A probabilistic coverage for on-the-fly test generation algorithms

N. Goga,
Department of Mathematics and Computing Science,
Eindhoven University of Technology,
P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands
goga@win.tue.nl

January 16, 2003, version 0.1

Abstract

This paper describes a way to compute the coverage for an *on-the-fly* test generation algorithm based on a probabilistic approach. The *on-the-fly* test generation and execution process and the development process of an implementation from a specification are viewed as stochastic processes. The probabilities of the stochastic processes are integrated in a generalized definition of coverage. The generalized formulas were instantiated for the *ioco* theory and for the specification of the TorX test generation algorithm.

Systematic testing is an important technique to check and control the quality of software systems. Testing consists of systematically developing a set of experiments or test cases and then running these experiments on the software system that has to be tested, i.e. the IUT. The subsequent observations made during execution are used to determine whether the IUT behaved as expected leading to a verdict about the IUT's correctness.

In a traditional approach the test generation and test execution are separated phases of the test experiment. We refer to such an approach as *batch-oriented*. A newer technique for test derivation is to combine the test generation and the test execution in one phase. We refer to such a technique as *on-the-fly*. An example of a tool which works on-the-fly is TorX [BFdV+99].

An on-the-fly algorithm derives and runs a finite number of tests which can detect a certain amount of errors, with others going undetected. In order to express this, as ITU-T Z.500 recommends, the concept of *coverage measure* should be used. An example of such a coverage measure which is useful for a set of test cases can be found in [HT96].

The coverage measure from [HT96] has a probabilistic nature and it can be a applied for a batch-oriented test suite. We will explain below the stochastic nature of this coverage. Due to the non-determinism of the IUT, each run of a test leads to a different outcome and the outcomes of several, independent test runs make up one observation. So, some outcomes are more likely to occur than others. The probability of an outcome to occur can be thought of as depending on the frequency with which the implementation resolves the nondeterministic choices leading to the different outcomes. This leads to the consideration of the occurrence of an outcome of a test run as a stochastic experiment. Also, the design and the implementation process of a complex system are viewed as a stochastic experiment. For example when designing a juice machine which is supposed to deliver juice and tea it is less likely to end up with a completely different kind of machine (as for example a washing machine) than to end up with a slightly different, but possibly incorrect, juice machine. Moreover, assuming that an implementer can make several independent mistakes with non-zero probability and while trying to do his task as good as possible, it is less likely for the implementer to make all possible mistakes than to make only a small number of mistakes. These are some examples which show that not every implementation has the same chance to occur as the result of 'implementing' a given specification. Also, to express the severity of the bugs in implementations, a weight assignment is added to the implementations. These three ingredients: the probability distribution of the outcomes produced by the test runs, the

probability distribution of the implementations and the severity of the bugs in implementations form the base of the probabilistic coverage from [HT96].

The existing theory was useful for batch-oriented testing. Modern test process turns away from batch-oriented to on-the-fly. Therefore we generalized the existing theory from [HT96] for on-the-fly algorithms. We adopted the same views as [HT96] regarding the probabilistic distribution of IUTs and the weights assigned to implementations. Regarding the probability distribution of the outcomes, first we remind that the test generation and test running are integrated in one phase by an on-the-fly algorithm. Therefore, the probabilistic distribution for outcomes has two independent sources, in this case. First, the non-determinism of the IUT gives a probabilistic nature to outcomes. Secondly, the test generation algorithm itself can have a probabilistic component (as it is the case for TorX). The probabilistic nature of TorX comes from its three non-deterministic choices, each choice could be chosen with a given probability at a given moment in the generation process. The probabilistic nature of the algorithm itself is another reason to add probabilities to outcomes. In [FGM00], we showed how to compute the probabilities of the outcomes for TorX (the outcomes are traces in this case). In a similar way, the probabilities of the outcomes can be computed for other test generation algorithms. With these being said, our coverage for an on-the-fly algorithm is defined as follows. The severity of the bugs in implementations, the probability distribution of the IUT and the probability distribution of the test outcomes which is seen as a result of an integrated test generation and running process are combined in a probabilistic coverage formula for an on-the-fly algorithm. The coverage is parameterized with the number of tests derived. The abstract coverage formula is instantiated for the *ioco* theory of test derivation using results from [FGM00].

