Scenario-Based Performance Engineering with UCMNav

Dorin Petriu[1], Daniel Amyot[2], and Murray Woodside[1]

[1] Department of Systems and Computer Engineering
Carleton University
Ottawa, ON K1S 5B6, Canada
{dorin, cmw}@sce.carleton.ca
[2] School of Information Technology and Engineering
University of Ottawa
Ottawa, ON K1N 6N5, Canada
damyot@site.uottawa.ca

Abstract. The analysis of a scenario specification for a new system can address some questions of system performance, in the sense of delay and capacity estimation. To assist the analyst, a performance model can be generated automatically from a Use Case Map specification in the UCM Navigator (UCMNav). This paper describes the process, and the information that must be supplied in the way of scenario annotations. It illustrates the tool-supported process with a substantial example related to electronic commerce, which demonstrates the impact of provisioning the software architecture for concurrency.

1 Introduction

Software performance engineering (SPE) is concerned with performance characteristics (metrics) such as response times, delays and throughputs, and it aims to insure that software products under development will meet their performance requirements. SPE uses predictive performance models to analyze the effect of software features on performance metrics for systems with timing and capacity requirements. SPE should begin early in the software lifecycle, before serious barriers to performance are frozen into the design and implementation. Although existing methods for early analysis are successful, the transfer of designer knowledge into the performance model is slow and expensive [18].

Scenario specifications provide a powerful starting point for system design and for analysis of various kinds of requirements. *Use Case Maps* (UCMs) are a graphical language specifically used for expressing scenarios, and for experimenting with scenario interactions and architecture [3, 9]. The UCM notation is part of the upcoming User Requirements Notation, currently standardized by ITU-T [8]. Among the numerous scenario notations surveyed in [1], UCMs are notably fit for many requirements engineering activities and for transformations to other modeling languages. A UCM tool, the *UCM Navigator* (UCMNav [13]) has been augmented to assist with the early analysis of performance questions,

from scenario specifications. This tool has already been used in various SPE case studies [16, 17, 19] and in SPE graduate courses. This paper addresses the details of how to begin such an analysis, by considering the performance attributes of scenarios and how they are represented in tools.

The analysis of performance from scenario specifications is an active area of investigation [18]. Many other approaches are based on adding performance attributes to behaviour models, with scenario-based and state-based languages. For instance, Message Sequence Charts (MSCs) [7] can be supplemented with performance information to generate SDL [6] specifications, as suggested by Dulz *et al.* [5] and by Kerber [11]. UML behaviour models [14] can also have such annotations. Kähkipuro uses performance-oriented UML models (in addition to the design model) to generate performance models [10], whereas Woodside *et al.* extract performance models from UML designs in a CASE tool [22]. A survey of methods for building performance models from UML specifications is given in [2]. The advantages of UCMs for SPE are the capture of scenario interactions and of architecture issues, the ability to describe scenarios without specifying explicit inter-component collaborations, and the flexibility to rapidly modify the architecture and to re-analyze, as studied by Scratchley in [19].

There exist many families of performance modeling languages. A very recent study showed that queueing networks (QN) provide higher scalability and adequacy for performance analysis than process algebras and generalized stochastic Petri Nets [4]. *Layered Queueing Networks* (LQNs) are supersets of QNs [12]. They capture the workload within activities (operations connected in sequence or in parallel) which belong to an entry of a task (e.g. method of an operating system process) running on a host device (usually a processor). The host device is usually a processor, a task is often an operating system process, but may also be an object, and an entry is like a method.

The following sections describe the steps needed to do a performance analysis of scenarios specified with the UCMNav editor. A tutorial example of an electronic commerce system (e-bookstore) is presented in order to illustrate these steps, which are:

- start with a UCM model that is "sufficiently complete" for performance analysis;
- augment the UCM with performance-related data;
- generate a Layered Queueing Network performance model and solve the model (this step is automated);
- analyze the LQN results with reference to performance requirements and goals, and revise the UCM if needed.

The reader unfamiliar with UCMs or LQNs can find additional tutorial material in [3, 9, 12, 23].

2 UCM Model

A UCM describes a system as a set of *paths* that traverse a set of *components*. The operations of the system are captured as *responsibilities* along a path, allocated

to particular components. The sequence in the path represents causality and control, including forking into parallel subpaths. The movement of the path from one component to another represents transfer of control, without showing details of how this is accomplished. Path detail can be hidden in sub-diagrams called *plug-ins*, contained in *stubs* (diamonds) on a path. More details on the UCM semantics can be found in [9].

The steps required for performance analysis will be presented using an example of a Web-based bookstore called the *RADSbookstore* (for selling books on Real-time And Distributed Systems), also described in [17]. The RADSbookstore provides the following facilities:

- an interface for customers - to browse the catalogue and to buy books;
- an interface for bookstore administrators - to examine the inventory and the sales data;
- separate databases for the inventory and the customer accounts;
- applications to manage customer accounts, shopping cart objects, and the inventory; and
- a subsystem to track and fill back-orders (orders to be filled for books that are not in stock).

