Automatic Generation of Layered Queuing Software Performance Models from Commonly Available Traces

Tauseef A. Israr
IBM Canada
770 Palladium Drive
Ottawa K2C 1C8, Canada
tisrar@ca.ibm.com

Danny H. Lau, Greg Franks, Murray Woodside
Dept. of Systems and Computer Engineering,
Carleton University
Ottawa K1S 5B6, Canada
{dlau,greg,cmw}@sce.carleton.ca

ABSTRACT

Performance models of software designs can give early warnings of problems such as resource saturation or excessive delays. However models are seldom used because of the considerable effort needed to construct them. Software Architecture and Model Extraction (SAME) is a lightweight model building technique that extracts communication patterns from executable designs or prototypes that use message passing, to develop a Layered Queuing Network model in an automated fashion. It is a formal, traceable model building process. The transformation follows a series of well -defined transformation steps, from input domain, (an executable software design or the implementation of software itself) to output domain, a Layered Queuing Network (LQN) Performance model. The SAME technique is appropriate for a message passing distributed system where tasks interact by point–to-point communication. With SAME, the performance analyst can focus on the principles of software performance analysis rather than model building.

Keywords

Performance Engineering, Software performance, Tracing Performance Modeling, Layered Queuing, Model Building

1. INTRODUCTION

Many software projects combine tight deadlines and increasing complexity. As a result, vital performance issues are neglected, only to arise in the late stages, when they are difficult to deal with. In general, we would like to predict the performance problems in advance when large design changes are easier to execute.

The earliest predictions are given by performance models created from the requirements and the designers' expertise [19][21][22]. The models require the description of the execution time and frequency of the major system operations, and may be expressed as extended queuing networks. Solutions give estimates of the response time and throughput of the system under different loads, and identify problem areas such as operations that are performed too often or take too long. A recent standard supports the definition of the necessary parameters within a UML design model [13].

Generating performance models this way is effective [22] but may be difficult. It requires system expertise and experience, a deep knowledge of the system to be built, and performance expertise. Automation of the model-building process would make the use of models more accessible and reduce the time and effort.

Automation can be applied as the design develops, and executable design products are created such as CASE tool models and prototypes. Their behavior can be traced and used to create performance models, as in the Angio Tracing and Trace-based Load Characterization method described by Hrischuk et al [5][6]. These methods require a specialized trace format based on an "angio dye id" which is injected into the beginning of each response and propagated through the system (the name is by analogy with dyes used for medical angiograms). This method is effective in creating a model structure, but the workload parameters must be determined separately.

The application of angio-trace based methods is limited by the need for special trace information (transaction correlation in the ARM monitoring standard [14] is the closest thing to a dye id). Thus this research developed a different approach based on commonly available trace data. It is called SAME (the Systems Architecture and Model Extraction technique). It can be applied in the middle and later stages of the software development cycle, once an executable form of design product is available. In the earliest stages, a loosely related scenario-based technique such as [15] may be more suitable.

SAME uses performance model concepts which are described in Section 2, based on types of interactions between components which are discussed in detail in Section 3. Section 4 outlines the method and the algorithm for reducing traces, with some examples. Section 5 describes validation on simple cases, and there is an e-commerce case study in section 6 which demonstrates its application to a prototype, and its scalability.

2. PERFORMANCE ISSUES AND MODELING

Software Performance Engineering (SPE) involves the use of a set of methods for software systems development that meets with another set of pre-defined performance requirements. The available SPE techniques include design optimizations, the use of lab or field measurements to suggest configuration tuning and design tuning, and performance modeling.

2.1 Performance Techniques

Software developers often incorporate well-known performance optimizations such as hash tables in a design. To be effective this requires a good knowledge of the system, and it is very difficult to focus on the important optimizations without knowing where the problems are. Source-code metrics (see e.g. [2]) may provide relevant information.

Once a prototype or implementation is available its performance can be measured, using operating system data, profiling of the software, and special tools such as Quantify [7]. An example of a measurement environment is Pablo [16]. An advantage of this approach is that it evaluates the configuration of the system in the environment, as well as design. However, it may be difficult to obtain the measurement that one needs to understand a problem (called "drilling down"), and this can only be done late in the development cycle. This may be too late for some kinds of design change (or make such change costly and time-consuming).

Performance models can be applied at any stage, including the earliest, to analyze for any problems for the system in different configuration and workload. When done in the very beginning this can assist the developers to design performance into the system. For deadline-driven systems, rate monotonic analysis [11] determines the schedulability of a set of tasks. For systems with probabilistic workloads and execution, simulation and queuing network models [9] analyze contention for resources using statistical workload descriptions such as mean service time and mean number of visits of jobs to servers. Smith and others have described how to create these models in early software design [1][21][22].

Layered Queuing Network (LQN) models [3][15][17] are a kind of extended queuing model which describes software resources and interactions, and are used here.

2.2 Layered Queuing Network Models

The LQN model is an extension to queuing models, as proposed by Woodside et al. and others [3][17][23]. These models can handle some of the important performance features such as multi-threaded processes, devices, locks and other communication tasks. It is useful for describing systems with parallel processes on a multiprocessor or a network-based client-server system.

LQN models contain *tasks* representing software and hardware components in the system, with *entries* representing operations by these tasks. Entries make requests to entries in lower layers. Tasks may take roles as "pure clients" which only originate operations as clients, or as "pure servers" which only serve, as for hardware devices, or as "active servers" which accept and serve requests and then while doing so, make their own requests to lower servers.

Interactions between entries are of three types: asynchronous, synchronous, and forwarding. An *asynchronous* service request does not wait for any reply to the request. As soon as the request is sent, the requester can continue with its operation. A *synchronous* service request can also be described as a RPC or a rendezvous request. The requester waits for a reply ("blocked") and then continues its own operation. A *forwarding* interaction is a combination. The server of a synchronous request forwards it asynchronously to another task, which has the responsibility of replying, or forwarding it again. The original task is blocked until the reply arrives.

A notation for LQN models is shown by an example in Figure 1. The parallelograms represent the tasks and entries, while the different types of service requests are identified by the different kinds of arrows. Client1 and Client2 originate asynchronous and synchronous requests, respectively, the latter being forwarded to Entry6. Entries have parameters for CPU demand and arcs for requests are labelled by the mean number of requests they make [3]. LQN models have been described for many kinds of applications (e.g. [20]).

