A Model-Based Approach for Testing the Performance of Web Applications

Mahnaz Shams Diwakar Krishnamurthy Behrouz Far
Department of Electrical and Computer Engineering, University of Calgary,
2500 University Drive NW, Calgary, AB, Canada. T2N 1N4
Phone: 1-403-220-5806
Email: dkrishna@ucalgary.ca

ABSTRACT

Poor performance of Web-based systems can adversely impact the profitability of enterprises that rely on them. As a result, effective performance testing techniques are essential for understanding whether a Web-based system will meet its performance objectives when deployed in the real world. The workload of a Web-based system has to be characterized in terms of sessions; a session being a sequence of inter-dependent requests submitted by a single user. Dependencies arise because some requests depend on the responses of earlier requests in a session. To exercise application functions in a representative manner, these dependencies should be reflected in the synthetic workloads used to test Web-based systems. This makes performance testing a challenge for these systems. In this paper, we propose a model-based approach to address this problem. Our approach uses an application model that captures the dependencies for a Web-based system under study. Essentially, the application model can be used to obtain a large set of valid request sequences representing how users typically interact with the application. This set of sequences can be used to automatically construct a synthetic workload with desired characteristics. The application model provides an indirection which allows a common set of workload generation tools to be used for testing different applications. Consequently, less effort is needed for developing and maintaining the workload generation tools and more effort can be dedicated towards the performance testing process.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of systems - Measurement techniques, Modeling techniques, Performance attributes

General Terms

Measurement, Performance, Algorithms, Experimentation.

Keywords

Performance testing, Web-based Applications, EFSM

1. INTRODUCTION

Enterprises are increasingly relying on Web-based systems to support critical business functions. Such systems often host complex, multi-tier applications such as e-commerce and business process management. With these systems, responses to user requests are typically generated dynamically by invoking the service of one or more server tiers. The system workload has to be characterized in terms of sessions; a session being a sequence of inter-dependent requests submitted by a single user. Dependencies arise because some requests depend on the responses of earlier requests in a session. For example, an order cannot be submitted to an e-commerce system unless the previous requests have resulted in an item being ordered. We define this phenomenon as *inter-request dependency* and refer to this class of systems as *session-based systems.*

Poor performance of session-based systems can adversely impact the profitability of an enterprise. As a result, effective performance testing techniques are essential for understanding whether a session-based system will meet its performance objective when deployed in the real world. *Performance testing* is a technique where synthetic workloads [4] are submitted to a system under study within a controlled environment. The *synthetic workload* used in a performance test serves to mimic the request patterns of real system users. Synthetic workloads are constructed from a workload model. A *workload model* specifies statistical characterizations for a set of workload attributes that are expected to affect performance the most. Measurements such as user response times and server utilizations are collected during the tests and used to support capacity planning and service level assessment exercises.

In general, a performance testing methodology must address several requirements with respect to synthetic workloads. Firstly, to reach reliable conclusions based on the results of a performance test, the synthetic workloads used must be representative of real workloads. A synthetic workload is said to be representative of a real workload if both workloads result in similar performance when submitted to a system. The representativeness of a synthetic workload is significantly influenced by the attributes in the workload model and the characterizations used for them [4]. Furthermore, for session-based systems requests within a session in the synthetic workload must reflect the correct inter-request dependencies to exercise application functions representatively. Krishnamurthy [6] experimentally showed that incorrect performance estimates can result when inter-request dependencies are ignored while constructing synthetic workloads. Secondly, since it is very difficult to know precisely what a real workload's characteristics will be, a performance testing methodology must

provide the flexibility to conduct a controlled sensitivity analysis on the characterizations of the workload model's attributes. The need to satisfy inter-request dependencies makes it difficult to satisfy the representativeness and flexibility requirements. Typically, a set of system-specific scripts are manually developed to create a synthetic workload. Different synthetic workload characteristics are achieved by manually selecting different sets of scripts. Such an approach can be time consuming especially when the characterizations of workload attributes need to be varied in a fine-grained and controlled manner. Furthermore, since the scripts developed are system-specific they need to be modified when changes are made to a system (e.g., changes in inter-request dependency, addition of new functionality). New sets of the modified scripts need to be deduced to construct the various desired synthetic workloads further increasing the performance testing effort. As a result, very often ad-hoc workloads are used and insufficient sensitivity analysis is carried out. Consequently, results from such tests are not likely to provide reliable insights into system performance.

In this paper, we propose a model-based approach that addresses these limitations. Our approach uses an application model that captures the application logic of a session-based system under study. Essentially, the application model can be used to obtain a large set of user request sequences that satisfy the correct inter-request dependencies for the system under study. This set of sequences is used to automatically construct a synthetic workload with desired characteristics. The advantages of this approach over the traditional approach are as follows:

- The approach provides automated support for fine-grained control of workload characteristics. For example, the approach would make it easy to create controlled workloads to study how varying the characteristics of a particular workload attribute impacts system performance.
- Lesser effort is needed to adapt the synthetic workload generation tools to handle changes made to a given system or to use them for testing other systems. This improved portability is a result of the indirection provided by the application model. The application model essentially makes the workload generation tools independent of the system under study. For example, by specifying different application models the same set of tools can be used to test an e-commerce system, a modified version of the e-commerce system and an online auction system.

