. For each node $s_i \in S_i$, $a_i \in \Sigma_i$
9. Add an edge $(s_i, a_i) \rightarrow (s^j_{init}, t)$ to $\hat{A}$;
10. Designate the node (s^1_{init}, Λ) in $\hat{A}_1$ as the initial node of $\hat{A}$;
11. Return $\hat{A}$.

We can easily check that for each path $u = x_1 \cdots x_n$ in $\hat{A}$, there is a behavior $\omega \in L(A)$, such that $u_{\downarrow(\Sigma \cup \Phi)} = \omega$, where $\Sigma = \cup_i \Sigma_i$, and vice versa. Next we will study how the behavioral complexity of the composed-based graph changes when the units are composed in various ways.

First, we study a relative simple form of sequential composition, where in the sequentially composed graph A, units $A_1, \cdots, A_k$ are sequentially composed without loops; i.e., the component graph G of A is a *directed acyclic graph* (DAG). Let $\mathcal{C}(A_i)$ $(i = 1, \cdots, k)$ and $\mathcal{C}(A)$ be the behavioral complexity of the unit A_i and the sequentially composed graph A, respectively. We have the following theorem:

Theorem 7. *For a sequentially composed graph*

$$A = \langle A_c, \Phi, G \rangle$$

where $A_c = \{A_1, \cdots, A_k\}$ *and the component graph* G *is a DAG, the behavioral complexity is*

$$\mathcal{C}(A) = \max_{1 \leq i \leq k} \mathcal{C}(A_i).$$

Proof. Without loss of generality, we assume that there are only two units, A_1 and A_2 in A, and

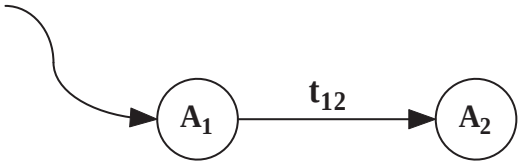


Figure 3.3: The component graph G of a sequentially composed graph A composed of units A_1 and A_2.

A_2 must be executed after A_1. The component graph G of A is illustrated in Figure 3.3. Let $\hat{A}_i$ ($i = 1, 2$) be the (unlabeled) graph converted from A_i. Let $N(n)$ be the number of behaviors of length n in A, and the behavioral complexity

$$\mathcal{C}(A) = \lim_{n\to\infty} \frac{\log N(n)}{n}.$$

Let $N_i(t_i)$ ($i = 1, 2$) be the number of behaviors of length t_i in A_i, and λ_i be the counting eigenvalue of the counting matrix of $\hat{A}_i$. Let $||\lambda_{max}|| = \max\{||\lambda_1||, ||\lambda_2||\}$. From Theorem 5, we have $N_i(t_i) \leq g_i(t_i)(||\lambda_i||)^{t_i} \leq g_i(t_i)(||\lambda_{max}||)^{t_i}$, where $g_i(t_i)$ is a polynomial function on t_i. We have,

$$\begin{aligned} N(n) &= \sum_{t_1+t_2=n-1} N_1(t_1) \cdot N_2(t_2) \\ &\leq \sum_{t_1+t_2=n-1} g_1(t_1)(||\lambda_1||)^{t_1} \cdot g_2(t_2)(||\lambda_2||)^{t_2} \\ &\leq \sum_{t_1+t_2=n-1} g_1(t_1)g_2(t_2)(||\lambda_{max}||)^{n-1} \\ &\leq n g_1(n)g_2(n)(||\lambda_{max}||)^{n}. \end{aligned}$$

Therefore,

$$\begin{aligned} \mathcal{C}(A) &= \lim_{n\to\infty} \frac{\log N(n)}{n} \\ &\leq \lim_{n\to\infty} \frac{\log n g_1(n) g_2(n) (||\lambda_{max}||)^n}{n} \\ &= \log ||\lambda_{max}|| \\ &= \max_i \mathcal{C}(A_i). \end{aligned} \tag{3.14}$$

On the other hand, $N(n) \geq N_1(n-1)$, and therefore

$$\mathcal{C}(A) = \lim_{n\to\infty} \frac{\log N(n)}{n} \geq \lim_{n\to\infty} \frac{\log N_1(n-1)}{n} = \log ||\lambda_1|| = \mathcal{C}(A_1). \tag{3.15}$$

Similarly, we have

$$\mathcal{C}(A) \geq \mathcal{C}(A_2). \tag{3.16}$$

From (3.14), (3.15) and (3.16), we have $\mathcal{C}(A) = \max_i \mathcal{C}(A_i)$. The conclusion remains when there are more than two units in A, as long as the component graph of A is a DAG. □

Theorem 7 reveals the fact that when units are sequentially composed without loops, the behavioral complexity of the sequentially composed graph will not increase.

Next we discuss the case where the component graph G of A (consisting of units $A_1, \cdots, A_k$) is not necessarily a DAG; i.e., loops are introduced when units are composed. On the one hand, we still can calculate the behavioral complexity of A from the counting matrix of the converted graph $\hat{A}$; on the other hand, it would be difficult to analyze the relation between the behavioral complexity of A and the behavioral complexity of the units only from the counting matrix. Currently, we can not give a precise representation of $\mathcal{C}(A)$ in the form of $\mathcal{C}(A_i)$; instead, we give the upper and lower bounds of $\mathcal{C}(A)$, and both bounds are tight.

Notice that the component graph G itself can be treated as a labeled graph when we treat each

unit A_i as a single node, and we can also calculate the behavioral complexity $\mathcal{C}(G)$ of G. We have the following theorem,

Theorem 8. *For a sequentially composed graph $A = \langle A_c, \Phi, G\rangle$ where $A_c = \{A_1, \cdots, A_k\}$, we have*

$$\max_{1\le i\le k} \mathcal{C}(A_i) \le \mathcal{C}(A) \le \log \rho + \mathcal{C}(G) + 1,$$

where $\rho = \max_i \rho(A_i)$ is the maximal branch factor (see Section 3.2.1 for definition) of the units.

Proof. The proof of the lower bound part is straightforward and one can refer to the proof of Theorem 7. Now we prove the upper bound part of this theorem.

For our notational convenience, we first define $\rho_i = \rho(A_i)$, and thus $\rho = \max\{\rho_i\}$. Let $N_i(t_i)$ be the number of behaviors of length t_i in A_i. Obviously, $N_i(t_i) \le \rho_i^{t_i}$. Let $N(n)$ be the number of behaviors of length n in A. For a behavior ω of length n in A, we can cut it into l ($1 \le l \le n$) segments, and each segment represents a behavior in some unit A_i, which is a node in G. Let $N(n,l)$ be the number of behaviors of lengths n in A, while the corresponding behaviors traverses l nodes in G. Clearly, $N(n) = \sum_{1\le l\le n} N(n,l)$. For a behavior ω in A that traverses l nodes in G, let $d(l)$ be the path in G that traverses those l nodes. Suppose that the behavior ω_i in A_i is of length t_i and it is a segment of ω, then $t_1 + \cdots t_l = n - (l-1)$, where $(l-1)$ is the number of transition symbols in $d(l)$. We have,

$$\begin{aligned}
N(n,l) &= \sum_{d(l)} \sum_{t_1+\cdots t_l = n-(l-1)} \prod_{1\le i\le l} N_i(t_i) \\
&\le \sum_{d(l)} \sum_{t_1+\cdots t_l = n-(l-1)} \prod_{1\le i\le l} \rho_i^{t_i} \\
&\le \sum_{d(l)} \sum_{t_1+\cdots t_l = n-(l-1)} \rho^{n-(l-1)} \\
&\le \sum_{d(l)} \binom{n}{l} \rho^n
\end{aligned}$$