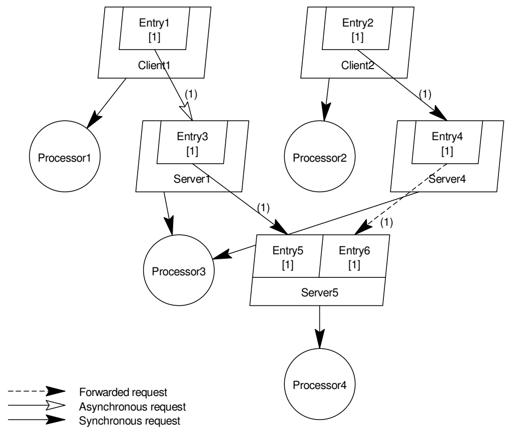

Figure 1: Example of an LQN Model

2.3 Approaches for Building Performance Models

In order to construct these performance models, we need to have a systematic approach. As proposed from Connie Smith, we should use quantitative methods to assess and change the system's attributes during the early design stage of the development cycle [20]. We can then use it to obtain the execution patterns and solve for the performance prediction. Other scenario-based approaches by Scratchley [18] and by Petriu and Woodside [15] create models from Use Case Maps. Menasce and Gomaa described a software performance engineering language CLISSPE [12].

The trace based approach taken here was originated by Hrischuk et al in [5] and extended in [6]. It identified patterns in the trace, including inter-task interactions described here and additional patterns involving forking and joining. An end-to-end response identifier called an "angio dye id" was carried through the system by the messages involved in the response, and is recorded in the trace events. The dye id gives a response context for every event, and also assists in relating forks and joins in the flow. A scenario was described as a graph, and LQN model structures (with request frequencies, see below) were created by a rule based graph analysis approach to define graph transformations [6]. The model could include parallel sub-paths.

Other performance analysis methods use traces to create profiles (time spent in different components, in context), or to visualize problems [7]. Klar et al. [10] created a series-parallel delay model from traces, and used it to predict performance. However none have gone as far as the angio-trace work, in creating queueing model structures.

The present work is closely related to angio-tracing described in [6]. Here, we give up the ability to capture parallel paths, in order not to require the angio dye ids in the trace.

3. SOFTWARE ARCHITECTURE AND MODEL EXTRACTION TECHNIQUE

In this paper, a new automated model generation technique call the Software Architecture and Model Extraction (SAME) technique is presented. To apply SAME, a scenario, or a set of scenarios, is executed in the system or the executable design model, and a trace of events for the interactions of the components is captured. Interaction must be by message passing.

Commonly available trace information such as the component identity, the type of even (send or receive, for inter-component events), and a timestamp, is all that is required. There must be enough information with a receive event to identify the corresponding send event, and to construct a message trace with records showing

[sender id, receiver id, time of reception]

The trace may come from a single node or a loosely coupled distributed system. From here on, the components in the trace will be called "tasks", in line with the LQN model. However in practice they may be objects, threads, or processes, or a combination of all three. A task must only be some kind of sequential program, running on a single "host" processor.

Figure 2 shows the steps in processing the trace. An analysis based on "interaction trees", described in the next section, identifies the interactions between the tasks as patterns. These interactions serve as input for generating the LQN models. With the use of performance parameters for CPU demand and request frequencies, an LQN model is produced.

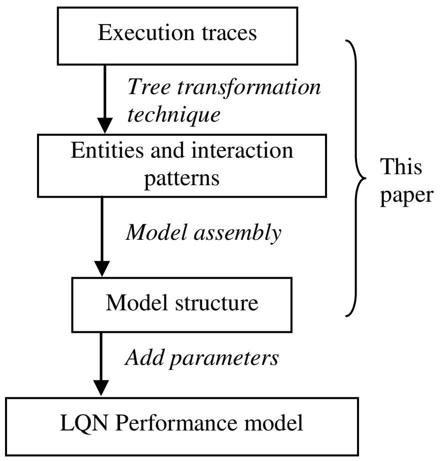

Figure 2: Steps for SAME Technique

3.1 Tasks, Trace Events, and Messages

To apply the SAME technique, the software must be composed of components with a unique identity, which receive messages one at a time (no internal concurrency) from a request queue, and which can block for replies (which do not have to wait in its request queue). Messages must be delivered in the order they are sent, for the same pair of sender and receiver.

In a distributed system separate traces are made at each node, or even within each process, and timestamps are often not precisely synchronized. The traces must be merged, or otherwise interpreted to associate send events and receive events for the same message, and give the message trace for SAME. This requires that timestamps be sufficiently well synchronized to obtain the correct causal order in the merged trace (the timestamp at the receiving end is later than that at the sender, when sending in either direction), or alternatively that messages carry additional identification that is recorded in the traces at each end, and can be used to associate the send and receive events correctly.

3.2 Interaction Patterns

The three interaction patterns mentioned above are important for identifying task blocking behavior, with its associated delays. Figure 3(a) illustrates all three patterns by showing a trace fragment with six messages between five tasks, as a UML sequence diagram.

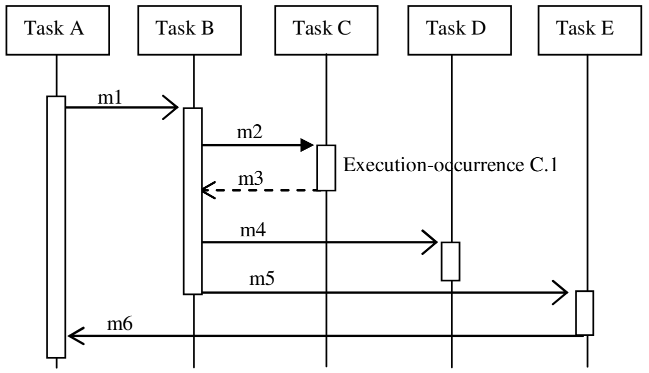

Figure 3(a) A trace fragment as a Sequence Diagram, showing an execution-occurrence and three interaction patterns

In the example shown, the first message m1, from A to B, begins an "execution occurrence" (which we will abbreviate as an EO) at A. It makes A block, but we will return to it in a moment. Messages m2 and m3, between B and C, make a synchronous interaction shown in Figure 3(b). The "client" task B sends a request m2 to C which is processed by the "server" task C while the client task waits (is blocked). The server sends back a reply m3 to the client, ending its EO. While it is busy further requests to C are put into a service queue. During the service, the server may make nested synchronous requests to other tasks. This type of communication pattern can be seen in many different types of client-server systems.

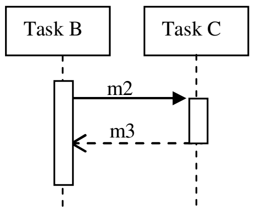

Figure 3(b) The synchronous interaction between tasks B & C

The interaction from B to D is shown separately in Figure 3(c). It is asynchronous, in that B sends a message and does not expect a reply. Both B and D can be executed concurrently. While the server is servicing a request, other requests have to wait. Since the client does not wait for a reply, it is important that these messages are reliable so they do not get lost during the transmission.

