Entropy and Software Systems: Towards an Information-Theoretic Foundation of Software Testing

Linmin Yang, Zhe Dang, Thomas R. Fischer, Min Sik Kim and Li Tan
School of Electrical Engineering and Computer Science
Washington State University, Pullman, WA 99164, USA
{lyang1, zdang, fischer, msk}@eecs.wsu.edu, litan@tricity.wsu.edu

ABSTRACT

We integrate information theory into software testing. In particular, we use entropy in information theory to measure the amount of uncertainty in a software system before it is fully tested, and we show how the amount decreases when we test the system. Moreover, we introduce 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 seek practical approaches in testing real systems using the above theories, and we apply our novel approaches in testing model-based embedded systems and network intrusion detection systems. Our information-theoretic approach is syntax-independent, which is a desired property in software testing.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, software science*

General Terms

Measurement, Theory

Keywords

software testing, information theory, syntax-independent

1. INTRODUCTION

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 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? Entropy is specifically used to measure the amount of uncertainty in an object in information theory [10, 6], 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 in software systems. In reality, people use probabilities to measure the distribution of an object over a probabilistically measurable space, and it is a useful way to handle the uncertainty. Probabilities may not be meaningful to depict a specific object, but they are statistically correct for a large number of objects. 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 section.

What merits can this information-theoretic approach bring to software testing? This approach provides a syntactic-independent coverage criterion for software testing, since the Shannon entropy of a discrete random variable remains unchanged after a one-to-one function is applied [6]. Such a characterization is of great importance. 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. One component C_1 is modeled using statecharts [7] in standardized modeling language UML [1], while the other component C_2 is modeled using logical expressions such as LTL formulas [4]. Suppose that we use the branch-coverage [3] criterion and the property-coverage criterion [11] 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 approach can help us develop optimal testing strategies in choosing test cases. If a syntax-based test adequacy criterion 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. Finally, 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.

2. HOW TO APPLY THE CONCEPT OF ENTROPY IN SOFTWARE TESTING?

2.1 Software System Modeled as Random Variable

In this subsection, the system under test is a reactive black-box, where we can only observe its input-output behaviors. In this context, the objective of software testing is to test whether the observable behaviors of a reactive black-box software system conform with a set of sequences of events. The set is called a *trace-specification*, which specifies the observable behaviors that the system under test is intended to have. The original trace-specification could be an infinite language. However, in the real world, we can only test a finite length of the input sequence. Therefore, we simply assume that the trace-specification we are to test is finite. Given a trace-specification P of a system under test, we can use a *trace-specification tree* T to represent P. Each edge in T is labeled with an event, and each path (a walk from the root to a leaf or nonleaf node) in T corresponds to a sequence of events in P. For example, Figure 1 is the trace-specification tree T corresponding to the trace-specification $P = \{e_1, e_1e_2, e_1e_3, e_1e_4, e_5, e_5e_6, e_5e_6e_7\}$.

A sequence in the trace-specification P (also a path in the corresponding trace-specification tree T) is a test case sent to the system. Suppose that there is a test oracle that could show us the longest prefix of the path such that the prefix is an actual observable behavior of the system. We will mark every edge (if any) in the prefix as connected and mark the remaining edges on the path as disconnected. All of the connected edges in the trace-specification tree T, which correspond to all of the actual observable behaviors

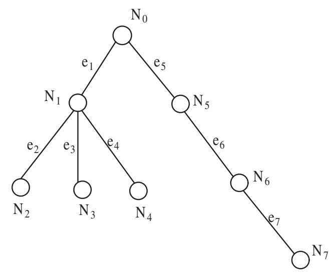

Figure 1: An example trace-specification tree T.

of the system with respect to the trace-specification, form a *system tree*. However, before any testing is performed, we do not know exactly what the system tree is, except that it is a subtree that shares the same root with the trace-specification tree. Adopting the idea of entropy in information theory, we model the system tree (which we do not clearly know before testing) as a random variable X_T, whose sample space Ω is the set of all subtrees that share the same root with the trace-specification tree T. The entropy of the trace-specification tree is simply defined to be the entropy of the random variable X_T which, intuitively, describes the amount of information in the system under test with respect to the given potential observable behaviors in the trace-specification tree T (i.e., the trace-specification P).