The rest of the paper is organized as follows. Section 2 discusses background and related work. An overview of the proposed approach is provided in Section 3. Section 4 discusses a methodology to construct application models for the purpose of performance testing. Implementation details and a preliminary evaluation of our approach are presented in Section 5. Section 6 provides conclusions and discusses future work.

2. BACKGROUND AND RELATED WORK

Generating synthetic Web workloads typically involves two steps: trace generation and request generation. The trace generation step handles the complexities of creating a synthetic trace of HTTP/HTTPS requests that adhere to a workload model. The request generation step submits the requests in the trace to the system under study. Pre-generating traces reduces overheads during request generation thereby ensuring that the achieved workload characteristics stay close to the specified characteristics. The `SURGE` [3] tool supports trace and request generation capabilities for testing Web servers. `S-Clients` [2] and `httperf` [9] are request generation tools that are capable of generating realistically heavy overloads during performance tests.

Requests for session-based systems can be split into two parts: 1) a request type; and 2) a name-value list. Typically a *request type* instructs an application server to execute a particular server-side script and the *name-value list*, which follows the request type, provides input data for the script. Each entry in the name-value list consists of a name of a parameter and the parameter's value. For example, the request `/Add?product_id=1&number=2` has `Add` as its request type and `product_id=1&number=2` as its name-value list. The two parameters for this request are `product_id` and `number`. For session-based systems the trace generation step must address the issue of handling inter-request dependencies. For example, considering an e-commerce system, a session can issue a `Purchase` request only after a product has been added to the shopping cart through a previous `Add` request. Furthermore, trace generation must also address data dependencies. *Data dependencies* govern the choice of values for parameters in the name-value lists. For example, a `Login` request type in an e-commerce system has to be submitted with valid user name-password combinations.

Finite State Machines (FSMs) have been extensively used to model application-specific dependencies. A FSM consists of sets of states, inputs, and outputs. Applying a set of inputs causes transition from one state to another state and produces a set of outputs. FSM models are widely used to test whether an implementation of a software application conforms to specifications. An FSM is used to describe the specifications. In essence, the transitions in the FSM capture the desired behaviour for the implementation. Testing typically involves obtaining sequences of input sets called test sequences from the FSM. Each test sequence is applied to the implementation and the resulting sets of outputs are observed. Decisions about the correctness of the implementation are made by comparing the observed behaviour with the desired behaviour as given by the FSM.

Andrews *et al.* [1] proposed an FSM-based model to automate conformance testing of Web applications. First, a FSM-based model is used to describe a Web application. A state in the FSM corresponds to what the authors define as a logical Web page (LWP). A LWP can be a physical Web page produced by the application or specific HTML links and forms used to interact with the application. Transitions occur between LWPs in response to user inputs (e.g., submitting a user name and password through a HTML form). Inter-request dependencies are enforced by controlling the transitions allowed between LWPs. The authors also annotate the transitions to handle several other types of dependencies. The annotations provide information on whether an input for a LWP is required or optional and the sequences in which a user can supply inputs to a LWP. The authors demonstrated the technique on a simple student information Web application.

Menasce *et al.* [8] applied an FSM-based approach for the performance testing of Web applications. The approach uses a probabilistic FSM called a *Markov chain* to model a user's session with a Web application. The states of the Markov chain represent the different request types supported by the Web

application. Transitions between states model the user behaviour of navigating from one request type to another within a session. In contrast to a non-probabilistic FSM, transitions are associated with probabilities with the sum of a state's outgoing transition probabilities being one. The transition probability gives the likelihood of a user choosing a particular transition from a state from among all the allowed transitions. By traversing the Markov chain, a trace of synthetic sessions can be created for performance testing.

Both the FSM-based approaches have several limitations that may prevent them from meeting the performance testing requirements for session-based systems. We discuss these briefly as follows:

1) Inability to fully support inter-request dependencies – Both approaches assume a first-order dependency between request types. As per this assumption, the next request type that needs to be generated within a session (i.e., the next state) depends solely on the current request type. However, such an assumption may not be valid in practice. For example, consider the following valid session of request types for an e-commerce system: [`Home, View, Add, View, Add, Delete, Purchase`].This session describes a shopper who adds two products to the shopping cart, then deletes one of the products from the shopping cart before purchasing the other product. An FSM constructed based on this session and based on the first-order assumption would incorrectly deduce a dependency between `Delete` and `Purchase` without recognizing that the `Purchase` depended on one of the two previous `Add` states. As a result, the FSM could cause a sequence ([`Home, View, Add, Delete, Purchase`]) that invokes `Purchase` without any item in the shopping cart thereby violating inter-request dependencies for the system.

