A Performance Experiment System Supporting Fast Mapping of System Issues

Martin Mroz and Greg Franks
Department of Systems and Computer Engineering, Carleton University
Ottawa, ON Canada K1S 5B6
{mroz,greg}@sce.carleton.ca

ABSTRACT

The most fruitful use of a performance model is to study deep properties of the system, and hypothetical situations that might lead to improved configurations or designs. This requires executing experiments on the model, which evaluate systematic changes. Parameter estimation methods also exploit search in a parameter space to fit a model to performance data. Estimation, sensitivity and optimization experiments can require hundreds of evaluations, and the efficiency of the analytic model solver may become an issue. Analytic models usually provide fast solutions (compared to simulations) but repetitive solutions for near-neighbour models offer opportunities for further reducing the effort. This work describes an experiment driver for a layered queueing solver which provides a factor of two improvement. It also raises the issue of domain-specific languages for model experiments, versus general languages with suitable libraries.

Categories and Subject Descriptors

G.3 [**Mathematics of Computing**]: Distribution functions

General Terms

Performance

Keywords

Performance Analysis, modeling languages, sensitivity, experiment control, efficient solution

1. INTRODUCTION

The ability to solve a single performance model is just a building block for constructing studies of performance problems. A study will solve the model for variations in parameter values (and perhaps also in model structure) corresponding to different possible system designs or deployments, and will evolve depending on the results obtained. To define the evolution of the model, an experiment system is used. When there are many model-solutions to be obtained, the efficiency of the experiment system is a critical factor in its usability.

This paper considers efficient experimentation through integration of the experiment definition and the solver, for analytic layered queueing (LQ) performance models of computer software and systems. It considers the potential sources of greater efficiency, language design issues to exploit these, and describes the choices made in designing LQX (Layered Queueing eXperimenter). LQX provides variables for model parameters and general-purpose computation on these variables, combined with an API to the model to generate solutions and extract results.

Important types of experiments on performance models include:

1. Obtaining results for a predefined set of model parameter values, which may be:
   - *Traversal*: varying one or two parameter values across a given range, to explore the effect of changes.
   - *A full factorial experiment* (see e.g. [11] for a discussion) has a list of values for each of several parameters, and evaluates all combinations.
   - *a partial factorial experiment*, in which certain combinations are selected to evaluate single factor and multi-factor effects, without the combinatorial explosion of the full-factorial design.

   Traversal experiments on one or two parameters are extremely common in all kinds of performance studies, both with models and with measurements on running systems. Figure 1 sketches an example.
2. *Sensitivity*: providing a matrix of estimated partial derivatives of a set of performance measures with respect to a set of parameters. Estimation requires a base case plus one solution for a slightly perturbed value of each parameter (a special case of a partial-factorial experiment). This is used to linearize the response of the performance model, either for gradient optimization or for estimating parameters [25, 24].
3. *Model Iteration:* solving a model whose parameters depend on previous solutions of the same model. An example is a parameter whose value depends on the system throughput, such as a collision rate or hit rate parameter. This is a common feature of extended queue-

ing models. An example with a network collision rate is shown in Section 5.4.

4. *Optimization* of model parameters by a direct search technique requiring repeated model solutions. This requires integration of the solver and the search technique, which may be feasible for relatively simple search logic. A simulated annealing process in [12] generated thousands of model solutions. This does not apply to more elaborate optimization packages which have their own architecture, as used, for example, in [15].

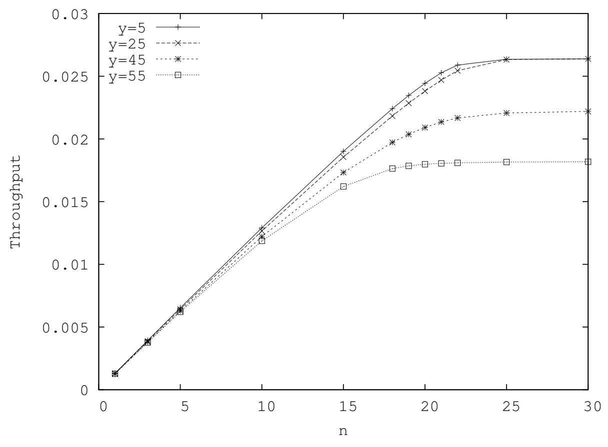


Figure 1: 2-parameter traversal, no of users on horizontal axis, image reads varying across a set of curves

There is a long history of experiment systems for performance models. The IMSE Experimenter [7] is a comprehensive system which can invoke runs on a model using either a planned pattern of parameter changes or a strategy to compute new parameters from previous runs. The Experimenter has a graphical interface in which each run is a node with attributes, including parameter-setting options. QNAP2 [22] for queueing networks and SPNP [3, 8] for Petri nets both combine model definition and model solution in a single programming language, with a solution section that can run multiple runs with parameter variations. In these systems the experimentation and solver are partly integrated, in the sense that the model construction is performed only once (as in LQNX) but the solver does not exploit a previous solution. These tools support both analytic and simulation models; simulation-only systems often have an experimental capability implicit in the simulation control (e.g. CSIM [20]).