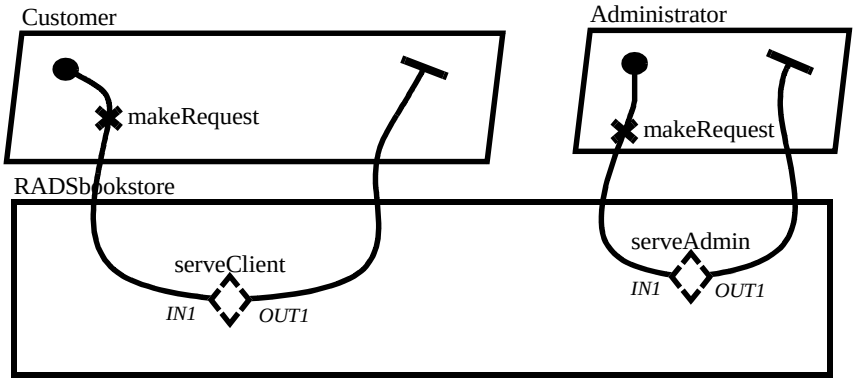

Fig. 1. UCM root map for the RADSbookstore.

Figure 1 shows the RADSbookstore root (top-level) map. The parallelogram shaped components represent concurrent processes, while the rectangle shapes are uncommitted architectural elements. The details of service operations are entirely hidden in the stubs; indeed there are seven different client operations and two different administrator operations. These are *dynamic stubs* with selection of the appropriate plug-in UCM according to a request type. In this style, the root UCM represents a large number of scenarios. Because of space limitations here, we will illustrate only the checkout scenario for customers, which has two levels of plug-ins as shown in Figure 2 and Figure 3, and the fill backorders scenario for administrators, shown in Figure 4.

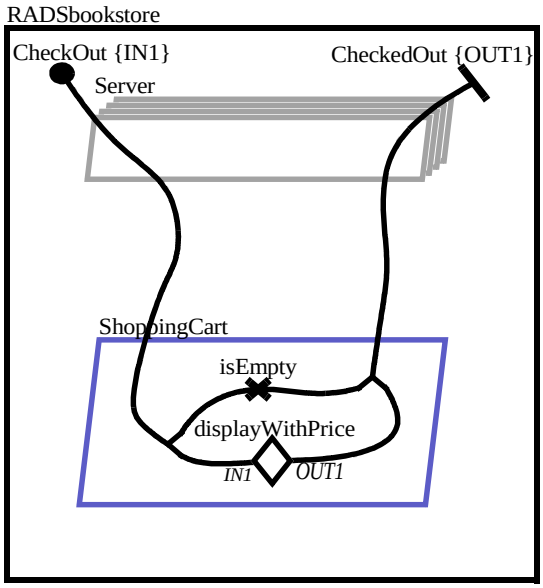

Fig. 2. First level plug-in for the checkout scenario, in stub serveClient.

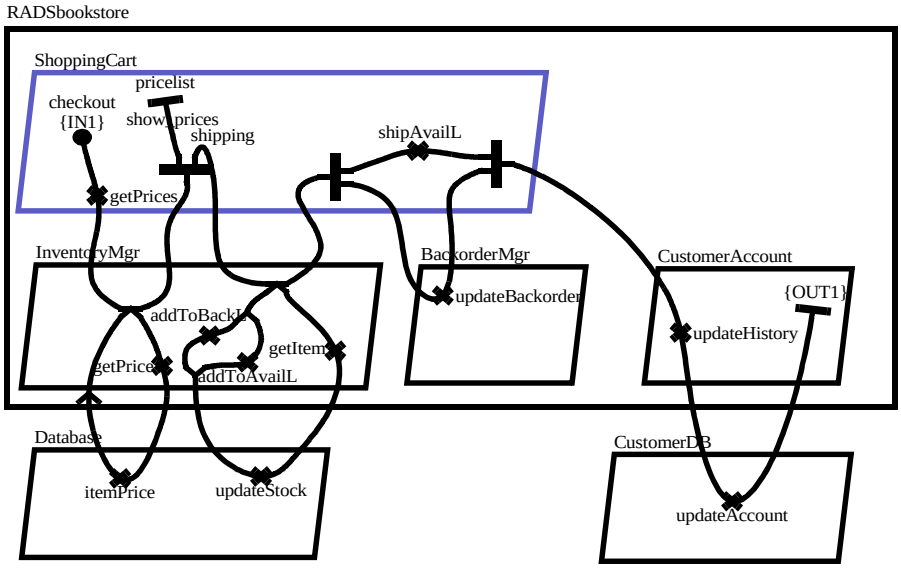

Fig. 3. Second level plug-in for the checkout scenario, in stub displayWithPrice.

