Realistic Load Testing of Web Applications

Dirk Draheim
Institute of Computer Science
Freie Universität Berlin
Takustr.9, 14195 Berlin, Germany
draheim@acm.org

John Grundy, John Hosking,
Christof Lutteroth, Gerald Weber
Department of Computer Science
The University of Auckland
38 Princes Street, Auckland 1020, New Zealand
{john-g,john,lutteroth,gerald}@cs.auckland.ac.nz

Abstract

We present a new approach for performing load testing of web applications by simulating realistic user behaviour with stochastic form-oriented analysis models. Realism in the simulation of user behaviour is necessary in order to achieve valid testing results. In contrast to many other user models, web site navigation and time delay are modelled stochastically. The models can be constructed from sample data and can take into account effects of session history on user behaviour and the existence of different categories of users. The approach is implemented in an existing architecture modelling and performance evaluation tool and is integrated with existing methods for forward and reverse engineering.

1 Introduction

Web applications are ubiquitous and need to deal with a large number of users. Due to their exposure to end users, especially customers, web applications have to be fast and reliable, as well as up-to-date. However, delays during the usage of the Internet are common and have been the focus of interest in different studies [2, 6]. The demands on a web site can change very rapidly due to different factors, such as visibility in search engines or on other web sites. Load testing is thus an important practice for making sure a web site meets those demands and for optimizing its different components [1]. Continual evolution of web applications is a challenge for the engineering of this class of software application. After each maintenance cycle, a convincing performance test must include a test of the full application under realistic loading conditions. In [16] it is recommended that load testing of a web site should be performed on a regular basis in order to make sure that IT infrastructure is provisioned adequately, particularly with regard to changing user behaviour and web site evolution. Also other experiences stress the importance of load testing for the prediction and avoidance of service-affecting performance problems [23] at earlier stages of a web site's life cycle.

However, realism in the simulation of user behaviour for the purpose of load testing has been found to be crucial [29, 1]. "A load test is valid only if virtual users' behaviour has characteristics similar to those of actual users" because "failure to mimic real user behaviour can generate totally inconsistent results" [16]. Most current tools for load testing support the creation of simple test cases consisting of a fixed sequence of operations. However, in order to give the generated load some variety it is usually necessary to modify and parametrize these test cases manually. This is usually both time-consuming and difficult. A more elaborate approach is needed in order to generate a realistic load, and such an approach requires more advanced tool support.

Our approach applies the methodology of form-oriented analysis [9], in which user interaction with a submit/response style system is modelled as a bipartite state transition diagram. The model used in form-oriented analysis is technology-independent and suitable for the description of user behaviour. It describes what the user sees on the system output, and what he or she provides as input to the system. In order to simulate realistic users we extend the model with stochastic functions that describe navigation, time delays and user input. The resulting stochastic model of user behaviour can be configured in different ways, e.g., by analyzing real user data, and can be used to create virtual users of different complexity. The level of detail of the stochastic model can be adjusted continuously. All this enables our load test tool to generate large sets of representative test cases. Furthermore, our load test tool has its natural place in a chain of tools that support software engineers working on web applications. Those other tools also use the form-oriented model and thus help a software engineer to reuse or recover a web site's model for load testing.

Sect. 2 gives an overview of the methodology of form-oriented analysis that we use throughout the paper. Sect. 3

explains how this methodology can be applied in order to perform realistic simulation of web site users. Sect. 4 provides some information about how load tests are actually performed and delineates the whole picture of load testing; Sect. 5 explains the parameters used to describe workloads. Sect. 6 discusses related work, and Sect. 7 concludes the paper.

2 The Form-Oriented Model

Form-oriented analysis [9] is a methodology for the specification of ultra-thin client based systems. Form-oriented models describe a web application as a typed, bipartite state machine which consists of *pages*, *actions* and transitions between them. Pages can be understood as sets of *screens*, which are single instances of a particular page as they are seen by the user in the web browser. The screens of a page are conceptually similar, but their content may vary, e.g., in the different instances of the welcome page of a system, which may look different depending on the user. Each page contains an arbitrary number of *forms*, which in turn can have an arbitrary number of *fields*. The fields of forms usually allow users to enter information, and each form offers a way to submit the information that has been entered into its fields to the system. A submission invokes an action on the server side, which processes the submitted information and returns to the client a new screen in response. Hyperlinks are forms with no fields or only fields that are hidden to the user.

Form-oriented models can be visualized using *formcharts*. In a formchart the pages are represented as ovals and the actions as boxes, while the transitions between them are represented as arrows, forming a directed graph. Formcharts are bipartite directed graphs, meaning that on each path pages and actions occur alternatingly. This partitioning of states and transitions creates a convenient distinction between system side and user side: the page/server transitions always express user behaviour, while the server/page transitions always express system behaviour.

In Fig. 1 we see the formchart of a simple home banking system, which will be the running example of this paper. The system starts showing page Login to a user, who can enter an account number and access code. This data is submitted to action Verify, which checks if it is correct and either redirects the user back to the Login page or to the Menu page of the home banking system. Here the user can access the different functions, i.e., showing the account's status, making transfers, trading bonds, and logging out. Each of the functions may involve different subsystems and make use of different technical resources.

A form-oriented model of a web application offers several benefits. It is suitable for testing as well as for the analysis and development of dynamic web applications. A

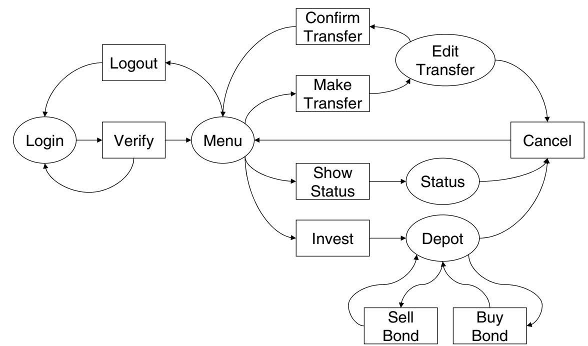

Figure 1. Formchart of example home banking web application.

typical shortcoming of many other models is that they do not capture fan-out of server actions, i.e., the ability of a server action to deliver many conceptually different client pages, which is covered by the form-oriented model.