Recently Smith et. al. [21] have proposed a language called Ex-SE for defining experiments at a level above the model-solving tool, as part of their PMIF proposal for exchanging queueing models between solvers. They describe the capabilities of Ex-SE as including:

- Changes in parameter values from one execution of a model to the next.
- Specification of control in performing model studies, including iteration and alternation.
- Variables that are local to the experiment to be used in computations and output.
- Model-results dependent execution.
- Use of previous output as input to subsequent runs.
- Specification of the output metrics to be returned.
- Solution type specifications.

All of these capabilities are also provided by LQNX, with the addition of results analysis capability, and its primary goal of solver/experimenter integration for efficiency.

Smith et. al. [21] also compare some prior experiment systems as to their model definition interface (GUI or Language), experiment definition (GUI or Language, integrated or separate) and additional features such as the ability to feed back results to control experiments (as in IMSE or SPEX [9]), and debugging capability.

Experimentation may also be combined with additional facilities for building models from component submodels, a feature which is not considered here since the opportunities for efficient integration cannot be applied when the model structure changes. In [1], candidate sets of component submodels are composed and evaluated to optimize component selection in a flexible product line. In IMSE, submodels are solved separately and coordinated by setting parameters of one model from results of another, possibly iteratively. Models may be in any of three different formalisms. Möbius [4] has a similar capability, but also submodels that share a state space may be composed and invoke an integrated solution technique.

As far as we know, the experiment control has never been fully integrated with the solver to enhance solution efficiency, as in LQNX. The present approach combines this integration with control of parameters either according to a programmed plan, or as functions of previous solutions, governed by logic expressed in a general programming language. The opportunities to enhance the solution efficiency are three-fold:

- The initialization of the solver may be the bulk of its execution time; this can be avoided for all subsequent solutions if only a parameter value has changed.
- In an iterative approximate solver, for instance for any extended queueing model or Markov model, the next iteration can begin from the previous solution. For a small parameter change, the previous solution may be close to the next solution and give much faster convergence of the iterations.
- The logical control over the exploration of parameters can reduce the number of solutions required.

This paper describes LQX and how it integrates the experiment definition with the modeling language, and the execution of experiments with the solution process, in our solver LQNX. The relative efficiency with LQNX, compared to a previous experiment system, ranges from a 30% to 210% improvement in execution time and other measures of effort.

2. LAYERED QUEUEING NETWORKS

Layered queueing is an extended queueing formalism designed for systems with software servers. Basic queueing models represent jobs that use one resource at a time, with resources modeled as servers. In layered queueing a server may make a blocking request to a lower layer server, and the waiting and service of this request becomes part of the requester's service time. To compensate for blocking delays, servers are often made multithreaded, so that when one thread blocks, others may continue. Multithreaded servers are represented by multiservers, as are pools of processors in a cluster. The model can also include non-blocking interactions and parallel operations.

The concepts of layered queueing will be explained with reference to a substantial example in Figure 2, representing an e-commerce system. A system is made up of software entities called *tasks* (shown as large parallelograms), offering classes of service called *entries* (shown as attached parallelograms) and running on host *processors* (shown as circles). The execution of an entry requires service by the host, indicated by a CPU demand $[s]$ in square brackets in the entry, and requests to other entries, which are indicated by an arrow to the entry labeled by a mean number of requests (y). The tasks, entries and interactions provide the structure of the model; the parameters are the values of $[s]$ and (y), and the multiplicity of the tasks $\{m\}$.

2.1 Illustrative Example

The example shown in Figure 2 will be used to demonstrate the capabilities of LQX with LQNX.

A top-level task models users or sources of load for the system, with a parameter for a think time (between requests) and a multiplicity which is the number of users of that class. Figure 2 represents two groups of users, with 50 users in the first group and 100 in the second.

All task names begin with 'T' and all entry names begin with 'E', to remind the reader of their roles in the model. The users behaviour is defined by the entries EUserBeh*i*, and they alternate between a period of thinking (1000 ms for TUser1, 5000 ms for TUser2) and a single request to EStoreAccess, defining the service given for each access to a web page of the electronic store. This service includes a combination of

- images read from the webserver's own disk, by ERdImg
- browsing access to the site, including product description pages with details from the inventory database, by EBrowse
- shopping cart operations gathered together in the entry EOrder, including buying the cart contents
- login of established customers before purchase.

Notice that an entry may represent a single interface operation of the system, or may gather several service functions together. For example, the entry EOrder may gather together several separate service functions related to the shopping cart, because their operations, and particularly their access to lower services, are similar.

Performance questions relating to ECOMM could include:

- scalability (maximum users supported with a specified response time)
- location of the bottleneck(s)
- sensitivity of the system capacity to execution demands of some of the services,

3. LQN SOLUTIONS

LQNs are a kind of extended queueing network, and analytic solutions use mean-value analysis and extended queueing techniques. The LQNS solver, used here, uses iterative techniques to incorporate the blocking requests and grew out of [23, 19]. As described in [6, 5], it creates a set of ordinary queueing network "layer submodels", in which blocking delays are represented by surrogate delays imported from other submodels. Other solvers for layered resources use variations of this approach [19, 16, 13, 17].

The solution process has six steps;

1 Read and parse input file.
2 Construct LQN model objects with parameters.
3 Construct Layer submodels.
4 Initialize iterative solution process and Layer submodels.
5 Execute iterative solution to convergence.
6 Write output file.

