An Automatic Performance Testing Method Based on a Formal Model for Communicating Systems

Levente Eros  
Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, H-1117 Budapest, Magyar tudosok krt. 2, Hungary  
Email: eros@tmit.bme.hu

Tibor Csondes  
Ericsson Hungary  
H-1117 Budapest, Irinyi J. u. 4-20, Hungary  
Email: tibor.csondes@ericsson.com

Abstract—In this paper we present a novel black box testing method for the performance validation of systems implementing a communication protocol. Our method solves the main problem of this kind of performance testing, namely its ad-hocness, by proposing a theoretical background for it. As a part of the presented methodology, we show how to map two kinds of performance requirements to a performance model and how to compare this model to the physical implementation. For comparing the physical implementation to the performance model automatically, we introduce a worst-case method and a probabilistic method. We also present our simulation results according to which, the proposed methods are correct and can be used efficiently.

I. INTRODUCTION

The last phase of the development process of a network component implementing a communication protocol is its testing. Two of the most important kinds of tests are conformance tests and performance tests. A conformance test checks whether the system under test (SUT from now on) implements the communication protocol it should, in other words, it checks whether the SUT fulfills its functional requirements [1]. A performance test is executed once the SUT has passed the conformance test and checks whether the SUT that meets its functional requirements meets its performance requirements too. Conformance testing has a well-developed methodology including many conformance testing methods [1]–[3]. Each one of these methods first creates a formal, Communicating Extended Finite State Machine (CEFSM) [4] model based on the functional requirements and then checks whether the SUT is identical to the CEFSM, regarding the SUT as a black box, that is, examining the system only through its responses given for different inputs. A CEFSM is a Finite State Machine the transitions of which have input and output values. If the input value the CEFSM gets from its environment belongs to an outgoing transition of the current state of the machine, the CEFSM transfers to the destination state of the transition and sends the output value of the transition to its environment. Contrarily to conformance tests, black box performance tests of such systems are mainly designed in an ad-hoc way, mostly relying on the expertise of the test designer, that is given the SUT (the conformance requirements of which are met) as a black box and different performance requirements as the inputs of the test, the performance test is designed and carried out manually without any theoretical background. Some performance requirements are furthermore ambiguous. In this paper we attempt to solve these problems of performance testing by proposing unambiguous definitions for two types of performance requirements, namely the maximal number of requests to be processed within a second (calls per second) and the maximal number of users to be served, and an automatic performance testing method that, without any human intervention, first maps the performance requirements to a formal model and then checks whether the SUT corresponds to this formal model.

II. RELATED WORK

Many papers have been written in the field of performance modeling communication protocols. [5] describes an SDL (Specification and Description Language) based performance model for communicating systems. [6] uses stochastic Petri nets, while [7] uses a timed Petri net to model the performance of communication protocols. [8] gives a timed Petri net based performance model of CSMA/CD bus LANs. These papers also describe methods for verifying the presented models that is, for analytically proving that the models correspond to their specifications. Our goal however, is to create a methodology for validating the system, that is to check whether the physical implementation corresponds to the performance requirements. [9] introduces TimedTTCN, a TTCN-3 (Testing and Test Control Notation) language extension that tests the response times of the physical implementation for different inputs which is a similar performance characteristic to the number of calls per second. As we will show in Subsection III-C however, TimedTTCN can only be used to check whether the SUT fulfills a performance requirement that is stricter than needed.

To sum the above up, papers [5], [6], [7] and [8] define formal verification methods for checking the correctness of models, while in our paper we propose a validation method for checking the correctness of the physical implementation.

III. MAPPING THE PERFORMANCE REQUIREMENTS TO THE FORMAL MODEL

In this section we are going to show how to model the two above mentioned performance requirements, the maximal
number of users to be served simultaneously and the maximal number of calls per second. In order to be able to show how to model these requirements however, we first have to define the Timed Communicating Finite Multistate Machine (TCFMM) model, and show how to create its structure.