3 Modelling User Behaviour with Stochastic Formcharts

Formcharts specify web applications, which usually work in a strictly deterministic manner. In a load testing scenario, however, the web application already exists, and the problem is to simulate the behaviour of a large number of users. But just as a formchart is a specification of the web application, it is also a specification of possible user behaviour; and while it is the web application that chooses in an action which page will come next and which data will be shown on the page, it is the user who chooses which of the available actions will be invoked afterwards and which data the action will get. In other words, when simulating users we have to model their navigational choices and the input they enter. Since we are aiming at real-time simulation, we also need to model the timing of user behaviour. In the case of web applications, this can be reduced to a model of user response time or "think time", i.e., the time delay between reception of a screen and submission of a form. Since the fine-grained interaction involved in user input happens at the client side, transparent to the server, we do not model it.

We cannot predict user behaviour as we can predict the behaviour of a web application. Therefore, we use a stochastic model, which makes only assumptions about the probability of a particular user behaviour and not about which behaviour will actually occur. When estimating such probabilities, it can be important to take into account the session history of a user, which may influence the decision about the next step. For example, a user that has just

logged into the system is unlikely to log out immediately afterwards, but much more likely to log out after he or she did other things. Consequently, we are dealing with conditional probabilities.

The essential decision that a user makes on every page is about which form he or she will use. This affects much of the behaviour that will follow, so in our simulation we should make this decision first. There is always a limited number of forms on a page, and the question is how probable it is for each form to be chosen. This is expressed by probability distribution P_{form}, which also takes into account the session history. $Histories$ is the set of all possible session histories, and $Forms$ is the set of all forms. A form that is not available at a certain point in session history has probability 0; if there is only a single form available, it has probability 1.

$$P_{form}\colon Histories \times Forms \rightarrow [0,1]$$

Once a form is chosen, we estimate a delay. This is done with p_{delay}. Time is a continuous variable, therefore p_{delay} is not a discrete probability distribution but a probability density function.

$$p_{delay}\colon Histories \times Forms \times (0,\infty) \rightarrow \mathbb{R}^{+}$$

Again, we acknowledge that the session history may have an influence on delay time, but we expect this effect to be much weaker than the effect of history on form choice. Therefore, we might simplify our model by neglecting session history. In our example, the delay probability density graphs for the forms on page Menu that lead to actions Logout, Status, Make Transfer and Invest, respectively, could look like the ones in Fig. 2.

One of the more complex tasks is the generation of input data for the fields of the chosen form. P_{input} is a discrete probability distribution that describes the probability that certain data is entered into the form. $Data$ is the universe of possible data, and if some particular data in it cannot be entered into a form, the value of P_{input} is 0.

$$P_{input}\colon Histories \times Forms \times Data \rightarrow [0,1]$$

Also P_{input} depends on the session history. If, for example, the user needs a secret one-off transaction number (TAN) to add additional security to each transaction, such a TAN is only used once, hence the probability for that TAN to be used again shrinks. However, usually we can neglect the effect of session history on user input and use some simple logic instead in order to cope with such dependencies. Approaches for the generation of form input exist and have been discussed, for example, in [8, 3, 7].

Depending on the data we have about real user behaviour, there are different possibilities for us to configure the user model. In some cases, e.g., when the system we want

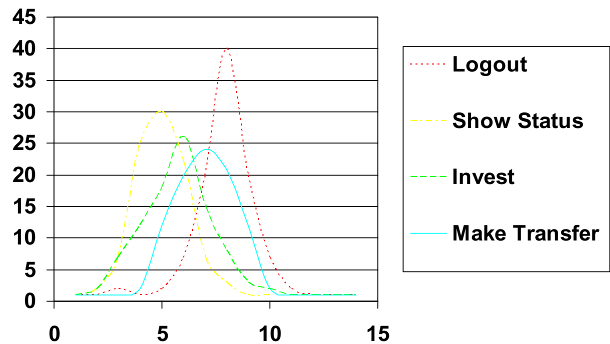

Figure 2. Probability densities for the delay caused with different forms.

to test is new, there might not be any sample of real user data yet. But when the system has already been used for some time there might be plenty of empirical historical user behaviour data. In both cases we try to make the user model as realistic as possible.

3.1 Repetition Models

After surveying real user session data, we can use it directly by replaying it on the system. Of course, the system would usually have to be reset to its original state first because it is not generally possible to play a session, e.g., for selling a bond, twice. We can adjust the load of the system by changing the time interval in which the sessions are replayed, or, when a session can be replayed an arbitrary number of times, the frequency of repetition. Form-oriented analysis offers suitable concepts for storing and representing session histories.

For surveying user behaviour we need a suitable instrument. In [8] we suggested the Revangie tool that can "snoop" on the communication between clients and server. Such a tool could be used for recording real user behaviour. Alternatively, server logs can be used. The problems and procedures of collecting real user data, and the benefit in the context of test case generation for functionality testing has been discussed, for example, in [14].