When solving multiple models using the existing experiment tools (SPEX or MultiSRVN [18]), all of these steps must be performed for each individual experiment because experiment control is a separate process disjoint from model solution, shown in Figure 3(a). When using LQNX, shown in Figure 3(b), steps 1 through 3 need only be executed once as the LQN model objects and the layer submodels will be re-used. Running step 1 only once will substantially reduce I/O time. Further, by restarting the iterations for step 5 at the previous solution, the iterative solution itself is expected to take fewer steps.

The sections that follow describe submodel construction (step 3), initialization (step 4), the iterative solution (step 5), and how LQNX restarts the MVA solution.

3.1 MVA Submodel Construction

A LQN is solved by breaking up the original input model into a set of submodels, each of which can be solved as a conventional queueing network, then iterating among these submodels until a solution to the overall network is found. The solver first sorts topologically the tasks and processors in the input model into a set of layers starting from tasks which never accept requests and working on down the call tree, as shown in Figure 2. Submodels are then constructed from top to bottom starting at layer 2. Submodel s is constructed by first finding all of the tasks and processors at layer $l = s + 1$ and treating these objects as the *servers* for the submodel. Next, all of the tasks that make direct requests to these servers are included in the submodel as clients. Clients in the submodel act as customers in the corresponding queueing network. A task can call itself, so it can appear as both a client and a server in a submodel. Figure 4 shows Submodels 2, 3 and 4 for the E-Commerce system shown in Figure 2. Note that the task *TStoreApp* appears as a server in submodel 2, and as a client in submodels 3 and 4.

The submodels are solved using the Linearizer approximate Mean Value Analysis [2], and results are exchanged between layers. The service and think times for the queueing stations and customers are found using the *waiting time* array associated with each task and processor in the LQN, illustrated in Figure 5 for task TStoreApp. The index of the array corresponds to the submodel, and the value stored at that index in the array corresponds to the response time for the object when it appears as client in that submodel. The

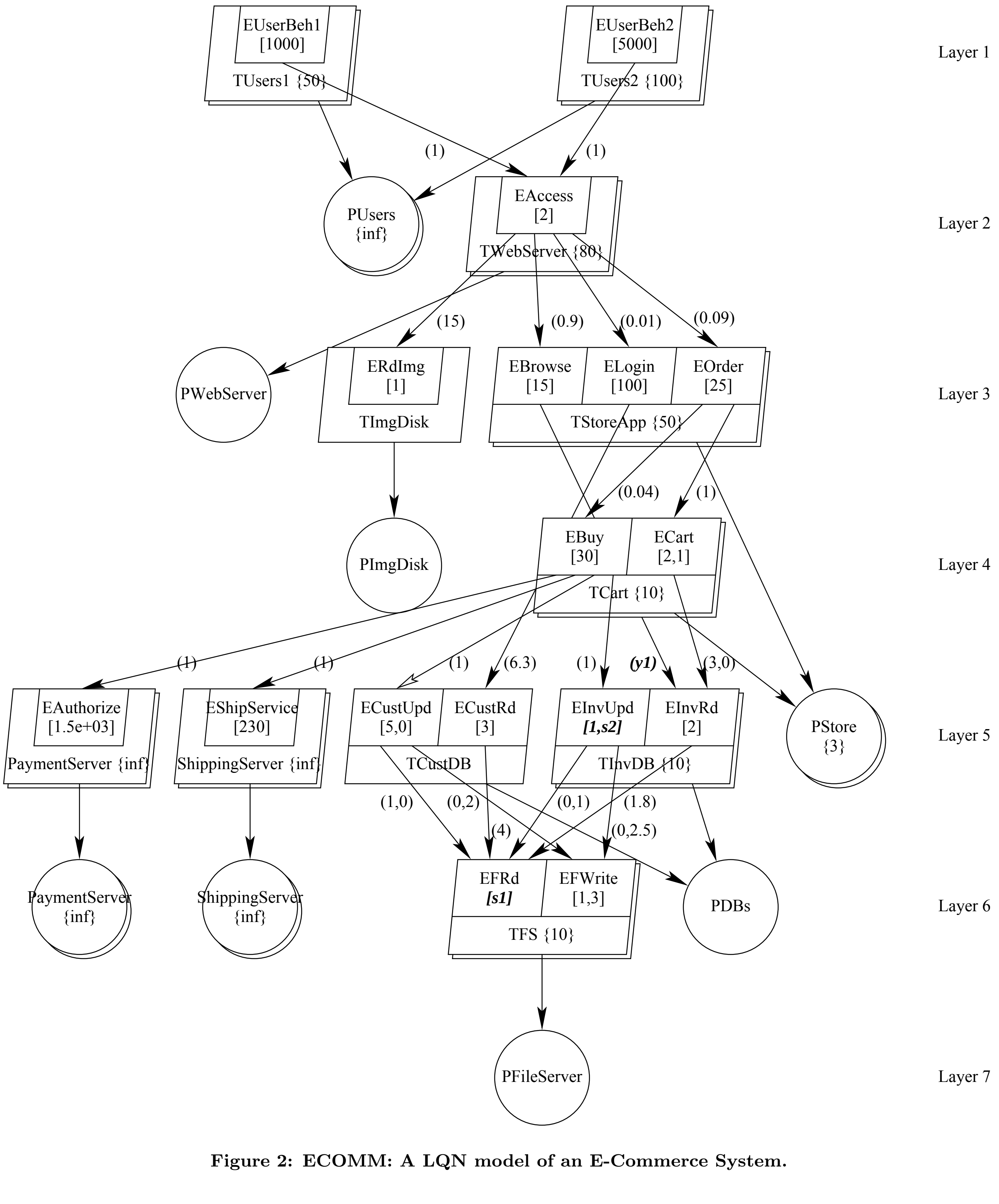
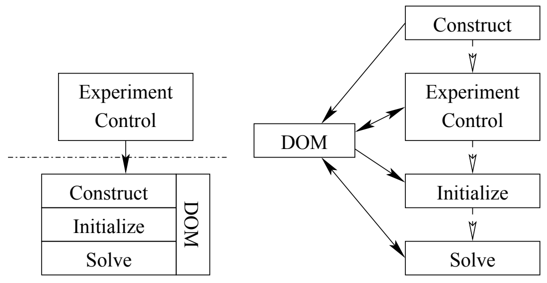
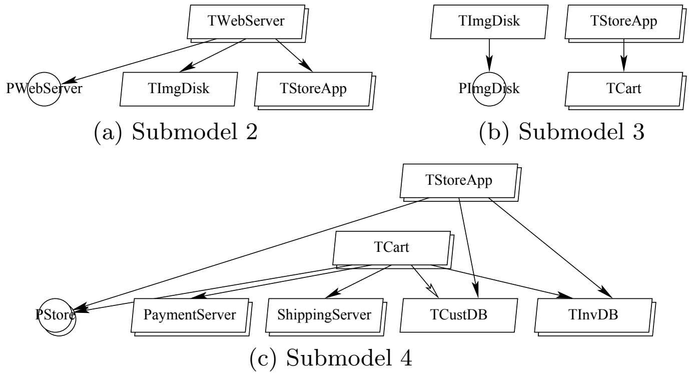