A. Creating the Functional Structure of the TCFMM

The TCFMM model is an extension of the Communicating Extended Finite State Machine (CEFSM) model used in conformance testing [4]. It is called "Multistate", because contrarily to a CEFSM (or a simple Finite State Machine), it can have more than one active states marked with tokens. A TCFMM has seven attributes:

\[
TCFMM = (S, I, O, T, U, s_0), \quad (1)
\]

\(S\) is the finite set of states, \(I\) is the finite set of inputs, \(O\) is the finite set of outputs, \(T\) is the finite set of state transition functions, and \(s_0\) is the initial state of the TCFMM at which all the tokens are initially. Each transition \(t_i \in T\) has a source state \(s_{from}\), a destination state \(s_{to}\), an input value \(i_k\), and output value \(o_k\), and a delay \(d_k\). Transition \(t_i\) works as follows: When receiving input \(i_k\) from user \(j\), it takes token \(u_j\) from \(s_{from}\), puts it to \(s_{to}\), and sends \(o_k\) to user \(j\). The delay between receiving \(i_k\) and sending \(o_k\) is \(d_k\) (see more on this in Subsection III-C).

A TCFMM is thus a CEFSM the transitions of which have delay values and the states of which can store more than one tokens at the same time (the TCFMM can have more than one active states at the same time). This way the functional structure of the TCFMM is created by taking the CEFSM describing the functional behavior of the protocol implemented by the SUT, placing all \(|U|\) tokens to \(s_0\) and adding a delay parameter (not a delay value) to each transition.

Figure 1 shows two TCFMMs. All their tokens are at their states \(s_0\). Three parameters of the transitions are written on the corresponding edges in the form input/output/delay.

B. Modeling the Number of Simultaneously Served Users

As we have mentioned earlier, each token in the TCFMM represents a user served by the SUT. More precisely, the current state of token \(u_i\) is the current state of the server instance (or protocol instance) serving user \(i\). During performance testing, our goal is to check whether the SUT is able to generate a predefined number of calls per second while serving a predefined number of users simultaneously. In order to stress the SUT by the maximal number of users which is going to be denoted by \(usr\) from now on, the number of active tokens has to equal \(usr\) at all times. An active token is a token the current state of which is not a terminal state, i.e. not a state with no outgoing transitions. The reason why the number of active tokens has to be \(usr\) at all times is because users represented by non-active tokens cannot generate requests towards the SUT and therefore, cannot be considered as users that are currently served by the SUT.

In order to keep the number of active tokens equal to \(usr\), each transition originating in a terminal state, formally each \(t_i : \theta(t_j : s_{from} = s_{to})\), where \(t_i, t_j \in T\) has to be redirected to initial state \(s_0\). This way, when a token would go inactive, it merely reappears at the initial state, keeping the number of currently active tokens constant. A token transferring to the initial state on a redirected transition thus, corresponds to a new active user appearing whenever a user goes inactive.

The right side of Figure 1 shows the TCFMM from the left side of the figure after its transitions leading to the only final state, \(s_3\) got redirected to \(s_0\).

C. Modeling the Number of Calls per Second

In this section, we are going to show how to use the TCFMM for modeling the number of requests that can be handled within a second, in other words the number of calls per second, but before going on with this, we have to disambiguate the notion of "the number of calls per second" as we mentioned in Section I.

We have defined the criterion of a system processing \(C_{req}\) calls per second in the following way: A system fulfilling the requirement of processing \(C_{req}\) calls per second processes at least \(C_{req}\) calls per second when induced by an arbitrary infinite sequence of inputs (requests), measured for a relatively long (optimally infinite) period of time. Formally, if \(F_t^s\) denotes the number of firings measured for time length \(t\) with an infinite input sequence \(s\), then the system can process \(C_{req}\) calls per second if and only if the following inequality is true for any \(s\):

\[
\lim_{t \to \infty} \frac{F_t^s}{t} \geq C_{req} \quad (2)
\]

By measuring the number of calls processed within a second as a long-time average, the above definition hides transient stages when the system underperforms and thus, the definition prevents systems underperforming for a relatively short time from failing the test. This corresponds to the real life, where users do not usually care if the server they are communicating with "slows down" for, let us say, a few seconds if afterwards it performs as expected for hours. Note: There are certain cases in which it is important for the system to be able to serve a specified number of requests in each individual second, in this paper however, we are going to deal with the more permissive case formally defined by Formula 2.
In our model the number of calls processed within a second is modeled by restrictions on transition delays. The delay of a transition is defined as follows:

Let us drive all \(usr\) tokens in the TCFMM to the originating state \(s_{from}\) of \(t_i\) with an appropriate sequence of input values (requests), then let all \(usr\) clients send input \(i_s\) of \(t_i\) to the system at the same moment. Let \(pd_j\) denote the (physical) delay between user \(u_j\) sending \(i_s\) and receiving \(o_t\). The delay of transition \(t_i\) is then:

\[
d_i = \max_{u_s \in L} \frac{pd_j}{usr}
\]

(3)

Defining transition delays with the help of many tokens is necessary, since if we measure the delay of a transition of a multiprocessor system using only one user (token) we might get a false picture of its performance, since it is possible that only one processor serves the request of this single user, while the others remain idle and therefore hidden from the tester. If however, a transition delay is measured by sending \(usr\) requests to the system at the same time, we get a real picture of the average time needed to process a single request belonging to \(t_i\), as we have stressed the system by sending it the maximal number of requests at the same time. In the case of a uniprocessor system, measuring transition delays with a single token would be correct as there wouldn’t be any idle processors, but as the tester sees the SUT as a black box and might not have any information on the number of processors in it, it is more appropriate to measure delays in the above described way.

After all the above, we are now going to show how to model the maximal number of calls per second using transition delays. If \(d_i \leq \frac{1}{C_{req}}\) for each \(t_i \in T\), then the system corresponding to the model will be able to process \(C_{req}\) calls within each second. Checking if the SUT fulfills this restriction would be easy with TimedTTCN, as the restriction defines a delay limit for each transition separately [9]. This condition is however, only sufficient and not necessary, since Formula 2 does not require the SUT to process \(C_{req}\) calls in every single second, but only to process \(C_{req}\) calls per second in average, measured for a long (ideally infinite) period of time, and thus for a relatively long sequence of transitions. Furthermore, the delay of a transition \(t_i\) may be \textit{constantly} above \(\frac{1}{C_{req}}\) if the other transitions of each directed transition cycle \(t_i\) is a member of compensate its delay. The following, lighter condition is a sufficient and necessary restriction for a system that processes at least \(C_{req}\) calls per second according to Formula 2. A system is able to process \(C_{req}\) calls per second if and only if for all of its transition cycles \(c_i\):

\[
\sum_{t_j \in c_i} \frac{d_j}{|c_i|} \leq \frac{1}{C_{req}}
\]

(4)

In the above formula and the rest of this paper, cycles are considered to be ordered sets of transitions.

IV. Comparing the TCFMM Model to the SUT

Comparing the TCFMM to the SUT is done by measuring the transition delays of the SUT. Measuring the delay of a transition is done by firing all \(usr\) tokens through the transition by each user sending the transition’s input value to the SUT at the same time. Based on the physical delays elapsed between each user sending the input and receiving the output value a momentary delay is calculated according to Formula 3. After repeating this a large enough times, the measured momentary delays are averaged to get the transition delay. Using the so obtained delay values for each transition, the maximal number of requests the SUT can process within a second can be calculated in two ways. These are described in the following.

A. Calculating the Worst Case Number of Calls per Second

This approach calculates a worst-case value for the maximal number of calls per second. This is done by finding the cycle in the TCFMM traversing which the SUT processes the lowest possible number of calls within a second. Having already obtained the transition delays by the earlier described method, the maximal number of calls per second can be calculated using the following formula:

\[
C_{worst} = \min_{c_j \in C} \left\{ \sum_{t_i \in c_j} \frac{|c_j|}{\sum_{t_j \in c_i} d_j} \right\}
\]

(5)

If we have some extra information on user behavior, we can use a more sophisticated method described in the following.