3.2 History-Free Stochastic Models

As discussed in the previous section, we can simply repeat real user sessions. But we can also exploit real user data in order to find suitable parameters for a stochastic model. A stochastic model is a more general approach and therefore more versatile. It is much easier to create new load testing scenarios by adjusting model parameters. Furthermore, replaying recorded data might not cause a system

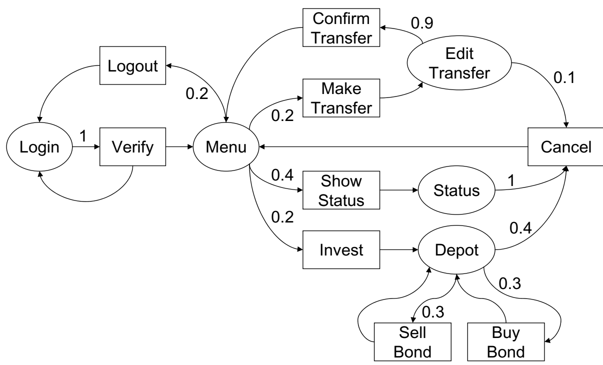

Figure 3. Simple stochastic formchart for the home banking system.

load that is representative. To create a representative repetition model requires a random sample of real user sessions with a sufficient size. If this is not given, the model might simply fail to cover the variety of possible input behaviour well enough, which might result in flaws of the system remaining undetected. In contrast to that, load testing with a stochastic model uses a randomized algorithm, which is generally less prone to yield tendentious results.

If we have data about how many times each form has been submitted, i.e., the total usage frequency of each form, we can use this to approximate P_{form}. The probability of a form to be chosen is set to the relative usage frequency, i.e., the number of usages of that form divided by the sum of the numbers of usages for all the forms on that page. This neglects the effect of session history on P_{form} and produces a stochastic formchart like the one in Fig. 3. In this formchart all page-action transitions are annotated with a transition probability, which reflects function P_{form}. At each page, the probabilities of all outgoing transitions sum up to 1 or possibly a bit less, with the remaining probability reserved for abrupt termination of the session. The transitions from actions to pages are performed by the system and therefore need no annotation. The problem of choosing transition probabilities for similar stochastic models is also discussed in [13, 27].

Such stochastic formcharts are similar to Markov chains, but there is a subtle and important difference: while a Markov chain creates a state machine with probabilities at every transition, a stochastic formchart is a bipartite state machine with probabilities only at the transitions going from page to action. Which transition will be chosen from action to a page is determined by the logic of the system, which is well-defined. Consequently, it makes no sense for load testing to model also this aspect stochastically and add probabilities to the action-page transitions, too. We cannot get rid of action-page transitions because an action can have more than one outgoing transition, such as the action Verify.

3.3 History-Sensitive Stochastic Models

When we have samples of real user sessions and not just unrelated usage frequencies, we are able to create an empirical model that takes into account the effects of session history on P_{form}. A good method to define P_{form} with the help of this data is a decision tree (see, for example, [12]), which captures the relation between past events and future ones. Each path in the tree is a sequence of pages and actions, alternating, and represents a possible user session of a certain length. If there are cycles in the original formchart of a system, a corresponding decision tree can have arbitrary depth, and actions and pages of the original formchart can occur multiple times. In the decision tree we distinguish these multiple occurrences of actions and pages by giving them running indexes. All these actions and pages with index but the same name correspond to a single action or page in the original formchart.

Look, for example, at Fig. 4, which shows a possible decision tree for our home banking system. The root of the tree represents the state in which the system starts, i.e., page Login. Since it is the first occurrence of this page in our tree, we add the index 1 to its name. There is only one form on page Login, i.e., the form to enter account number and PIN. Logically, the probability that action Verify, which is invoked by that form, is chosen is 1 (or a little bit less if we would consider abrupt termination). The next two outgoing transitions of Verify are action-page transitions and therefore need no probability, as we have discussed in Sect. 3.2. The first of these transitions is chosen by the system when the authentication failed and leads back to page Login_1. This page, Login_1, is the same as the root of our tree. Formcharts allow us to visually represent actions and pages arbitrarily often, which can be good to avoid ugly transition arrows crossing over the diagram. If we want a page or action to have a certain transition, we can add a corresponding arrow to any of its corresponding bubbles/rectangles. The fact that there is a transition from Verify_1 back to Login_1 signifies that if the system chooses to go back to page Login, the user will, with regard to P_{form}, behave stochastically just the same as when he or she first entered the system at that page. This equivalence of user behaviour also includes future behaviour, i.e., the probabilities of form choices on pages to come. We need this recurrence to states in order to handle cycles in the original formchart, which could not be represented with a finite decision tree otherwise. On the arrows representing the outgoing transitions of page Menu_1 we see the probabilities with which the user chooses different forms just after he or she logged in. Let us have a closer look at what happens when the user chooses the form that leads to ac-

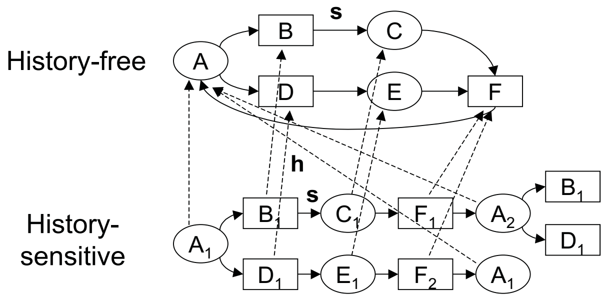

Figure 5. Homomorphism from history-sensitive to history-free formchart model.

tion Invest. Page Depot_1 offers a form for action Cancel_1, which means that a simulated user that invokes it will come back to page Menu_1 and stochastically behave like a user who has just logged in. There are also forms for actions Buy Bond and Sell Bond, which are assigned equal probabilities. When action Buy Bond is invoked, we get back to page Depot. However, the probabilities have changed, and we model this situation with a new page Depot_2. The probability for the invocation of Cancel is now higher, i.e., once a bond has been bought, the user is more likely to go back to Menu. Furthermore, the probability to buy another bond is now higher than the probability to sell one, which might reflect, for example, the common pattern that new bonds are bought from the money old ones were sold for, but not vice versa. In all the three possible cases we refer back to other parts of the decision tree, which means that the following pattern of user behaviour is already part of the model. However, if we found that after some action the user behaviour differs significantly from what has already been modelled, we would add another layer of pages and actions to the tree that reflects these differences. Following this pattern, the effects of session history in P_{form} can be approximated arbitrarily.