Figure 2: ECOMM: A LQN model of an E-Commerce System.



(a) Old Architecture (b) New Architecture

Figure 3: Old and New Experiment Solution Architectures



(a) Submodel 2 (b) Submodel 3

(c) Submodel 4

Figure 4: Submodels containing task TStoreApp

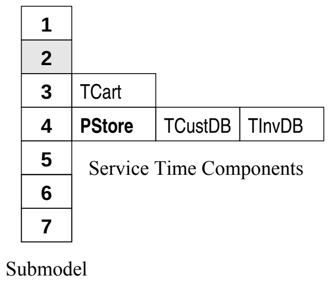


Figure 5: Waiting time components for TStoreApp.

service time for a server in a submodel is found by summing up all of the elements of the waiting time array. The think time for a customers in submodel s consists of a *surrogate delay* [10] found by summing up all of the elements except for the one at index s in its waiting time array plus its idle time when it acts as a server. For example, when TStoreApp appears as a client in submodel 4, it's surrogate delay is it's response time at task TCart in submodel 2, plus its idle time found during the solution of submodel 2.

Other layering strategies exist. One extreme is to form submodels with exactly one server. Unfortunately, the accuracy of this approach can suffer because the surrogate delays don't necessarily capture the contention effects at other servers. The other extreme is to use only one submodel with tasks acting as both clients and servers simultaneously. However, this strategy results in queueing models with a large number of stations and chains, which can take a long time to solve in Linearizer. The approach described here is a compromise between these two extremes and is much like the approach used in the Method of Layers (MOL) [19] except that MOL places all of the processors in a separate layer effectively at the deepest call depth in the model.

3.2 Initialization

The initialization phase of the solver is used to initialize the service time and populations in the submodel queueing networks, and to set the minimum value of the response times at each of the tasks in the LQN. First, since the processors in the model consume "time", the service times specified for the entries of tasks must be propagated down to the processors. The service time at the processor is

$$\zeta_e = \frac{s_e}{1 + \sum ye, d} \tag{1}$$

where s_e is the service time for the entry as specified in the input model, and $y_{e,d}$ is the request rate from entry e to entry d.

Next, the blocking times at lower level servers must be propagated up to their callers. This process starts at the lowest submodel (corresponding to the deepest layer in the LQN) and works it up so that the service time for layer l is the sum of the residence times at all of the layers, $l + 1, l + 2, \ldots.$ Since processors are treated uniformly, the service time set earlier reappears as one of the components of the residence time in the waiting time structure shown in Figure 5 after this step. Finally, the "Type-one" throughput bound is computed as the reciprocal of the initial value of the residence time for all of the tasks in the model.

3.3 Iterative Solution

The solution of a LQN involves three levels of nesting, The outermost iteration of the solver is used to solve the submodels starting from the top-most submodel and working down. This pass propagates idle times down to the lower level servers. Next, submodels are solved in the reverse order in order to propagate contention times up through the layers. This process continues until the change in the utilizations of all of the tasks and processors is sufficiently small, or if a pre-determined iteration limit is hit. The number of outer iterations is shown under the column labeled n in the results below.

The conventional queueing network corresponding to a submodel is solved using Linearizer approximate MVA [2]. The Linearizer algorithm consists of two layers of iterative solutions. The outermost layer invokes the core Schweitzer approximate MVA algorithm $I+(I-1)K$ times, where K is the number of chains (or job-classes) in the queueing model, and I was set at 3. The innermost layer, the core algorithm, iterates until the change in the queue length at all of the stations converges to some pre-determined value, calling one-step MVA on each iteration. One-step MVA is used to compute the queue length for all M stations over all K chains in the queueing model. The computational cost of the core algorithm is $O(MK^2)$ operations. The number of times the waiting time computation is called is shown under the column labeled `wait()` in the results below.