Higher-order dependencies can be captured, for example, by introducing states that represent sequences of request types. For example, creating a new state [`Add, View, Add, Delete`] can ensure that `Purchase` is invoked only when there is an item in the shopping cart. However, this may cause the well-known state explosion problem [7]. Another approach could be to build the FSM such that it only yields sequences with first-order request dependencies. Such an approach may not produce representative workloads since real workloads could contain several sequences with higher-order dependencies. We propose an alternate approach in Section 4.

2) Lack of support for data dependencies - Existing approaches have not focused on modeling data dependencies that are important for performance testing. Handling data dependencies is important since the correct choice of parameter values is essential for stressing the system under study in a representative manner. The application model must capture data dependencies so that they can be satisfied in an automated manner while creating workloads.

In this paper, we propose a new application modeling methodology to handle inter-request dependencies and data dependencies. The methodology relies on Extended Finite State Machines (EFSMs). EFSMs can model applications with higher-order request dependencies without encountering the state explosion problem. Consequently, they are better suited for modeling Web-based applications. Additionally, the modeling methodology allows different types of data dependencies to be captured. We present examples where our methodology is used to model an e-commerce application.

Our workload generation approach uses a modified version of the Session-Based Web Application Tester (`SWAT`) tool developed by Krishnamurthy [6]. The modifications allow `SWAT` to accept as input the application model for a system under study. `SWAT` uses a workload model which includes attributes that can influence the performance of session-based systems. The characterizations for these attributes can either be based on those observed in real systems or perturbations for the purpose of a sensitivity analysis. The application and workload models are used by `SWAT`'s trace generation algorithm to create a synthetic workload that has the correct inter-request dependencies and that has the specified characteristics. The chief advantage of `SWAT` is the fine control it offers over workload characteristics. For example, it permits the characterizations of one or more attributes to be changed at a time while keeping those of others unchanged so that a system's sensitivity to those characteristics alone can be established.

3. OVERVIEW OF THE MODEL-BASED PERFORMANCE TESTING APPROACH

Figure 1 provides an overview of our proposed approach. As mentioned previously, the tester provides an application model that captures the inter-request dependencies and data dependencies for the system under study. The modeling methodology and the process of creating a model are described in Section 4. The *sequence generator* uses the model to produce a large trace containing valid sequences of request types. We define each valid sequence of request types as a *sessionlet*. Each sessionlet in this trace satisfies the inter-request dependencies for the system under study. The trace of sessionlets is input to `SWAT` along with the workload model and the application model.

The workload model exposed by `SWAT` depends on the workload generation mode employed [6]. For the sake of brevity only the *session mode* of workload generation is discussed. In the session mode, new sessions are generated according to a session inter-arrival time distribution. *Session inter-arrival time* is defined as the time between successive arrivals of sessions at the system under study. Each generated session behaves as a user by issuing a request, waiting for the complete response from the system, and then waiting for an inactive period, defined as the *think time*, before issuing the next request. The think times are chosen according to a think time distribution while the number of requests per session is governed by a *session length* distribution. Finally, `SWAT` includes as attributes *workload mix* and the distributions for the values of parameters used within requests. Workload mix is defined as the overall proportions of the different request types in the workload. The parameter value distributions can be used to control the locality properties of name-value pairs (e.g., control the relative popularities of products in a bookstore).

We note that the `SWAT` workload does not include requests for objects such as image and multimedia files that are embedded within the HTML responses of the system. This is because such content is often hosted on external "graphics" servers, or at content delivery networks. Krishnamurthy [6] provides a more detailed discussion on the rationale behind the workload model chosen for `SWAT`.