It is important to note that the relation between a system's original history-free formchart and a corresponding history-sensitive stochastic formchart model is formally well-defined. There exists a *homomorphism* h between the pages and actions U of a formchart that takes into account session history and the pages and actions V of the corresponding one that does not. $h: U \rightarrow V$ maps all pages or actions $X_i \in U$ to the corresponding page or action $X \in V$. Any sequence of action invocations s performed from a certain page $u \in U$ that ends at a page $s(u) \in U$ can also be performed on page $h(u) \in V$ and will end at page $h(s(u)) \in V$. This relation is illustrated in Fig. 5.

3.4 Multimodels

Within the population of all users there are usually different categories of users that use some features of the web application particularly often, which causes differences in P_{form}, or with particular skill, which generally reflects in p_{delay}. The presence of distinct groups of novice users and expert users, for example, can be visible in the fact that the graph of p_{delay} has more than one peak.

In order to find out which categories exist, we can analyse data about real user sessions and perform a classification. One method for doing this automatically is clustering (see, for example, [12]). When we have divided the sessions into different categories, we can create a user model for each category using the respective subset of data. Once we have a set $Models$ of different user Models, we can define a distribution P_{model} of the probability that a user is of a certain category and has thus to be simulated with the corresponding model.

$$P_{model}: Models \rightarrow [0, 1]$$

This distribution can simply be inferred from the relative sizes of the different categories. In order to produce a realistic load during load-testing, we stochastically choose a user model every time we create a new virtual user. When load-testing models incorporate several user models, we call them *multimodels*.

Such multimodels enable prioritization techniques proposed for functional testing, e.g., as discussed in [10]. Such techniques, which execute more important test cases more often than others, can also be reasonable from a load testing perspective, for example for making sure that particularly popular or critical parts of a web site are tested thoroughly. Such a multimodel can also be understood as a complex form of operational profile [18] that takes into account different types of user behaviours with their respective occurrence probabilities. In certain circumstances, multimodels can be subsumed under history sensitive formchart models.

4 Performing Load Tests

In order to actually perform a load-test, we have to use one of the described user models to create a *workload* on the system, i.e., an activity that consumes some of its resources. For this task, we can choose between different workload models, which describe workload over time. In order to push a system to its limits we could, for example, utilize the increasing workload model, which adds more and more virtual users to the system. The parameters that can be used to describe a workload are explained in Sect. 5 It is important that the load test environment the system is running in is very close to the production environment, i.e.,

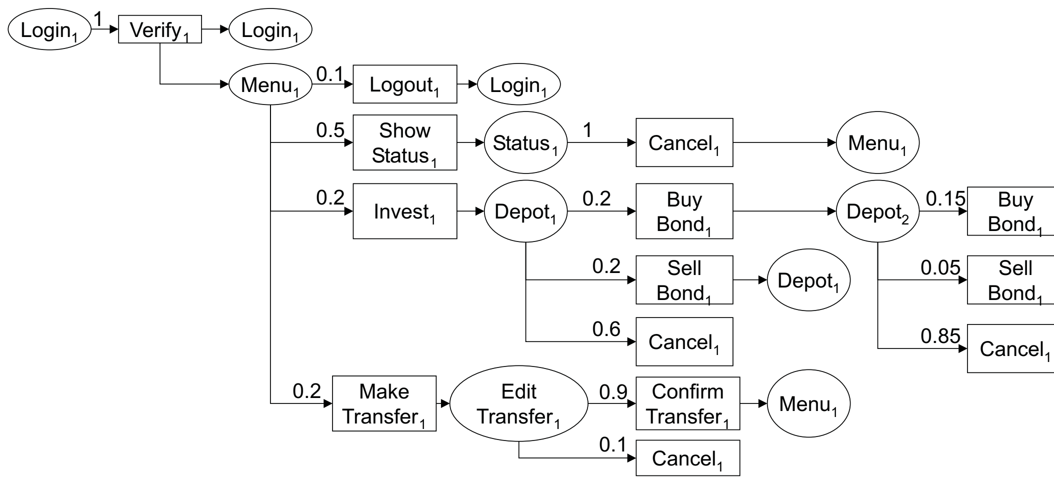

Figure 4. History-sensitive formchart model for the home banking system based on a decision tree.

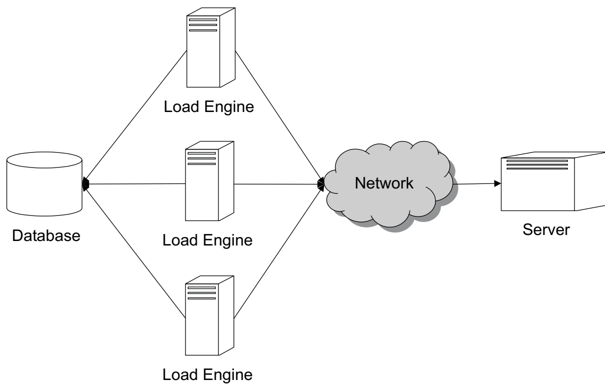

Figure 6. Load-testing environment setup.

the one the system is eventually intended to work in. The most accurate results will be produced when the load test is performed in the production environment itself, although this is usually not possible in a system that is being used already. As one can see in the schematic illustration of the load testing environment in Fig. 6, we perform the load-test with the help of dedicated computers, which are called *load engines*. The load engines simulate virtual users and write data about the state of the testing environment and its measured performance to a database. Another important requirement of the test environment is that the load engines themselves are bottleneck-free. If this is not the case, then performance measurements of the server will be distorted by delays introduced by the load engines [1]. That is why, from a certain number of virtual users on, we distribute the simulation of the users onto several load engines.

