Performance Testing from UML Models with Resource Descriptions*

Flávio M. de Oliveira¹, Rômulo da S. Menna¹, Hugo V. Vieira¹, Duncan D.A. Ruiz¹

¹Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul
90619-900 – Porto Alegre – RS – Brazil
{flavio.oliveira, romulo.menna, duncan.ruiz}@pucrs.br, hvieira@hp.com

Abstract. Markovian models and techniques normally used in performance analysis do not apply directly to performance testing. While analysis models are steady-state, testing scenarios are transient; also, assumptions about resources considered when applying steady-state analysis may not be valid in real test scenarios. We present here a strategy for performance modeling adapted to performance testing. Our input is a performance specification using the UML 2.0 SPT Profile, from which we extract a modified Stochastic Petri Net; this model represents the information we need to generate realistic test scenarios. We implemented the strategy into the UpperT tool, and show how it can be used to automatically derive test scripts for a Web application.

1. Introduction

Performance evaluation and performance testing are strongly related fields, for obvious reasons; nevertheless, one finds that solutions from one side are not translated to the other so directly as it could be expected. For example, the vast body of knowledge available about Markovian performance analysis does not apply directly to performance testing. One problem is related to the nature of models; analysis models are steady-state, while testing scenarios are transient. To build a performance testing scenario, we use temporal as well as probabilistic information; also, assumptions about resources considered when applying steady-state analysis may not be valid in real test scenarios.

The UML2.0 [OMG 2007] provides, among other things, a notation for modeling some important characteristics of applications with stronger semantics, allowing the development of automatic tools for tasks such as model verification, analysis and code generation. Performance is one of these characteristics; it is the object of the SPT (Schedulability, Performance and Time) Profile, which defines a standard way to represent performance information in a UML model.

We present here a strategy for performance modeling adapted to the generation of performance test scenarios. The input for our approach is a specification of the performance requirements for the application to be tested using the UML 2.0 SPT Profile, from which we extract a modified Stochastic Petri Net; this model includes the kind of information we need to generate realistic test scenarios. We implemented the strategy into the UpperT (UML-Profile Performance Testing) tool, and show how it can be used to automatically derive test scripts for a Web application.

* This work was developed in collaboration with HP Brazil R&D - PUCRS (Research Project CPTS).
2. UML2.0 SPT Profile

The UML 2.0 SPT (Schedulability, Performance and Time) Profile is an extension of UML for modeling performance requirements of applications. Its purpose is to provide modeling patterns with clear semantics for those characteristics, thus allowing automated performance analysis of models. The usual approach is to extract a Markovian model of some kind (Markov chain, Petri net or equivalent) from the UML description and perform analysis on the extracted model, using standard techniques such as queuing theory and simulation. That is the approach adopted in [Lopez-Grao et al. 2004], for instance. A model designed with the UML SPT Profile is composed of UML diagrams (use cases, classes and activities) tagged with specific performance-related attributes: probability of use cases, response times for activities, availability of resources etc.). There are successful applications of the SPT Profile for simulation and performance analysis modeling, see for example [Marzola & Balsamo 2004] and [Lopez-Grao et al. 2004]. The main benefits of this approach are: (1) UML is a well-known and widely applied design notation, being a de facto standard in the object-oriented software industry; (2) the profile adds a strong semantics to the modeling notation, allowing automatic derivation of models for analysis and meaningful interpretation.

3. Stochastic Petri Nets

Our strategy for generation of test scenarios consists of deriving, from the UML diagrams describing the application under test, a single stochastic model representing the necessary information. This model is an adaptation of the class of Generalized Stochastic Petri Nets (GSPN) [Marsan et al. 1994]. The GSPN formalism is known to be isomorphic to continuous-time Markov chains [Marsan et al. 1994], and has been used in software performance modeling. A Petri Net (PN) is a structure composed of places, transitions and directed arcs. Arcs connect places to transitions and transitions to places. Places may contain tokens. The state of a Petri Net – called a marking - is determined by the number of tokens present in each place. The initial marking $M'$ describes the initial state of the network. Transitions have input and output places, which are, respectively, places connected to the transition by arcs to the transition (input places) or from the transition (output places). We use $t_\bullet$ and $t^\bullet$ to denote the set of, respectively, input and output transitions of $t$. In the same way, places have input and output transitions. As in the case with input and output places, we use $p\bullet$ and $p^\bullet$ to denote respectively the sets of input and output transitions of $p$ [Marsan et al. 1994].

A transition is enabled to fire in a marking $M$ if and only if there is a token in each of its input places. If a transition fires, it consumes one token of each of its input places and generates a token in each of its output places.