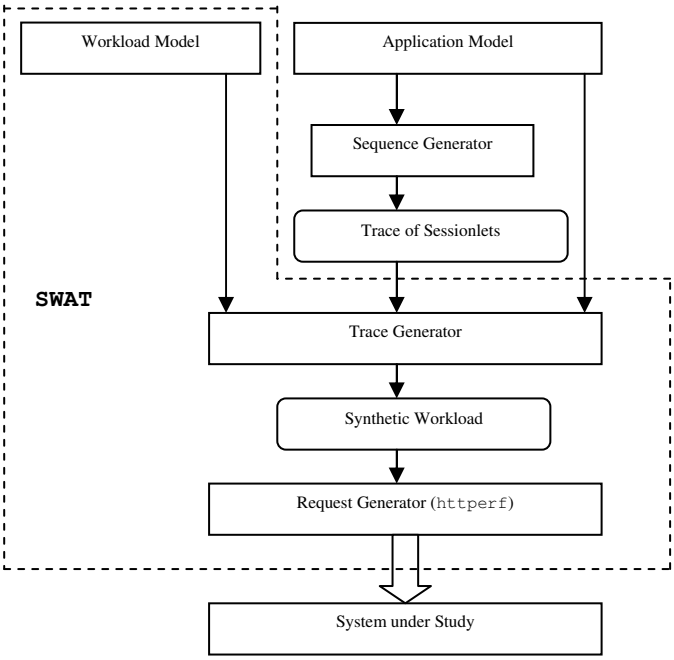

Figure 1. Model-Based performance testing approach

The objective of trace generation is to produce a trace of sessions that can be submitted to a system under study. Trace generation proceeds in two distinct steps. In the first step, the trace generator produces an intermediate trace of sessionlets. The intermediate trace is created by repeating selected sessionlets from the input trace of sessionlets. The trace generator determines the sessionlets selected and the number of times each sessionlet is repeated so as to closely match the specified workload mix and session length distribution. The next step in trace generation transforms the intermediate trace of sessionlets to a trace of sessions. This involves selecting name-value lists for request types to form URLs and inserting think times between successive URL requests in a session. The application model is consulted to handle data dependencies. The trace generator also ensures that the parameter value distributions specified in the workload model are achieved. With `SWAT`'s trace generation approach, inter-request dependencies are satisfied since each session in the synthetic workload has the same sequence of request types as one of the sessionlets in the input trace.

The sessions produced by the trace generator and the specified session inter-arrival time distribution constitute the synthetic workload. A modified version of `httperf`, an open-source request generator, is used to submit the synthetic workload to the system under study. The modifications were required to support certain commonly occurring data dependencies (explained in Section 4) and to facilitate finer-grained reporting of performance metrics. We note that our approach is not limited to `httperf`. The trace generator can be easily modified to produce synthetic workloads in formats that conform to other request generators. Furthermore, with the proposed approach testing different applications does not require the development of new scripts. Workloads can be generated for different systems by merely specifying different application models. For a given application, characteristics of the synthetic workload can be varied by changing the characterizations for the attributes in the workload model.

4. APPLICATION MODELING FOR PERFORMANCE TESTING

As mentioned in Section 2, the modeling methodology describes a Web application as an EFSM for the purpose of automating performance tests. Section 4.1 describes an EFSM and introduces related terminology. We introduce additional modeling elements to address inter-request and data dependencies and to accommodate the trace generation process described in the previous section. Section 4.2 presents examples to illustrate the modeling of inter-request dependencies. Modeling of data dependencies is discussed in Section 4.3

4.1 Overview of Modeling Methodology

In this section we briefly describe an EFSM. For a more detailed discussion readers are referred to the survey paper by Lee and Yannakakis [7].

An EFSM is described as the following quintuple:

$$M = (I, O, S, \vec{x}, T)$$

I, O, S, $\vec{x}$, and *T* are finite sets of *input symbols*, *output symbols*, *states, variables*, and *transitions*, respectively. A transition *t* in the set *T* is defined by 6-tuple:

$$t = (s_t, q_t, a_t, o_t, \vec{P}_t, A_t)$$

where s_t, q_t, a_t, and o_t are the current state, next state, input, and output, respectively. $\vec{P}(\vec{x})$ is a predicate constructed from the current variable values, $A_t(\vec{x})$ defines an action on the variable values.

The operation of an EFSM can be described as follows. Let the machine's initial state be $s_{initial}$ where $s_{initial}$ belongs to *S*. Let the initial values of variables be given by $\vec{x}_{initial}$. Assume that the machine is currently at state *s* and that the current variable values are $\vec{x}$. On receiving an input *a* the machine makes a transition $t = (s, q, a, o, \vec{P}, A)$ if the predicate $\vec{P}(\vec{x})$ evaluates to true. If the predicate evaluates to true, then the machine produces the output *o*, the values of the state variables are modified as per the function $A(\vec{x})$ and the machine moves to the state *q*. For this work we use an EFSM as follows. We define as input the act of a user submitting a request to the system. Consequently, an input is associated with a request type and a name-value list. Predicates constructed from the state variables are used to capture inter-request and data dependencies. A transition from one state to another is allowed only if the predicate associated with that transition evaluates to true. A successful transition may result in modification of state variable values as per the transition's action function. The EFSM is *non-deterministic* since more than one transition can be followed from a given state. For example, in an e-commerce system users maybe able to both sign-in as well as view products from the homepage. Since our focus is on workload generation we ignore outputs in this work. However, they could be interpreted as the Web page resulting from the

input.

To accommodate the approach described in Section 3, we introduce several additional model elements. Firstly, to facilitate the generation of an input trace of sessionlets an EFSM always has a `Start` state and an `Exit` state. They model respectively, the starting and termination of interactions a user has with the Web-based application. Secondly, each transition has two distinct sets of predicates and actions. *Request dependency predicates* and *request dependency actions* are involved in enforcing correct inter-request dependencies. *Data dependency predicates* and *data dependency actions* are used to satisfy data dependencies. As described later, such a distinction is necessitated by the two step trace generation process described in Section 3. Finally, data dependency actions may invoke `select` functions. The *`select`* function is used to choose a specific value for a given request parameter from among all the possible values for the parameter. The following sections provide a detailed description of these elements along with examples.