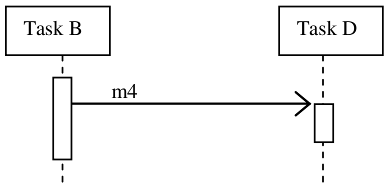

Figure 3(c) The asynchronous interaction from B to D

The third type of interaction is a forwarded request, represented here by the messages in Figure 3(d). The message m5 from B to E forwards the request from A to B, on to E. E must take the responsibility to send the reply (m6) while B ends its EO, and may take its next request. A message may be forwarded any number of times, with the last server replying to the client, which remains blocked. The forwarding pattern in Figure 3(a) begins with m1, and includes m5 and m6; the other messages are part of the service given by B before it forwards the request. For example, B might have to do a lookup (at directory task C) to find the address of the ultimate server, and log the operation (at logging task D) before sending the request to the ultimate server E.

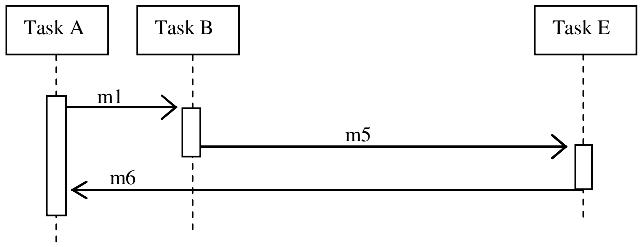

Figure3(d) The forwarding interaction from Task A, to B, to E

The communications patterns in Figure 3(a) may be expressed by the graph in Figure 3(e), in which the nodes represent the execution occurrences and the arcs represent the messages. Because a task may have many EOs in a trace, the nodes carry a number (A.i represents the ith EO of task A); here the number 1 is used for all the tasks.

The three interaction patterns appear in this graph. For example the cycle B.1-C.1-B.1 represents the synchronous request from B.1 to C.1, and the cycle A.1-B.1-E.1-A.1 represents the forwarding. Clearly the interpretation of a message as part of an interaction depends on its context in the message flow. In analyzing such a graph we will interpret any interaction as synchronous or forwarding, if that is feasible. Any message which might be a reply is interpreted that way.

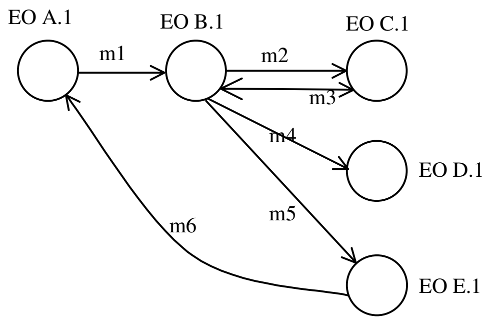

Figure 3 (e) The interpretation of the Trace Fragment as an Interaction Tree (This form is a graph rather than a tree)

4. INTERACTION TREE ANALYSIS

The technique to be described uses Interaction Trees, which are similar to Figure 3(e) except that they are created on the fly from the stream of messages in the trace, and they evolve for each message. As in Figure 3(e), nodes represent EOs of tasks, and arcs represent messages. These graphs are trees because any cycles are removed as soon as they are created. A set of trees is maintained by the algorithm, for different concurrently active interactions.

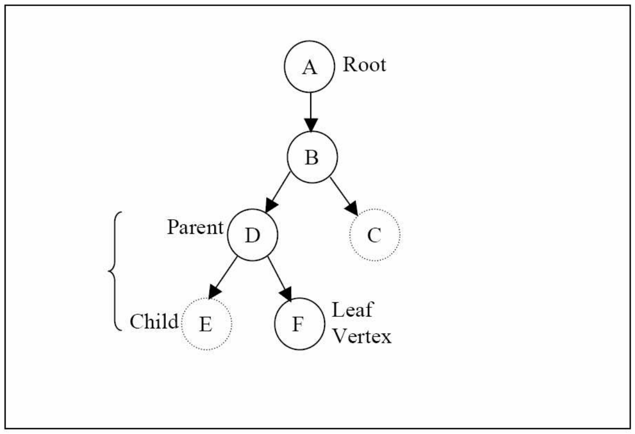

Figure 4: A Typical Interaction Tree

Figure 4 illustrates an interaction tree and the usual terms root, leaf, parent and child nodes.

4.1 Sketch of Interaction Tree analysis

The processing of one item from the message trace is as follows:

Step 1. Assuming there is already a live EO for the sender, a node is created for the EO of the receiver, which is attached by a directed arc from the active EO of the sender. If there already is a live EO for the receiver, the arc goes to that node, which may create a cycle.

If there is no active EO for the sender, a node is created for it which becomes the root of a new tree, and a node for the receiver is created and attached to it. Any existing live node for the receiver is made a *zombie* (it can no longer have arcs attached to it). Zombie nodes are removed in Steps 2 and 3.

Step 2. If a cycle was created in some tree by the added message, it may be a reply message which completes an interaction. The cycle is analyzed and if possible, interaction records are produced and the tree is simplified, to remove the messages in the cycle. The simplification may identify some number of additional asynchronous interactions, and may divide one tree into several smaller trees.

Step 3. A cleanup operation is applied to all the trees that have been modified or created during Steps 1 and 2. This may identify further asynchronous interactions and create further divisions into smaller trees.

4.2 Definitions for Interaction Tree analysis

Interaction Tree Analysis creates and evolves a set of trees which is initially empty. For each task X it maintains the EO number $live_X$ of the current live node (if any; it is zero if there is none), and the next EO number $next_X$ to be assigned.

Each node is name X.i for the task and its EO and is labeled with (t, status) for the time of its invoking message and its status (live or zombie). Each arc is labeled by the time of the message it represents. Nodes are live when they are created; they may be turned into zombies when the same task participates in another message. There can only be one active node for a task at any time, across all trees.

Because each global interaction begins with a single message, the root node of each tree has only one child. During a step a root may be produced with multiple children (called "improper"), but cleanup will reduce this to one or more proper trees.

The most recent active child node of any node is called its "immediate candidate" or IC; we may say that Child = IC(Parent) or "Child is IC". In Figure 4, the nodes C and E with dashed outlines are "non-immediate candidate" or NIC; B, D and F are IC.

If there is a direct path from active node A to active node B in any tree, we say A is an ancestor of B, written A => B. For example in Figure 4, A => B and B => F.

4.3 Algorithm

Only an informal definition of the algorithm is given here; a full definition (in a somewhat different form) is given in [8].

Part A. *Initialize* the set of tasks and their counters live (to zero), and next (to 1).

Part B. *Process message records* until the trace terminates. For each message (A, B, t), we define the existing active nodes for A and B (if any) to be A.u and B.v and their next EOs to be A.i and B.j. Then:

1. *Add a message arc* with time label t. Nodes A.u or B.v may be present or absent. There are two cases:
   - if NOT(B.v => A.u) (note: no cycle will be created),
     - if (A.u exists) attach the arc from A.u to a new node B.j with label (t, live)
     - else create new node A.i with label (t, live) and a child B.j with (t, live).