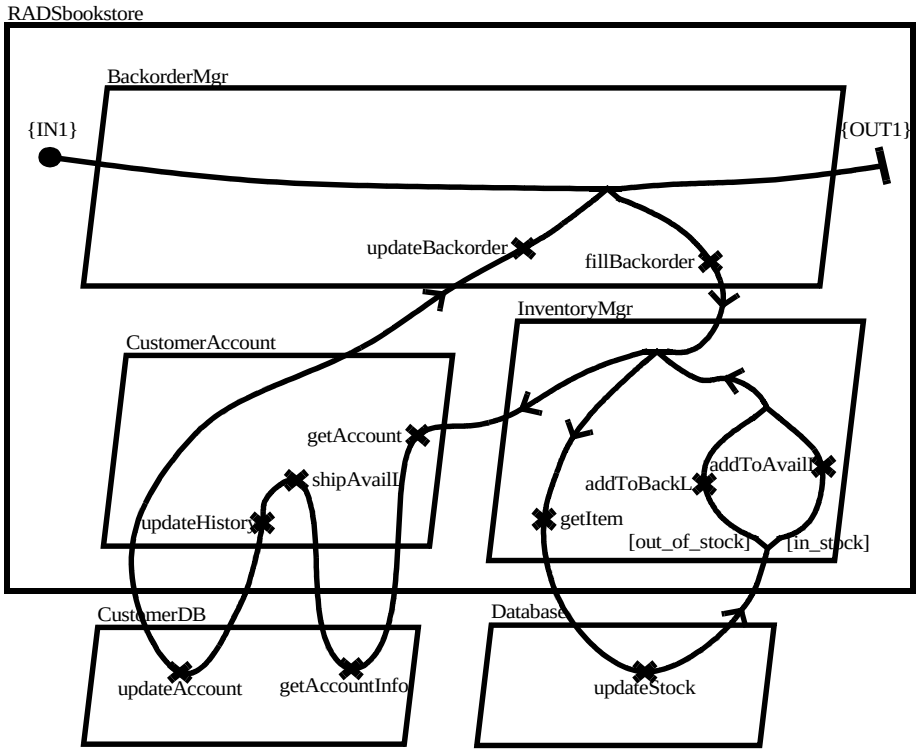

Fig. 4. Plug-in for the fill backorders scenario, in stub serveAdmin.

2.1 UCM Style Constraints For Generating Performance Models

The UCMNav tool has a performance model generation capability, but it can only be used on a UCM which satisfies certain constraints on completeness and style. (To encourage the capture of incomplete scenarios, the default style of a legal UCM is relatively unconstrained.) In particular, a UCM **must** be *properly formed*, meaning:

- it must have at least one point, empty or otherwise, inside each component that is crossed by a path;
- it must have all loops expressed by the explicit loop construct
  - this means avoiding "informal" looping structures formed by using an OR-fork followed by a path looping back to an OR-join at an earlier point on the path (see Figure 5);
- it should not have paths branching from a loop that join with paths that did not branch off the same loop;
- plug-ins must be properly bound to their stubs;
  - that is, each input to a stub must be bound to a start point in the plug-in map, and each output to an end point in the plug-in.

– plug-in maps must not also be identified as root maps.

In addition, UCMs for performance **should** also:

– have paths that fully cover the scenario interactions of the system that are to be modeled;
– be augmented with the necessary performance-related data, as listed below. Some of the parameters have default values.

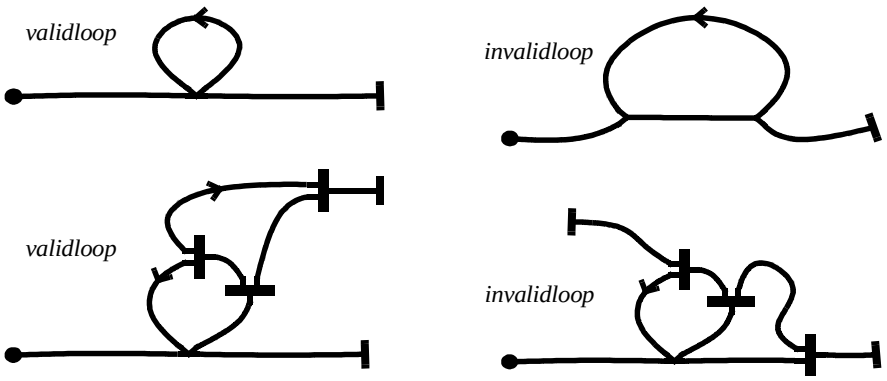

Fig. 5. Valid and invalid loop structures.

UCMNav has the capability of automatically generating LQN models for any well-formed UCM, triggered by selecting the menu *Performance → Generate LQN*. The *scenario to performance transformation algorithm* (SPT, [16]) used to generate LQNs uses a point to point traversal of the UCM paths and infers a calling structure between the components based on the order in which they are traversed by the path. If a path crosses a component but does not have a point inside that component, then the path traversal will not detect the component, and the entire set of calling relationships between components may be misinterpreted. Thus, if the designer does not intend a path to touch a component, it is recommended not to draw the path over the component at all.

The requirement to use the loop construct relieves the SPT algorithm of the need to interpret some very complex constructs which can be created by allowing paths to branch and rejoin in any way at all. In effect it is a "good structure" constraint similar to the use of a *while..do*. Figure 5 shows some valid loop constructs and indicates whether they are interpreted as being properly formed for the performance model transformation.

