

A Method for Analysing the Performance of Testing Techniques for Concurrent Systems

Timo Kellomäki¹ and Antti Valmari²

¹ Tampere University of Technology Timo.Kellomaki@tut.fi

² Tampere University of Technology Antti.Valmari@tut.fi

Technical Report 37

23.11.2004

Abstract. In this paper we develop a method for analysing and comparing the performance of different testing techniques for concurrent systems, and use it to give some evidence that so-called “exploration testing” finds errors faster than traditional testing based on test cases. To compare testing methods, we model the system under test as a state space with a weight and cost assigned to each transition. We describe an algorithm that finds the probability and expected cost of reaching terminal states and the probability of reaching terminal strongly connected components. From this information, the probabilities and expected costs of finding errors using each method can be computed. A drawback of our method is that it is not feasible for arbitrarily large systems, but, in return, it gives results much quicker and with much higher precision than possible by running actual tests. We then present a small case study, where our method is applied to compare the efficiency of exploration testing with test cases to find a planted error from a simple protocol.

1 Introduction

Testing software completely is impossible. Thus testing should be organized so that as many errors as possible are found using the given limited resources. This is particularly important when trying to find errors that occur infrequently, which are especially common in concurrent systems.

Unfortunately, it is difficult to get information on the performance of different approaches to testing. In this paper, we develop a method for that purpose, and apply it to compare traditional test cases to so-called exploration testing. A drawback of our method is that it cannot be used for real systems, but only for their formal models. This is not crucial, however, because our method is not meant for testing itself, but for comparing testing techniques. An advantage of our method is that, unlike comparison of methods by trying them, it produces precise and extensive numerical data with a small amount of work.

Traditionally systems are tested using short test cases that test the system one feature at a time. This method is often also applied to concurrent systems. Because of timing and scheduling issues, concurrent systems are nondeterministic. That is, executing the same test case again can alter the outcome. Thus the same test case can sometimes reveal an error and sometimes not.

In an alternative approach, called exploration testing [2], a test engine automatically roams around the specification as it sees fit, generating test cases on the fly. This method

continues until an error is found or the resources are exhausted, thus producing much longer testing runs than the traditional method.

Exploration testing requires a formal model of the specification, which certainly limits the possible uses and increases the need of highly trained personnel. However, our feeling is that exploration testing is a potentially profoundly more efficient way to test concurrent systems than test cases are. There are essentially two reasons for this. Exploration testing avoids the need for running initialization and shutdown routines again and again. Additionally, exploration testing can change what is being tested on the fly. An execution of a test case, on the other hand, can turn out fruitless because of the verdict “inconclusive”. The verdict is caused by a legal but unexpected response because of nondeterminism.

It is interesting to try and obtain numerical information on the efficiency of exploration testing, especially when the system under test is concurrent and nondeterministic. To compare exploration testing with test case based testing, we did a formal experiment. We wrote a formal model of a broken version of the famous alternating bit protocol [1] together with a testing environment, and computed numerical results from it. The results are of three kinds: the probability of finding an error when executing a test case once, the expected amount of effort for finding an error by repeating the test case until an error is detected, and the expected amount of effort of running exploration testing until an error is found.

We use the well-known notion of the state space of the system. The state space of a concurrent system is a formal model of its behaviour. It is a directed graph whose nodes correspond to the states the system can reach in any execution, and arcs correspond to transitions between states. Because of nondeterminism, the state space usually has states that have multiple outgoing transitions, of which one is chosen during a system run. A choice between alternative transitions may correspond to a “fundamental” choice like selecting between passing a message through a channel or losing it; or it may represent details of scheduling or relative timing of events.

Both of these phenomena can be modelled by giving each transition a probability distribution function for the time before it is executed after arriving in its begin state. Unfortunately, for most real systems these functions would be very complex and lead us into problems in the analysis. There is, however, a simple, yet satisfying probability distribution function, namely the negative exponential function. It is widely used, for example, in stochastic Petri nets [4]. The negative exponential function leads to assigning each transition a weight.

In our model, the weights are only used to model the probabilities of different choices. We also use costs. They can model execution time or other resources consumed during a test run. For instance, if we assign each transmission request the cost 1 and all other transitions the cost 0, the results reveal the expected number of transmission requests needed to find an error. On the other hand, the expected number of losses of messages is found by making the loss transitions cost 1 and others 0.