4.2 Modeling Inter-Request Dependencies

As described in Section 3, the sequence generator uses the application model to create a trace of input sessionlets. A sessionlet is generated as follows. The model is initialized by providing initial values to the state variables. The sequence generator causes a transition from the `Start` state by executing a randomly selected transition from among the set of allowed transitions from that state. Another transition is executed in a similar manner if the resulting state is not the `End` state. Sessionlet generation is complete if the `End` state is reached. The sequence generator outputs the sequence of inputs (i.e., request types) corresponding to the sequence of transitions executed. It then re-initializes the application model to generate more sessionlets. Valid sessionlets are produced as long as the application model enforces the correct inter-request dependencies.

We now present an e-commerce application example to illustrate modeling of inter-request dependencies. Figure 2 shows a simplified model for the application. In this application users execute the `Home` request type to request the homepage. The `Sign-in` request type allows a user to login as a registered user. A user can view product information through the `Browse` request type. The `Add` and `Delete` request types allow a user to add and delete items from the shopping cart, respectively. The `Checkout` request type allows a user to initiate ordering of products in the shopping cart. A user submits the `Purchase` request type to provide payment details for finalizing the order.

Two request dependency state variables are used to enforce inter-request dependencies. The `items_in_cart` is an integer variable that indicates the number of items in the shopping cart. The `signed_on` Boolean variable states whether a user has signed on or not. The initial values of the `items_in_cart` and `signed_on` variables are 0 and FALSE, respectively. The values of these variables are changed by actions associated with several transitions. For example, from Figure 2, submitting the `Sign-in` request type (transitions S_1 to S_7 and S_5 to S_7) changes the value of `signed_on` to TRUE. Similarly an `Add` request type (transition S_2 to S_3) increments `items_in_cart` variable by 1 while a `Delete` request type (transitions S_3 to S_4 and S_4 to S_4) decrements the variable by 1.

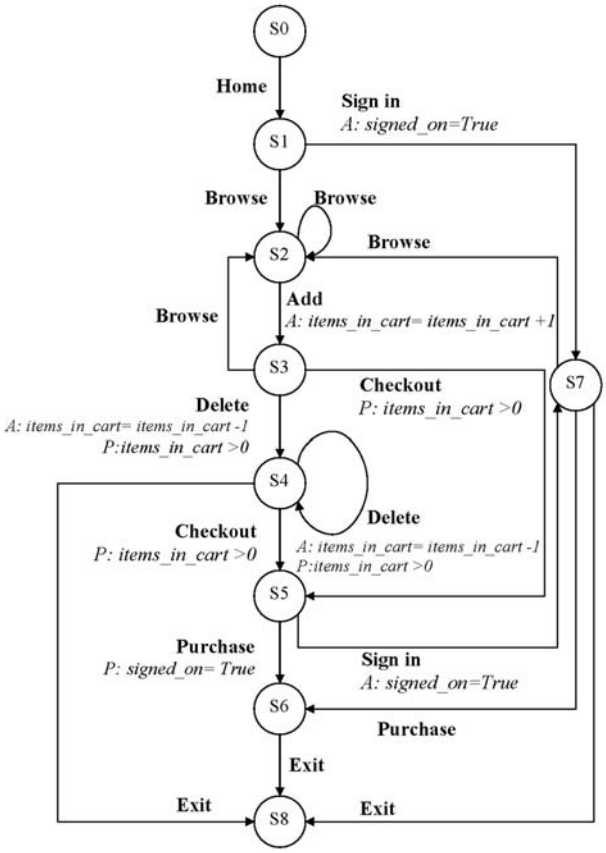

Figure 2. An EFSM for an e-commerce system

From Figure 2, certain transitions depend only on the current state of the EFSM. These first-order transitions are not associated with any predicates. For example, a user can submit a `Browse` request type after submitting a `Home` request type as indicated by the transition from S_1 to S_2. Similarly, a user can browse another product after browsing a particular product as indicated by the transition from S_2 to S_2.

Our application model also allows higher-order dependencies between request types to be captured. For example, consider the transition from S_4 to S_5 in Figure 2. In this transition, the user submits a checkout request after deleting an item from the shopping cart. This transition is allowed only when the previous sequences of requests have resulted in at least one item in the shopping cart. This dependency is enforced by the predicate associated with the transition which checks whether the `items_in_cart` variable is greater than 0. Consequently, the sequence [`Home, Browse, Add, Browse, Add, Delete, Checkout`] is allowed while the sequence [`Home, Browse, Add, Delete, Checkout`] is not.

The EFSM can model different ways in which a user can complete a given task. Such a scenario is very common in Web-based applications. In the example considered a user can either sign-in just immediately before purchasing (transition S_5 to S_7 in Figure 2) or sign-in immediately after visiting the homepage (transition S_1 to S_7 in Figure 2). As a result, the sequences [`Home, Browse, Add, Checkout, Sign-in, Purchase`] and [`Home, Sign-in, Browse, Add, Checkout, Purchase`] represent two possible ways for a user to purchase

an item. The EFSM handles the different scenarios through the predicate associated with the transition from S_5 to S_7 in Figure 2. The predicate uses the `sign_in` variable to determine whether or not a user has to sign-in before purchasing an item.