Suppose that we treat G as a labeled graph, and we can also convert G to a (unlabeled) graph $\hat{G}$. Let λ_G be the counting eigenvalue of the counting matrix of $\hat{G}$. The number of paths of length l in G is $|\{d(l)\}| \leq g(l)(||\lambda_G||)^l$, where $g(l)$ is a polynomial function on l. Therefore,

$$N(n,l) \leq g(l)(||\lambda_G||)^l \begin{pmatrix} n \\ l \end{pmatrix} (\rho)^n.$$

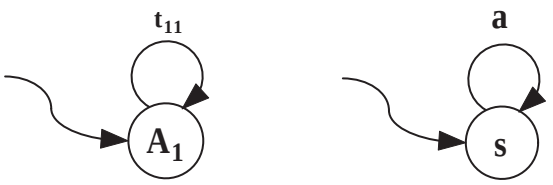
The total number of behaviors of length n in A is

$$\begin{aligned} N(n) &= \sum_{1 \leq l \leq n} N(n,l) \\ &\leq \sum_{1 \leq l \leq n} g(l)(||\lambda_G||)^l \begin{pmatrix} n \\ l \end{pmatrix} \rho^n \\ &\leq g(n)\rho^n \sum_{1 \leq l \leq n} (||\lambda_G||)^l \begin{pmatrix} n \\ l \end{pmatrix} \\ &\leq g(n)\rho^n (1 + ||\lambda_G||)^n \\ &\leq g(n)\rho^n (2||\lambda_G||)^n \end{aligned}$$

The behavioral complexity of A is

$$\begin{aligned} \mathcal{C}(A) &= \lim_{n \to \infty} \frac{\log N(n)}{n} \\ &\leq \lim_{n \to \infty} \frac{\log g(n)\rho^n (2||\lambda_G||)^n}{n} \\ &= \log \rho + \log ||\lambda_G|| + 1 \\ &= \log \rho + \mathcal{C}(G) + 1, \end{aligned}$$

which completes our proof. □



(1) (2)

Figure 3.4: A sequentially composed graph and the unit component.

Clearly, the lower bound in Theorem 8 is tight according to Theorem 7; the upper bound in Theorem 8 is also tight, shown in the following example.

Example 15. Figure 3.4 (1) shows a sequentially composed graph A which only has a unit A_1, and A_1 is composed with itself. The unit component (which is also a labeled graph) is shown in Figure 3.4 (2). The branching factor of A_1 is $\rho = 1$. It is not hard to compute that the behavioral complexity of G is $\mathcal{C}(G) = 0$. However, the behavioral complexity of A is $\mathcal{C}(A) = 1$ bit, which is $\log \rho + \mathcal{C}(G) + 1$. □

Theorem 7 and 8 reveal the fundamental science behind our intuition: it is loops that make sequentially composed graphs more complex. Theorem 8 also shows why integration testing is necessary (by looking at the tight upper bound): since it is always true that

$$\log \rho \geq \max_i \mathcal{C}(A_i),$$

in order to understand a sequentially composed system (specified by A), the term $\log \rho$ implies that we need to understand the units (specified by A_i's) with more effort than

$$\max_i \mathcal{C}(A_i),$$

and the term $\mathcal{C}(G)$ implies that we need to understand the high-level design of the sequentially composed system; however, that is not enough: we still need an extra 1-bit effort (which is fairly large since we are referring to the 1-bit rate here), which in our understanding, is the additional effort of integration testing, to understand the composed system.

3.4.2 Concurrent Graphs

In Section 3.4.1, we studied the behavioral complexity of a component-based graph where units are sequentially composed, and we showed that behavioral complexity can only increase when units are composed through loops. In this section, we further study how the behavioral complexity changes when units are composed in parallel in a component-based graph.

A component-based graph where units are composed in parallel is called a *concurrent* graph. In this chapter, we only discuss the simplest form of concurrency – synchronization over a finite set of events. A concurrent graph

$$A = \langle A_s, \Delta \rangle$$

consists of a set of unit components $A_s = \{A_1, \cdots, A_k\}$ for some $k > 1$, where each unit $A_i = \langle S_i, s^i_{init}, \Sigma_i, E_i \rangle$ is a labeled graph; Δ is the set of *synchronous (event) symbols*, which will be explained in a moment. For any two units A_i and A_j ($1 \leq i \neq j \leq k$), the intersection of the two alphabets $\Sigma_i \cap \Sigma_j = \Delta$. Sometimes, we simply denote A by $A = A_1||_\Delta A_2||_\Delta \cdots ||_\Delta A_k$.

Let $\omega_i = \omega_i^1 a_i^1 \omega_i^2 a_i^2 \cdots \omega_i^{m-1} a_i^{m-1} \omega_i^m$ for some m, and $\omega_j = \omega_j^1 a_j^1 \omega_j^2 a_j^2 \cdots \omega_j^{n-1} a_j^{n-1} \omega_j^n$ for some n be two behaviors of A_i and A_j, respectively, where $\omega_i^p \in (\Sigma_i - \Delta)^*$ for each $p = 1, \cdots, m$, $\omega_j^q \in (\Sigma_j - \Delta)^*$ for each $q = 1, \cdots, n$, a_i^g and $a_j^t \in \Delta$ for each $g = 1, \cdots, m-1$ and $t = 1, \cdots, n-1$. The two behaviors can synchronize with each other if and only if they have the same number of synchronous symbols, i.e., $m = n$, and $a_i^l = a_j^l$ for each $l = 1, \cdots, m-1$. A *synchronous behavior* of ω_i and ω_j is

$$\omega_{ij} = \omega_i||\omega_j = \omega_{ij}^1 a_i^1 \omega_{ij}^2 a_i^2 \cdots \omega_{ij}^{m-1} a_i^{m-1} \omega_{ij}^m, \tag{3.17}$$

where ω_{ij}^l is a shuffle[3] of ω_i^l and ω_j^l, $l = 1, \ldots, m$. Similarly, we can also define a synchronous behavior ω of units $A_1, A_2, \cdots, A_k$ as $\omega = \omega_1||\omega_2||\cdots||\omega_k$, where $\omega_i \in L(A_i)$ for $i = 1, \cdots, k$, and every two behaviors ω_i and ω_j can synchronize with each other. That is, recursively, $\omega_1||\omega_2||\cdots||\omega_k = (\omega_1||\omega_2||\cdots||\omega_{k-1})||\omega_k$. A behavior ω of A is a synchronous behavior of $A_1, \cdots, A_k$. The behavior set $L(A)$ is the set of behaviors of A. In below, without loss of generality, we study the concurrent graph A that only has two units A_1 and A_2, and simply write A as $A = A_1||_\Delta A_2$. This is because, a general case of $A_1||_\Delta A_2||_\Delta \cdots ||_\Delta A_k$ can be considered as, recursively, $(A_1||_\Delta A_2||_\Delta \cdots ||_\Delta A_{k-1})||_\Delta A_k$.