This paper introduces an algorithm that can be used to calculate the probability of a test run revealing an error, provided that we have a model with the appropriate weights and costs. It also finds the expected cost for the execution.

Appropriate weights are difficult to find. To compensate for that, we did our experiment with a wide range of weight combinations, and sought for findings that are valid for all of them. They have thus some general validity. One must take into account, though, that all our results were obtained with only one system under test. In this sense the work in this

paper is initial.

The rest of this paper is structured as follows. Section 2 formalizes the concept of the state space augmented with weights and costs. In Section 3 we present the algorithm for performance analysis and Section 4 shows how it can be used for comparing the performance of testing methods. Section 5 contains a small case study, where the algorithm is used to compare exploration testing with test cases.

2 Weight-Cost Transition Systems

As was mentioned in the introduction, we use a state space model of the system and its test environment. In addition to the states and transitions between them, we add a weight and cost to each transition.

In stochastic Petri nets [4], when the execution arrives to a state, each transition has a negative exponential distribution for the time before it gets enabled. The transition that is enabled first then gets executed. The expected mean value of this distribution is called the weight. It is the inverse of the single parameter of the negative exponential function. As shown in [4], the probability of executing a transition then becomes its weight divided by the sum of all weights of transitions leaving from the state.

Each transition also has a cost, which can represent any resource that we are interested in. For example, if we only want to know how many messages have to be sent on average before an error is found, we can set the cost of sending a message to one and all other costs to zero.

We normalize the weights of transitions of each state so that they sum to one. After this operation the weights can be directly interpreted as the probability of executing the transition after arriving to the corresponding state, which makes the computations simpler. Although our weights do affect relative timing of events, they do not contain information on execution times in absolute time units. The idea is that if such information is of interest, the costs can be used to model it.

Let us now formalize our model of the state space:

Definition 1 A *weight-cost transition system*, WCTS, is a 7-tuple $(S, T, B, E, C, W, \hat{s})$, where S is a set of states; T is a set of transitions; B and E are functions $T \mapsto S$, which define a begin and end state for each transition, respectively; C is a function $T \mapsto \{0\} \cup \mathfrak{R}^+$, which defines a cost for each transition; W is a function $T \mapsto \mathfrak{R}^+$, which defines a weight for each transition; and $\hat{s} \in S$ is an initial state. In addition, the extra condition (weight normalization) given below must hold.

Let s be a state in a WCTS. Let us denote $\bullet s = \{t \in T \mid E(t) = s\}$ and correspondingly $s\bullet = \{t \in T \mid B(t) = s\}$ for the sets of incoming and outgoing transitions of s . A state is a *terminal state* iff $s\bullet = \emptyset$. The weight normalization condition for a WCTS is $\forall s \in S : s\bullet = \emptyset \vee \sum_{t \in s\bullet} W(t) = 1$.

An execution of a WCTS is a sequence of transitions $t_1 t_2 t_3 \dots t_n$ such that $B(t_1) = \hat{s}$ and $\forall i; 1 \leq i < n : E(t_i) = B(t_{i+1})$. The probability of a finite execution $t_1 t_2 t_3 \dots t_n$ is $W(t_1 t_2 \dots t_n) = W(t_1)W(t_2)W(t_3) \dots W(t_n)$ and the cost of the execution is $C(t_1 t_2 \dots t_n) = C(t_1) + C(t_2) + C(t_3) + \dots + C(t_n)$.

The state space can be divided into *strongly connected components* (SCCs). An SCC is a maximal set of states such that any state in it can be reached from any other state in it by a sequence of transitions. An SCC is *terminal* if the execution cannot leave the SCC once that it has arrived in it.

Let X be an SCC. We denote with $R(X)$ the set of those executions $t_1 t_2 \dots t_n$ such that $E(t_n) \in X \wedge \forall i; 1 \leq i < n : E(t_i) \notin X$ that is reachable from the initial state (for the empty execution we replace \hat{s} for $E(t_n)$ in the definition). The probability of reaching X is $\sum_{\sigma \in R(X)} W(\sigma)$. The expected cost is $\sum_{\sigma \in R(X)} C(\sigma) W(\sigma)$.