The simple example presented illustrates some of the limitations of the existing FSM-based approaches discussed in Section 2. As mentioned previously, in our approach request types constitute the input. In contrast, with the existing FSM-based approaches request types constitute the states of the FSM. Furthermore, due to the first-order dependency assumption the next state (request type) to be submitted in a session can be determined from the current state (request type) alone. However, the example presented shows that whether or not a certain request type can follow another request type can depend on certain complex preconditions being met or not. The existing FSM-based approaches are not expressive enough to capture such dependencies. For example, it is not possible to capture the conditional dependency between `Delete` and `Checkout` or `Checkout` and `Purchase` using the existing approaches. As a result, for systems characterized by complex inter-request dependencies only a limited number of unique sessionlets (i.e., those with only first-order transitions between request types) can be obtained from such models. Synthetic workloads constructed from such a limited number of sessionlets are not likely to be representative. Furthermore, our previous experience suggests that having a limited number of sessionlets may limit the ability of the trace generator to achieve arbitrarily desired mixes and session length distributions.

We note that an FSM whose inputs represent request types could also be used instead of an EFSM. However, the FSM equivalent of an EFSM typically has a larger number of states [7]. State explosion can occur for complex systems characterized by a large number of state variables and large numbers of possible values for state variables. Consequently, an EFSM-based approach can model Web-based applications in a more succinct manner.

4.3 Modeling Data Dependencies

As discussed in Section 3, the sessionlets generated with the help of the application model are used by the trace generator to create an intermediate trace of sessionlets exhibiting the desired workload mix and session length distribution. The intermediate trace has to be converted to a trace of sessions that can be submitted to the system under study. This is achieved by appending name-value lists for the request types in the intermediate trace.

We now describe how the application model is used to capture data dependencies. As mentioned in Section 3, data dependencies govern the generation of parameter values. The set of state variables used by the model includes the parameters for all request types supported by the application. Each parameter is denoted using the notation "`Request Type.Parameter Name`". For example, `Add.Item_ID` refers to the `Item_ID` parameter of the `Add` request type. Depending on how their values are chosen, the modeling methodology classifies parameters into several categories.

Tester-specifiable parameters are provided by the tester as inputs to the model. Considering an e-commerce system example, user names and passwords as well as product identifiers belong to this category. Tester-specifiable parameters are further subdivided into independent parameters and inter-dependent parameters. The value chosen for an *independent parameter* does not have any dependency with the value chosen for another parameter. In contrast, the value chosen for an *inter-dependent* parameter is controlled by the value chosen for another independent parameter. For example, in an e-commerce system the password selected for a sign-in request type will depend on the user name chosen. Currently, the model only allows such one-to-one dependencies. We note that, where appropriate, the choice of values for tester-specifiable variables can be controlled through the parameter value distributions specified to the workload model.

The values for *session-dependent* parameters depend on the sequence of URLs submitted in a session and hence cannot be specified explicitly by the tester. Considering an e-commerce system example again, the id of a product that needs to be deleted will depend on the products present in the shopping cart. These parameters are further classified into dynamically generated and non-dynamically generated parameters. The values of *dynamically generated parameters* are known only during request generation and hence cannot be resolved during trace generation. For example, the unique order identifier that is passed along with a purchase request type is typically assigned dynamically by the system only when the session is in progress. For such parameters, the trace generator uses placeholder values instead of the actual values. These placeholder values instruct the request generator that the actual values have to be obtained by parsing the responses of the Web pages returned by the system under study when the session is in progress. In contrast, values for *non-dynamically generated parameters* can be resolved during trace generation.

To generate name-value lists for a sessionlet in the intermediate trace, the sequence of states corresponding to the sessionlet are identified in the application model. As mentioned in Section 4.1, the state transitions have associated with them data dependency predicates and actions. The actions are used to choose parameter values for request types associated with the transitions. The actions use the `select` function to choose parameter values. As described shortly, the behaviour of the `select` function depends on the type of parameter being handled and the input arguments passed to the function. We present the following example to explain the process of handling data dependencies.

Consider the state sequence [`Start`, S_1, S_7, S_2, S_3, S_2, S_3, S_4, S_5, S_6, `Exit`] generated from the EFSM shown in Figure 2. This sequence corresponds to the sessionlet [`Home`, `Sign-in`, `Browse`, `Add`, `Browse`, `Add`, `Delete`, `Checkout`, `Purchase`]. Assume that `Sign-in` takes two parameters `username` and `password`. `Browse`, `Add`, and `Delete` accept a parameter called `product_id` denoting the item to be browsed, added to the shopping cart, and deleted from shopping cart, respectively. `Purchase` requires an `order_id` parameter whose value is dynamically assigned by the system. A state variable called `item_ids_in_cart` maintains the `product_id` values of the products in the shopping cart. We now discuss cases involving the generation of values for tester-specifiable and session-dependent parameters.