There are many variations of the basic Petri Net formalism described above. One important extension is the class of Generalized Stochastic Petri Nets, which includes temporal and probabilistic information.

3.1. Generalized Stochastic Petri Nets

A stochastic Petri Net (SPN) is a structure $SPN = (PN, R)$, where $PN$ is a Petri Net, and $R = \{r_1, r_2, ..., r_m\}$ is a set of firing rates assigned to the transitions of $PN$ [Marsan et al. 1994]. Each transition consumes time to fire; this time is a random variable with an exponential distribution. Thus the firing rate (number of firings per time unit) is also
exponentially distributed; the rates in $R$ are the mean values of the firing rates for each transition. The time spent in a given marking of a SPN is also a random variable with an exponential distribution. The enabling rules of transitions in SPNs are the same as in Petri Nets.

A Generalized SPN (GSPN) is an SPN with two types of transitions – immediate and timed transitions. Timed transitions are as in the SPN, while immediate transitions are instantaneous and therefore don't have a firing rate. Immediate transitions have higher priority than timed transitions; in a given marking, if an immediate and a timed transition are both enabled, the immediate one will fire first. We can assign probabilities to immediate transitions; if any two immediate transitions are enabled, one is chosen randomly to fire first, according to their probabilities. We can say that immediate transitions behave as in discrete-time Markov chains, while timed transitions behave as in continuous-time chains.

4. Our Approach

4.1. Modified GSPNs

The strategy adopted in our work is to generate performance test scenarios by simulating a GSPN, plus additional information about resources to be used in the testing scenarios. These resources describe real assets available in the testing lab, for example, printers, servers etc., instead of abstract conditions assumed in analysis models.

In order to generate test scenarios, we need information about activities and types of resources involved in the application. Therefore, our modified GSPNs use objects instead of tokens. Objects can be of two types: execution objects and resource objects. Execution objects represent instances of processes (sequences of activities), corresponding to job requests or users. A sequence of activities corresponds to a use case or activity diagram in the UML model. Resource objects represent resources available to the system modeled by the network, necessary for the execution of some activity. Unlike tokens, execution objects and resource objects have identifiers.

Resource objects are stored in resource places, which are grouped in resource classes. The definitions of resource place and resource class are based on definitions presented by Ruiz [Ruiz 1995]. A resource class denotes the type of resource necessary to the execution of some activity; for example, a print job needs a resource of the printer class. The class of a resource object is the class of the resource place where it is. Thus, a transition can replicate a resource object from an input resource place to an output resource place only if they are of the same class. Only resource objects can be stored in resource places.

The enabling rule for transitions is modified to consider information of resources and execution objects. A transition is enabled to fire if and only if there are execution objects with the same identifier in all its input places which are not resource places and resources in all its input resource places.

Every modified GSPN must have an initial place $i$ such that $\bullet i = \emptyset$ and a final place $o \in P$ such $\bullet o = \emptyset$; these elements are adapted from [5]. In the initial marking, all execution objects are in the initial place; all other places that are not resource places are empty. In a final marking, all execution objects are in the final place.
4.2. Deriving a modified GSPN from a UML2.0 model

We describe here the elements (tags and/or classes) of the SPT Profile that we use to derive the modified GSPN, and the rules defined for the derivation. We make reference only to the elements of the Profile effectively used in the derivation. Details about the profile structure can be found in [OMG 2007]. In the figures, we use the usual notation for GSPNs, with circles representing places, white bars for timed transitions and black bars for immediate transitions.

- **PApopulation**: a tag that represents the number of users. It defines the number of execution objects in the initial place of the derived GSPN.

- **Activity**: each activity is represented in the modified GSPN by three immediate transitions and their places: **Start**, representing the availability of resources for the activity, **Run**, representing the execution of the activity, and **End**. With that structure, we can automatically generate representations for resource allocation and synchronization of activities.

- **Resource**: a class that represents the necessary resources the execution of a given activity. Resources modeled with this class correspond to resource classes in the modified GSPN; resource places and objects represent their instances. A resource is allocated in the **Start** transition and released in the transition **End**.

- **PAoccurrence**: a tag that describes the occurrence pattern of the execution objects – in our case, the arrival rate. We represent this information in the modified GSPN by means of a timed transition. The input place of the transition is the initial place, and the firing rate is the value of the **PAoccurrence** tag.

- **PAextDelay**: we use this tag to represent think time, the time between the availability for execution and the actual start of the activity by a user action. This tag is represented by a timed transition before the **Run** transition. Its rate is \( r = \frac{1}{PAextDelay} \).