Now let us consider what each of the virtual users actually does. According to the model, an initial page request is sent to the system. The load engine lets the virtual user wait for a random time given by p_{delay}. Then, a form and corresponding input is chosen randomly by P_{form} and P_{input}, respectively. This information is used in order to create and send a new request. Since the server action corresponding to the chosen form may generate screens of different pages, the received screen has to be classified in order to determine which of the possibly many action-page transitions was chosen by the system. How such classification can be done has described, for example, in [8]. Once the classification is done, the load engine can again simulate a delay. Another aspect of virtual users worth mentioning is that it can sometimes be necessary to use a managed set of virtual user profiles. This means maintaining and using data that cannot be generated randomly, e.g., account numbers and corresponding PINs for our home banking example. These data can also be stored in the database all the load engines are connected to.

4.1 Finding Performance Bottlenecks

When looking for system bottlenecks, we usually observe the system in a state of extreme load, which is also called a *stress test*. An important measure to keep track of is the roundtrip-response time [16], i.e., the time between the request of a load engine to the server and the reception of its response, because this reflects the time a real user would have to wait. Also the response time on the server, i.e., the time between the reception of a request and the sending of a response, is usually monitored. Other common measures are, for example, the CPU and memory usage on the server. More about metrics for load testing can be found in [21].

A common way to perform stress testing is to use an increasing workload model and add virtual users until the response time crosses a threshold that we consider long enough to render the system unusable, i.e., longer than a

user would probably be willing to wait. With the data collected by the load engines we are able to analyse the response time of individual actions. The number of users where the response time crosses a certain limit usually differs between the actions, depending on how much they are invoked and the technical resources they are driven by. Actions for which this limit is crossed early present a bottleneck of the system and usually allow us to draw conclusions about the underlying subsystem.

4.2 Load Testing of Legacy Systems

Often we face the task of load testing a legacy system, i.e., a system which is already deployed and running, and for which the information necessary to create a realistic user model is not available. In such a case we could either use a very simple user model, such as a generic one that invokes actions randomly with a uniform distribution, or try to extract the necessary information by means of reverse engineering. In [8] we proposed a methodology and a tool called Revangie which is able to reconstruct form-oriented analysis models for existing web applications. Models can be constructed online, i.e., during system exploration, but also offline, e.g., from recorded user data. There also exist other tools for model recovery of web sites that can be useful for the creation of load models, e.g., the ones described in [3].

4.3 Load Testing in the MaramaMTE Tool

Our implementation extends earlier work on performance estimation of distributed systems generated using the ArgoMTE tool. This tool generates testbeds for such systems from descriptions of their software architecture allowing performance estimation to be carried out at design time and lessens the cost of experimenting with multiple software architecture choices [5]. This work has recently been ported to an Eclipse-based implementation in the form of the MaramaMTE tool. A screenshot of MaramaMTE in use specifying a software architecture is shown in Fig. 7. In this example, the architecture modelled consists of a set of RMI remote objects (CustomerManager, UserManager, AccountManager) and database tables (customer, user, account). MaramaMTE permits specification of client-based testing from the architecture description using the simple assumption that internal code within modules is much lower cost than inter-module communication (valid for most applications; note if actual code has been implemented for a module this can be used in place of the testbed generated code). Clients can be multi-threaded and client load can be repeated; in this example ClientTest1, ClientTest2 are repeated each 1000 times. While this tool is very successful, it lacks the ability to model user interaction appropriately. Accordingly, we have implemented our load testing approach for the history free stochastic model in the MaramaMTE toolsuite.

Our implementation adds a formchart view to MaramaMTE for editing the model (Fig. 8). The test designer can define the interaction model and annotate the transitions with probabilities. Delay distributions can be defined as described earlier. For the load testing process the test designer can specify a workload model. The actions specified in the formchart view are linked to remote service calls specified in the MaramaMTE architecture views, combining the two models together; for example the login_Test2 page in Fig. 8 is linked to the findUser service of the UserManager component in Fig. 7. The load test can be activated from the MaramaMTE toolsuite. This toolsuite possesses a sophisticated remote deployment functionality; the tool can automatically deploy load test agents acting as clients on different machines and orchestrate their activity. From the MaramaMTE tool, the whole load test architecture can be controlled.

Combining our stochastic model-based approach with MaramaMTE provides a very powerful model-based performance estimation approach where realistic estimates of a web application's performance can be performed at design time before significant implementation expenses have occurred.

5 Workload Models

A workload is completely described by a user model and one of several possible workload parameters; both elements together give a workload model. A user model is a stochastic formchart with user delays. A workload parameter can be a function over time, modelling a changing workload. For the introduction and comparison of the different parameters here we focus on constant parameters. We will introduce workload parameters that are partly based on virtual users, partly on the concept of user sessions. A *user session* is the model of what we consider as one typical connected usage of the system by a user; we assume that users always explicitly start end end their sessions. User sessions can be modelled by identifying the session delimiting transitions like login and logout in a formchart; the user session model is then a part of the user model. Given a stochastic formchart, a session model may have a defined *average number of requests* ($AVGR$) before the session terminates; note that not every distribution has to have such an expected value. But it is possible to statically check whether a stochastic user session model has such a finite $AVGR$; then we call it a finite user session. The *client request rate* (CRR) is the request rate of the individual virtual user. It is determined by the server response time and the user think time after receiving the response. If this think time is kept con-