B. Calculating the Expected Number of Calls per Second

If a system corresponds to Formula 2, it will be able to process at least \(C_{req}\) calls per second, induced by any sequence of requests. Probabilities of different sequences of requests are however not equal, as a user in a given state of execution might choose a request to induce the system by more likely than another request. This behavior of users assigns different firing probabilities to transitions originated from the same state. As a consequence, the bottleneck sequence of the previously described method, i.e. the one that minimizes \(\min_{c_j \in C} \left\{ \sum_{t_j \in c_i} d_j \right\}\) might be an extremely unlikely one and might draw a false picture of the performance of the SUT. Based on this, if we have some information on user behavior, it is possible to give a way to calculate the expected number of calls per second instead of the worst-case number of calls processed within a second, and loosen the delay requirements.

Let \(q_i\) denote the probability of a user with its token \(u_i\) at \(s_{from}\), choosing transition \(t_i\) to fire (\(q_i\) is equal for each user in our model). Furthermore let us define \(p_{kl}\) as follows:

\[
p_{kl} = \sum_{s_{from} = s_k, \land s_{to} = s_l} q_i
\]

(6)

Thus, \(p_{kl}\) is the probability of token \(u_i\) transferring from \(s_k\) to \(s_l\) \(p_{kl}\) is equal for each user). Let \(z_i\) be the \textit{stationary state probability} of state \(s_i\), that is the probability of token
(or one of the cycles) traversing which the SUT serves the lowest possible number of calls within a second, otherwise \( C_{\text{expected}} > C_{\text{worst}} \).

V. SIMULATION RESULTS

We have proven the correctness and evaluated the efficiency of our method by comparing it to the ad-hoc performance testing method, which simply simulates a real-life scenario by running a given number of independent clients (users) communicating with the SUT in parallel, for a certain amount of time. Each user measures the number of responses the SUT has sent to it in return for its requests sent to the SUT. Upon receiving a response from the SUT, the user replies with the next request immediately. At the end of the simulation, the number of received response messages measured by each user are summed up and divided by the duration of the test to get the average number of calls the SUT has served within a second. Our simulator has three components, namely ad-hoc, tester, and SUT. Component ad-hoc performs the ad-hoc testing method, component tester implements the method (probabilistic and worst-case) proposed in this paper, while the SUT component simulates the System Under Test. We used two hosts connected by a direct link for our simulations, one of them for running components ad-hoc and tester (one at a time), and the other one for acting like the SUT by running the SUT component.

For the ad-hoc performance measurements we used 10 independent users running in parallel. We have performed 1279 ad-hoc measurements, each one of which ran for a different amount of time, 1 to 1279 seconds. To perform the proposed testing method, we also used 10 tokens representing 10 users to measure the transition delays of the SUT according to Formula 5 and Equation 8. We have performed 42 measurements (probabilistic and worst-case), changing the number of iterations (the number of times each element of \( C \) is traversed) from 1 to 42. Each measurement performed by the proposed method took approximately \( n \ast 30 + 15 \) seconds, where \( n \) is the number of iterations of the measurement. As an input of the ad-hoc and the proposed method, we used the same user state transition probabilities, of course.

Figure 2 shows as a function of the time needed to perform the measurements, the calls per second values measured on the SUT by the ad-hoc method and the expected number and minimal number of calls per second according to Formula 5 and Equation 8. We have performed 42 measurements (probabilistic and worst-case), changing the number of iterations (the number of times each element of \( C \) is traversed) from 1 to 42. Each measurement performed by the proposed method took approximately \( n \ast 30 + 15 \) seconds, where \( n \) is the number of iterations of the measurement. As an input of the ad-hoc and the proposed method, we used the same user state transition probabilities, of course.

Figure 2 shows as a function of the time needed to perform the measurements, the calls per second values measured on the SUT by the ad-hoc method and the expected number and minimal number of calls per second according to Formula 5 and Equation 8. We have performed 42 measurements (probabilistic and worst-case), changing the number of iterations (the number of times each element of \( C \) is traversed) from 1 to 42. Each measurement performed by the proposed method took approximately \( n \ast 30 + 15 \) seconds, where \( n \) is the number of iterations of the measurement. As an input of the ad-hoc and the proposed method, we used the same user state transition probabilities, of course.