To calculate the entropy of X_T, we need its probability distribution. We use $p(t, T)$ to denote the probability of a subtree t being the system tree (that shares the same root with T). The Shannon entropy of X_T, simply written $H(T)$, is

$$H(T) = -\sum_{t \prec T} p(t, T) \log p(t, T),$$

where $t \prec T$ denotes that t is a subtree of T and they share the same root. Probabilities of edges could be pre-assigned and then we can calculate the probability $p(t, T)$ for each subtree t and the entropy $H(T)$ as well. 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). For the trace-specification tree T in Figure 1, the probabilities of edges that make $H(T)$ maximum are $p^*(e_2) = p^*(e_3) = p^*(e_4) = p^*(e_7) = 1/2$, $p^*(e_1) = 8/9$, $p^*(e_5) = 3/4$ and $p^*(e_6) = 2/3$, and the maximal entropy is $H(T) = 5.17$ bits.

After testing a test case in the trace-specification tree, we will know that some edges are connected while some are not. This knowledge decreases the entropy $H(T)$. We use the entropy reduction as a testing coverage criterion, and every time we test the system, we choose the test case that can reduce the entropy most, i.e., archives the largest amount of gain. In our previous work [12], we give three different optimal testing strategies to select test cases. For the trace-specification tree T in Figure 1, suppose that we can only test three branches. The optimal testing strategy is e_1, e_2, e_3.

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, we could assume that an edge in the trace-specification tree is connected with a probability that has a lower bound. For instance, we

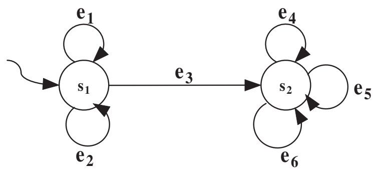

Figure 2: A labeled graph A.

can assume that every edge in Figure 1 is connected with probability at least 0.7. In this case, the probabilities that make $H(T)$ maximum are $p(e_1) = 0.86$, $p(e_5) = 0.74$ and $p(e_i) = 0.7$ for $i = 2, 3, 4, 6, 7$, and the maximal entropy with the addition information is $H(T) = 4.79$ bits. Suppose that we can only test three branches. With the additional information, the optimal testing strategy is e_1, e_2, e_5.

A trace-specification tree can be unrolled from a labeled graph which serves as a specification of the black box under test. Existing testing criteria like branch coverage are to select test cases to cover edges in the graph. What are the implications of using branch coverage as a testing criterion? Interestingly, we can prove the following: for every nontrivial (i.e., containing at least two paths of length > 1) labeled graph, if every test set that achieves the same branch coverage has the same entropy reduction, then the black box under test cannot be black (i.e., one must know additional information about the black box). Then, what kind of information is already known? How can such information be integrated into black-box testing?

2.2 Software System Modeled as Random Process

In the previous subsection, we model the system under test as a random variable, and calculate its maximal entropy. In this subsection, we will provide another approach to model the complexity of a software system. Here, specifications of software systems are modeled as graphs instead of trees.

Let A be a labeled graph with a designated initial node. A behavior of A is a concatenation of labels on a path starting from the initial node. The behavior set $L(A)$ is the set of behaviors of A. A labeled graph A can be interpreted as a control flow graph or design specification of a software system, and therefore the behavior set $L(A)$ can be interpreted as the set of desired behaviors of the software system. Let $N(n)$ be the number of behaviors of length n in the behavior set $L(A)$. We define the *behavioral complexity* of A as

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

The behavioral complexity 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. The labeled graph A can be uniquely transferred to a Markov chain M_A, and we can prove that the behavioral complexity $\mathcal{C}(A)$ is the maximal achievable entropy rate of the Markov chain M_A [13]. For instance, the behavioral complexity of the labeled graph A in Figure 2 is $\mathcal{C}(A) = 1.58$ bits.