3 The Algorithm

The algorithm is based on repeating three different operations, which make the WCTS smaller while still preserving the probability of the execution ending up in each terminal SCC. Additionally, each operation preserves the costs of reaching terminal states.

The algorithm is analogous to and inspired by the algorithm that is used to convert finite automata to regular expressions [3]. For each operation, we mention some important qualities of the WCTS that they conserve. The details of the proofs are straightforward (and dull) probability calculations, and are omitted.

We assume below that $P = (S, T, B, E, C, W, \hat{s})$ is a WCTS.

3.1 Removing Double Transitions

Let us assume that $\exists t_1, t_2 \in T : t_1 \neq t_2 \wedge B(t_1) = B(t_2) \wedge E(t_1) = E(t_2)$. We call (t_1, t_2) a *double transition*. They can be combined with a simple operation without changing the probabilities and costs of reaching different states.

$$\begin{aligned} \text{Remove-double-tr}(P, t_1, t_2) = \\ (S', T', B', E', C', W', \hat{s}'), \text{ where} \\ * S' = S, \\ * T' = T - \{t_2\}, \\ * \forall t \in T' : B'(t) = B(t), \\ * \forall t \in T' : E'(t) = E(t), \\ * C'(t_1) = \frac{C(t_1)W(t_1) + C(t_2)W(t_2)}{W(t_1) + W(t_2)}, \\ * \forall t \in T' - \{t_1\} : C'(t) = C(t), \\ * W'(t_1) = W(t_1) + W(t_2), \\ * \forall t \in T' - \{t_1\} : W'(t) = W(t), \\ * \hat{s}' = \hat{s}. \end{aligned}$$

$\text{Remove-double-tr}(P, t_1, t_2)$ produces a WCTS that, instead of t_1 and t_2 , only has t_1 . The weight and cost of t_1 are updated so that it represents the two transitions together.

Removing a double transition (t_1, t_2) with Remove-double-tr does not affect the probability or cost of reaching terminal SCCs.

An example of removing a double transition is presented in Fig. 1.

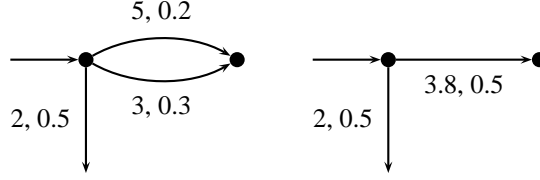


Figure 1. Removing a double transition. The original is on the left and the result on the right hand side.

3.2 Removing Self-Loops

A transition is a *self-loop* if and only if $B(t) = E(t)$. A self-loop can be removed if it is not the only transition leaving from a state. Let t_l be a self-loop and $B(t_l) = E(t_l) = s$. Now

$$\text{Remove-self-loop}(P, t_l) = (S', T', B', E', C', W', \hat{s}'), \text{ where}$$

- * $S' = S$,
- * $T' = T - \{t_l\}$,
- * $\forall t \in T' : B'(t) = B(t)$,
- * $\forall t \in T' : E'(t) = E(t)$,
- * $\forall t \in T' : C'(t) = \begin{cases} C(t) + C(t_l) \frac{W(t_l)}{1-W(t_l)}, & \text{if } t \in s\bullet \\ C(t), & \text{if } t \notin s\bullet \end{cases}$,
- * $\forall t \in T' : W'(t) = \begin{cases} \frac{W(t)}{1-W(t_l)}, & \text{if } t \in s\bullet \\ W(t), & \text{if } t \notin s\bullet \end{cases}$,
- * $\hat{s}' = \hat{s}$.

The operator updates the other transitions beginning from the state s so that the effect of t_l is included in them. On average, t_l will be executed $(1 - W(t_l)) \sum_{i=0}^{\infty} iW(t_l)^i = W(t_l)/(1 - W(t_l))$ times in a row before choosing some other transition t . Since $C(t_l)W(t_l)/(1 - W(t_l))$ is added to the cost of t by the operator, so the average cost of the execution coming to the state and then going out using the transition t remains the same.

Remove-self-loop does not affect the costs and probabilities of reaching terminal SCCs, because t_l cannot be the last transition of an element of $R(X)$.

An example of removing a self-loop is presented in Fig. 2.