In the following, we build a labeled graph $A_{Sync} = \langle S, s_{init}, \Sigma, E\rangle$ with its behavior set $L(A_{Sync}) = L(A)$, where $A = A_1||_\Delta A_2$ is a concurrent graph. The labeled graph A_{Sync} can be obtained as follows.

```
Sync(A)
//A = A_1||_Δ A_2 is a concurrent graph, where
//A_1 = ⟨S_1, s^1_init, Σ_1, E_1⟩ and A_2 = ⟨S_2, s^2_init, Σ_2, E_2⟩.
1.   S := S_1 × S_2 = {(s_1, s_2)|s_1 ∈ S_1, s_2 ∈ S_2};
2.   s_init := (s^1_init, s^2_init);
3.   Σ := Σ_1 ∪ Σ_2;
4.   E := ∅;
5.   For each edge (s_1, b_1, s'_1) ∈ A_1, b_1 ∉ Δ
6.      For each node s ∈ S_2
7.         Add edge ((s_1, s), b_1, (s'_1, s)) to E;
8.   For each edge (s_2, b_2, s'_2) ∈ A_2, b_2 ∉ Δ
9.      For each node s ∈ S_1
```

[3]A sequence ω is a *shuffle* of a sequence ω_1 and a sequence ω_2 if ω_1 is a subsequence of ω, and after dropping the subsequence from ω, we obtain ω_2. For instance, $\omega = b\underline{c}d\underline{cd}a\underline{bc}$ is a shuffle of $\omega_1 = ccdbc$ (the subsequence in ω is underlined) and $\omega_2 = bda$.

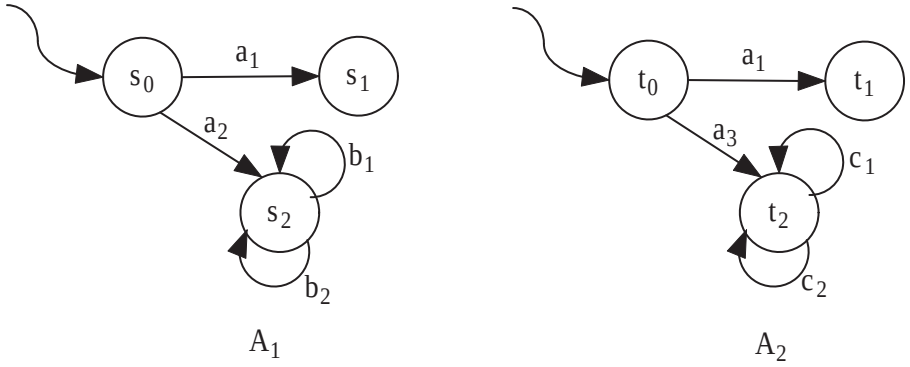


Figure 3.5: A concurrent graph.

```
10.        Add edge ((s, s2), b2, (s, s'2)) to E;
11.  For each edge (s1, a, s'1) ∈ A1 and (s2, a, s'2) ∈ A2, a ∈ Δ
12.     Add edge ((s1, s2), a, (s'1, s'2)) to E;
13.  Return A_Sync := ⟨S, s_init, Σ, E⟩.
```

Notice that the size of the resulting A_{Sync} is $|A_1| \times |A_2|$, where $|A_i|$ is the size of A_i for $i = 1, 2$. We can easily check that there is a one-one mapping between the two behavior sets $L(A_{Sync})$ and $L(A)$; therefore, $\mathcal{C}(A_{Sync}) = \mathcal{C}(A)$. The behavioral complexity $\mathcal{C}(A_{Sync})$ of A_{Sync} is efficient to compute (using the `complexity()` algorithm in before), and then $\mathcal{C}(A_{Sync})$ can be used to asymptotically measure the cost of exhaustive testing of the system specified by A. Next we will give a tight lower bound and a tight upper bound of $\mathcal{C}(A)$ when A is a concurrent graph.

Theorem 9. *For a concurrent graph $A = \langle A_s, \Delta \rangle$ where $A_s = \{A_1, \cdots, A_k\}$ ($k > 1$), we have*

$$0 \leq \mathcal{C}(A) \leq \max_i \mathcal{C}(A_i) + \log k.$$

Proof. We first prove the case for $k = 2$. The proof of the lower bound part is given by an example. Consider the concurrent graph $A = A_1||_\Delta A_2$ shown in Figure 3.5, where the set of synchronous symbols is $\Delta = \{a_1, a_2, a_3\}$. Both units A_1 and A_2 have behavioral complexity 1 bit. However,

after synchronization, the behavior set becomes $L(A) = \{a_1\}$, which implies that $\mathcal{C}(A) = 0$. This example shows that it is possible that synchronization rules out most behaviors of units, and makes the behaviors of the concurrent graph extremely simple.

Now we turn to the upper bound part. Let $N(n)$ be the number of behaviors of length n in A, $N_i(t_i)$ $(i = 1, 2)$ be the number of behaviors of length t_i in A_i, and λ_i be the counting eigenvalue of the counting matrix of $\hat{A}_i$. Let $||\lambda_{max}|| = \max\{||\lambda_1||, ||\lambda_2||\}$. From Theorem 5, we have $N_i(t_i) \leq g_i(t_i)(||\lambda_i||)^{t_i} \leq g_i(t_i)(||\lambda_{max}||)^{t_i}$, where $g_i(t_i)$ is a polynomial function on t_i. Clearly, $N(n)$ reaches the maximum when there is no synchronization between units; i.e., each behavior of A is a pure shuffle of behaviors of its units. Note that given two sequences ω_1 of length t_1 and ω_2 of length t_2, there are $\begin{pmatrix} t_1 + t_2 \\ t_1 \end{pmatrix}$ shuffles of ω_1 and ω_2. We have

$$\begin{aligned}
N(n) &\leq \sum_{t_1+t_2=n} N_1(t_1) \cdot N_2(t_2) \cdot \begin{pmatrix} n \\ t_1 \end{pmatrix} \\
&\leq \sum_{t_1+t_2=n} g_1(t_1)(||\lambda_1||)^{t_1} \cdot g_2(t_2)(||\lambda_2||)^{t_2} \cdot \begin{pmatrix} n \\ t_1 \end{pmatrix} \\
&\leq \sum_{t_1+t_2=n} g_1(t_1)g_2(t_2)(||\lambda_{max}||)^{n} \cdot \begin{pmatrix} n \\ t_1 \end{pmatrix} \\
&\leq g_1(n)g_2(n)(||\lambda_{max}||)^{n} \sum_{t_1+t_2=n} \begin{pmatrix} n \\ t_1 \end{pmatrix} \\
&= g_1(n)g_2(n)(2||\lambda_{max}||)^{n}
\end{aligned}$$

Therefore,

$$\begin{aligned} \mathcal{C}(A) &= \lim_{n\to\infty} \frac{\log N(n)}{n} \\ &\leq \lim_{n\to\infty} \frac{\log g_1(n)g_2(n)(2||\lambda_{max}||)^n}{n} \\ &= \log ||\lambda_{max}|| + 1 \\ &= \max_i \mathcal{C}(A_i) + 1. \end{aligned} \tag{3.18}$$

The proof can be directly generalized to a concurrent graph with $k > 2$ units. In that case, $\mathcal{C}(A) = \max_i \mathcal{C}(A_i) + \log k$. This result can be interpreted as follows. For a synchronous behavior $\omega = \omega_1||\omega_2||\cdots||\omega_k$, we can assume that that there is a *controller* to generate it in the following way. Each time, the controller picks some A_i to generate a symbol in ω. The bit rate caused by A_i is $\max_i \mathcal{C}(A_i)$ at most. Since there are k such A_i's, the extra bit rate that the controller causes is at most $\log k$. □

Clearly the lower bound in Theorem 9 is tight according to the proof; the upper bound is also tight. Consider the concurrent graph $A = A_1||_\Delta A_2$ with $L(A_1) = a^*$, $L(A_2) = b^*$ and $\Delta = \emptyset$. The behavior set of A is $L(A) = (a + b)^*$. Clearly, $\mathcal{C}(A_1) = \mathcal{C}(A_2) = 0$ bit, and $\mathcal{C}(A) = 1$ bit, which is $\max_i \mathcal{C}(A_i) + 1$.