A labeled graph A can be updated by adding or deleting edges, resulting in a new graph A'. Since the behavioral complexity can be interpreted as the cost of exhaustive testing, we can compare the behavioral complexities $\mathcal{C}(A)$ and $\mathcal{C}(A')$ and then analyze the impact on testing due to the update. For instance, if we drop the edge e_4 in Figure 2 and get a new graph A', the behavioral complexity would decrease to $\mathcal{C}(A') = 1$ bit. We say e_4 is a *critical edge* since it can change the complexity of the graph. However, the edge e_1 is not a critical edge since dropping it will not affect the behavioral complexity. This can also be generalized to the concept of a *critical subgraph*. For instance, the subgraph with node s_2 and edges e_4, e_5, e_6 is a critical subgraph of A, while the subgraph with node s_1 and edges e_1, e_2 is not. This approach gives us a way to predict whether an update would make a system harder to understand/test, or not.

Behavioral complexity can also give us many insights in testing component-based systems. In the McCabe metric [9], which is a widely accepted complexity metric, the complexity of a graph is monotonic in the number of nodes and edges. In our behavioral complexity, we can show that only loops (but not all loops) can increase the behavioral complexity when units are sequentially composed together. This result could be very helpful for software development. The metric can be used to predict the complexity of the semantics of a software system to be built from units, and it can also help testers properly distribute testing cost among units of a software system.

3. APPLICATIONS

In this section, we study how to apply the aforementioned information-theoretic approaches to the testing of practical software systems. In particular, two types of systems are studied, namely, model-based embedded systems and network intrusion detection systems.

3.1 Testing for Embedded Systems

We will apply our entropy-based testing to the development of model-based embedded systems. These systems typically contain both digital and analog components, and they continuously interact with their environment. Examples of such hybrid and reactive systems include Engine Control Module (ECM) in automobiles and Autopilot System in aircrafts. These systems are widely used in safety-critical applications. Testing still remains a predominant verification and validation (V&V) method for assuring the quality of these systems. In fact, testing is an important component in software quality standards such as RTCA DO-178B/EUROCAE ED-12B [5], the standard used in aerospace industry for certifying avionic software. Traditional coverage criteria such as MCDC emphasize on the structure of a system. They are susceptible to implementation changes but not sensitive to the behaviors of the system. In contrast, our entropy-based testing criterion focuses on the actual semantics of the system. This is especially valuable for safety-critical embedded application, since our entropy-based testing can guide testing activity towards verifying system requirements.

Entropy-based testing can be applied to model-based embedded design. Model-based design has been adapted by research and engineering community as a way to tame the complexity of hybrid embedded system designs. For our case study, we will choose StateChart [7] as the targeted design notation. StateChart is widely used in system design, and it is also the basis for a variety of design languages such as Charon [2] and Stateflow [8]. In model-based design, a design model serves as high-level executable specification of a system. There are several ways in which our approach may be used for model-based design: 1) following the discussion in Section 2.2, we can identify "critical components" of a StateChart. Locating these components will help designers understand the source of semantics complexity, and also help V&V activities being centralized around these components; 2) Our information-theoretic approach may be extended to generate test cases for a StateChart design with probabilities. These probabilities designate the likeness of firing enabled transitions. The probabilities may be decided by usage profiles and/or given as part of requirement specification. Following the discussion in Section 2.1, we can generate test cases that quickly reduce the entropy of the system under test. Intuitively, these test cases expedite the removal of uncertainty in (understanding) the behaviors of the system. Such behavioral uncertainty is a major source of concern in verifying safety-critical applications.