     In either case if a previous B.v exists it is made zombie when B.j is created.
   - else (note: A.u, B.v exist and a cycle will be created):
     - insert the arc from A.u to B.v
2. *If there is a cycle* (there should be only one), suppose the last message is from node A.u with label (t, live) to a node B.v with label (t, live). Then
   - if the cycle length is 2, produce an interaction record [Sync, (B, v, tstart), (A, u, tend)], with tend taken as the time on the arc (A,B)
   - if the cycle length is N>2, produce an interaction record [Fwd, (B, v, tstart), List, (A, u, tend)], with
     - (B, v, tstart), (A, u, tend) as above,
     - Construct List as a list with N-2 entries of form (X, k, time) for the intermediate nodes X.k, in order along the path from B to A.
3. *Tree cleanup*. There is a sequence of operations, which will be outlined only.
   - if there is a cycle, make zombies of all the nodes in the cycle below the highest node in the cycle (which initiated the interaction). Remove all the arcs on the cycle, and any nodes that are isolated by this.
   - remove any zombie root node X.k and for each child Y.m generate an asynchronous interaction record [Async, X, k, Y, m, t], where t is the time label of Y.
   - for any improper root node X.k, remove all the arcs except the one to its IC, and detach their sub-trees as new trees. For each removed arc from X.k to some node Y.m, generate an interaction record [Async, X, k, Y, m, t] as above. This must be done recursively as the detached trees may be improper also, and the previous step must be invoked if any zombie root node emerges.
   - Remove any zombie leaf node Y.m (and the arc from its parent X.k), and generate an interaction record [Async, X, k, Y, m, t] as above. This may have to be done recursively also.
   - Update the numbers $next_X$ for all tasks X that had nodes created, and $live_X$ for tasks with nodes that have changed status, in preparation for the next step.

When the trace terminates, for each child node Y.m of parent node X.k, create an asynchronous interaction record [Async,X,k,Y,m,t]. At this point we have a set of interaction records, each of which represents a single interaction.

Part C. *Assemble the interaction records into a layered queueing model.*

1. For each task A that is recorded in the interaction records, create an LQN task.
2. For each EO of form A.i that is recorded in the interaction records, create an entry A.i for task A.
3. For each Sync interaction record [Sync, (A, u, tstart), (B, v, tend)], add a synchronous interaction arc from entry A.u to entry B.v in the LQN,
4. For each Async interaction record [Async, A, u, B, v, t], add an asynchronous interaction arc from entry A.u to entry B.v in the LQN,
5. For each Fwd interaction record [Fwd, (B, v, tstart), List, (A, u, tend)], a series of arcs are created. Let C(k) be the entry

corresponding to the kth element of List (if this element is (X, i, t), then C(k) is entry X.i). Then

- add a synchronous request from entry A.u to entry C(1),
- for j = 1 to N-3, add a forwarding request from entry C(j) to entry C(j+1). If List has only one element there is nothing to do.
- add a final forwarding request from entry C(N-2) to entry A.u

The reply from C(N-2) to entry A.u is implicit in the ending of the forwarding path.

The multiplicity parameters of the synchronous and asynchronous request arcs are all set to 1.

The result of part C is a LQN model with tasks which may have many entries, one for each EO. Many of these entries may perform the same operations, repeated within the trace, and they should be merged into a single entry.

Part D. *Simplify the entries* to merge those that perform the same operations.

1. Sort the tasks into an order such that, where possible, tasks earlier in the order originate interactions with later tasks.
2. Working from tasks at the end of the list, towards the beginning, merge identical entries. Two entries are "identical" if they belong to the same task, and initiate the same requests (to the same set of entries and with the same multiplicities), or the same forwarding operations. The request arcs from the merged entry have the same multiplicities as the two original entries; each arc into the merged entry has the sum of the multiplicities on the input arcs from the same source entry.
3. This is repeated until no candidate pair of entries can be found.

The result of Part D is the final LQN model structure. To create an LQN model, the performance parameters must be estimated and inserted, as shown in Figure 2.

4.4 Illustrations

Two brief examples will illustrate Part B in processing a tree and Part C in assembling a model.

Processing a tree (Part B)

Figure 5 illustrates one step of the algorithm beginning with the tree on the left. A message (E, B, 7) is next in the trace. In applying Part B, Step 1, the first option is taken because E.u does not exist, (and thus B.1 is not an ancestor of E.u). Because E.u does not exist a new tree with E.1 and B.2 is created. As part of creating B.2, the previous live node B.1 is made zombie.

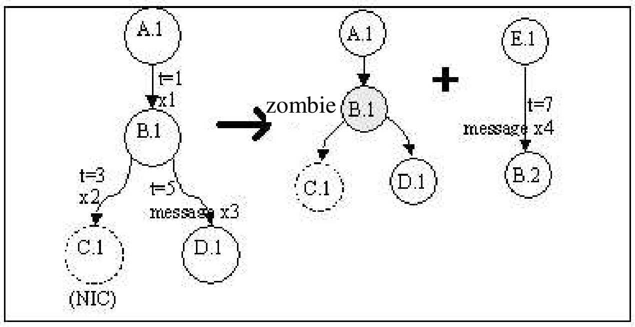

Figure 5 An interaction tree and its transformation on receipt of a message (E, B, 7), labelled x4

Assembling a model

Suppose the messages shown in Figure 3(a) are recorded in a message trace with entries (A, B, t1), (B, C, t2), etc. Then when the algorithm is applied to them it gives the following set of interaction records, in this order:

[Sync, (B, 1, t2), (C, 1, t3)]

[Async, B, 1, D, 1, t4]

[Fwd, (A, 1, t1), (B, 1, t5), (E, 1, t6)]

Part C first identifies the tasks A, B, C, D, E from the tasks in the records, and the entries A.1, B.1, C.1, D.1, E.1. It then connects them together from the interactions, to give the LQN shown in Figure 6.

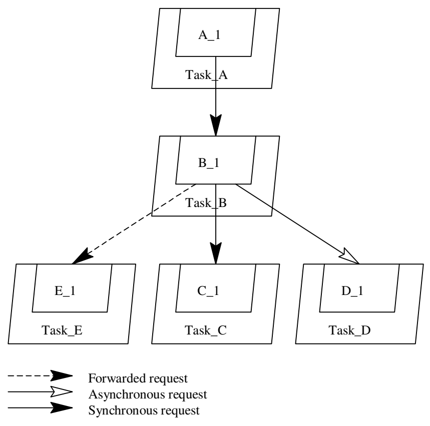

Figure 6 Final LQN model for the example of Figure 3(a)

4.5 Generalizations

As described in [8], the algorithm can also accommodate

- trace records which describe non-message events, and can be interpreted to deactivate nodes when they indicate that a task is not blocked waiting for a reply.
- tasks that have operations after sending a reply (called "second phase" operations in LQN).
- combining the results of processing multiple traces, in a single model.