When used informally, plug-in maps can be associated with a stub without explicitly binding the input and output path segments. However, the SPT algorithm relies on the bindings to traverse the path into the stub, and out again. For a set of plug-ins in a dynamic stub, it treats the input segment as an OR-fork to choose between possible plug-ins. The binding dialog window for a stub

is shown in Figure 6 and is accessed by opening the *Applicable Transformations* pop-up menu for the stub and selecting the *Bind Plugin to Stub* entry. Existing bindings are shown in the *Stub Bindings* text box. To create an entry binding, one needs to select a stub entry and a plug-in map start point before clicking on the *Bind Entries* button. Similarly, to create an exit binding one needs to select a stub exit and a plug-in map stop point before clicking on the *Bind Exits* button. Binding a plug-in map into a stub also requires that the plug-in start and end points have unique names in order to distinguish which point to use in a binding.

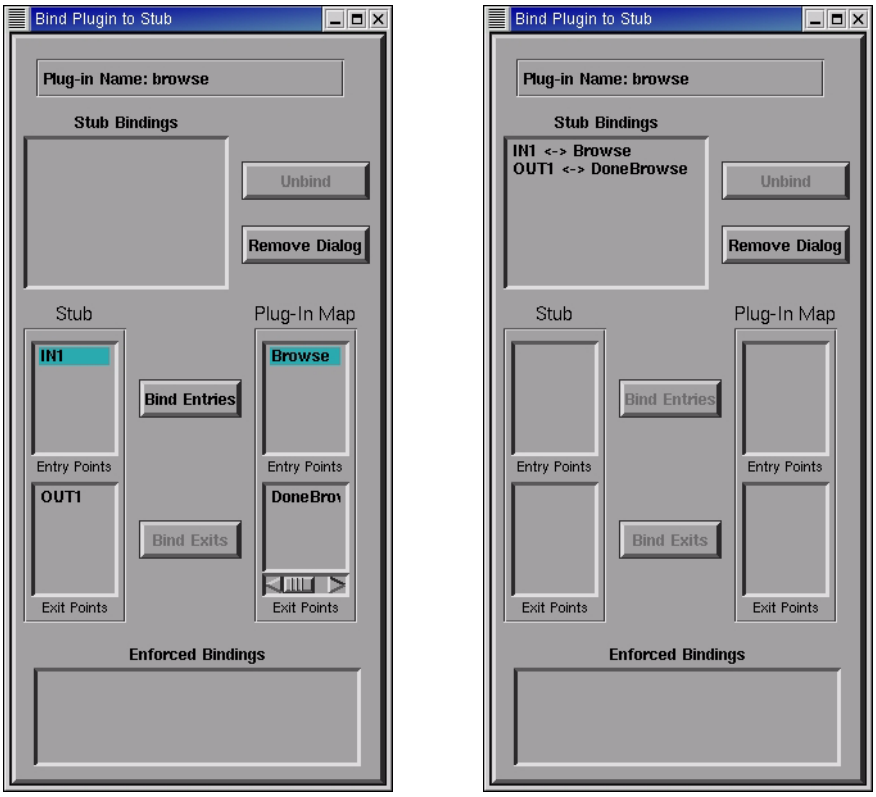

Fig. 6. *Bind Plugin to Stub* dialog box before and after completing the bindings.

A given UCM model can include multiple maps of both *Root* and *Plug-in* type, and a root map can even be used as a plug-in map in any stub. However this will confuse the interpretation by the current SPT algorithm. If the same map is to be used both as a root map and a plug-in map, then it is best to avoid confusion by exporting the map and then importing back as a plug-in for the

desired stub. There will be two copies of the map, but each copy will have a clearly defined type.

There is also a more subtle issue regarding the interactions between components when generating an LQN model from a UCM. Calling relationships between components are determined by the order in which they are traversed by a path. As a path crosses new components, it is assumed that calls are being made from component to component. Whenever a path returns to a component it has previously crossed, it is assumed that a reply to a call is being received. The SPT algorithm attempts to maximize the synchronous interpretation of interactions between components, but this interpretation requires that the path return to components that are supposed to make synchronous calls. Thus, the performance model is based on the more restricted interpretation that inter-component communication is determined solely by the order in which components are crossed along a path. This is not necessarily an interpretation that is assumed in other types of UCM usage.

2.2 UCM Performance-Related Parameters

The generation of performance models requires that the UCM be augmented with adequate performance-related data to enable meaningful analysis. The following steps must be performed in order:

- create processors and disks or other devices;
- assign UCM components to processors;
- assign service demands to UCM responsibilities.

The following steps should also be done but their order does not matter:

- define arrival characteristics for start points;
- assign probabilities/weights to branches on OR-forks;
- assign probabilities/weights to plug-ins for dynamic stubs;
- assign loop repetition counts to loops.

Time values used for parameters do not show units and can be interpreted to be of whatever time unit the designer chooses (typically milliseconds or seconds). However, it is important that the time unit used be consistent throughout the entire UCM.

Processors and devices need to be created in UCMNav prior to LQN generation. To create a device, one should open the *Device Characteristics* dialog box (*Performance* menu) as shown in Figure 7 (left).