Recall that the behavioral complexity $\mathcal{C}(A)$ intends to asymptotically measure the cost of exhaustive testing of the system Sys specified by A. Suppose that before testing the concurrent system Sys specified by $A = A_1||_\Delta A_2$, we know nothing about the system except the desired behavior set $L(A)$; i.e., we have *zero pre-test knowledge* about the system. There might be some hidden communications among units other than synchronizations over the known interface Δ. For instance, suppose that a unit system Sys_1 specified by A_1 and a unit system Sys_2 specified by A_2 synchronize over $\Delta = \{a\}$, and Sys_1 has observable events in $\Sigma_1 = \{b, a\}$, Sys_2 has observable

events in $\Sigma_2 = \{c, a\}$. Sys_1 and Sys_2 may communicate through some unobservable communication channel and such that "bca is observed in Sys" does not necessarily imply "cba is also observed in Sys". Such hidden communications can be an unobservable access control to a critical region to ensure that event b in Sys_1 must happen before event c in Sys_2. Therefore, in order to verify that the system is implemented as designed, we have to verify each behavior in the behavior set $L(A)$ and then decide whether the behavior is an actual behavior of the system or not. In that case, $\mathcal{C}(A)$ is used to asymptotically measures the cost of exhaustive testing of the system.

However, sometimes, before the testing, we do have some knowledge about Sys, and with that pre-test knowledge, we could avoid exhaustive testing and the cost of testing Sys could be reduced.

A simple form of such pre-test knowledge studied here is called *communication pre-test knowledge*. Suppose that before testing the concurrent system Sys specified by $A = A_1||_\Delta A_2$, we know for sure that:

> If there is any communication between Sys_1 (specified by A_1) and Sys_2 (specified by A_2), then the communication must be synchronizations over Δ.

In that case, we can impose *pre-order* on synchronous behaviors. The pre-ordered synchronous behavior of $\omega_i = \omega_i^1 a_i^1 \omega_i^2 a_i^2 \cdots \omega_i^{m-1} a_i^{m-1} \omega_i^m$ and $\omega_j = \omega_j^1 a_j^1 \omega_j^2 a_j^2 \cdots \omega_j^{m-1} a_j^{m-1} \omega_j^m$ in (3.17) is defined as

$$\omega_{ij}^* = \omega_i^1 \omega_j^1 a_i^1 \omega_i^2 \omega_j^2 a_i^2 \cdots \omega_i^{m-1} \omega_j^{m-1} a_i^{m-1} \omega_i^m \omega_j^m. \tag{3.19}$$

With the communication pre-test knowledge, we can safely say that, any synchronous behavior w_{ij} of ω_i and ω_j, in the form of (3.17), is an actual behavior of Sys if and only if the pre-ordered synchronous behavior ω_{ij}^* of ω_i and ω_j is an actual behavior of Sys. Therefore, we can build a new behavior set $L^*(A)$ consisting of pre-ordered synchronous behaviors, such that a synchronous behavior in $L(A)$ is an actual behavior of A if and only if the corresponding pre-ordered synchronous behaviors in $L^*(A)$ is an actual behavior of Sys. In this way, the cost of exhaustive testing on

$L(A)$ can be reduced to exhaustive testing on $L^*(A)$, and we use

$$\mathcal{C}^*(A) = \lim_{n\to\infty} \frac{\log N^*(n)}{n}$$

to denote the cost of testing Sys with communication pre-test knowledge, where $N^*(n)$ is number of sequences of length n in $L^*(A)$.

In the following, we build a labeled graph $A_{Pre} = \langle S, s_{init}, \Sigma, E\rangle$ with its behavior set $L(A_{Pre}) = L^*(A)$, where $A = A_1||_\Delta A_2$ is a concurrent graph. The idea of building A_{Pre} is that we use two phases ϕ_1 and ϕ_2 to control the ordering of A_1 and A_2. The labeled graph A_{Pre} can be obtained as follows.

Pre-Ordered(A)

//$A = A_1||_\Delta A_2$ is a concurrent graph, where

//$A_1 = \langle S_1, s^1_{init}, \Sigma_1, E_1\rangle$ and $A_2 = \langle S_2, s^2_{init}, \Sigma_2, E_2\rangle$.

1. $S := S_1 \times S_2 \times \{\phi_1, \phi_2\} = \{(s_1, s_2, \phi)|s_1 \in S_1, s_2 \in S_2, \phi \in \{\phi_1, \phi_2\}\}$;
2. $s_{init} := (s^1_{init}, s^2_{init}, \phi_1)$;
3. $\Sigma := \Sigma_1 \cup \Sigma_2$;
4. $E := \emptyset$;
5. For each edge $(s_1, b_1, s'_1) \in A_1$, $b_1 \notin \Delta$
6. For each node $s \in S_2$
7. Add edge $((s_1, s, \phi_1), b_1, (s'_1, s, \phi_1))$ to E;
8. For each edge $(s_2, b_2, s'_2) \in A_2$, $b_2 \notin \Delta$
9. For each node $s \in S_1$
10. Add edge $((s, s_2, \phi_1), b_2, (s, s'_2, \phi_2))$ to E;
11. Add edge $((s, s_2, \phi_2), b_2, (s, s'_2, \phi_2))$ to E;
12. For each edge $(s_1, a, s'_1) \in A_1$ and $(s_2, a, s'_2) \in A_2$, $a \in \Delta$

13. Add edge $((s_1, s_2, \phi_1), a, (s'_1, s'_2, \phi_1))$ to E;

14. Add edge $((s_1, s_2, \phi_2), a, (s'_1, s'_2, \phi_1))$ to E;

15. Return $A_{Pre} := \langle S, s_{init}, \Sigma, E \rangle$.

The size of the resulting A_{Pre} is $|A_1| \times |A_2|$, where $|A_i|$ is the size of A_i for $i = 1, 2$. Also we can use the algorithm `complexity()` to compute the behavioral complexity $\mathcal{C}(A_{Pre})$. The construction can be generalized to $A_1||_\Delta A_2||_\Delta \cdots ||_\Delta A_k$. In below, we give a tight lower bound and a tight upper bound of $\mathcal{C}^*(A)$ when A is a concurrent graph.

Theorem 10. *For a concurrent graph $A = \langle A_s, \Delta \rangle$ where $A_s = \{A_1, \cdots, A_k\}$ ($k > 1$), we have*

$$0 \leq \mathcal{C}^*(A) \leq \max_i \mathcal{C}(A_i).$$

Proof. Without loss of generality, we only show the case for $k = 2$. The proof of the lower bound is similar to the proof of Theorem 9.