- **PArespTime**: this tag denotes the expected response time for an activity. It corresponds to a timed transition with rate \( r = \frac{1}{PArespTime} \) after the immediate transition corresponding to the activity.

- **PAProb**: we use this tag to define the probability of a given execution instance (job request). This information is represented in the GSPN by an immediate transition with probability equal to the tag value.

We build the structure of the modified GSPN by means of composition of simple control flow structures (figure 1): OR split (implementing path selection; the values of the transition rates are the values of the corresponding **PAProb** tags), OR join (join of two or more paths without synchronization), AND split (parallel execution paths) and AND join (join with synchronization).

The figure 2 shows the general format of the modified GSPN. It begins with an OR Split; each alternative activity path corresponds to a use case. The path corresponding to the use case is derived from the activity diagram associated with the use case. An execution object (representing a job request) in the initial place \( i \) selects one path, according to the transition probabilities of each use case. At the end of the network, an OR Join connects the alternative paths of all use cases to the final place \( o \). After the activity network is complete, we add resource places, with their corresponding resource objects, as input places to the **Start** transition of each associated activity.
The control flow structures used here are based on [Aalst & Hee 2005], although we made some alterations. Each construct (OR Split, AND Join etc.) used to build the network begins with a place and terminates with a transition. In this way, we can connect a set of constructs into one single GSPN. This derivation strategy is modular and very easy to be implemented in a program.

4.3. UpperT

We developed an implementation of the strategy described above in the UpperT (UML Profile Performance Testing) system. UpperT is a test script generator from UML models; its current version receives an UML model compliant with the SPT Profile specification and produces a test script in the script language of the Jmeter performance testing engine [4]. Jmeter is an open source tool developed by the Apache project. It allows performance and load testing for web applications. The overall goal of UpperT is the automatic generation of performance test scripts directly from high-level (design) artifacts. The execution of UpperT consists of the following steps:

1. Load the UML input model, represented in XMI format. We adopted the
2. Extract the modified GSPN from the UML input model and generate a set of test scenarios. A test scenario consists of the following information:
- a set of resources initially available for the application (these are extracted from the model and correspond to real resources available for execution in the testing lab; in this way, we generate realistic and feasible tests automatically);
- a set of jobs;
- for each job, a schedule defining its entry time, activity requests and think times;
- expected results and response times;

We generate the scenario by simulated execution of the modified GSPN. The simulator is based on the JFern open source Petri Net simulation engine [Sourceforge 2007].

3. Convert the test scenarios into JMeter script format.

4. Execute the JMeter engine.

5. Load the result file created by JMeter engine for interpretation.

5. Example

The figures 3 and 4 below give a simple example of input model for an FTP server. We define that only 3 clients can access server FTP simultaneously, in order to simulate resource contention. We show two use cases, “List FTP Files” and “Get File”, both with probability 0.5. The figure 4 shows the corresponding activity diagrams. The figure 5 shows the modified GSPN extracted from the model.

We show in the figure 5 the part of the generated GSPN corresponding to the “list FTP files” use case – the second activity diagram in the figure 4. Each activity corresponds to a subnet, marked with a dotted box in the figure. The “access to server” place has three objects, representing the maximum of 3 simultaneous connections allowed. The subnet corresponding to the “Get File” use case (not detailed in the figure) has a similar structure, and has arcs pointing to the “access to server” place.

6. Final Considerations

Although there are many works applying performance analysis to UML models based on the SPT Profile, its application to performance testing requires specific adaptation, mostly related to the discrete-time nature of stochastic models for testing. In this work, we consider that GSPNs are a useful model for testing, because it can combine probabilistic (discrete) and temporal (continuous) information, with meaningful results.
An alternative would have been discrete Markov chains, which have been applied often to statistical testing. Nevertheless, we believe that the modified GSPNs preserve important information about what is happening to objects, activities and resources. In this way, we can easily recover and reinterpret the results of the testing session in the same level of abstraction of the original model. That would be more difficult to do with Markov chains, because they consist only of states and transitions.

We performed a first set of tests with UpperT, generating samples for two applications: the FTP example and a remote printer monitoring service. We executed runs considering different client populations; although we did not make a formal analysis of the results yet, the average numbers are consistent with the UML model. We are now developing a result interpretation layer for UpperT.

We are still not using the full power of the SPT Profile; for instance, with the sequence diagrams and class diagrams, it would be possible to do performance testing for internal components of the application. The current prototype of UpperT is limited to web applications; we are developing script generation layers for other engines. The good thing is that we can do it using the same modified GSPN model for test generation.

Acknowledgments. We would like to thank Lucas Waechter at HP Brasil R&D for his comments and support.

References

Figure 5. GSPN generated by UpperT.