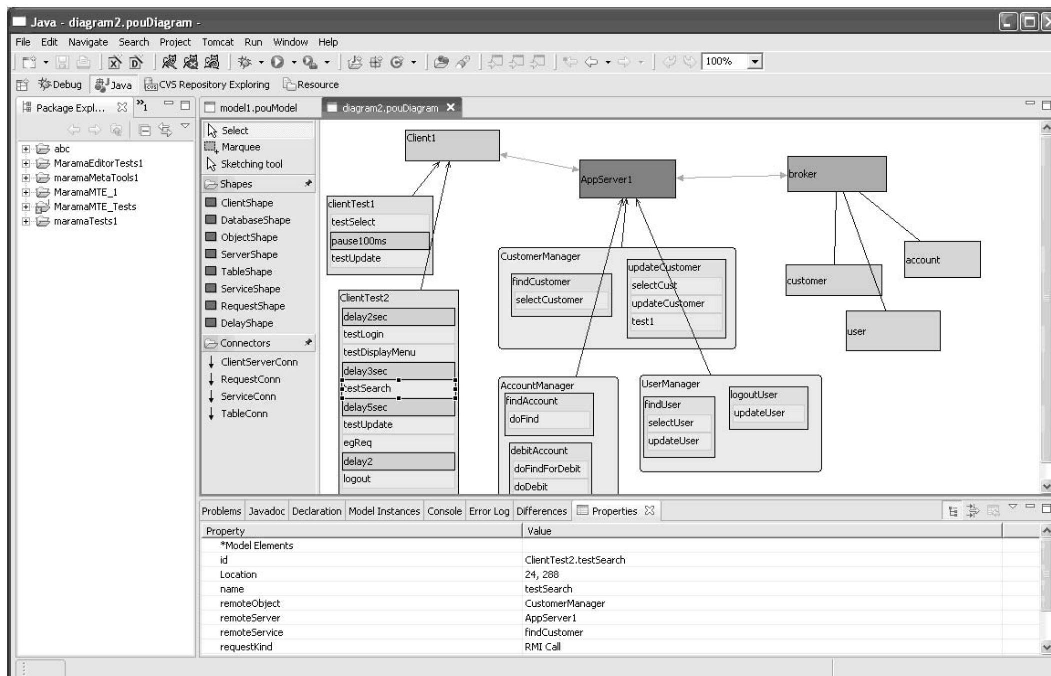

Figure 7. Screenshot of the architecture view of the MaramaMTE tool.

stant, then CRR decreases as soon as the server response degrades. For load testing tools, a load-independent CRR is often recommended, but this requires in general a non-trivial implementation [1]. We will also consider the *average session duration* ($AVGD$). This value takes into account server response times as well as think times. If the server response times are negligible, then exactly the finite user sessions have a defined $AVGD$.

There are two classical workload parameters, one is the *number of virtual users* (VU), that is the number of user processes active at a point in time. Another workload parameter is the *request rate* (RR), that is the number of requests generated per time unit. All client requests — in our terminology form submissions — are counted. For workload models with finite user sessions we introduce a different workload parameter, the *starting user session rate* (SUR). This is the average number of finite user sessions that is started per time unit. All three parameters can be used to describe a constant load. If VU is the workload parameter, a constant load is achieved by generating a certain number of virtual users, and then ceasing to generate new users. We can also control the load with SUR. A constant load is achieved by continuously generating new user sessions with a constant SUR. After an initial start-up time in the order of $AVGD$ we have constant load. VU is in this case an observable parameter that is affected by SUR and other parameters.

The different elements of the workload model, user model and workload parameter should model truly different aspects of the model as a separation of concerns; otherwise the model will be unrealistic as we will see in the following. We define: a workload description is *realistic* if the following holds for a change in the user model. If an element of the user model is changed, for example if a think time is shortened, perhaps in order to model an improved page readability, or if the number of requests per user session is changed, perhaps because the user navigation is improved, then this change in the user model should have the same effect on the load test as it would have on the real system. We restrict our definition here to these two types of changes in the user model. We can show that the two major conventional workload parameters, namely VU and RR, do not give realistic workload models, but SUR does.

We now discuss whether a workload model with workload parameter VU is realistic. If the server response times are negligible, we note that for such models globally scaling all think times by factor a ceteris paribus changes RR by factor $1/a$, because each virtual user delivers a changed CRR; obviously we have

$$RR = VU \cdot CRR$$

The scaling of think times hence changes the system load in the load test. On the real system however the load is not expected to change as we will see soon; if all users simply take a little bit longer to think, but still do the same number of requests, RR does not change! Hence realism is vio-

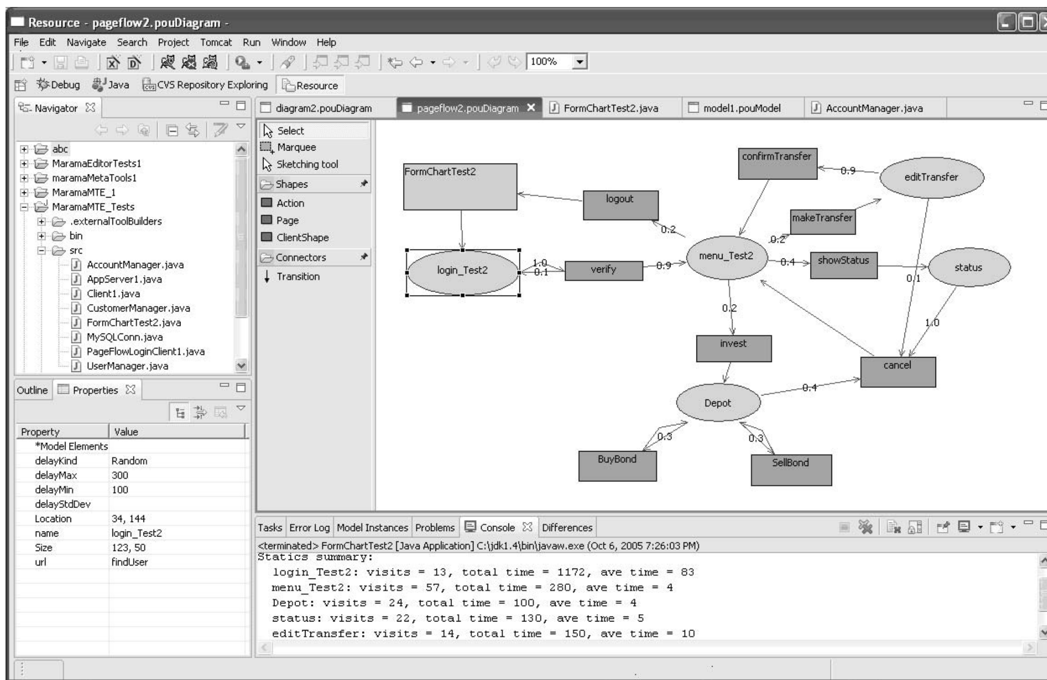

Figure 8. Screenshot of the formchart view of the MaramaMTE tool.

lated for a workload model with workload parameter VU. We now discuss whether a workload model with workload parameter RR is realistic. We note that for such models the request rate is trivially kept constant if think times are scaled. We now consider the second condition of realism; if we change the user model by reducing $AVGR$ and we assume for simplicity that all requests create the same load, then the load during the load test remains constant if we have fixed RR. But on the real system, the load would decrease. Hence workload models with workload parameter RR are unrealistic.