3.3 Removing States

The idea of removing a state is to replace each pair of an incoming and an outgoing transition with a single transition that bypasses the state to be removed.

There are three preconditions that a state must meet before it can be removed. First, the initial state should not be removed. Second, the state must have at least one transition out to another state. Finally, no self-loops are allowed in the state.

If a state fulfills the first two conditions, self-loops can be removed as described in Section 3.2. Thus the only reachable states that cannot be removed are the initial state and such states that form a terminal SCC alone.

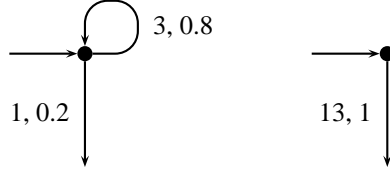


Figure 2. Removing a self-loop. Original on the left and the result on the right hand side.

Let us assume that s is a state that fulfills the preconditions. Additionally, let $\bullet s = \{b_1, b_2, \dots, b_n\}$ and $s\bullet = \{e_1, e_2, \dots, e_m\}$. Now

Remove-state(P, s) = ($S', T', B', E', C', W', \hat{s}'$), where

- * $S' = S - \{s\}$,
- * $T' = (T - s\bullet - \bullet s) \cup T_{\text{new}}$, where $T_{\text{new}} = \bullet s \times s\bullet$,
- * $B'(t) = \begin{cases} B(t), & \text{if } t \in T' - T_{\text{new}} \\ B(b_i), & \text{if } t = (b_i, e_j) \in T_{\text{new}} \end{cases}$,
- * $E'(t) = \begin{cases} E(t), & \text{if } t \in T' - T_{\text{new}} \\ E(e_j), & \text{if } t = (b_i, e_j) \in T_{\text{new}} \end{cases}$,
- * $C'(t) = \begin{cases} C(t), & \text{if } t \in T' - T_{\text{new}} \\ C(b_i) + C(e_j), & \text{if } t = (b_i, e_j) \in T_{\text{new}} \end{cases}$,
- * $W'(t) = \begin{cases} W(t), & \text{if } t \in T' - T_{\text{new}} \\ W(b_i)W(e_j), & \text{if } t = (b_i, e_j) \in T_{\text{new}} \end{cases}$ and
- * $\hat{s}' = \hat{s}$.

The operator removes the transitions in $s\bullet$ and $\bullet s$ and adds a new transition for each element (b_i, e_j) of $\bullet s \times s\bullet$ so that each new transition begins from $B(b_i)$ and ends at $E(e_j)$. Each new transition (b_i, e_j) represents the possibility of first executing the transition b_i and then the transition e_j . Thus its cost becomes the sum of the costs of the original transitions. The weight becomes the product of the original weights.

Note that if $\bullet s$ is empty, the operator will simply remove the state and all transitions starting from it. This is correct, because such an s is unreachable.

This operation can possibly add numerous new transitions, which makes it potentially time-consuming.

The operator does not change the probability of reaching terminal SCCs or the cost of reaching terminal states. It may change the cost of reaching such terminal SCCs that have transitions. This happens whenever s is in a terminal SCC, $B(b_i)$ is not, and $C(e_j) \neq 0$.

An example of removing a state is presented in Fig. 3.

3.4 The Complete Algorithm

With the previously defined operators, a complete algorithm can now be formed.

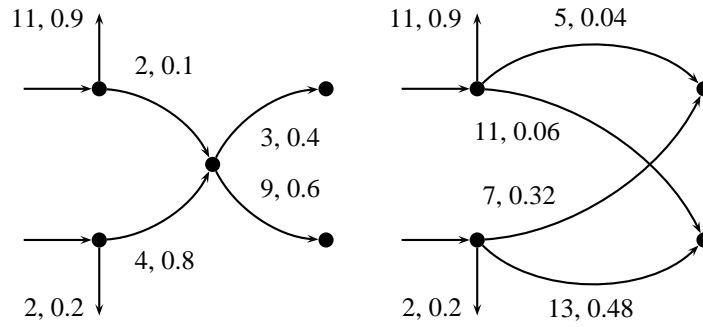


Figure 3. Removing a state. The middle state from the left graph is removed. The result is on the right hand side.