4.6 Implementation of SAME

The implementation follows a somewhat different description of the algorithm, given in [8], which breaks the processing of one message into 19 different cases. These cases are determined by the location of live nodes B and A in the interaction tree. Cases 1 to 9 deal with live nodes A and B in different trees or not existing; cases 10 to 13 have the vertices A and B in the same tree but neither is a descendant of the other. Cases 14 to 16 have the vertex A a descendant of vertex B, and in cases 17 and 18 process A performs some non-interaction operation. The last case deals with trace termination.

All the techniques and algorithms described in this section have been implemented in a Java program with two parts, SAME1 and SAME2. SAME1 implements parts A and B above, taking as input the execution traces in an ASCII file format and generating an ASCII file with the interaction records. SAME2 takes this output file and outputs an LQN model.

5. VALIDATION TESTS AND CASE STUDY

With the wide possibilities of the different communication patterns and scenarios, it is important to perform some testing to make sure the resulting LQN models generated are accurate. This section describes 6 simple test cases that were used for validation along with a comprehensive case study on the ATM-GSM network model. Please refer to [8] for the complete details of the tests and resulting LQN models. The test case and results are discussed in the following.

5.1 Validation Test Cases

The first case represents the synchronous communication pattern as shown in Figure 7. The first interaction is at t = 10 where B received a message from A with A replying back to B at t = 100. In the SAME tool, it detected a synchronous communication pattern from the interaction tree based on case 14 of the tree transformation technique as shown in the diagram, and it was able to generate the LQN models for it.

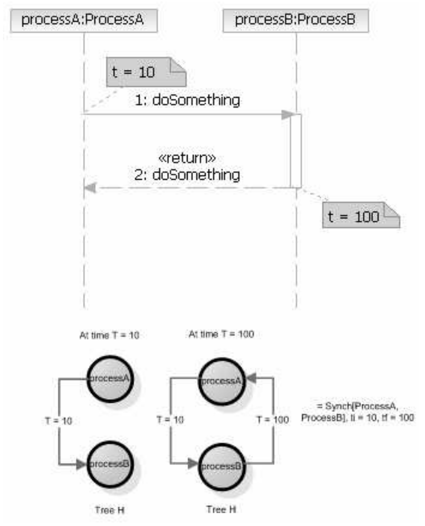

Figure 7: Simple Synchronous Communication Pattern

The second case illustrates the asynchronous communication pattern between two processes as shown in Figure 8. As time t = 10, process A sends a message to process B but there was no reply to the request. The interaction tree is created but there was no message as the reply when the interaction ends. Based on the last case of the tree transformation technique in the SAME, this interaction can be classified as an asynchronous interaction as it shows in the communication pattern in the diagram. An LQN model is created based on that.

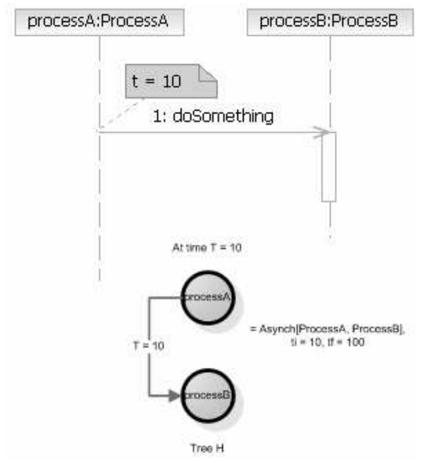

Figure 8: Simple Asynchronous Communication Pattern

The third case is an example of a simple forwarding communication pattern between three processes as shown in Figure 9. At t = 10, process B receive a message from process A and then it sends a message to process C at t = 100. Process C then replies process A at t = 150. The interaction tree is constructed as shown in the diagram. From case 16 in the tree transformation, SAME has determined that it is a forwarding interaction and was able to use it to build the appropriate LQN model.

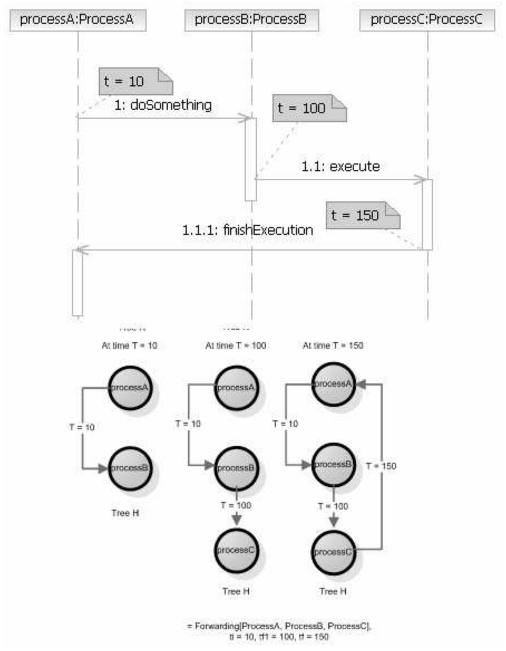

Figure 9: Simple Forwarding Communication Pattern

The fourth case is a synchronous pattern with nested interaction between three processes as shown in Figure 10. At t = 10, process B receives a message from A and then sends a message to process C at t =100. At t = 200, process B replies back to Process A. Since process A has received a reply, it is considered as a synchronous communication pattern. Since there was no reply from process C and process B was not blocked after the message, it is considered as an asynchronous message as stated in case 2 in the algorithm. We can see the communication pattern shown in the diagram and it was used to create a simple LQN model.

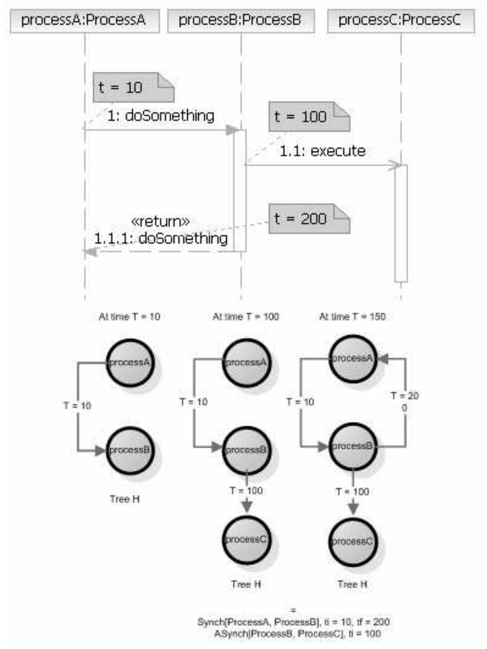

Figure 10: A Nested Asynchronous Message

The fifth case is a two step forwarding communication pattern involving four processes as shown in Figure 11. First, the message is passed from process A to process B, process B to process C, process C to process D. After that, process B sends a message back to B. From case 16 of the tree transformation technique, this is considered as the forwarding pattern. After that, process B replies to process A to end all the interactions. From case 14 of the tree transformation, this is a synchronous technique. From this scenario, we can see that there is a forwarding interaction nested inside a synchronous communication pattern. The results are produced in the diagram and were used to make the LQN model.