3.4 Iteration Restart

One of the objectives of incorporating iteration control within the solver itself, rather than as an external program as with SPEX and MultiSRVN is to reduce the number of iterations described in the preceding section by restarting the new solution from the value of the previous one. This objective is achieved by changing the initialization operation in step 4. Rather than initializing everything, only those waiting time components at processors whose *slice time* found using (1) which has changed is updated. The residence time at the processor is changed in proportion to the change in the slice time. All of the waiting times found elsewhere are unaffected, so the iteration can start at a solution which is substantially closer to the final solution.

4. THE LQX EXPERIMENT LANGUAGE

The LQ Experiment Control Language (LQX) is a general-purpose programming language whose primary purpose is to specify certain parameters of an LQN model in a flexible, consistent manner.

4.1 Defining the Model

One of the defining characteristics of the LQNX system is that the input model is defined with parameters that can be set either explicitly and directly from the input file or set as variables. The definition of the model is entirely separate from the LQX language, which is used solely for the purpose of setting the value of variable parameters. Certain systems allow for creation and manipulation of the structure of the input model directly from the experiment control language [22, 3].

Each approach has its advantages and disadvantages. Using the language to modify the parameters of a pre-defined model has the advantage of separating the structure from the flow of control surrounding experiment parameters. This makes the model easier to understand and to makes it substantially easier for external tools to parse the model and to change its structure. The disadvantage of having a fixed model structure is that small modifications to the structure must be tested by creating separate model file/script pairs. These must then be run through a new instance of the solver.

Keeping the structure of the model fixed, however, has the potential benefit of allowing for accelerated convergence in iterative models. By changing only the input parameters, it is possible to keep all of the state surrounding the last convergence of the model and to use it as a starting point. For small changes in parameters, this has the potential of decreasing the number of iterations substantially. This also creates the opportunity of not having to re-build the model, saving a potentially large amount of time.

The approach taken by the LQNS/LQNX system is to utilize XML to clearly define the structure of the model, in a method that is easy to parse by any external tools. The language itself is unable to modify the structure of the model to ensure that the definition is always in sync with what is actually being solved. The language is able to modify any parameters specified as being editable in the model itself.

4.2 Bindings between LQX and the Solver

Whatever approach is selected, the language must expose a set of bindings to the solver and to the model parameters. How these bindings are exposed can depend heavily on the nature of the language, such as whether it is a functional or an object-oriented programming language. For example, an object-oriented programming language may expose a document object model for interacting with the system, similar to that of JavaScript and the HTML DOM, for manipulating parameters. The experiment control language must also have a way of invoking the solver, either as an explicit call or an implicit one. Additionally, the language must decide whether or not to expose the results of a solution to be fed back into the solver, and how they should be exposed. Once again, this would depend heavily on the language.

In the LQX language, there are two kinds of variables, language variables and external or model variables. External variables always begin with a '\$', and as such are clearly distinguishable from language variables. In most locations within the model, the value of an attribute may either be specified as a constant number, or as a model variable beginning with '\$'. When the LQX language assigns a value of an external variable, that change takes place in all identically-named locations within the model. This allows multiple parameters to stay in sync, and reduces the chance of accidentally forgetting to set certain parameters. Additionally, when the solver is invoked, it verifies that all external variables in the model have had values assigned.

The interface from the LQX language to the solver itself is a single function call, `solve()`, which returns whether or not the model solved successfully, to the point of convergence. The LQNS/LQX system provides a method of accessing all of the results generated by solving the model, such that they can be fed back. They are exposed through a partially object-oriented API as described subsequently.

4.3 The LQX Language

The LQX language itself takes a number of cues from primarily-procedural languages such as PHP (PHP Hypertext Preprocessor) and ANSI-C. The language includes a

Listing 1: Sample LQX program

```
foreach( $x in [0.1, 0.2, 0.3]) {
  if (solve() == true) {
    t1 = task("t1");
    ph1 = phase(t1, 1);
    print("Phase 1 Utilization of T1: ",
      ph1.utilization);
  }
}
```

small number of data types `numerical`, `boolean`, `string` and `object`. The LQX language is dynamically typed, and in its initial incarnation has no support for user-defined functions or data types. The language includes support for common C constructs such as postfix notation, function calls and `for` loops, but also brings in more modern constructs such as associative arrays and `foreach` loops for iterating over them. Additionally, the language is dynamically typed to make it as flexible and expressive as possible. Due to its roots in C and PHP, the language is very clear and readable. The use of dynamic typing helps make it terse enough to both write quickly and read easily.

The decision to base the language on C and PHP was made because of the pervasiveness of not only of the C and PHP programming languages, but also their derivatives. People with a passing familiarity with one of any number of modern programming languages would be able to pick up LQX quickly, with minimal learning curve. The goal of the language was to make it as simple as possible for users to get started parameterizing their models, and make it flexible enough to fulfill the vast majority of their needs.