The paper is organized as follows. Section 1 introduces a framework for the definition of a coverage measure and the assumptions on which the theory presented is based. Section 2 formally defines a weight assignment to each implementation and the probability of an implementation to occur. In Section 3 the coverage formula is defined and The conclusions are presented in Section 4.

1 Automated testing

In this section we will present in more detail some basic notions regarding conformance, testing and test generation. We will give also the assumptions on which we base the coverage measure theory for on-the-fly test generation algorithms.

Conformance The conformance of an implementation under test (IUT) with respect to a specification implies the assumption of a universe of implementations IMPS and a universe of formal specifications SPECS. Implementations are concrete, informal objects, such as pieces of hardware, or pieces of software. In order to be able to reason in a formal way about them, it is assumed that each implementation IUT $\in$ IMPS can be modelled by a formal object i_{IUT} in a formalism MODS, which is referred to as the universe of models. Conformance is expressed by means of an implementation relation **imp** $\subseteq$ MODS $\times$ SPECS. We use $I_s =_{\mathrm{def}} \{i \in \mathrm{MODS} \mid i \ \mathbf{imp}\ s\}$ for the set of **imp**-correct implementations of s, and $\overline{I_s} =_{\mathrm{def}} \mathrm{MODS} \setminus I_s$ for its complement.

Testing The test cases, sometimes denoted simply as tests, are formally specified as elements of a universe of test cases TESTS. A set of test cases is called a test suite. The process of running a test repeatedly against an implementation is called test execution. Each test run leads to an outcome. Test execution leads to an observation, which is the set of all possible outcomes produced by the test runs, in a domain of observations OBS. Let $\mathcal{O}$ be the set of possible test-run outcomes, then an observation is a set of outcomes: $\mathrm{OBS} = \mathcal{P}(\mathcal{O})$. To each outcome a verdict is assigned by a verdict assignment function $v_t : \mathcal{O} \to \{\mathbf{pass}, \mathbf{fail}\}$, where t is a test. The verdict of an observation will be **pass** if all outcome-verdicts are **pass**. Test execution is modelled by a function $exec : \mathrm{TESTS} \times \mathrm{MODS} \to \mathrm{OBS}$.

On-the-fly test generation An on-the-fly algorithm combines two phases, the test generation

and the test running, into one common phase. We model this mathematically by considering the generation of a test and the run of it as being formalized by the function *genexec*. The test execution is already modelled by the *exec* function, so that we should proceed with formalizing the test generation. Let ALGS be the universe of all the algorithms that generate and run tests on-the-fly. The process of generating and running tests by an on-the-fly algorithm will be called on-the-fly test generation and execution. An on-the-fly algorithm $A \in$ ALGS is applied to a specification $s \in$ SPECS and to an implementation $i \in$ MODS. The set of all the tests, which we will denote as $A(s,i)$, derived with A when it uses the specification s and the implementation i is a subset of TESTS, i.e. $A(s,i) \subseteq$ TESTS. Now the observation made by an on-the-fly algorithm A is the set of all outcomes which can be produced by the runs of all the tests which can be generated with A, which is $\cup_{t \in A(s,i)} exec(t,i)$. Then let *genexec*: ALGS $\times$ SPECS $\times$ MODS $\rightarrow$ OBS be the function that correctly models the test generation and execution of an algorithm on-the-fly, then $genexec(A,s,i) = \cup_{t \in A(s,i)} exec(t,i)$.

The assumptions for on-the-fly test-generation Now we will introduce the assumptions which are at the base of all the coverage computations. We have the same probabilistic approach as in [BTV91] and [HT96], but our contribution is in extending it by considering the test generation and execution as a stochastic process, and in instantiating it for the *ioco* theory and for the specification of the TorX test-generation-algorithm. We focus on on-the-fly testing.

- $\mathcal{A}$*:* we assume that the occurrence of an implementation i is the outcome of a stochastic experiment; we adopt the notation $p_s(i)$ from [HT96] to denote the probability that implementation i occurs;
- $\mathcal{B}$*:* we assume that the generation of a test and the running of the test against an implementation i by an on-the-fly algorithm A, which uses a specification s, is a stochastic experiment; in particular $p_{A,s,i}(\sigma)$ is the probability that A generates a test which leads to outcome σ when it runs the test against i and uses a specification s.