Algorithm $\text{Reduce}(P)$:

1. while a double transition (a, b) exists do
 $P := \text{Remove-double-tr}(P, a, b)$
2. while a removable self-loop l exists do
 $P := \text{Remove-self-loop}(P, l)$
3. if a removable state s exists then
 $P := \text{Remove-state}(P, s)$
 goto step 1

When the algorithm terminates, the only states that are left are the initial state and a single state for each terminal SCC. All transitions that are left are either from the initial state to a state representing a terminal SCC or self-loops in such a state. This can be seen as follows.

If there would be a transition t from a state $s \in S - \{\hat{s}\}$ so that $E(t) \neq s$, s could always be removed. It follows that all the other states are reachable from the initial state by a single transition.

If there would be two states s_1, s_2 that belong to the same terminal SCC, then one or the other of them could be removed, since there would be a transition as in the condition above. Thus there are no transitions between different states except from the initial state.

At least one state of each terminal SCC remains. When there is only one state in a terminal SCC left, there can be no transitions from it to another state and thus it cannot be removed. Otherwise it would either not be the only state in that SCC or the SCC would not be terminal.

If there is at least one other state besides the initial state, the initial state cannot have a self-loop, since it could be removed. If the initial state is the only state, it represents the only terminal SCC and can have a self-loop.

Because every operator in the algorithm conserves the probabilities of reaching terminal SCCs, the weights of the transitions that start from the initial state are the probabilities of the execution resulting in the corresponding terminal SCCs.

The cost of a transition is not necessarily the cost of reaching the terminal SCC. However, it is, if the corresponding state has no self-loop transition, or the cost of such a tran-

sition is zero. This is because then every execution that enters the SCC and perhaps roams around in it for a finite number of steps has the same cost.

If a state representing a terminal SCC has a self-loop with a cost greater than zero, the cost of the transition that leads to it is only an upper bound to the cost of reaching the component. This is because the terminal SCC may originally have had loops formed by several states. Some of the cost of looping inside the terminal SCC might have been transmitted to the incoming transition by the operation `Remove-state`. The algorithm does not specify in any way that calculating the cost should be stopped when the terminal SCC is first reached. If needed, this can be done by suitable preprocessing as described in Section 4.

4 Using the Algorithm

As an example of the use of the algorithm, we compare the efficiency of exploration testing with test case based testing. The algorithm is by no means limited to that use. It can be used, for example, to compare the efficiency of other testing methods or even the effectivity of single test cases.

We start by building two WCTSs that model executing a test case against a system with a planted error, and executing exploration testing against the same system. We shall call these WCTSs P and Q , respectively. Both of the WCTSs are designed so that all activity stops when the test environment announces an error. Thus each terminal SCC is actually a terminal state.

Finding reasonable weights for the transitions is a problematic issue. The weights should somehow represent realistic probabilities of different actions and model the relative speeds of different parts of the whole system. However, if we can show that one testing method is more efficient than the other for a large range of weight combinations that most likely contains all realistic combinations, then we need not know which precise combinations are realistic. This is exactly the method we have used in the case study.

The state space of P should be such that each terminal SCC represents either an error or correct behaviour. Since exploration testing terminates only after an error is found, each terminal SCC of Q should represent an error.

If we now execute the algorithm for P , we get the probability of finding the error with one execution of the test case, the expected cost of the corresponding execution and the expected cost of an execution that does not find an error. From these one can compute the expected cost of repeating the test until an error is found. Alternatively, one could write a WCTS that contains a model of a test environment that resets the system under test and starts the test run anew after each test run that fails to find an error. By executing the algorithm for Q , we get the expected cost of finding the error with exploration testing.

If more than one SCC corresponds to an error then their costs and probabilities can be combined in the same style as the operation `Remove-double-tr` of Section 3.1 does. The same applies to multiple non-error SCCs, of course.

The computational complexity of the algorithm is polynomial but nonlinear. The number of transitions created by removing states can grow in proportion to the square of the number of states. Though our implementation partly prevents this with some heuristics, the algorithm might run into trouble with big state spaces.

5 A Case Study: Broken Alternating Bit Protocol

The alternating bit protocol is a simple protocol for transmitting messages over unreliable channels [1]. In this case study, we break the protocol by removing the alternating bit from the acknowledgement messages. After this modification the protocol always delivers the first message correctly, but might lose messages after that. Because of this, sending the first message serves in this example as an initialization step that exploration testing avoids repeating.