The proof of the upper bound is very similar to the proof of Theorem 7. Let $N^*(n)$ be the number of behaviors of length n in $L^*(A)$, $N_i(t_i)$ $(i = 1, 2)$ be the number of behaviors of length t_i in A_i, and λ_i be the counting eigenvalue of the counting matrix of $\hat{A}_i$. Let $||\lambda_{max}|| = \max\{||\lambda_1||, ||\lambda_2||\}$. From Theorem 5, we have $N_i(t_i) \leq g_i(t_i)(||\lambda_i||)^{t_i} \leq g_i(t_i)(||\lambda_{max}||)^{t_i}$, where $g_i(t_i)$ is a polynomial function on t_i. Clearly, $N^*(n)$ reaches the maximum when there is no synchronization between units. We have,

$$\begin{aligned}
N^*(n) &\leq \sum_{t_1+t_2=n} N_1(t_1) \cdot N_2(t_2) \\
&\leq \sum_{t_1+t_2=n} g_1(t_1)(||\lambda_1||)^{t_1} \cdot g_2(t_2)(||\lambda_2||)^{t_2} \\
&\leq \sum_{t_1+t_2=n} g_1(t_1) g_2(t_2)(||\lambda_{max}||)^n \\
&\leq n g_1(n) g_2(n)(||\lambda_{max}||)^n.
\end{aligned}$$

Therefore,

$$\begin{aligned} \mathcal{C}^*(A) &= \lim_{n\to\infty} \frac{\log N^*(n)}{n} \\ &\leq \lim_{n\to\infty} \frac{\log n g_1(n) g_2(n) (||\lambda_{max}||)^n}{n} \\ &= \log ||\lambda_{max}|| \\ &= \max_i \mathcal{C}(A_i), \end{aligned} \tag{3.20}$$

which completes our proof. □

Both the lower bound and upper bound in Theorem 10 are tight. Consider the concurrent graph $A = A_1||_\Delta A_2$ with $L(A_1) = a^*$, $L(A_2) = b^*$ and $\Delta = \emptyset$. The (pre-ordered) behavior set is $L^*(A) = a^*b^*$. Clearly, $\mathcal{C}(A_1) = \mathcal{C}(A_2) = \mathcal{C}^*(A) = 0$ bit, which shows that both the lower bound and upper bound in Theorem 10 are tight.

In [70], we show a result that under certain conditions, when a system is concurrently composed from a number of blackboxes, integration testing of the system is not necessary: it can be implemented through a "cascade" sequence of unit blackbox testing. This is verified by Theorem 10: the behavioral complexity of a current system will not increase as long as we know for sure that communications among units are only achieved by synchronization over a known interface. However, Theorem 9 also reveals that, if there are other hidden communications, we do need extra effort to understand the concurrent system.

3.4.3 Nondeterministic Choice

A component-based graph A consisting of units $A_1, \cdots, A_k$ ($k \geq 1$) where nondeterministic choice is involved is called a *nondeterministic component-based graph*, where each unit $A_i = \langle S_i, s^i_{init}, \Sigma_i, E_i \rangle$ is a labeled graph, and A is denoted by

$$A = A_1 + \cdots + A_k.$$

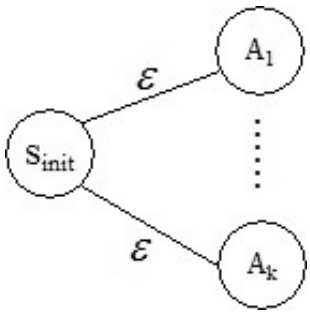


Figure 3.6: A nondeterministic component-based system $A = A_1 + \cdots + A_k$

Nondeterministic choice states that, suppose that a component-based software system Sys is specified by A, and there are k unit systems $Sys_1, \cdots, Sys_k$ specified by $A_1, \cdots, A_k$, respectively. Sys can nondeterministically pick one unit Sys_i ($1 \leq i \leq k$) to execute. Clearly, the behavioral set of A is the set of all its constituent units' behaviors. A can be represented by a labeled graph $A = \langle S, s_{init}, \Sigma, E \rangle$ shown in Figure 3.6, where $S = \{s_{init}\} \bigcup_i S_i$, s_{init} is the initial state of A, $\Sigma = \{\varepsilon\} \bigcup_i \Sigma_i$ with ε being the empty symbol, and $E = \bigcup_i E_i \cup \{(s_{init}, \varepsilon, s^i_{init})\}$. Notice that the labeled graph shown in Figure 3.6 is actually a nondeterministic finite automaton. we can simply use a standard textbook algorithm (e.g., [62]) to convert the nondeterministic automaton to a deterministic one, which is trivial. Note that units A_i's themselves are deterministic systems, therefore the algorithm of converting the nondeterministic automaton to a deterministic one can be done in polynomial time. It is straightforward that the behavioral complexity of A is

$$\mathcal{C}(A) = \max_i \mathcal{C}(A_i),$$

which implies that nondeterministic choice will not make the composed system more complex.

3.5 Discussions

Unlike the McCabe metric, the behavior complexity of a labeled graph is not necessarily strictly monotonic in the number of nodes and edges in the graph. However, similar to as the McCabe metric, the behavioral complexity is related to the strongly connected components (SCC) in the

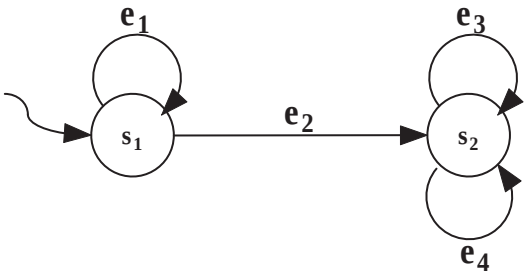


Figure 3.7: A labeled graph A.

graph. From Theorem 7, we can calculate the behavioral complexity of a labeled graph A in the following way. We first decompose (in linear time using Tarjan's algorithm) A into SCC's $A_1, \cdots, A_k$, and for each SCC A_i, we calculate the behavioral complexity $\mathcal{C}(A_i)$, and then the behavioral complexity of A is

$$\mathcal{C}(A) = \max_i \mathcal{C}(A_i).$$

This is because A becomes a DAG when each SCC is treated as a node. In other words, one can identify a "key portion" of A that "concentrates" the entire complexity of A.

Furthermore, we can define the concept of *critical edges*. Let e be an edge in a labeled graph A. We define A^{-e} as the resulting graph after dropping the edge e from A. Clearly, $\mathcal{C}(A^{-e}) \leq \mathcal{C}(A)$. We say that e is *critical* if

$$\mathcal{C}(A^{-e}) < \mathcal{C}(A).$$

That is, after dropping e, the complexity becomes smaller. For instance, in the labeled graph given in Figure 3.7, one can show that the edges e_2, e_3 and e_4 are critical edges, but not e_1. This can also be generalized to the case of a "critical portion" of a graph A. If we drop a subgraph A' from A and the behavioral complexity of the resulting graph becomes smaller, we say that A' is a critical portion of A. For the labeled graph A given in Figure 3.7, the subgraph consisting of the node s_2, the edge e_3 and the edge e_4 is the critical portion of A. Similarly, we can also decide whether an update applied to a labeled graph makes the graph more complex or not. Suppose that we are

going to replace a subgraph A' of A with A''. Clearly, the behavioral complexity of the updated graph can be computed and can also be compared with the behavioral complexity of the original graph A before the replacement. This gives us a way to predict whether an update would make a system harder to understand or not.

The result of Theorem 6 (essentially relating the number of paths in a graph with the maximal entropy rate of a Markov chain represented by the graph) has already been hinted at in some classical work, dated back in Shannon's paper in 1949 (Theorem 8 in [60]), and Chomsky and Miller's work [17] in 1958, and even the Exercise 4.16 in the classical textbook on Information Theory [19]. Reference [11] also uses $\lim_{n\to\infty} \frac{\log N(n)}{n}$ to define the capacity of a discrete noiseless channel with an intuitive explanation. However, we have found a mathematically strict proof only for a special form of graphs to make a Markov chain stationary. In our work, we proved the general case, by removing the condition of ergodicity, but replacing the entropy rate of a Markov chain from the limit-form to the upper-limit-form (noticing that the upper-limit entropy rate always exists for any Markov chain). Our proof is quite involved, using the method of type.