By providing a general purpose programming language rather than a domain specific programming language, the users gain a large amount of flexibility. A variety of different APIs can be exposed. For example, file IO can be exposed for writing custom-formatted reports or for reading data from separate input files. Listing 1 shows how a user would parametrize a simple model, with one variable defined in the model, $x, and obtain the Phase 1 utilization of a task named "t1," and write it on the console.

5. COMMONLY PERFORMED EXPERIMENTS

In this section the most commonly performed experiments are described, and the efficiency of LQNX is compared to that of our previous experiment system SPEX [9]. The experiments will be performed on the ECOMM model shown in Figure 2. The parameters of that model with symbolic names (beginning with $) may be controlled by LQX; in a given experiment some subset of them is varied. To describe the experiments in a general way, the following notation will be used: $X = (x_1, x_2, ...)$ = set of model parameters that are controlled in a given experiment.

5.1 Parameter Traversal

The most common parameter traversal reported in a performance study is the response time or throughput, against the workload intensity expressed by the number of users or the arrival rate. Figure 1 earlier showed such a traversal for ECOMM, varying the users in Group 1 (`$nUsers1`) from 1 to 30 and in Group 2 (`$nUsers2`) from 2 to 60 in direct proportion, and including a variation of the parameter $yRdImg, the number of image files per web page, from 3 to 18. For a small number of pages the system bottleneck is the database, while for $yRdImg > 45 approximately, it shifts to the image disk. The LQX code to run the traversal experiment is shown in the Listing 2.

Listing 2: LQX Program for the Parameter Traversal Experiment (§5.1).

```
webserver = task("TWebServer");
foreach ( $yRdImg in [5,15,25,35,45,55]) {
  foreach ( $nUsers1 in [1,3,5,10,15,18,
    19,20,21,22,25,30] ) {
    $nUsers2 = 2 * $nUsers1;
    solve();
    print( $yRdImg, " ", $nUsers1, " ",
           webserver.throughput );
  }
}
```

Table 1: Performance Comparison of LQNX vs SPEX for the Parameter Traversal Experiment (§5.1).

Solver	n	`wait()`	**E**lapsed	**C**PU
SPEX	522	64785128	15.32±0.31	13.34±0.03
LQNX	437	9334452	7.12±0.11	5.54±0.01
η	1.2×	6.9×	2.2×	2.4×

Table 1 compares the iteration effort by the solver, and the running time, between SPEX without integration with the solver, and LQX. The column labeled n shows the number of outer iterations of the solver and `wait()` show the number of times the inner-most waiting time calculation was executed. The last two columns show the "wall-clock" and CPU time to run the experiments, the former including blocking caused by I/O operations. The execution time was averaged over 10 replications of the solution, to obtain confidence intervals. The LQNS solver invoked by the two experiment tools was the same, with the same solution options.

The improvement is due to reduced iterations between the layer submodels, starting from the previous solution. If the system is not saturated, the next solution does not require a large number of new iterations, but as a resource becomes a bottleneck, any change can have a large affect on the response times, thus requiring more iterations to converge. If the `foreach` loops were written in the opposite order, then the incremental change would be in $yRdImg which might require more iterations to converge.

5.2 Full-Factorial Exploration

A two parameter traversal is a simple example of a full factorial experiment. If N parameters are varied and have M levels each, a full factorial experiment requires N^M model solutions. A small example with three parameters with three levels each was performed: $X = (\$s1, \$y1, \$s2)$, parameters labeled on Figure 2. Listing 3 shows the LQX code for this experiment.

Table 2 compares the effort between the two tools. The iterations are reduced by a third, but are still substantial.

Listing 3: LQX Program for the Full-Factorial Experiment (§5.2).

```
sFrd = [1,2,3,4,5];
yBwsIvRd = [5.74,6,47,7.3,7,93,8.66];
sInvUpd2 = [8,9,10,11,12];
foreach ( $s1 in sFRd ) {
  foreach ( $y1 in yBwsIvRd ) {
    foreach ( $s2 in sInvUpd2 ) {
      solve();
    }
  }
}
```

Table 2: Performance Comparison of LQNX vs SPEX for the Full Factorial Experiment (§5.2).

Solver	n	`wait()`	**E**lapsed	**C**PU
SPEX	1517	356858100	85.85±0.34	79.21±0.04
LQNX	961	11318310	25.68±1.38	16.29±0.06
η	1.6×	31.5×	3.3×	4.9×

5.3 Partial Factorial Exploration

A partial factorial design (see, e.g. [11]) omits some parameter combinations. For example, instead of all combinations, the base case may be perturbed by changing only one parameter at a time (a traversal in each dimension). For N parameters with K levels for each one, this reduces the number of solutions from N^K to $NK + 1$. To include interaction effects, parameters may also be varied in pairs, for all pairings. We should expect similar efficiency, as in the full factorial case.

5.4 Model Iteration

In some models a parameter is known only as function of the solution. The model can be solve by fixed-point iteration on the entire model, by solving it, recomputing the parameter, and repeating until convergence. The results of a solution of a model are used to change the inputs of the model.

As an example, we will consider the model in Figure 6, containing a local network modeled by the approximate extended queueing model based on a description by Lazowska et. al. ([14], Sec 15.3). The network is represented by the infinite server (pure delay) task "Network".