We divided the transitions of the protocol into six groups and assumed that each transition in the same group has the same weight, not depending on the system state. The groups are

- * Reading from a channel
- * Losing a message in a channel
- * Accepting input from the user of the protocol
- * Sender timeout
- * Other sender transitions
- * Other receiver transitions

The efficiency of exploration testing is evaluated by building the state space so that messages are sent and received forever. By running the algorithm we then get an average cost C_0 for finding the error as the cost of the transition that leads to the single terminal SCC that represents an error.

A typical test case of the system would consist of giving the protocol some messages to deliver. Since a single message cannot reveal the error, the number of messages should be at least two. When we build a state space where n messages are sent and input it to the algorithm, we get the probability of finding the error by (and the average cost of) a single run of a test case that tries to send n messages. Then the average cost C_n of repeating the test until an error is found is calculated as described in the previous section.

We ran the algorithm to get C_0 and C_n for $n = 2 \dots 7$. To fight the problem of choosing realistic weights the algorithm was run for 10 000 random weight combinations (minimum weight 1, maximum 999, even distribution) for each value of n , and the corresponding C_0 for each weight combination was also calculated.

The cost of each transition was always set to 1, which means that we were only interested in the total number of transitions taken before the error was found.

We examined the proportion C_n/C_0 , which tells us how quickly exploration testing finds errors compared to repeating a test case of length n .

The 25 % and 75 % fractiles and the medians of the proportion C_n/C_0 for different values of n are presented in Table 1. Additionally, Fig. 4 shows the 10 000 random weight combinations plotted so that the probability to find the error with two messages is on the horizontal axis and the proportion C_2/C_0 is on the vertical axis.

The test cases get more efficient as their length increases, which was expected. Most of our effort was concentrated on the case $n = 2$. The values show that with random weights, exploration testing finds the implanted error in about one third of the time compared to running short test cases repeatedly. As can be seen in Fig 4, exploration testing is, at worst, more than twice as efficient as the short test cases.

Table 1. The medians and some fractiles of C_n/C_0 for different values of n .

n	25 % fractile	median	75 % fractile
2	2.3746	2.8031	3.8539
3	1.6120	1.7206	1.8728
4	1.3883	1.4366	1.4900
5	1.2793	1.3072	1.3351
6	1.2165	1.2363	1.2555
7	1.1765	1.1910	1.2055

We also noticed that the proportion C_n/C_0 gets larger as the probability of losing the message in the channels gets smaller. This is good news for exploration testing, since it is easy to believe that most of the messages go through in any realistic case. However, no explanation has been found for this behaviour as of yet. This phenomenon can be observed in Fig. 5, which shows the proportion C_2/C_0 as a function of the weight of losing the message with the same 10 000 random data points.

For an example of this, let us consider the case where the weights of timeout and of losing the message are 1, while the other weights are 100. Then $C_2/C_0 \approx 143$, which is significantly larger than the typical values. These weights represent small probability of message loss and (premature) timeout. Thus they are clearly a more realistic weight combination than most random weight combinations, which makes this an important observation. On the other hand, the results are not as impressive for longer test sequences: $C_7/C_0 \approx 1.60$ with these weights.

More case studies are needed to find out whether these figures are repeated in other systems. If exploration testing proves to be significantly more efficient than test cases in realistic systems, using it for example for regression testing of nondeterministic systems could bring remarkable benefits.

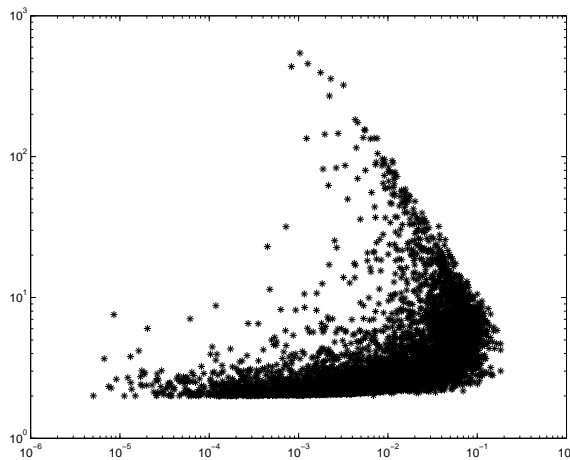


Figure 4. The efficiency of exploration testing compared to test cases of length 2. 10 000 random weight combinations are presented. Probability of error on the horizontal axis, C_2/C_0 on the vertical.