3.2 IDS Evaluation

A network intrusion detection system (IDS) monitors network traffic to identify suspicious packets. It is given a set of rules, which describe properties and behaviors of malicious traffic. An IDS compares each incoming network packet with those rules, and decides an action for the packet; for example, it may pass the packet, alert the network administrator, or reject the packet. Because of many types of attacks and their variations in the Internet, the number and complexity of rules have been increasing significantly in recent years. Therefore, an IDS employs sophisticated data structures to perform rule matching efficiently to catch up with the line speed of the network. An IDS has evolved from a simple pattern matching program into a complex software system, and testing an IDS has become a challenging task.

Testing and evaluating an IDS is typically conducted by running it on a testbed network with randomly generated traffic. This process is similar to a black-box testing in that we can test whether the response of the IDS to each packet conforms to the rules. Therefore, the technique in Section 2.1 can be used to generate test packets in this process. However, testing correctness is only a small part of the IDS evaluation. More critical and more challenging is to study its behaviors under heavy load. In today's high-speed networks, the packet processing time of an IDS often exceeds the corresponding transmission delay, which forces the IDS to discard or skip packets. This creates a potential security hole because the IDS may fail to identify attacks in such a case. Therefore, it is crucial to understand how the IDS responds to intense traffic and how its performance degrades as the load increases. To study the relationship between the traffic load and the IDS performance, we need to generated traffic with different levels of intensity.

Varying intensity in traffic generation is not straightforward. A naïve way is to change the packet rate, the number of packets per unit time. However, that is insufficient because the load incurred by a packet depends on which execution path it takes in an IDS. With today's large rule set, it also depends on preceding packets; if successive packets follow the same execution path, an IDS spends little time on the latter because the execution path and related data already reside in the cache memory. Hence, we need to generate a good mixture of packets that takes various execution paths in the IDS, and for this purpose, the entropy defined in Section 2.1 will be able to indicate the potential load of the generated traffic on the IDS as follows.

The trace-specification tree T is built from the IDS rules. Because of the nature of rules, listing conditions for multiple fields in a packet, the resulting tree will look like a decision tree. Then for each packet in the generated traffic, identify a path that it will follow, and increment the frequency of each edge along the path. We use this frequency divided by the total number of packets as an estimation of $p(e_i)$ for each edge e_i. After obtaining every $p(e_i)$, we can compute $H(T)$, which is an approximation of the load incurred by the given traffic. Because $H(T)$ does not depend on the IDS implementation, it is useful not only in evaluating a single IDS but also in comparing different IDSs.

4. CONCLUSION

We define a semantics-based software testing criterion, independent of the syntax of software systems, and we also introduce a novel complexity metric for labeled graphs (which can be interpreted as control flow graphs, design specifications, etc.) named behaviorial complexity. We provide an information-theoretic foundation for both of them, where software systems are modeled as either random variables or random processes. We also seek practical approaches for optimal testing in the sense of entropy reduction in the context of model-based design and for testing network intrusion detection systems. Our information-theoretic approach will significantly advance the understanding of the fundamental side of software testing. Additionally, our information-theoretic approach could have applications in other areas than software testing:

- software understanding from a specification. Our approach may be upon the specification to identify "difficult" parts of a software,

- data mining on structural data. Our approach may provide a new data mining approach that is based on the structures of the data instead of their appearance,

- threat analysis. Currently, we are working on a similar information-theoretic approach in detecting a behavioral threat.

5. REFERENCES

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

[2] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivancic, V. Kumar, P. Mishra, GJ Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, 2003.

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

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

[5] SC-167 committee. Software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission for Aeronautics, 1992.

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

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

[8] MathWorks. Simulink and stateflow. http://www.mathworks.com.

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

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

[11] 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.

[12] L. Yang, Z. Dang, and T. R. Fischer. Information gain of black-box testing. Submitted.

[13] L. Yang, Z. Dang, and T. R. Fischer. A syntax-independent complexity metric. In *proceedings of TMFCS'10*, pages 127–134, 2010.