The contention delay depends on the mean number n of contending users. In our model contending users all occupy "Transmit" while waiting to transmit, so the mean number is the mean utilization of this server. Then the probability of a successful attempt is $A = (1 - 1/n)^{(n-1)}$, the average unsuccessful attempts is $C = (1 - A)/A$ and the mean delay to a successful transmission is \$cont_delay $= C\times$ (packet transmission time). This is coded in LQX in Listing 4 below.

The same iterative computation was done in a version of SPEX, using the same parameter update function. The results in Table 3 compare the two systems. LQNX took just over two-thirds the total model iterations.

5.5 Sensitivity Matrix

A sensitivity value for this discussion will be taken to be the derivative of some performance measure with respect to some parameter. A sensitivity matrix J is the Jacobian

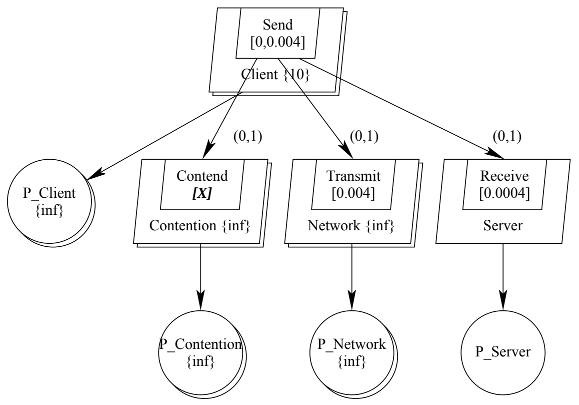


Figure 6: Carrier Sense, Multiple Access Collision Detect feedback model.

Listing 4: LQX program for the Model Feedback Experiment (§5.4).

```
$prop_delay = 0.0000025;
old_delay = -1;
transmit=entry("Transmit");
do {
  util = phase(transmit,1).utilization;
  if ( util >= 1 ) {
    prTrans = 1.0;
  } else {
    prTrans = (1-(1/util))**(util-1);
  }
  avgSlots = (1-prTrans)/prTrans;
  if ( avgSlots != 0 ) {
    $cont_delay = avgSlots * $prop_delay;
  } else {
    $cont_delay = 0.0;
  }
  solve();
  delta = old_delay - $cont_delay;
  old_delay = $delay;
} while ( abs( delta ) > 0.00001 );
```

Table 3: Performance Comparison of LQNX vs SPEX for the Model Feedback Experiment (§5.4).

Solver	n	`wait()`	**E**lapsed	**C**PU
SPEX	16	69392	0.119±0.003	0.058±0.003
LQNX	8	17632	0.087±0.004	0.044±0.003
η	2.0×	3.9×	1.4×	1.3×

Listing 5: LQX program for the Sensitivity Analysis experiment (§5.5).

```
foreach ( i in [0,1,2,3,4,5,6,7] ) {
  $sEInvRd=2;
  $sEBrowse=15;
  $sECustRd=3;
  $sERdImg=1;
  $sEBuy=30;
  $sEAccess=2;
  $sEShipService=230;
  if (i==1) $sEInvRd=$sEInvRd*1.01;
  else if (i==2) $sEBrowse=$sEBrowse*1.01;
  else if (i==3) $sECustRd=$sECustRd*1.01;
  else if (i==4) $sERdImg=$sERdImg*1.01;
  else if (i==5) $sEBuy=$sEBuy*1.01;
  else if (i==6) $sEAccess=$sEAccess*1.01;
  else $sEShipService=$sEShipService*1.01;
  solve();
}
```

Table 4: Performance Comparison of LQNX vs SPEX for the Sensitivity Experiment (§5.5).

Solver	n	wait()	**E**lapsed	**C**PU
SPEX	151	27629100	7.62±0.24	6.94±0.02
LQNX	63	1643094	1.14±0.06	0.99±0.00
η	2.4×	16.8×	6.7×	7.0×

of a vector of measures (model outputs) with respect to a vector of model parameters, computed at a base value of the parameter vector. It is defined as

$$J_{ij} = \frac{dy_i}{dx_j}$$

dy_i/dx_j can be estimated by an experiment in which x_j is displaced by a small fraction (a 1% displacement was used here).

If a performance model is a nonlinear function of its parameters, the Jacobean matrix gives a local linearization. The Jacobian matrix is required for nonlinear estimation of model parameters, as described in [25, 24]. It also provides a gradient vector for a gradient optimization technique, for parameter optimization. The pattern of parameter values is a special case of a partial factorial experiment.

6. SUMMARY AND CONCLUSIONS

This paper has described an experiment control language integrated into the analytic Layered Queueing Network Solver, LQNS. The performance model being analyzed is specified using XML; the control program is specified as an optional element within the model. This approach allows the interchange of the model with other XML-based tools which would be difficult if a completely custom modelling language were used instead.

The control language, LQX, is loosely based on C and PHP, in order to retain the familiar if not terse syntax of these common programming languages. LQX, like many other modern scripting languages, is dynamically typed and supports boolean, string and numerical types. Model elements such as tasks, processors and entries are also types in the language, so that outputs from a solution can be referenced and fed back as inputs into the LQX program. LQX also supports associative arrays, so arbitrary data structures can be created as needed.

The analytic solver described here attempts to improve the performance of solving multiple models by integrating the experiment control into the model solution. Rather than starting all iterations from scratch for each experiment, the solver starts from the previous solution. In the models shown here, the number of times the underlying waiting time calculation for the MVA solver was called was reduced from 3.9 to 31.5 times, and run-time improvements of from 1.4 to 6.7 times were achieved over comparable experiments which invoked the solver on separate model files. Further improvements to the solution speed should be possible as the algorithms are refined.