The type of device to be edited can be selected from the Device Type dropdown menu. Devices can be:

- **Processor**: processing device that acts as a host to components
- **Disk**: disk device
- **DSP**: digital signal processor
- **Service**: any external service

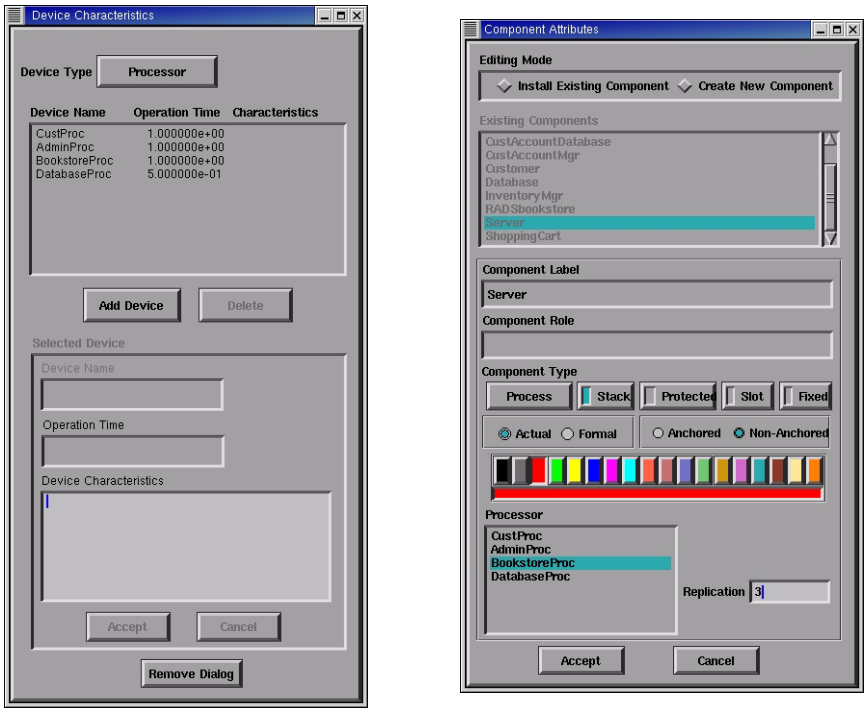

Fig. 7. *Device Characteristics* and *Component Attributes* dialog boxes.

Each device must be specified to have an associated operation time that is a relative scale factor for its processing speed. A larger operation time indicates a slower device. The SPT algorithm also creates a default infinite processor, i.e. a multiprocessor with an unlimited number of replicas, with an operation time of 1. All the components that do not have a processor assigned in the UCM are generated as LQN tasks assigned to this infinite processor.

Components in the UCM should have a host processor specified. Figure 7 (right) shows the *Component Attributes* dialog box which is used to configure components. The *Editing Mode* determines whether the component is a stand-alone component (*Create New Component*) or a reference to an already existing component (*Install Existing Component*). The *Component Label* is the name of the component - if the component is a reference of an existing component then changing the label will change that name of the component and all its other references. The *Component Type* drop-down menu determines whether the component is a Team, Object, Process, ISR, Pool, Agent, or Other. However, all component types are mapped to LQN tasks. In the case of multiple references to the same component, only one LQN task is generated for the component and visits to any of the references are assumed to generate messages to that one task. The *Stack* checkbox indicates whether or not the component is replicated, and

it activates the *Replication* field which specifies the number of copies. Multiple components are generated as multi-threaded tasks in the LQN. If the *Replication* field is set to * then the corresponding LQN task is infinitely-threaded. The *Processor* text field shows all the processors defined in the UCM, with the highlighted processor being the host for the component. If no processor is selected for the component then the LQN generated will make the corresponding task run on the infinite processor.

Responsibilities can make specific demands on the various services defined. Figure 8 shows the *Service Requests by Responsibility* dialog box (invoked from the *Edit Responsibility* dialog box by pressing the *Service Requests* button). The *Service Type* column in the top text field shows the services that are called and the *Quantity* column indicates the number of requests made. To add a new service request, one selects the *Service Category* to be used from the drop-down box and then selects the actual device in the *Service Name* field. If the service type is processor, then only the name of the host processor for the component that contains the responsibility will be shown, but it must still be explicitly selected by highlighting it in the *Service Name* field. The request quantity must also be entered in the *Request Quantity* field.

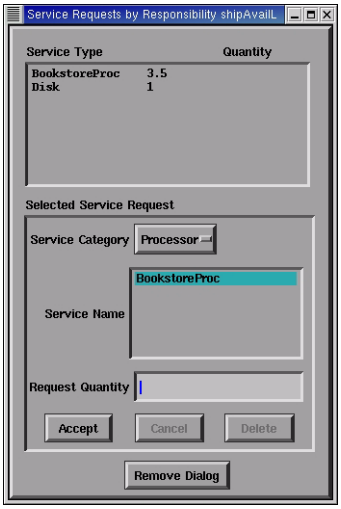

Fig. 8. *Service Requests by Responsibility* dialog box.

Deciding upon the right values for service demands can be a delicate undertaking, and there are different strategies used to find them. A value can come from a known value or benchmark, a performance budget for the maximum/average time that may be taken by the responsibility [20], or maybe just an estimate by the designer that can be fine-tuned later [21]. Any responsibility that does not have service demands specified is generated as an LQN activity

with a default demand of 1. This default means that even an incompletely specified UCM generates an LQN that can be solved, although the solution is only a very rough approximation.