We now consider a workload model with workload parameter SUR. After an initial start-up we have, as one easily convinces oneself:

$$RR = SUR \cdot AVGR$$

Hence globally scaling all think times ceteris paribus does not change RR, since the duration of the single user session is irrelevant after start-up: The equation contains $AVGR$ and not $AVGD$. This behaviour of the load test is exactly the behaviour of the real system; for the real system, the same equation holds. We now consider the second condition of realism; if we change the user model by, say, halving $AVGR$, and if we assume for simplicity that all requests create the same load, then in both, the load test and the real system, the load will be halved. According to our definition this indicates that workload models with workload parameter SUR are realistic. In fact, workload models with workload parameter SUR have other advantages. Furthermore RR is not sensitive to server load, even if the think time of the session clients would be sensitive to the server load. Even if the actual user agents are programmed in a way that they have $AVGD$ that are dependent on the server response, if SUR is kept constant, then the load is constant. This is because VU is changed appropriately if we scale the think times; in fact it is easy to see that we have:

$$VU = SUR \cdot AVGD$$

We discuss now an example showing that an unrealistic workload model, if applied naively, can create misleading load test results. We take an example from one of the load test projects the authors are involved with. It is an enrolment system for university students. Using the system is mandatory for students. We compare now the behaviour of the different load test approaches during maintenance of the application. We assume a workload model with workload parameter RR. We assume the system performance is sufficient. We assume every student performs 10 requests for his enrolment, only the last one creates heavy load throughout the system. Now the user interface is improved, and only two requests per user are necessary, a first lightweight one and the last one being the same heavy load request as before. If we naively change only the user model in the workload model, and not the workload parameter, then the system load would increase roughly by factor 5 and could bring down the system. In the real system the system load

would not increase. The problem remains if the workload model is set up with workload parameter VU. In contrast, if the workload model is set up with the realistic workload parameter SUR, the system load remains roughly the same in the load test, hence resembling the behaviour of the running application. As we see, workload models with workload parameters RR or VU can deliver spurious results, if the parameters are not changed with every change to the user model. These changes can be done, but they require the knowledge of the above equations or lucky intuition to the same effect. The correct load test behaviour comes for free in the workload model with the realistic workload parameter SUR.

6 Related Work

In many cases, load-testing is still done by hand-written scripts that describe the user model as a subprogram [22, 25]. For each virtual user, the subprogram is called, possibly with a set of parameters that describe certain aspects of the virtual user's behaviour. Often the users are also modelled by a multimodel, which defines a subprogram for each user category. Data is either taken from a set of predefined values or generated randomly. With regard to input data this approach has a certain degree of randomization. However, user behaviour itself mainly remains a matter of repetition. This approach is purely script driven and suffers, like any hand-written program, from being prone to programming errors. The load engine itself has to be developed and brought to a mature state, which usually is a very time consuming task. In contrast to that, our approach does not rely on hand-written programs but on configurable models, which are much easier to handle and less error prone. It does not require a rewrite of the load engine itself, but rather a reconfiguration of a load engine that interprets stochastic formcharts.

The leading product for industry-strength load testing [19] is Mercury Interactive's LoadRunner [17]. It takes a similar script-driven approach. However, it significantly increases usability by offering a visual editor for end-user scripts. No conventional programming is needed, and the scripts describe the load tests in a much more domain-specific manner. End-user scripts are run on a load engine that takes care of load balancing and monitoring automatically. LoadRunner does not, however, offer a model-based solution like that of stochastic formcharts. In contrast to the LoadRunner tool suite, which focuses on load testing and optimisation, formchart models offer well-understood concepts for the specification of systems in general, which are also useful for other web application engineering tools and facilitate their interoperability. Most current load testing tools operate in a manner similar to LoadRunner. A detailed discussion of bottleneck problems created among other things by operating systems is given in [1]. Additionally the authors present a nontrivial implementation for load test clients. More information about load test practice can be found in [24, 11, 16].

There already exist model-based approaches for testing of web applications, e.g., in [15, 26], but they usually focus on the generation of test cases for functionality testing. Different studies have shown that stochastic models, in particular Markov chains, provide benefits for functionality testing [26, 13, 27]. They can be used for the automatic generation of large randomized test suites with a high coverage of operational paths. In [20, 26], for example, analysis models are used for regression testing in web site evolution scenarios. The model for user navigation is a stochastic one similar to Markov chains, but all user input data has to be given in advance for the system to work. While this may be appropriate for regression testing, it is not flexible enough for performing load tests. A Markov chain model like that in [27, 26] can only be used for a system where identical inputs cause identical state transitions, which is not the case in most web applications that rely on session data or a modifiable database. Consider, for example, an online ticket reservation system: after a specific place has been booked, it is not available any more; thus, repeating the same inputs will cause different results.

The motivation of using a statistical model based on data about real user behaviour for realistic load testing of web sites was already anticipated in [13], but their model fails to distinguish the user behaviour, which can only be adequately modelled as a stochastic process, from the system behaviour, which is deterministically given by the implementation. Transforming a state model of a web site directly into a Markov chain is not sufficient and does not account for the system's behaviour, which is not stochastic. In [28] it was shown that creation of a simple stochastic user model with real user data represents a valid approach for load testing. However, most approaches offer no model for specifying user behaviour over time, and it is usually neglected that form choice probabilities may change during a session.