The last test case is an example of a concurrent system as shown in figure 12. SAME is able to handle the processes the same way as a non-concurrent system, but there are multiple trees used to represent the concurrent behaviors. In this test case, process A and process C starts two simultaneous execution threads interacting with different processes. The trace starts at t = 10 where process B receives a message from B and then process D receive a message from process C at t = 20. At this point, two initial interaction trees are formed. At t = 100, process D sends a message to process E with process B sending a message to process D at t = 110. Since process D is in another tree, this interaction causes vertex D in the interaction to become inert. This inert node allows process E to finish its interaction by a having a forwarding request to send a message back to process C, as it is done at t = 150, instead of having a reply back to process D which is currently in use.

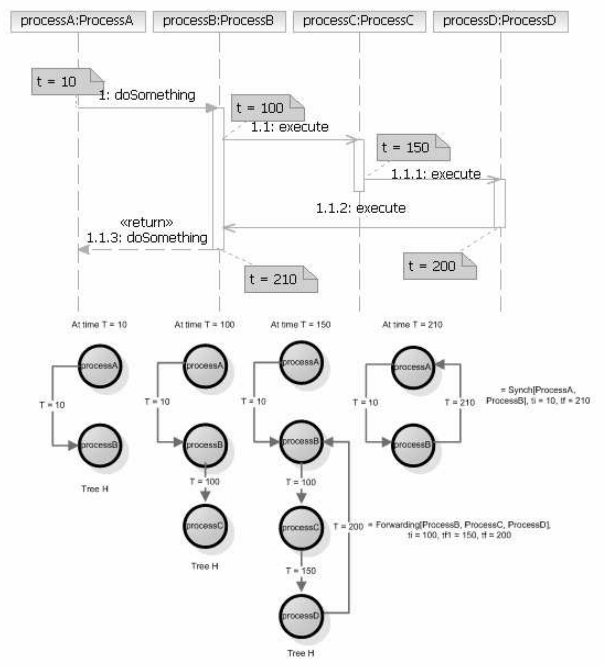

Figure 11: A Nested Forwarding Communication Pattern

After the reply from process E to process C, we have finished the forwarding interaction as stated in case 16 of the tree transformation technique.

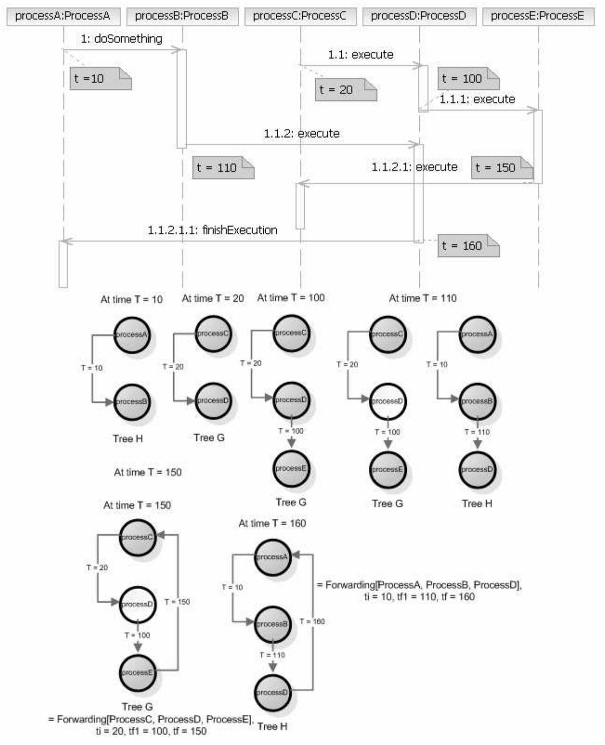

Figure 12: Concurrent System

At t = 160, process D sends a message back to A, and this is also another forwarding interaction. We can see the results produced from the interaction tree in the diagram. These are used to generate the LQN model as shown in figure 13.

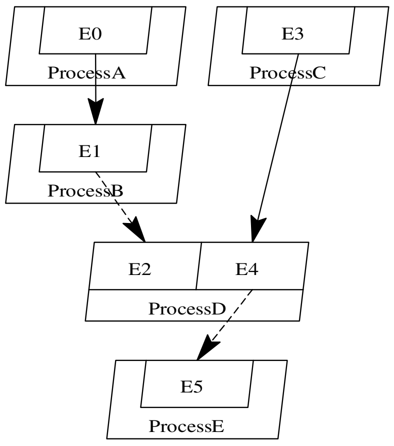

Figure 13: Concurrent System LQN Model

From the above six test cases, we were able to validate the techniques that were used to identify the different communication pattern to generate the appropriate performance model. The following section describes a case study using SAME to model the performance of a complex system.

6. CASE STUDY: RADSBookStore

RADSBookStore is an application created to evaluate and verify the logic of the SAME Technique. It is a prototype of a computerized book store system supporting most of the operations one can do on a book store computer terminal, with the Use Cases in Figure 14.

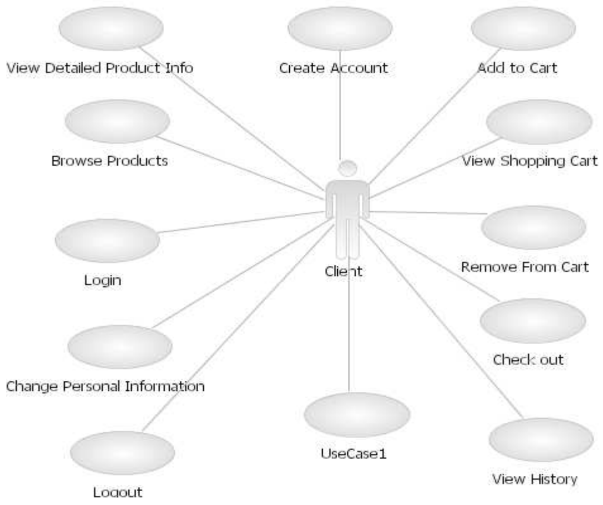

Figure 14 Use cases for the RADSBookStore system

The RADSbookstore application is a three tier system comprising of the client, the application and the database layer.