The arrival process for each start point needs to be specified using the *Start Point Workload* dialog box, shown in Figure 9, which is accessed from each start point's transformation menu. The arrivals can be specified as either open streams with no limit on the job population or as closed streams with a finite job population. In order to be picked up by the SPT algorithm, the distribution of the interarrival time for open arrival streams and the think time for closed streams should be specified as either exponential with a mean, deterministic with a mean, or uniform with a value. Erlang distributions with a high and low value or expert distribution with a string descriptor are not currently handled. Start points with closed arrivals imply a return path for each job and as such should be connected to an end point along the same path or be contained in the same component as an end point. In the absence of such an end point, the SPT algorithm will generate the required return path from the first end point encountered as the path is traversed. By default, start points with no specified arrival process are generated as having open arrivals with an exponential distribution with a mean of 1.

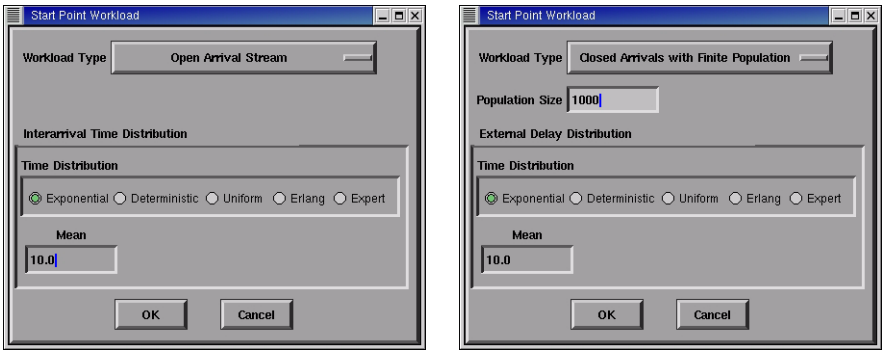

Fig. 9. *Start Point Workload* dialog boxes for open and closed arrivals.

Probabilities for OR-fork branches should be specified using the *Specification of OR-Fork* dialog box accessed from each OR-fork's transformation menu. The branches are labelled as *BR1*, *BR2*, and so on, and those labels are shown on the UCM whenever the window is open. Branch probabilities can be specified as decimal fractions or as relative weights for each branch, and they are normalized during the LQN generation process. Similarly, plug-ins for dynamic stubs are also traversed like branches between a virtual OR-fork at each stub entry point and a virtual OR-join at each stub exit point. Each plug-in should also have a specified selection probability, which are specified in the *Choose Plugin* dialog box of each dynamic stub. Any missing branch or plug-in selection probabilities

are given a relative weight of 1. If all the branch or plug-in selection probabilities for an OR-fork or dynamic stub are missing, then each branch or plug-in will be generated with an equal probability in the LQN.

The number of loop repetitions for each loop construct needs to be specified as a loop count in the *Edit Loop Characteristics* dialog box. Loops with missing loop counts are generated with a default value of 1.

2.3 Verification of Parameter Completion

UCMNav provides the capability of verifying whether all the performance-related parameters have been entered for all relevant elements in a UCM. Selecting the *Performance* menu → *Verify Annotation Completeness* entry highlights in red all the UCM elements that have missing parameters. Selecting the *Performance* menu → *Remove Annotation Highlighting* entry removes the highlighting.

3 LQN Performance Model

Figure 10 shows the LQN model generated from the annotated RADSbookstore UCMs. The details of entries and activities, and the workload parameters are suppressed for presentation purposes. The multiple interaction arrows show the numbers of different kinds of access made from one task to another. For example, tracing the paths in Figures 1, 2 and 3 leads us from the set of Customer tasks, to the RADSbookstore task representing the system as a whole (a task without functions) to the Server. The Checkout scenario then calls the ShoppingCart, which manages the checkout. The ShoppingCart calls the InventoryMgr (twice), the BackorderMgr, and the CustomerAccount. The InventoryMgr in turn calls the Database, and the CustomerAccount calls the CustomerDB.

The forwarding path, shown by the dashed arrow from the InventoryMgr to the CustomerAccount, is part of the Administrator's backorder scenario, as shown by the Administrator path in Figure 1 and the fill backorders plug-in in Figure 4. The forwarding occurs when the InventoryMgr is updated, initiates the shipping, and then passes on the information to the CustomerAccount for the accounting and billing. The CustomerAccount then forwards the reply back to the RADSbookstore.

3.1 Solving LQNs

LQN models can be solved using tools such as the *LQNS analytic solver* and the *LQSim* simulator [12]. Both solvers accept the same input LQN file format, which is automatically generated from UCMNav, and generate similar output files with the following sections:

- **General solver statistics**: elapsed time, system time, blocks, simulation length for LQSim, etc.
- **Echo of the specified service demands**: specified service demands for every entry and activity.