2 Valuation of the implementation

Weight of implementations To express the importance of each implementation, a weight assignment ([HT96]) is added to each implementation. Let $s \in$ SPECS be a specification, and **imp** $\subseteq$ MODS $\times$ SPECS an implementation relation, then a function w : MODS $\rightarrow \mathbf{R} \setminus \{0\}$ is a weight assignment function on MODS with respect to s and **imp**, if for all $i \in$ MODS:

$$w(i) > 0 \Leftrightarrow i \textbf{ imp } s \tag{1}$$

A weight assignment assigns a positive real number to each conforming implementation and a negative number to each erroneous implementation. For conforming implementations the weight can express that one implementation is better than another; negative weights express the gravity of errors in erroneous implementations: if $w(i_1) < w(i_2) < 0$ then both i_i and i_2 are not correct, but the errors of i_1 are more severe than those of i_2.

Example Let us consider the automaton *spec* of the specification from Figure 1. The specification has four states and the initial state is *I*. The set of inputs is $L_I = \{a\}$ and the set of outputs is $L_U = \{b,c\}$. In *I* the specification can receive the input a, produce the output b and after that c and arrive back in *I* or it can perform the c output. After performing c, the specification can receive a, produce c and arrive back in *I*.

In the same figure, two erroneous implementations i_1 and i_2 are shown. The implementation i_1 has one bug: after c in the initial state, it can produce the unspecified outputs b and c. The implementation i_2 has more problems: it has the same bug as i_1 to which is added the bug of the state *III* where it can produce not only the output b but also the other output c. So intuitively the weight of i_1 could be $w(i_1) = -1$ and of i_2 could be $w(i_2) = -2$ because these implementations

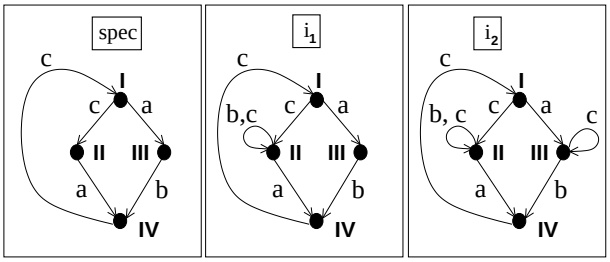

Figure 1: Example of a specification and two erroneous implementations.

have one and two bugs, respectively.

Probability of implementations In conformance with assumption $\mathcal{A}$ an implementation i occurs with a probability $p_s(i)$. We consider MODS discrete (MODS must be finite or countably finite). Let $I \subseteq$ MODS, later to be used as follows: I is the set of all erroneous implementations. Then:

$$P_s(I) =_{\text{def}} \sum_{i \in I} p_s(i) \tag{2}$$

means the probability that the activity of implementing specification s produces an implementation that is modelled by a member of I.

Valuation Using the probability p_s and the weight assignment w it is possible to define a valuation on a discrete set of implementations (which gives the importance of a set of implementations I in terms of their weight and their probability of occurrence):

$$\mu(I) =_{\text{def}} \sum_{i \in I} w(i) p_s(i) \tag{3}$$

3 The coverage for on-the-fly test generation

Test generation and test run On-the-fly combines the test generation with the test run. For the remainder of this paper we will abuse the words *test generation* for on-the-fly test generation and test run. Let $A \in$ ALGS be an algorithm for on-the-fly test generation. Let $\mathcal{O}$ be the set of possible test-run outcomes and let on-the-fly test generation and execution be correctly modelled by *genexec*: ALGS $\times$ SPECS $\times$ MODS $\to \mathcal{P}(\mathcal{O})$. Please note that $\mathcal{P}(\mathcal{O}) =$ OBS.