In our work, the semantics of a labeled graph is defined as the set of sequences of labels collected from the paths (from the initial node) of the labeled graph. This is because, semantically, the graph is understood as a machine that sequentially executes along the directed edges. Our behavioral complexity returns the same result for graphs sharing the same behavioral sets. However, similar to the McCabe metric, we do not further decipher the meaning of each label (i.e., each label is atomic); doing this would involve undecidability of checking whether the deciphered semantics are preserved or not. For instance, two arithmetic C programs that have different control flow graphs may actually compute the same function. These two graphs may have different behavioral complexities. In other words, if one does come up with a new definition of behavioral complexity such that the aforementioned two graphs have the same complexity under the new behavioral complexity, then, we can easily show that the new behavioral complexity is simply not computable in general. Nevertheless, our results suggest that, in theory, the C program with lower behavioral

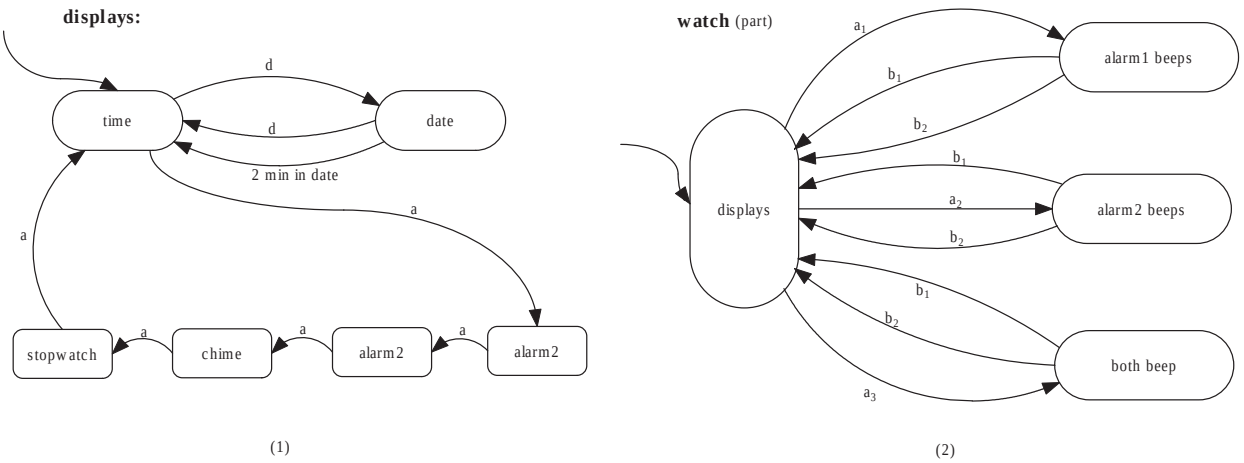


Figure 3.8: (1) The statechart of the unit `displays`. (2) The component graph of the component-based graph.

complexity is easier to understand. However, similar to the McCabe metric, whether this is also the case in practice relies on a work like [79] to further validate by experimental software engineering researchers.

3.6 A Case Study on Statecharts Design

Statecharts [31] is a useful tool for modeling system designs, and it has become part of the standardized modeling language UML [1]. In a statechart, a state is not necessarily atomic, and it can specify a unit in a component-based system. Throughout the original paper [31] of Statecharts, the design of a Citizen Quartz Multi-Alarm III wristwatch is used to illustrate how to build statecharts to illustrate the design of a system. In this section, we will also adopt this example to show the behavioral complexity of the design.

Figure 3.8 (1) is the statechart A_1 of the unit `displays` (Figure 9 in [31]), and it shows how the unit `displays` of the watch transits its states. A labeled edge in A_1 denotes a state transition, and the label denotes some event. For instance, the edge $\texttt{time} \xrightarrow{d} \texttt{date}$ means that when the button d is pressed, `displays` would transit from the state `time` to `date`. Clearly, A_1 can be treated as a labeled graph, and it is not hard to calculate the behavioral complexity, which is $\mathcal{C}(A_1) = 0.5973$

bit. After dropping the edge $\texttt{date}\xrightarrow{d}\texttt{time}$, the behavioral complexity decreases to 0.3063 bit, which implies that $\texttt{date}\xrightarrow{d}\texttt{time}$ is a critical edge in A_1, and the semantics would change if we drop this edge.

The unit `displays` is sequentially composed with other units in the component-based graph A. Figure 3.8 (2) (modified from Figure 8 in [31]) actually gives the component graph G of A. If we treat every state in G as a single node, we can also calculate the behavioral complexity of the labeled graph G (which can be interpreted as the high-level design) and $\mathcal{C}(G) = 1.2925$ bits. Notice that 1.2925 is a relatively large number, which implies that units in the component-based graph are composed in a complicated way.

Now we are to calculate the behavioral complexity $\mathcal{C}(A)$ of the component-based graph. We do not, as in the paper [31], further decompose the nodes `alarm 1 beeps`, `alarm 2 beeps` and `both beep` in G, and simply treat them as units only with one state. Clearly the behavioral complexity of the three units are all 0. We can easily calculate that $\mathcal{C}(A) = 1.0839$ bits. If we drop the unit `alarm 1 beeps` from A, we can check that the behavioral complexity of A decreases to 1.0087 bits, which implies that dropping the unit `alarm 1 beeps` would change the semantics of A.

The behavioral complexities of units in A is not that large (0.5973 bit for `displays` and 0 bit for the other three units). However, the behavioral complexity of A is 1.0839 bits, which implies that, even simple units can make fairly complex component-based graphs. It is also interesting to find that the behavioral complexity of A (1.0839 bits) does not exceed the behavioral complexity of its component graph G (1.2925 bits). Notice that the component graph can be interpreted as the high-level design of a component-based system. This indicates that, even if units are composed in a complicated way, the behaviors of the component-based graph could be semantically simple.

We write a script in Matlab, from a graph, to compute the behavioral complexity through Jordan decomposition and running times of the experiments in this section are all negligible on a ThinkPad T400 laptop.

3.7 Summary

In this chapter, we introduce a novel complexity metric for labeled graphs named behavioral complexity, and we provide an information-theoretic foundation for it. We also study how the behavioral complexity changes when graphs are composed sequentially and in parallel, and we show that behavioral complexity can increase when units are sequentially composed through loops, or composed in parallel through synchronization with zero pre-test knowledge. Since labeled graphs can be interpreted as control flow graphs, design specifications, etc., our complexity metric can also be used to measure the complexity of software systems. We show the usefulness of the metric through a statecharts design example. Our results could be very helpful for software developers: the metric can be used to predict the complexity of the semantics of a software system to be built. Our results also suggest that integration testing is necessary (after units are tested).

CHAPTER 4

CONCLUSIONS AND DISCUSSIONS

In this dissertation, we propose an information-theoretic framework of software testing. Our work mainly consists of two parts: the first part provides a syntax-independent coverage criterion for software testing, which can be used for information-optimal test case selection (the process of information-optimal test case selection is called a cooling-down testing approach); the second part introduces a novel complexity metric for software systems which asymptotically measures the cost of exhaustive testing. Both theories have solid information-theoretic foundations, which, we think, is of great importance in the area of software testing (which, in our opinion, still lacks a mathematical foundation) as well as model-based design for safety-critical systems and network intrusion detection systems [73].