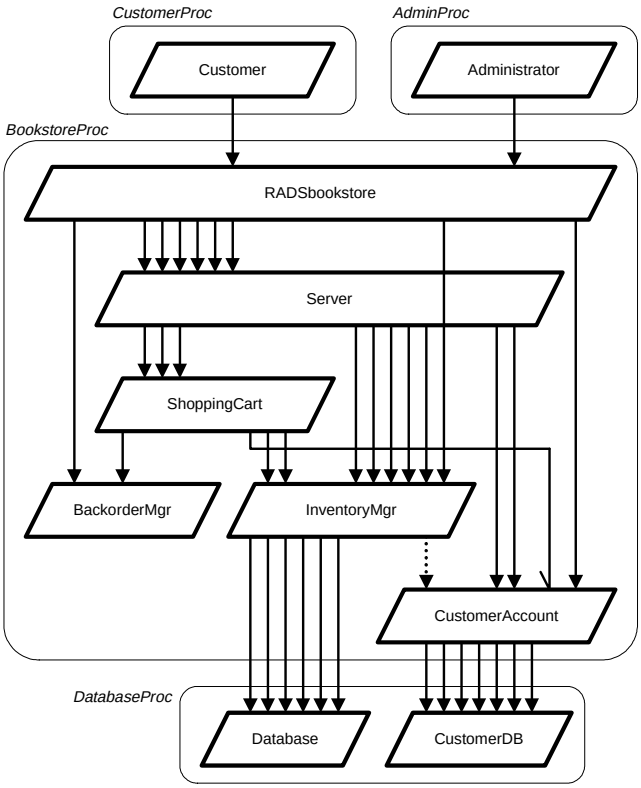

Fig. 10. LQN model of the RADSbookstore.

- **Measured quantities**: measured service demands, number of blocking and non-blocking calls, call delays, synchronization delays.
- **Service times**: solved service times for every entry and activity, includes confidence intervals when simulated with multiple blocks.
- **Service time variances**: variances and squared coefficients of variance for the service times calculated in the section above.
- **Throughputs and utilizations**: solved throughputs and utilizations for every entry and activity, includes confidence intervals when simulated with multiple blocks.
- **Utilizations and waiting times for devices**: solved hardware utilizations and waiting times by every entry.

LQNS is faster but more limited than LQSim in the models it can handle. Some LQN models do not have a stable analytical solution and therefore they need to be solved using simulations with LQSim.

The layered nature of LQNs and the fact that the solvers provide results for both software and hardware resources means this approach is suitable for detecting both software and hardware performance bottlenecks.

Figure 11 shows the response time and throughput results for the RADS-Bookstore. The bookstore was solved as a closed system with a variable client population and a single administrator. The results show that this system becomes saturated with about 50 clients.

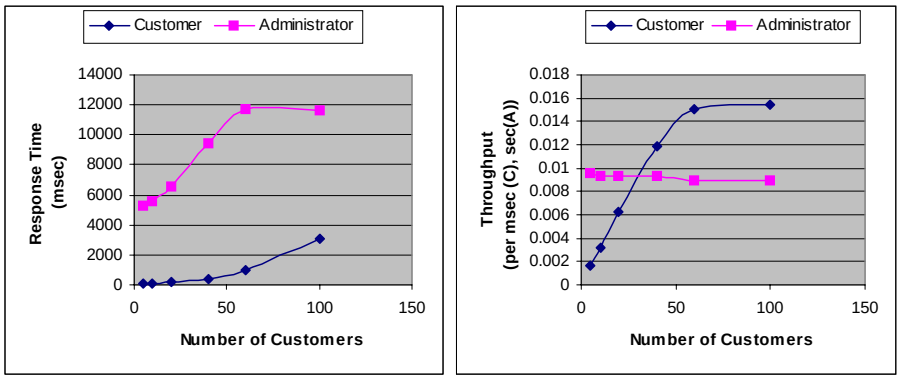

Fig. 11. Response time and throughput simulation results for the RADSbookstore.

4 Performance Analysis

The LQN performance model can be used as a basis for exploring the performance solution space of the system. The kinds of analysis that can be performed include, but are not limited to, the following:

- **Sensitivity analysis**: how important are different values for certain parameters to the solution. This is useful to estimate the performance impact of the uncertainty in estimated values.
- **Scalability analysis**: how well does the system cope with more users, how does the system throughput, response times and utilization behave as the workload is increased.
- **Concurrency analysis**: how does the system respond to changes in the number of threads or replicas for certain tasks.
- **Deployment/configuration analysis**: how does the system respond to different deployment configurations, what are the effects of bandwidth limitations, network delays, or reallocating the system hardware.

4.1 Example of Concurrency Analysis for the RADSbookstore

The results for the base case of the RADSbookstore indicate that Customers are queueing up at the Server task, which has 5 threads. This suggests that increasing the number of Server threads would remove a software bottleneck. Figure 12

shows the results of increasing the number of Server threads to 50. Instead of improving the overall performance of the system, increasing the number of Server threads actually degrades it. The response time and throughput for Customers remains essentially unchanged, but the response time for the Administrator rises from 12 seconds with 100 Customers to 60 seconds with 100 Customers. Thus the increase in threads consumes resources and makes the Administrator response much worse, giving absolutely no benefits.