In accordance with assumption $\mathcal{B}$ running the algorithm A against an implementation i and using the specification s produces an outcome $\underline{\sigma}(A, s, i)$ from the set $genexec(A, s, i)$ with a probability $p_{A,s,i}(\sigma)$, with $\sigma = \underline{\sigma}(A, s, i)$. Note that the distribution of the variable $\underline{\sigma}(A, s, i)$ depends on the algorithm A, the specification s and the implementation i; for each A, s, i it may have another distribution. For a subset $O \subseteq genexec(A, s, i)$, the probability distribution is

$$p_{A,s,i}(O) =_{\text{def}} P(\underline{\sigma}(A, s, i) \in O) \tag{4}$$

The verdict for the observation will be **pass** if all outcome-verdicts are **pass**. Then let $v_t : \mathcal{O} \longrightarrow \{\textbf{pass}, \textbf{fail}\}$ be a verdict assignment to outcomes, where t is a test generated with A; the probability measure $p_{A,s,i}$ can induce a probability measure on the set of verdicts {**pass**,**fail**}. The probability that a single test generation of A with s and i results in a verdict **pass** is the cumulative probability that an outcome in $genexec(A, s, i)$ leads to a **pass** verdict.

$$\begin{aligned} p_{A,s,i}(\{\textbf{pass}\}) &=_{\text{def}} P(v_t(\underline{\sigma}(A, s, i)) = \textbf{pass}) \\ &= p_{A,s,i}(\{\sigma \in genexec(A, s, i) \mid v_t(\sigma) = \textbf{pass}\}) \end{aligned} \tag{5}$$

If the algorithm A will run more times against the implementation i, say m times, under the assumption that the test generations are independent, then the implementation passes all the test generations only if it passes all the individual test generations.

$$p^m_{A,s,i}(\{\textbf{pass}\}) = (p_{A,s,i}(\{\textbf{pass}\}))^m \tag{6}$$

The probability to **fail** is: $p^m_{A,s,i}(\{\textbf{fail}\}) =_{\text{def}} 1 - p^m_{A,s,i}(\{\textbf{pass}\})$. It can be shown that $p^m_{A,s,i}$ has the property that for non-zero probability of **fail** in a test generation, the algorithm will finally result in **fail** if enough test generations are performed (this property is expressed in the following proposition).

Proposition 3.1 *If* $p^1_{A,s,i}(\{\textbf{fail}\}) > 0$ *then* $\lim_{m\to\infty} p^m_{A,s,i}(\{\textbf{fail}\}) = 1$
(without proof)

The on-the-fly coverage The probability of occurrence of implementations was expressed by the probability measure P_s on the set of implementations (2). The probability that an implementation i yields the verdict **pass** with algorithm A was expressed by the probability measure $p_{A,s,i}$ (6). Under the assumption that the probability of occurrence of implementations is independent of the probability of yielding a **fail** we can integrate these measures to obtain the probability measure on MODS $\times\{\textbf{pass}, \textbf{fail}\}$. Taking also the weight of implementation into account we obtain analogously the valuation measure λ:

$$\lambda^m_{A,s}(I, V) = \sum_{i\in I} \sum_{v\in V} w(i) p^m_{A,s,i}(v) p_s(i) \tag{7}$$

Let $\overline{I_s}$ be the set of non-implementations of s. We define the coverage of A, applied m times on a specification s, as

$$cov(A, s, m) =_{\text{def}} \frac{\lambda^m_{A,s}(\overline{I_s}, \{\textbf{fail}\})}{\lambda^\infty_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\})} \tag{8}$$

where by $\lambda^\infty_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\})$ we mean $\sum_{i\in\overline{I_s}} w(i)p_s(i)$ (we assume that an erroneous implementation occurs with a non-zero probability, formally $P_s(\overline{I_s}) > 0$).

In other words $cov(A, s, m)$ is the weighted probability of being able to conclude **fail** divided by the probability of an erroneous implementation to occur. The coverage is a function of the number of tests produced by the algorithm and it follows immediately from the definition that it has all the values in the range $[0, 1]$.