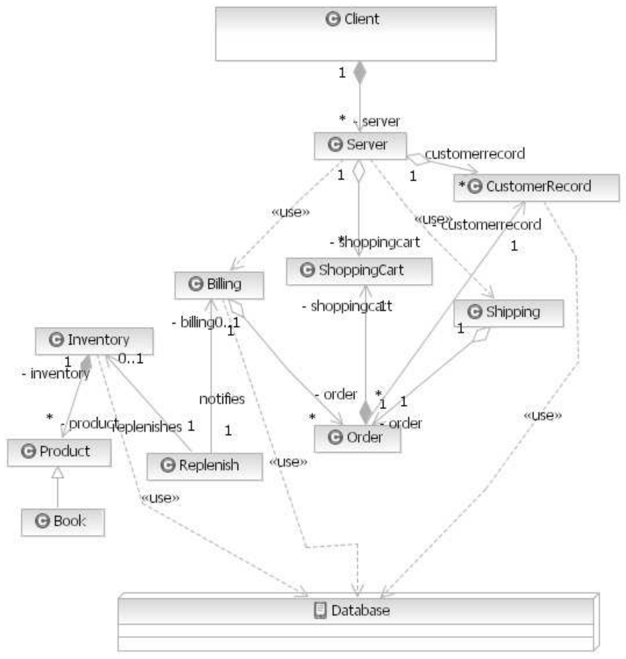

Figure 15 Classes in the RADSBookStore software

The client layer serves as the front-end of the system from where requests of various sorts originates. The application layer consists of a main server which is responsible for serving clients requests by creating new customer accounts, sending requests to the billing and shipping departments, query books' and customers' information and maintaining shopping carts for the customers involved in current sessions. Further, there is also a replenish thread that serves the purposes of replenishing the inventory upon which the billing and the shipping department serves the backorders which were created due to inventory shortfall.

For the purposes of obtaining execution traces from the application, RADSBookStore was instrumented with trace messages which were then recorded in a file while executing the use cases shown in Figure 14. For the BrowseProducts operation, the sequence diagram shown in Figure 16 shows the messages that will be recorded during one execution.

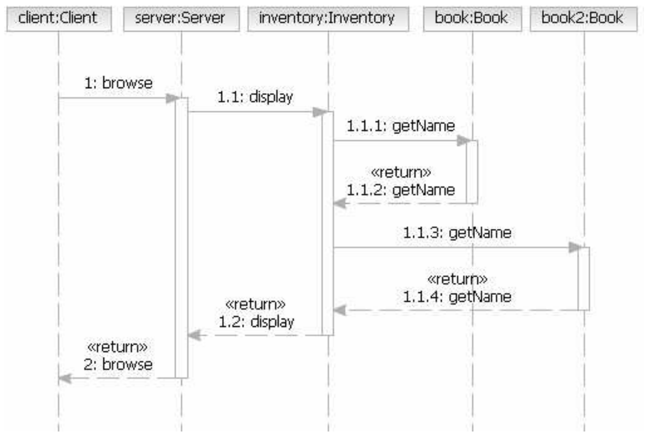

Figure 16 The sequence diagram for the BrowseProducts operation

When this operation was executed on the prototype it produced the trace in Table 1.

Table 1 Trace for BrowseProducts

Time	Event	Process	Message Type
4052950	send	Client	browse_STARTC
4053220	receive	Server	browse_STARTC
4053500	send	Server	display_START
4053720	receive	Inventory	display_START
4053990	send	Inventory	getName_START
4054270	receive	Book	getName_START
4054490	send	Book	getName_END
4054760	receive	Inventory	getName_END
4055030	send	Inventory	getName_START
4055310	receive	Book2	getName_START
4055530	send	Book2	getName_END
4055750	receive	Inventory	getName_END
4056020	send	Inventory	display_END
4056240	receive	Server	display_END
4056460	send	Server	browse_ENDC
4056740	receive	Client	browse_ENDC

Each of these traces was processed using the Interaction Tree Analysis algorithm to first identify different types of interactions and then, in Part D, to create an LQN model for each scenario. The LQN models created were sufficiently syntactically and semantically correct to satisfy the model parser for the solver, in all 12 cases. The models gave reasonable solutions.

The scenarios were then merged by creating a Client task which makes all the initial requests, in proportions corresponding to a usage profile for the scenarios. The tasks were merged by name (so the merged task has the combined entries from all the submodels) and Step D was applied to simplify the entries of this combined model. A composite model of two scenarios is shown in Figure 17. The model at this point has no meaningful CPU-demand parameters, but with default values of 1 ms for each entry the model can be solved with the LQNS solver [3] to give a system throughput of about 10 responses/sec.

Scalability

Traces of up to 650 events were collected from the bookstore prototype and processed into models. Other applications which were successfully analyzed include traces from an executable design model with about 50000 events, and from a running system with nearly half a million events. This demonstrates the scalability of the technique to large traces. For very large traces, the memory requirements for Steps C and D can be improved by doing them incrementally.

7. CONCLUSIONS

This paper has described an approach to automating the construction of performance models of software, from traces of behavior of running systems, prototypes or executable models. A strength of this SAME technique is that it uses conventional trace data which is available from many tracing tools, compared to a previous technique which required special traces. The price paid for this simplicity is that the joining of flows that previously forked cannot be identified, however the existence of the parallel sub-path and its workload is captured here.

The layered (LQN) performance model that is derived captures the software workload, and also the effect of concurrent interacting processes with various kinds of interactions (synchronous, forwarding and asynchronous). This class of model is useful for studying concurrency, threading levels, and other kinds of logical resources.

The interactions between software components are identified by an Interaction Tree analysis which is efficient and scales up well to large traces (hundreds of thousands of trace events). The performance model clusters similar interactions into single architectural units called entries in LQN notation, which reduces the complexity so that it distinguishes only those interactions which have different behavior.

The technique is demonstrated on a moderate sized three-tier application, called RadsBookStore. From the results generated from the test case scenarios, we obtained LQN models that are syntactically and semantically correct and are essentially the same as those generated by hand. When performance parameters were inserted, the models could be solved with the LQN solver to obtain the performance results.