Acknowledgment

This research was supported by the Natural Sciences and Engineering Research Council of Canada. The authors would also like to thank Murray Woodside for his input both for this paper, and for the the design of the experiment control language.

7. REFERENCES

[1] N. Barthwal and M. Woodside. Efficient evaluation of alternatives for assembly of services. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) Workshop 15*, volume 16, Denver, CO USA, Apr. 4–8 2005.

[2] K. M. Chandy and D. Neuse. Linearizer: A heuristic algorithm for queueing network models of computing systems. *Commun. ACM*, 25(2):126–134, Feb. 1982.

[3] G. Ciardo, J. Muppala, and K. Trivedi. SPNP: stochastic Petri net package. In *Proceedings of the Third International Workshop on Petri Nets and Performance Models, 1989. (PNPM89)*, pages 142–151, Kyoto, Japan, Dec. 11–13 1989.

[4] D. Daly, D. D. Deavours, J. M. Doyle, P. G. Webster, and W. H. Sanders. Möbius: An extensible tool for performance and dependability modeling. In *Computer Performance Evaluation. Modelling Techniques and Tools*, Lecture Notes in Computer Science, pages 332–336. Springer, 2000.

[5] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Softw. Eng.*, 35(2):148–161, 2009.

[6] R. G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada, Dec. 1999.

[7] J. Hillston. A tool to enhance model exploitation. *Performance Evaluation*, 22(3):59–74, Feb. 1995.

[8] C. Hirel, B. Tuffin, and K. S. Trivedi. SPNP: Stochastic Petri nets. version 6.0. In *Computer Performance Evaluation. Modelling Techniques and Tools*, volume 1786 of *Lecture Notes in Computer Science*, pages 354–357. Springer, 2000.

[9] A. Hubbard. SPEX software performance experiment driver. http://www.sce.carleton.ca/rads/lqns/lqn-documentation/spex.txt, Aug. 1997.

[10] P. A. Jacobson and E. D. Lazowska. Analyzing queueing networks with simultaneous resource possession. *Commun. ACM*, 25(2):142–151, Feb. 1982.

[11] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[12] P. Jogalekar and M. Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, 2000.

[13] P. Kähkipuro. UML based performance modeling framework for object oriented systems. In *UML '99, The Unified Modeling Language, Beyond the Standard*, number 1723 in Lecture Notes in Computer Science, pages 356–371, Berlin, 1999. Springer-Verlag.

[14] E. D. Lazowska, J. Zhorjan, S. G. Graham, and K. C. Sevcik. *Quantitative System Performance; Computer System Analysis Using Queueing Network Models*. Prentice Hall, Englewood Cliffs, NJ, 1984. Out of Print.

[15] M. Litoiu and J. A. Rolia. Object allocation for distributed applications with complex workloads. In *11th International Conference on Computer Performance Evaluation; Modelling Techniques and Tools TOOLS 2000*, number 1786 in Lecture Notes in Computer Science, pages 25–39, Schaumberg, IL, Mar.27–31 2000. Springer-Verlag.

[16] D. A. Menascé. Two-level iterative queuing modeling of software contention. In *Proceedings of the Tenth IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, Fort Worth, TX, Oct. 12–16 2002.

[17] S. Ramesh and H. G. Perros. A multilayer client-server queueing network model with synchronous and asynchronous messages. *IEEE Trans. Softw. Eng.*, 26(11):1086–1100, Nov. 2000.

[18] Real Time and Distributed Systems Group, Carleton University, Ottawa, Canada. *User's Guide for MultiSRVN Version 4.0*, Jan. 1997.

[19] J. A. Rolia and K. A. Sevcik. The method of layers. *IEEE Trans. Softw. Eng.*, 21(8):689–700, Aug. 1995.

[20] H. Schwetman. CSIM: a C-based process-oriented simulation language. In *Proceedings of the 18th conference on Winter simulation*, pages 387–396, Washington DC, USA, 1986.

[21] C. U. Smith, C. Llado, and R. Puigjaner. Interchange formats for performance models: Experimentation and output. In *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*, pages 91–100, Edinburgh, Scotland, Sept. 17–19 2007.

[22] M. Veran and D. Potier. QNAP 2: A portable environment for queueing systems modelling. In D. Potier, editor, *Proceedings of the International Conference on Modelling Techniques and Tools for Performance Analysis*, pages 5–24, Paris, France, May 16–18 1984. INRIA, North Holland.

[23] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Trans. Comput.*, 44(8):20–34, Aug. 1995.

[24] M. Woodside. The relationship of performance models to data. In *Performance Evaluation: Metrics, Models and Benchmarks (Proc. SIPEW 2008)*, number 5119 in Lecture Notes in Computer Science, pages 9–28. Springer-Verlag, June 2008.

[25] T. Zheng, C. M. Woodside, and M. Litoiu. Performance model estimation and tracking using optimal filters. *IEEE Trans. Softw. Eng.*, 34(3):391–406, May–June 2008.

Digital Object Identifier: 10.4108/ICST.VALUETOOLS2009.7807
http://dx.doi.org/10.4108/ICST.VALUETOOLS2009.7807