Example So for example, if on average one out of three implementations is erroneous and if we assume $w(i) = -1$ for all implementations i, then $\lambda^\infty_{A,s}(\overline{I_s}, \{\textbf{pass}, \textbf{fail}\}) = -\frac{1}{3}$. Taking an arbitrary implementation and performing one test run will yield **fail** with a probability $\frac{1}{30}$, for example. So in $\frac{1}{3}$ of all cases we encounter an erroneous implementation and then we observe the bug in one out of ten cases on average. Then $\lambda^1_{A,s}(\overline{I_s}, \{\textbf{fail}\}) = -\frac{1}{30}$. So $cov(A, s, 1) = 0.1$, as expected. Moreover $\lambda^2_{A,s}(\overline{I_s}, \{\textbf{fail}\}) = -\frac{1}{3}(1-(1-\frac{1}{10})^2) \approx -0.063$. So $cov(A, s, 2) = \frac{-0.063}{-0.333} = 0.19$.

In this easy example we see that $cov(A, s, m)$ is monotonic in m and $\lim_{m\to\infty} cov(A, s, m) = 1$. As we will see, this holds in general under very reasonable conditions, which we set out to explore in this paper. So for a non-zero distribution for $p^m_{A,s,i}$ and P_s the coverage has the monotonicity property. This property of the coverage is expressed in the following proposition.

Proposition 3.2 *Let* $s \in$ SPECS *be a specification and let* $\overline{I_s}$ *be the set of non-implementations of* s. *Let* $A \in ALGS$ *be an algorithm of on-the-fly test generation. Assume that an erroneous implementation occurs with a non-zero probability, formally* $P_s(\overline{I_s}) > 0$. *Assume that all faulty*

implementations that are possible can be detected, that is $\forall i \in \overline{I_s} : (p_s(i) > 0 \Rightarrow p_{A,s,i}(\{\textbf{fail}\}) > 0)$. *For positive integers* m *and* n

$$m < n \Rightarrow cov(A, s, m) < cov(A, s, n)$$

(Without proof)

4 Conclusions

This paper extends the work from [HT96] and [FGM00] with a coverage measure for an algorithm which generates and runs tests on-the-fly. The probability of sending outputs by the implementation and consequently of obtaining a verdict by the on-the-fly algorithm reflects the probabilistic nature of this coverage. The severity of the bugs in the erroneous implementation, the probability distribution of the implementations and the probability distribution of producing a **fail** by the on-the-fly algorithm are combined in the probabilistic coverage.

This theory was instantiated for the *ioco* theory and the TorX algorithm and an example for this instantiation was also worked out. The following results were obtained. When the number of test runs increases, the coverage increases arriving at the limit one, provided that there is a non-zero probability of **fail**. This expresses the expected property that after performing a sufficient quantity of test runs, the algorithm is able to detect at the end all bugs of an erroneous implementation. When decreasing the probability of detecting an error, the coverage also decreases, as expected. When increasing the weight of an arbitrary fault, the coverage follows the behaviours of the erroneous implementations which do contain the fault. Computer programs can be made for the computation of the coverage, as a part of test generation tools (such as TorX).

Our instantiation is based on the current version of TorX which works completely randomly. One interesting extension of this application is to enrich the algorithm with more sophisticated features such as permitting it to select a group of tests from the complete set of possible tests which leads to a higher probability of getting a **fail** and consequently to obtain a higher coverage for the algorithm. In other words, for our example, to give to the algorithm the possibility to answer the following type of questions: when we are allowed to have only three test runs what are the three traces from $\mathcal{F}$ which will be allowed to be executed such that the probabilistic coverage is the highest.

References

[BFdV+99] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Int. Workshop on Testing of Communicating Systems*, volume XII, pages 179–196. Kluwer Academic Publishers, 1999.

[BTV91] E. Brinksma, J. Tretmans, and L. Verhaard. A framework for test selection. In G. Jonsson, J. Parrow, and B. Pehrson, editors, *Protocol Specification, Testing, and Verification*, volume XI, pages 233–248, 1991.

[FGM00] L. Feijs, N. Goga, and S. Mauw. Probabilities in the TorX test derivation algorithm. In S. Graf, C. Jard, and Y. Lahav, editors, *SAM2000 – 2nd Workshop on SDL and MSC*, pages 173–188, Col de Porte, Grenoble. VERIMAG, IRISA, SDL Forum Society, 2000.

[HT96] L. Heerink and J. Tretmans. Formal methods in conformance testing: a probabilistic refinement. In B. Baumgarten, H. Burkhardt, and A. Giessler, editors, *Ninth International Workshop in Testing and Communication System*, volume IX, pages 261–276, 1996.