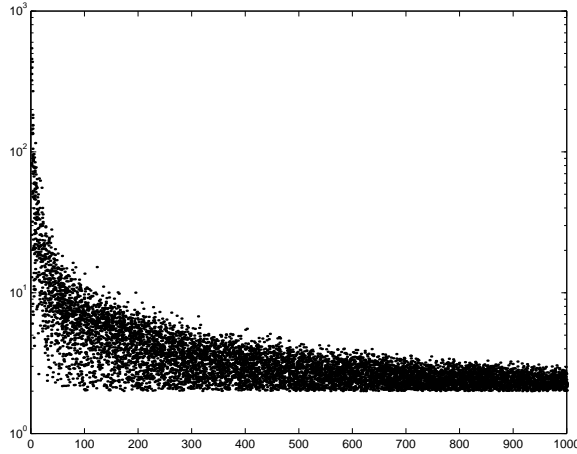


Figure 5. The efficiency of exploration testing compared to test cases of length 2 as a function of the weight of losing the message. 10 000 random weight combinations are presented. Weight of losing the message on the horizontal axis, C_2/C_0 on the vertical.

6 Conclusions

We developed an algorithm for analytically calculating the probabilities and costs of reaching different terminal SCCs in a WCTS. The algorithm can be used in multiple ways for performance analysis of terminating executions of formal models of concurrent systems. We then showed how this algorithm can be applied to analyze the probabilities and costs of different results when executing a test case against a system modelled with WCTSs, and the cost of exploration testing.

Our approach requires the use of a formal model of the system under test. It is thus applicable for a limited number of systems. On the other hand, it produces results much faster and with many more significant digits than is possible by actually conducting the tests or simulating their execution. For instance, when the error detection probability is 10^{-5} , as was on the left hand side of Fig.4, one million test runs are needed to get just one digit by conducting the tests. Furthermore, our method allows for easy experimentation with different system parameters, such as channel error rate. It is thus suitable for investigating general kind of phenomena related to error detection probabilities and costs.

One can, of course, question the appropriateness of the use of the negative exponential distribution, and the validity of the chosen weights. On the other hand, it seems impossible to find out the “real” distributions in a real system. Our approach makes it feasible to conduct the analysis with so wide a range of weight combinations that it is plausible that also the ill-defined real situation is represented reasonably well somewhere in the data.

The method was used in a case study of a broken alternating bit protocol to examine whether exploration testing finds errors more efficiently than test cases. We saw that exploration testing was clearly more efficient than short test cases, especially when the probability of losing the message was small. With longer test cases, the difference became less significant.

The idea of using this kind of formal algorithm for analysing the performance of testing

methods is new to our knowledge. Applying it to several larger and more diverse systems, especially to demonstrate the performance effects of choosing what to test on-the-fly, is beyond the scope of this paper. A lot of work still remains to be done in this area.

Acknowledgements

Thanks to Mikko Tiusanen for providing some valuable background information and Antti Kervinen and Antero Kangas for their comments.

This work is part of the SASOKE project funded by TEKES, Conformiq Software Oy Ltd and Nokia Research Center.

References

1. Bartlett, K., Scantlebury, R. & Wilkinson, P.: “A Note on Reliable Full-Duplex Transmission over Half-Duplex Links”. *Communications of the ACM* 12 (5), 1969, pp. 260–261.
2. Helovu, J. & Leppänen, S.: “Exploration Testing”. *Proc. ICACSD 2001, 2nd IEEE International Conference on Application of Concurrency to System Design*, IEEE Computer Society 2001, pp. 201–210.
3. Hopcroft, J., Motwani, R. & Ullman, J.: “*Introduction to Automata Theory, Languages, and Computation*” — 2nd ed. ISBN 0-201-44124-1, Addison-Wesley 2001, pp. 96–101.
4. Marsan, M., Bobbio, A. & Donatelli, S.: “Petri Nets in Performance Analysis: An Introduction”. In W. Reisig, editor, *Lectures in Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag, 1998, pp. 211–256.