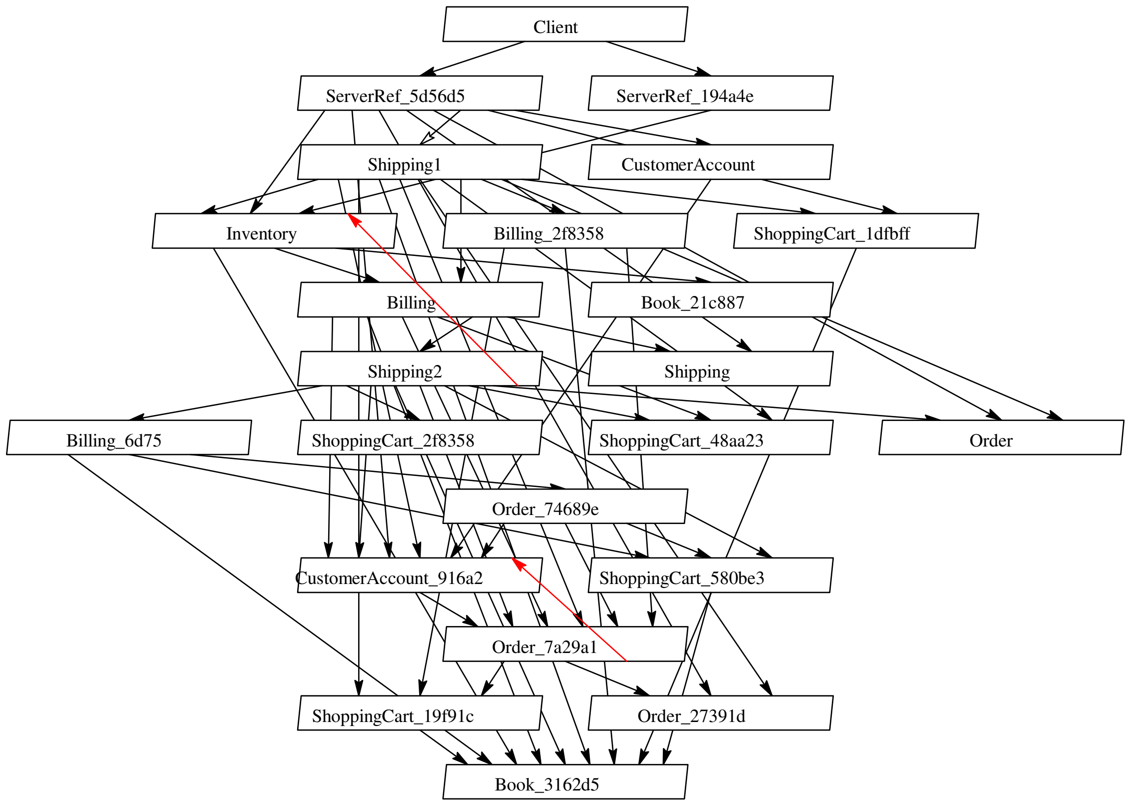

Figure 17. A view of a composite LQN model for the bookstore case study combining the behavior of two scenarios. The parallelograms are tasks; the entries are not shown. The numerical substrings identify object instances.

The tool has also been applied to a large trace of a production system with over a hundred threads, and a quarter-million events, to generate a model.

The approach has limitations. First, the trace events must have globally ordered timestamps, or unique message ids. Second, if there are multiple threads, traces must identify the thread. Third, if there is no unique message identifier, then we must be able to assume the messages between components are served in FIFO order. Fourth, the approach cannot identify the joining of two parallel flows.

Currently, one research area is to automate the trace generation from an executing system to make traces that are compatible with the SAME tool. The target for this trace generation is on the J2EE platform. This can be useful since a performance model can be generated from the execution of the program without extra instrumentation. It allows the developers to get instant performance measurements during the testing phase of the system. This can be very useful in the development environment.

8. ACKNOWLEDGEMENTS

This research was supported by a grant from NSERC, the Natural Sciences and Engineering Research Council of Canada, and CITO, Communications and Information Technology Ontario.

9. REFERENCES

[1] V. Cortellessa and R. Mirandola. *Deriving a Queueing Network based Performance Model from UML Diagrams*, in Proceedings of the Second International Workshop on Software and Performance (WOSP2000), Ottawa, Canada, September 17-20, 2000, pp. 58-70.

[2] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Complany, 1997.

[3] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside. *Performance Analysis of Distributed Server Systems*, Proc. Sixth International Conference on Software Quality (6ICSQ), Ottawa, Ontario, 1996, pp. 15-26.

[4] F. Hayes-Roth, and D. Waterman. *Principles of pattern-directed interface systems.* In D. Waterman and F. Hayes-Roth, editors, Pattern-Distributed Inference Systems, pages 577-601. Academic Press, 1978.

[5] C. Hrischuk, J. Rolia, and C. M. Woodside. *Automated generation of software performance model using an object-oriented prototype*. Int. Workshop on Modeling and simulation, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '95), pp. 399-409, Durham, NC, 1995.

[6] C. Hrischuk, C. M. Woodside, J. Rolia, R. Iversen. *Trace-based load characterization for generating software performance models.* IEEE Transactions on Software Engineering, vol. 25, no. 1, January 1999.

[7] IBM, *IBM Rational PurifyPlus for Windows: Quantify Component*, http://www.pts.com/wp2292.cfm?det=y

[8] T. A. Israr. *Lightweight Technique for Extracting Software Architecture and Performance Models from Traces.* Master's Thesis, Carleton University, 2001.

[9] R. Jain, *The Art of Computer Systems Performance Analysis*. John Wiley & Sons Inc., 1991

[10] R. Klar, A. Quick, F. Soetz, "Tools for a Model-driven Instrumentation for Monitoring", Proc. 5th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation (TOOLS91), pp 165-180 Elsevier, 1992.

[11] C. L. Liu and J. W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*. J. ACM, 20, pp.40-61.

[12] D. Menasce and H. Gomaa. *A method for design and performance modeling of client/server systems,* IEEE Transactions on Software Engineering, vol. 26, no. 11 pp. 1066-1085, 2000.

[13] Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification*, OMG Adopted Specification ptc/02-03-02, July 1, 2002.

[14] The Open Group, *Systems Management: Application Response Measurement (ARM) API,* Technical Standard, July 1998.

[15] D.B. Petriu and M. Woodside, *Software Performance Models from System Scenarios in Use Case Maps*, in Proc. 12th Int. Conf. on Modelling Tools and Techniques (TOOLS 2002), London, England, April 2002.

[16] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B. Schwartz, and L.F. Tavera, *Scalable Performance Analysis: The Pablo Performance Analysis Environment*, Proc Scalable Parallel Libraries Conference, Starkville, MS, Oct. 1993, IEEE Computer Society Press, pp. 104-113

[17] J. R. Rolia and K. Sevcik. *The method of layers*. IEEE Transactions on Software Engineering, Vol. 21, No. 8, pp. 689-700, 1995.

[18] W. C. Scratchley and C. M. Woodside, *Evaluating Concurrency Options in Software Specifications*, in Int. Conf on Modelling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS), College Park, MD, Oct. 1999, pp. 330-338

[19] B. Selic, M. Woodside, C. Hrischuk, and S. Bayarov, *A Wideband Approach to Integrating Performance Prediction into a Software Design Environment,* in Proc. First Int. Workshop on Software and Performance (WOSP98), October 1998, pp. 31-41.

[20] F. Sheikh and C. M. Woodside, *Layered Analytic Performance Modelling of Distributed Database Systems*, in Proc. Int. Conf. on Distributed Computer Systems, Baltimore, May 1997, pp. 482-490.

[21] C. U. Smith. *Performance Engineering of Software Systems.* Addison-Wesley Publishing Co., New York, NY, 1990.

[22] C. U. Smith and L. G. Williams, *Performance Solutions*. Addison-Wesley, 2002.

[23] C. M. Woodside. *Throughput calculation for basic stochastic rendezvous networks.* Performance Evaluation, Vol. 9, No. 2, pp. 143-160, 1989.