Figure 2 shows as a function of the time needed to perform the measurements, the calls per second values measured on the SUT by the ad-hoc method and the expected number and minimal number of calls per second according to Formula 5 and Equation 8. We have performed 42 measurements (probabilistic and worst-case), changing the number of iterations (the number of times each element of \( C \) is traversed) from 1 to 42. Each measurement performed by the proposed method took approximately \( n \ast 30 + 15 \) seconds, where \( n \) is the number of iterations of the measurement. As an input of the ad-hoc and the proposed method, we used the same user state transition probabilities, of course.

Figure 2 shows as a function of the time needed to perform the measurements, the calls per second values measured on the SUT by the ad-hoc method and the expected number and minimal number of calls per second according to Formula 5 and Equation 8. We have performed 42 measurements (probabilistic and worst-case), changing the number of iterations (the number of times each element of \( C \) is traversed) from 1 to 42. Each measurement performed by the proposed method took approximately \( n \ast 30 + 15 \) seconds, where \( n \) is the number of iterations of the measurement. As an input of the ad-hoc and the proposed method, we used the same user state transition probabilities, of course.

Figure 2 shows as a function of the time needed to perform the measurements, the calls per second values measured on the SUT by the ad-hoc method and the expected number and minimal number of calls per second according to Formula 5 and Equation 8. We have performed 42 measurements (probabilistic and worst-case), changing the number of iterations (the number of times each element of \( C \) is traversed) from 1 to 42. Each measurement performed by the proposed method took approximately \( n \ast 30 + 15 \) seconds, where \( n \) is the number of iterations of the measurement. As an input of the ad-hoc and the proposed method, we used the same user state transition probabilities, of course.

As it can be seen in Figure 2, the expected number of calls per second measured by our method is stably around the mean
value of the ad-hoc measurements (8.2068 calls per second), proving the correctness of our method. The figure also shows that the deviation of the calls per second values measured by the ad-hoc method within the same amount of time is huge compared to the deviation of the results obtained by the proposed method. Figure 3 shows the deviation of the calls per second measurements taken by the ad-hoc method and that of the expected number of calls per second values measured by the proposed method as a function of the amount of time needed to perform the measurement, on a logarithmic vertical axis. As it can be seen in the figure, the deviation of the values measured by the proposed method is lower than the deviation of the values measured by the ad-hoc method by more than an order of magnitude. If we compare the deviation of all measurements taken between 1 and 1279 seconds with the two methods, we get even more dramatic results: The deviation of all ad-hoc measurements is 0.8432, while the deviation of the expected calls per second values obtained by the proposed method is only 0.0225.

![Figure 2. Measurements by the ad-hoc and the proposed method](image2)

The fluctuation of the deviation curve of our method in Figure 3 is caused by the variation of the link delay between the two hosts used for the simulations, and only seems bigger than that of the ad-hoc curve because of the logarithmic axis. The measurements have shown that by using the proposed method instead of the ad-hoc method, we can get correct and more accurate (lower deviation) measurement results on the performance of the SUT in a smaller amount of time.

![Figure 3. Measurement deviations on logarithmic scale](image3)

VI. SUMMARY AND FUTURE WORK

In this paper we have introduced a performance testing method that can check whether the System Under Test (SUT) fulfills two types of its performance requirements, namely the maximal number of users to be served and the number of calls the system has to process within a second. Our method first maps these requirements to our performance model (the TCFMM). The maximal number of users to be served is modeled by the number of tokens in the TCFMM and by looping back its transitions leading to terminal states. The number of calls the system has to process within a second is modeled by restrictions on transition delays, having measured which, the worst-case and the expected number of calls per second can be calculated. At the end of our paper we presented our simulation results proving that our method produces correct measurements, and is more accurate and more effective than the ad-hoc performance testing method.

In the future, we are going to extend our model for representing more kinds of performance requirements. We are also going to investigate how the performance fluctuation of the tester and the delay fluctuation of the link between the tester and the SUT affects the outcome of the test and propose fluctuation-proof testing methods.

ACKNOWLEDGMENT

This paper has been (partially) supported by HSNLab, Budapest University of Technology and Economics, http://www.hsnlab.hu.

REFERENCES