In the future, we plan to compare our cooling-down testing approaches with existing structural testing techniques and see if our approaches can be combined with the existing techniques to improve their effectiveness. We will also study an information-theoretic approach in optimal test case selection for concurrent systems. Testing a concurrent system is an extremely difficult and error-prone task [27, 34]. In Chapter 3, we have developed algorithms to calculate the behavioral complexity of concurrent systems; in other words, we have a way to tell whether a concurrent system is complex or not. However, in order to apply our cooling-down testing approach, we still need to calculate the entropy of a concurrent system. In our opinion, the entropy of a concurrent system is difficult (even in theory) to calculate because of the communications between components. We plan to study algorithms for general concurrent system models and determine how the communications between components affect the optimal testing over the entire system.

In a highly concurrent system, a massive number of objects are involved in computations. For instance, a network system can be viewed as programs running on top of a large number of devices, such as cellular phones, laptops, PDAs and sensors. How to design, implement and test such

systems has become a central research topic in areas like pervasive computing [57, 67], a proposal of building distributed software systems from a massive number of devices that are pervasively hidden in the environment. P system [51], is an emerging unconventional computing model motivated from natural phenomena of cell evolutions and chemical reactions, which can be used to characterize such systems. There has been a flurry of research activities in the area of membrane computing (a branch of molecular computing) initiated several years ago by Gheorghe Paun [51]. Membrane computing identifies an unconventional computing model, namely a P system, from natural phenomena of cell evolutions and chemical reactions. It abstracts from the way living cells process chemical compounds in their compartmental structures. Thus, regions defined by a membrane structure contain objects that evolve according to given rules. The objects can be described by symbols or strings of symbols, in such a way that multisets of objects are placed in regions of the membrane structure. The membranes themselves are organized as a Venn diagram or a tree structure where one membrane may contain other membranes. By using the rules in a nondeterministic and maximally parallel manner, transitions between the system configurations can be obtained. A sequence of transitions shows how the system is evolving. At a high-level, a P system has the following key features:

- Objects are typed but addressless (i.e., without individual identifiers),
- Object transferring rules are in (either maximally or locally) parallel, and
- The system is stateless [75].

The above three features make P systems be suitable to model a highly concurrently system. First, objects in P systems are typed but addressless (i.e., the objects do not have individual identifiers), which is an attractive property for modeling concurrent systems. In other words, unique identifiers such as IP addresses for network devices are left for the implementation level. For instance, in a fire truck scheduling system, a fire emergency calls for one or more trucks that are

currently available. In this scenario, exactly which truck is dispatched is not so important as long as the truck is available. Second, in a P system, there are a huge number (the number could be unspecified) of objects running in a highly concurrent manner, which exactly is the way that objects in a concurrent system behave (e.g., in a cellular phone network, there are a huge number of cellular phone being active and interacting with each other). Finally, it is almost impossible to maintain a global state of a highly concurrent system that involves a massive number of objects; thus P systems, which are naturally stateless, are a good model for such concurrent systems.

In [74, 75, 38, 77, 76, 78], we introduced computing models called Bond Computing Systems and service automata, which are variations of P systems. Testing of such systems remains a great challenge. Currently, software testing is to "Find a graph and cover it [7]." However, in a stateless system, such a graph does not even exist. Some research work [23] has been done on the model-checking (i.e., algorithmically check whether behaviors of a system conform with its specification) of a P system. However, this model-based approach faces either the state explosion problem or the undecidability problem. Usually we can not find efficient algorithms and could not finish checking all the possible behaviors since the number of the states in the system is exponential in the number of components or the components are simply Turing-complete. Using the cooling-down idea, how to select behaviors of the system in order to check the specification most efficiently becomes possible. Moreover, model-checking is traditionally used to check whether (behaviors in) the design of a system conforms with its requirement. It is the software testing that tests the system's implementation. From this point of view, an optimal testing strategy is desired to test a stateless system. We believe that our idea of cooling-down testing is a good approach to develop such optimal strategies. Furthermore, We can also apply the behavioral complexity metric on a stateless system, which can indicate how difficult it is to test a specific stateless system.

Information theory has been considered as a mathematical foundation for digital and analog communications as well as data compression, image processing, etc., which are among the most studied areas in Electrical Engineering. However, the information theory has not been shown its

equal importance in Computer Science; e.g, providing a mathematical foundation for an important area like Software Testing. The reasons, in our opinion, are the fact that in computer science, discrete structures (trees, graphs, automata, logic formulas, code, etc.), instead of quantitative variables, are often the subjects of study, while traditional information theory, with its history of more than 50 years, does not usually provide a mathematical tool to handle "information on discrete structures". This dissertation, also as a contribution to information theory, provides a way to calculate entropy for trees and graphs (drawn from finite automata), together with a syntax-based complexity metric for software systems. In the future, one could further study information theory on more complex structures like programs and more powerful automata. This work makes us strongly believe that Shannon's information theory could be a mathematical foundation for Software Testing. The theory contains a rich body of deep and foundational results in, for instance, channel communications and optimal coding. With this belief in mind, one would naturally seek applications of some of the results in Software Testing. For example, is there an inherent relation between optimal testing of a software system and optimal coding of a signal transmitted through a channel? Notice that optimal coding, roughly speaking, is to catch most amount of information using shortest bits, while our optimal testing is to identify most of the system under test using a smallest amount of testing cost. Testing a large software system is often considered as a team work: testers communicate with the system under test while running test cases, and, testers communicate among themselves to discuss the test results. Those communications can be considered as a network of channels: tester-system channels and tester-tester channels. This dissertation work only considers optimal strategies when only one tester-system channel is present. For future work, one need study an "optimal testing strategy" over the entire network of channels and the optimality is not only for an individual tester but also for the entire team. We believe that this is related to the area of "network information theory" (roughly speaking, it is about efficient information communication over a network of channels) [19], which is considered a very difficult and deep part of modern information theory.

BIBLIOGRAPHY

[1] http://www.uml.org/.

[2] http://www.wolfram.com/.

[3] http://www.mathworks.com/.

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[5] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *ISSRE'03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 99–107. IEEE Computer Society, 2003.

[6] C. Andrés, L. Llana, and I. Rodríguez. Formally transforming user-model testing problems into implementer-model testing problems and viceversa. *Journal of Logic and Algebraic Programming*, 78(6):425–453, 2009.

[7] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, second edition, 1990.

[8] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, 1995.

[9] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[10] R. Bhatia. *Matrix Analysis (Graduate Texts in Mathematics)*. Springer, 1996.

[11] R. E. Blahut. *Principals and Practice of Information Theory*. Addison-Wesley, 1987.

[12] L. B. Boza. Asymptotically optimal tests for finite markov chains. *Ann. Math. Statist.*, 42(6):1992–2007, 1971.

[13] M. Broy, B. Jonsson, J-P. Katoen, M. Leucker, and A. Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Lecture Notes in Computer Science, Vol. 3472, Springer, 2005.

[14] R. L. Burden and J. D. Faires. *Numerical Analysis*. Brooks-Cole Publishing, eighth edition, 2004.

[15] J. K. Chhabra, K. K. Aggarwal, and Y. Singh. Code and data spatial complexity: two important software understandability measures. *Information and Software Technology*, 45(8):539 – 546, 2003.