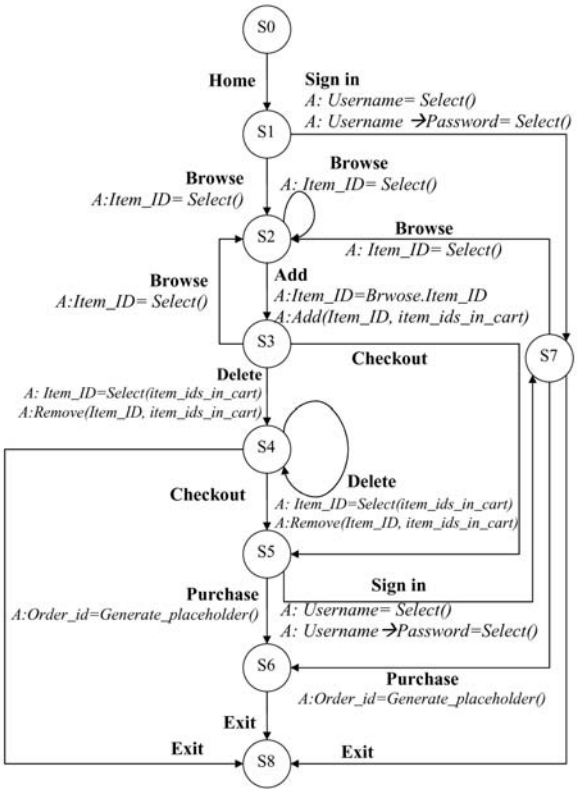

Figure 3. Data-dependencies in an e-commerce system

Tester-specifiable parameters - Consider the action associated with the transition from S_1 to S_7 in Figure 3. The `select` function first generates a value for the tester specifiable, independent parameter `Sign-In.username`. The `Sign-In.password` parameter is specified to be dependent on `Sign-In.username`. Consequently, the second call to `select` generates a value for password depending on the username selected in the first call. When many different values are possible for a tester-specifiable parameter, a value is either chosen randomly from among the possible values or as per a parameter value distribution, if such a distribution is specified in the workload model.

Session-dependent parameters – Consider the action associated with the transition from S_2 to S_3 in Figure 3. For this example, the `Add.item_ID` is session-dependent. As shown in Figure 3, the value of this parameter is the same as the value of the `Browse.item_ID` parameter chosen previously in the session. For certain session-dependent parameters there may be a choice between many possible values. In such cases, the `select` function is used to choose a value from among the possible values. This is illustrated in the action associated with the transition from S_3 to S_4. The `select` function takes as argument the `item_ids_in_cart` list variable. This variable is updated whenever an Add transition occurs (e.g., transition from S_2 to S_3 in Figure 3) and contains the ids of items added to the shopping cart. The function randomly selects a value from this list and assigns it to the `Delete.item_ID` parameter. The action also invokes the `remove` method on `item_ids_in_cart` to delete the item id from the list.

As mentioned previously, placeholder values are used for dynamically generated parameter values. Consider the action associated with the transition from S_5 to S_6. The call to the `Generate_Placeholder` function inserts a placeholder value for the dynamically generated `Purchase.order_ID` parameter. We note that the format of the placeholder value may differ for different request generators.

5. IMPLEMENTATION AND EVALUATION

We have developed tools to support the approach described in this paper. Testers use a text file to specify the application and workload models. Details of the specification language have been omitted due to space constraints. A model verifier has been developed to check for consistency in the application model. For example, the program flags an error when no values are provided for a variable declared as tester-specifiable variable. The model verifier also checks for consistency between the application model and workload model. For example, an error is reported when a request type specified in the workload model does not appear in the application model and vice-versa. If the model consistency check is successful, the model verifier generates the application and workload models in a less verbose format suitable for the sequence and trace generator tools shown in Figure 1.

We applied the proposed approach to develop an application model for the Java Pet Store e-commerce application (version 1.4) [5] used frequently in performance studies. The application model was used to generate a large trace of sessions for the application. A subset of sessions representing different sequences in which users could perform tasks such as browsing and purchasing was selected from this trace. The validity of these sessions was then verified by submitting them to the Pet Store system using `httperf` and observing the HTML responses generated.

Simple experiments were conducted on the Pet Store system to study the effect of inter-request dependencies. We selected two sessions from the trace of sessions created using our approach. The first session, named `Browse`, contained requests to browse items in the pet store. It did not contain any sign-in, shopping cart, or purchase related requests. The other session, named `Purchase`, predominantly consisted of purchase related requests. For the Pet Store application, the browse related requests do not have any dependency with other requests and can occur at any point in a session. However, purchase related requests have more complex inter-request dependencies similar to those modeled in Figure 2. We also created scrambled versions of the two sessions called `Browse-Scrambled` and `Purchase-Scrambled`. The scrambled versions have requests occurring in a random order and hence ignore dependencies.