The difference to stochastic process calculi such as, for example, the one described in [4], is that our stochastic model captures the specifics of web applications and can also be very suitably used for web application development [9]. However, it is conceivable that such theory can help in the development of new analysis techniques for our stochastic models. Since action-page transitions are chosen by the system, it is not necessary to assign probabilities to them in order to perform load testing. But if we assign probabilities to these transitions the same way we do with page-action transitions, e.g., by measuring relative frequencies, estimation or simply using uniform probabilities, then we can analyse the model statically and make statistical estimates similar to those described in [27] for Markov chains.

The difference to Markov chains is that our model also captures behaviour over time, i.e., delays. So if we extend our model further by measuring or estimating time delay distributions for the server actions, then we could also make statistical estimates of timing behaviour. This could, for example, allow the calculation of the expected duration of a session or of the expected usage of particular subsystems. Such new analysis techniques present possible future work in the area of web site performance evaluation.

7 Conclusion

In this paper we presented a new approach for load testing of web sites which is based on stochastic models of user behaviour. It allows the easy creation of realistic models of the individual user behavior. Furthermore we discussed how the user model can be used in a realistic workload model. We described our implementation of realistic load testing in a visual modelling and performance testbed generation tool, which allows realistic estimates of web application performance from software architecture descriptions.

References

[1] G. Banga and P. Druschel. Measuring the Capacity of a Web Server under Realistic Loads. *World Wide Web*, 2(1-2):69–83, 1999.

[2] P. Barford and M. Crovella. Measuring Web Performance in the Wide Area. *SIGMETRICS Perform. Eval. Rev.*, 27(2):37–48, 1.99.

[3] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings of 11th International WWW Conference*, May 2002.

[4] M. Bernardo and R. Gorrieri. A Tutorial on EMPA: a Theory of Concurrent Processes with Nondeterminism, Priorities, Probabilities and Time. *Theor. Comput. Sci.*, 202(1-2):1–54, 1998.

[5] Y. Cai, J. C. Grundy, and J. G. Hosking. Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. In *Proceedings of the 2004 IEEE International Conference on Automated Software Engineering*, pages 36–45. IEEE Press, September 2004.

[6] K. Curran and C. Duffy. Understanding and Reducing Web Delays. *Int. J. Netw. Manag.*, 15(2):89–102, 2005.

[7] Y. Deng, P. Frankl, and J. Wang. Testing Web Database Applications. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[8] D. Draheim, C. Lutteroth, and G. Weber. A Source Code Independent Reverse Engineering Tool for Dynamic Web Sites. In *9th European Conference on Software Maintenance and Reengineering*. IEEE Press, 2005.

[9] D. Draheim and G. Weber. *Form-Oriented Analysis - A New Methodology to Model Form-Based Applications*. Springer, October 2004.

[10] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[11] A. K. Iyengar, M. S. Squillante, and L. Zhang. Analysis and Characterization of Large-scale Web Server Access Patterns and Performance. *World Wide Web*, 2(1-2):85–100, 1999.

[12] K. Jajuga, A. Sokoowski, and H. H. Bock. *Classification, Clustering and Data Analysis*. Springer, August 2002.

[13] C. Kallepalli and J. Tian. Measuring and Modeling Usage and Reliability for Statistical Web Testing. *IEEE Trans. Softw. Eng.*, 27(11):1023–1036, 2001.

[14] S. Karre. Leveraging User-Session Data to Support Web Application Testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005. Member-Sebastian Elbaum and Member-Gregg Rothermel and Member-Marc Fisher II.

[15] D. C. Kung, C.-H. Liu, and P. Hsia. An Object-Oriented Web Test Model for Testing Web Applications. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 537–542, Washington, DC, USA, 2000. IEEE Computer Society.

[16] D. A. Menascé. Load Testing of Web Sites. *IEEE Internet Computing*, 6(4):70–74, July 2002.

[17] Mercury Interactive Corporation. Load Testing to Predict Web Performance. Technical Report WP-1079-0604, Mercury Interactive Corporation, 2004.

[18] J. D. Musa. *Software Reliability Engineering*. McGraw-Hill, 1998.

[19] Newport Group Inc. Annual Load Test Market Summary and Analysis, 2001.

[20] F. Ricca and P. Tonella. Analysis and Testing of Web Applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[21] L. Rosenberg and L. Hyatt. Developing a Successful Metrics Program. In *Proceedings of the 19th International Conference on Software Engineering*. ACM Press, 1997.

[22] A. Rudolf and R. Pirker. E-Business Testing: User Perceptions and Performance Issues. In *Proceedings of the First Asia-Pacific Conference on Quality Software*. IEEE Press, 2000.

[23] J. Shaw. Web Application Performance Testing – a Case Study of an On-line Learning Application. *BT Technology Journal*, 18(2):79–86, 2000.

[24] J. C. C. Shaw, C. G. Baisden, and W. M. Pryke. Performance Testing – A Case Study of a Combined Web/Telephony System. *BT Technology Journal*, 20(3):76–86, 2002.

[25] B. Subraya and S. Subrahmanya. Object Driven Performance Testing of Web Applications. In *Proceedings of the First Asia-Pacific Conference on Quality Software*. IEEE Press, 2000.

[26] P. Tonella and F. Ricca. Statistical Testing of Web Applications. *Software Maintenance and Evolution*, 16(1-2):103–127, April 2004.

[27] J. A. Whittaker and M. G. Thomason. A Markov Chain Model for Statistical Software Testing. *IEEE Trans. Softw. Eng.*, 20(10):812–824, 1994.

[28] L. Xu and B. Xu. Applying Users' Actions Obtaining Methods into Web Performance Testing. *Journal of Software (in Chinese)*, (14):115–120, 2003.

[29] L. Xu, B. Xu, and J. Jiang. Testing Web Applications Focusing on their Specialties. *SIGSOFT Softw. Eng. Notes*, 30(1):10, 2005.