[16] J. K. Chhabra and V. Gupta. Evaluation of object-oriented spatial complexity measures. *SIGSOFT Softw. Eng. Notes*, 34(3):1–5, 2009.

[17] N. Chomsky and G. A. Miller. Finite state languages. *Information and Control*, 1:91–112, 1958.

[18] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[19] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, second edition, 2006.

[20] I. Csiszár. The method of types. *IEEE Transactions on Information Theory*, 44(6):2505–2523, 1998.

[21] I. Csiszár, T. Cover, and B-S. Choi. Conditional limit theorems under markov conditioning. *IEEE Transactions on Information Theory*, 33(6):788 – 801, 1987.

[22] C. Cui, Z. Dang, and T. R. Fischer. Typical paths of a graph. Submitted.

[23] Z. Dang, O. H. Ibarra, C. Li, and G. Xie. On model-checking of p systems. In Cristian Calude, Michael J. Dinneen, Gheorghe Paun, Mario J. Pérez-Jiménez, and Grzegorz Rozenberg, editors, *UC*, volume 3699 of *Lecture Notes in Computer Science*, pages 82–93. Springer, 2005.

[24] C. R. Douce, P. J. Layzell, and J. Buckley. Spatial measures of software complexity. In *Proc 11th Meeting of Psychology of Programming Interest Group, Leeds*, 1999.

[25] J. W. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. Softw. Eng.*, 10(4):438–444, 1984.

[26] M. C. Gaudel. Testing can be formal, too. In *TAPSOFT '95: Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, 1995. Lecture Notes in Computer Science, Vol. 915, pp. 82–96, Springer.

[27] S. Ghosh and A. P. Mathur. Issues in testing distributed component-based systems. In *First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.

[28] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. In *Proceedings of the international conference on Reliable software*, pages 493–510. ACM, 1975.

[29] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[30] W. J. Hansen. Measurement of program complexity by the pair (cyclomatic number, operator count). *SIGPLAN Not.*, 13(3):29–33, 1978.

[31] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[32] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1989.

[33] W. Harrison. An entropy-based measure of software complexity. *IEEE Trans. Softw. Eng.*, 18(11):1025–1029, 1992.

[34] M. J. Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72. ACM, 2000.

[35] C. Heitmeyer. Software cost reduction. *Encyclopedia of Software Engineering, second edition*, 2002.

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

[37] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, third edition, 2007.

[38] O. H. Ibarra, Z. Dang, and L. Yang. On counter machines, reachability problems, and diophantine equations. *Int. J. Found. Comput. Sci.*, 19(4):919–934, 2008.

[39] J. P. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler. Software complexity measurement. *Commun. ACM*, 29(11):1044–1050, 1986.

[40] J. Ladyman. Physics and computation: The status of landauer's principle. In *CiE '07: Proceedings of the 3rd conference on Computability in Europe*, 2007. Lecture Notes in Computer Science, Vol. 4497, pp. 446–454, Springer.

[41] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

[42] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *KDD '03: Proceedings of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 388–396, 2003.

[43] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[44] L. Marco. Measuring software complexity. *Enterprise Systems Journal*, April 1997.

[45] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[46] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[47] M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. *Ann. of Math.*, 74:437–455, 1961.

[48] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[49] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29(4):55–64, 2004.

[50] E. I. Oviedo. Software engineering metrics I. chapter Control flow, data flow and program complexity, pages 52–65. McGraw-Hill, 1993.

[51] Gh. Paun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.

[52] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. *J. Autom. Lang. Comb.*, 7(2):225–246, 2001.

[53] A. Petrenko, N. Yevtushenko, and J. Huo. Testing transition systems with input and output testers. In *TestCom '03*, 2003. Lecture Notes in Computer Science, Vol. 2644, pp. 129–145, Springer.

[54] A. Pnueli. The temporal logic of programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[55] R. A. Qutaish and A. Abran. An analysis of the designs and the definitions of the halsteads metrics. In *Proc. 15th Int. Workshop Softw. Meas.*, 2005.

[56] W. J. Rugh. *Linear System Theory*. Prentice Hall, second edition, 1995.

[57] M. Satyanarayanan. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8(4):10–17, 2001.

[58] E. Schrödinger. Die gegenwärtige situation in der quantenmechanik (The present situation in quantum mechanics). *Naturwissenschaften*, 23:807–812; 823–828; 844–849, 1935.

[59] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[60] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[61] J. Shao and Y. Wang. A new measure of software complexity based on cognitive weight. *Can. J. Elect. Comput. Eng.*, 28(2):69–74, 2003.

[62] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2005.

[63] K-C. Tai. A program complexity metric based on data flow information in control graphs. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 239–248, 1984.

[64] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IRI'04: proceedings of IEEE Internation Conference on Information Reuse and Integration*, pages 483–498. IEEE Computer Society, 2004.

[65] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, 2008. Lecture Notes in Computer Science, Vol. 4949, pages 1–38, Springer.

[66] J. Tretmans and E. Brinksma. Torx: Automated model-based testing. In *First European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[67] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.

[68] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, 1991.

[69] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Trans. Softw. Eng.*, 5(1):45–50, 1979.

[70] G. Xie and Z. Dang. Testing systems of concurrent black-boxes - an automata-theoretic and decompositional approach. In *FATES'05: Proceedings of the 5th International Workshop on Formal Approaches to Software Testing*, 2006. Lecture Notes in Computer Science, Vol. 3997, pp. 170–186, Springer.

[71] L. Yang, Z. Dang, and T. R. Fischer. Information gain of black-box testing. *Formal Aspects of Computing*. To appear.

[72] L. Yang, Z. Dang, and T. R. Fischer. An information-theoretic complexity metric. Submitted.

[73] L. Yang, Z. Dang, T. R. Fischer, M. S. Kim, and L. Tan. Entropy and software systems: towards an information-theoretic foundation of software testing. In *FoSER'10: Proceedings of the Workshop on Future of Software Engineering Research*, pages 427–431, 2010.

[74] L. Yang, Z. Dang, and O. H. Ibarra. Bond computing systems: a biologically inspired and high-level dynamics model for pervasive computing. In *UC'07: Proceedings of the 6th International Conference on Unconventional Computation*. Lecture Notes in Computer Science, Vol. 4618, pp. 226-241, Springer, 2007.

[75] L. Yang, Z. Dang, and O. H. Ibarra. On stateless automata and P systems. *Int. J. Found. Comput. Sci.*, 19(5):1259–1276, 2008.

[76] L. Yang, Z. Dang, and O. H. Ibarra. Bond computing systems: a biologically inspired and high-level dynamics model for pervasive computing. *Natural Computing*, 9(2):347–364, 2010.

[77] L. Yang, Y. Wang, and Z. Dang. Automata on multisets of communicating objects. In *UC'08: Proceedings of the 7th International Conference on Unconventional Computation*. Lecture Notes in Computer Science, Vol. 5204, pp. 242-257, Springer, 2008.

[78] L. Yang, Y. Wang, and Z. Dang. Automata and processes on multisets of communicating objects. *Natural Computing*, 9(4):865–887, 2010.

[79] M. Zhang and N. Baddoo. Performance comparison of software complexity metrics in an open source project. In *EuroSPI*, volume 4764 of *Lecture Notes in Computer Science*, pages 160–174. Springer, 2007.

[80] H. Zhu and P. A. V. Hall. Test data adequacy measurement. *Softw. Eng. J.*, 8(1):21–29, 1992.

[81] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.