The experiment setup was as follows. The Pet Store application was installed on a Pentium III, 1 GHZ, 768 MB RAM machine running under the Windows XP operating system. `httperf` was executing on a Pentium III, 1 GHZ, 512 MB RAM machine running under a Linux operating system. Since we were interested in characterizing the aggregate resource demand placed by a session on the system's resources, the number of concurrent sessions accessing the Pet Store was set to be 1. `httperf` was used submit a session and measure the mean response time for

requests in the session. We define *response time* as the time between sending the first byte of a HTTP request and receiving the last byte of the corresponding HTTP response. The network connecting the `httperf` machine and the Pet Store machine was very lightly loaded during the experiments. Since there was no contention among sessions for system resources and since the network was lightly loaded, the mean response time reflects the end-to-end resource demands placed by the session across all resources in the Pet Store system. Multiple runs were carried out for each experiment to achieve statistical confidence in the results.

Table 1 shows the results from our experiments. There is no significant difference between the mean response times for the `Browse` and `Browse-Scrambled` sessions. This is not surprising given that browse related requests can occur in any order within a session. However, the mean response time for the `Purchase` session is almost 1.73 times the mean response time for the `Purchase-Scrambled` session. The mean response times for these sessions were statistically different since they had non-overlapping 95% confidence intervals. The `Purchase-Scrambled` workload places less stress on the system since certain requests (e.g., shopping cart, sign-in, checkout, purchase) impose less demand on the system resources if they occur at an incorrect point in a session. This result reinforces the importance of preserving correct inter-request dependencies in synthetic workloads for session-based systems. When significant inter-request dependencies are present, ignoring them can yield incorrect performance estimates and can therefore cause incorrect conclusions to be drawn from performance tests. As a result, we believe our approach adds value to the performance testing process by providing a mechanism to capture and preserve complex dependencies in session-based systems.

Table 1. Effect of inter-request dependencies

Workload	Mean response time (ms)
`Browse`	97.6
`Browse-Scrambled`	101.4
`Purchase`	390.6
`Purchase-Scrambled`	226.3

6. CONCLUSIONS AND FUTURE WORK

This paper proposed a model-based approach for testing the performance of Web applications. The approach simplifies the generation of synthetic workloads used in performance tests. It uses a formal model to capture an application's inter-request and data dependencies. The model can be used to obtain several sequences of requests representing how users typically interact with the Web-based application. The sequences can in turn be used to construct synthetic workloads with specified characteristics. The application model provides an indirection which allows a common set of workload generation tools to be used for testing different applications. Consequently, less effort is needed for developing and maintaining the workload generation tools and more effort can be dedicated towards the performance testing process.

The approach requires a certain effort on the part of the tester to specify the application model. However, we believe this effort would be much less than the effort needed to carry out performance tests using a traditional approach. Future work will focus on reducing the effort needed to develop the application model. Specifically, we intend to explore ways in which application documentation (e.g., UML object diagrams, message sequence charts) can be exploited to automatically generate application models.

Future work will also focus on a more extensive evaluation of the proposed approach. We are currently working on case studies that demonstrate the flexibility the approach provides for varying workload characteristics in a controlled manner. We also plan to demonstrate the generality of the approach by using it to construct synthetic workloads for different session-based applications such as Java Pet store, TPC-W [11], and RUBiS [10]. Finally, we intend to carry out a systematic study that establishes the advantages of our approach over the FSM based approaches discussed in Section 2.

7. ACKNOWLEDGMENTS

This work was financially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Calgary.

8. REFERENCES

[1] Andrews, A., Offutt, J. and Alexander, R. Testing Web-Applications by Modeling with FSMs. *Software Systems and Modeling.* Vol. 4, No. 3, pp. 326-345, July 2005.

[2] Banga, G. and Druschel, P. Measuring the Capacity of a Web Server under Realistic Loads. In *Proceedings of the International World Wide Web Conference.* 1999.

[3] Barford, P. and Crovella, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of ACM SIGMETRICS.* pp. 151-160, 1998.

[4] Ferrari, D. On the Foundation of Artificial Workload Design. In *Proceedings of ACM SIGMETRICS.* pp. 8-14, 1984.

[5] JavaPet Store, http://java.sun.com/developer/releases/petstore/

[6] Krishnamurthy, D. *Synthetic Workload Generation for Stress Testing Session-Based Systems.* PhD. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 2004.

[7] Lee, D. and Yannakakis, M. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of IEEE.* Vol. 84, pp. 1090-1123, August 1996.

[8] Menasce, D., Almeida, V., Fonesca, R. and Mendes, M. A Methodology for Workload Characterization of E-Commerce Sites. In *Proceedings of the ACM Conference on Electronic Commerce.* pp. 119-128, 1999.

[9] Mosberger, D. and Jin, T. httperf – A Tool for Measuring Web Server Performance. In *Proceedings of the Workshop on Internet Server Performance*, 1998.

[10] RUBiS, http://rubis.objectweb.org/

[11] TPC-W, http://www.tpc.org/tpcw