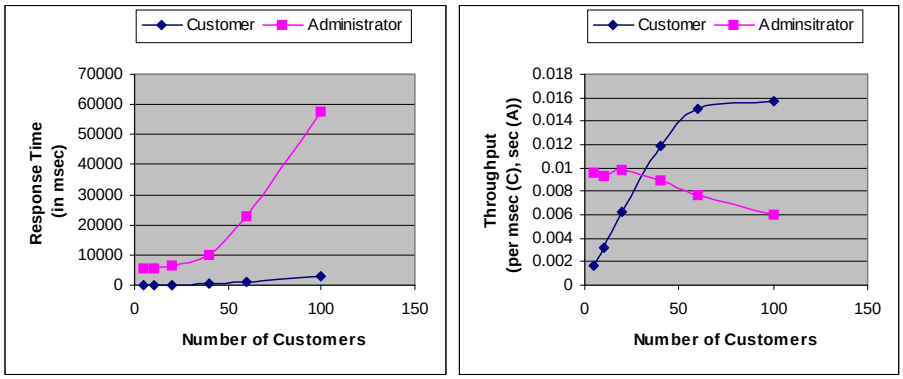

Fig. 12. Response time and throughput simulation results for the RADSbookstore with 50 server threads.

A deeper analysis of the performance results for the base case of the RADSbookstore shows that within the system the Inventory Manager task is 100i% saturated, mostly due to waiting for the Database which is 80% busy. Thus the Customers queueing at the Server are actually held up by the InventoryMgr and the Database. Therefore the limited number of Server threads provides a kind of admission control, keeping congestion out of the system without actually slowing it down. This indicates that a better way of improving the performance of the system is to improve the InventoryMgr and the Database.

An examination of the way in which Customers interact with the RADSbookstore shows that they mostly browse the catalogue of books, and do not really need full database capability and concurrency control. The catalogue is rarely updated, and could be separated out as a read-only database without complex concurrency control. Indeed, creating a separate Catalogue server inside the RADSbookstore to replace the catalogue accesses to the InventoryMgr and the Database significantly improves the system performance, as shown in Figure 13.

5 Conclusions

Software Performance Engineering from requirements descriptions is very challenging and demanding. However, several developments such as those presented

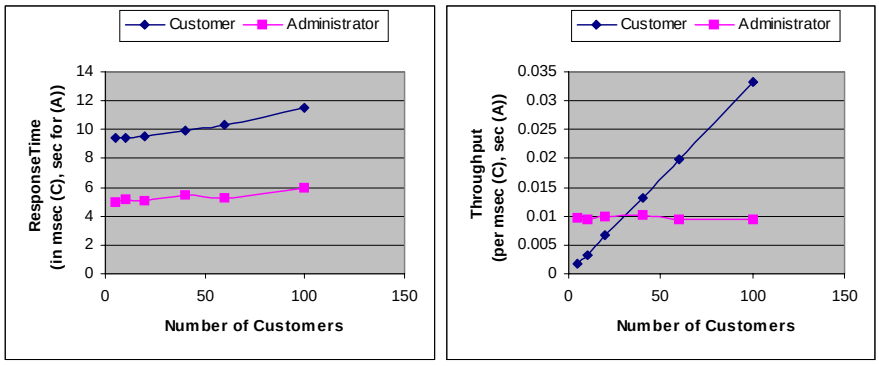

Fig. 13. Response time and throughput simulation results for the RADSbookstore with a Catalogue server.

in this paper indicate that automated generation of performance models early in the development process is not only possible but also useful.

This paper presented systematically the steps involved in the construction of UCM models annotated with performance information, as supported in UCM-Nav. Once all the required information is provided, a situation that can be verified by the tool, UCMNav can automatically generate a performance model suitable for various kinds of performance analysis. Traceability between the two models is preserved through the use of common names. Default values are provided by the SPT algorithm for several categories of parameters if they are not specified by the designer. Analysis reports for the resulting LQN model are produced automatically with tools such as the LQNS analytic solver and the LQSim simulator. The e-commerce example illustrated typical situations of parameter provision and analysis results. Many variants of an UCM model (e.g. with a different underlying architecture, or different values for the performance parameters) can quickly be generated, evaluated, and compared.

The two languages selected here (UCM and LQN) proved to be a good match for performance engineering based on requirement scenarios. Queueing networks are known to be abstract, like UCMs, but they are usually difficult to obtain from behavioural descriptions [4]. With UCMNav, they are generated automatically from the requirements specification, at the cost of some stylistic constraints. The addition of performance annotations is not really costly because such information is typically needed by any performance model.

Many of the parameter annotations for UML models have been inspired from existing work on UCMs and are now part of the performance profile for UML [15]. Recent work also suggests that the SPT algorithm used in UCMNav can likely be applied to scenario specifications in other languages, including MSC and UML sequence, collaboration, and activity diagrams [17]. This will be investigated, together with transformations to variants of queueing networks other than LQNs. We also plan to study how best to use UCM scenario definitions in a performance

engineering context, as well as the verification of soft real-time requirements (captured with pairs of *timestamps* on UCM paths [19]) through LQN analysis.

Acknowledgments. This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), through its programs of Strategic and Collaborative Research Grants, and by Nortel Networks. We are thankful to Don Cameron and Os Monkewich for their collaboration.

References

1. Amyot, D. and Eberlein, E. (2003) An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. To appear in *Telecommunication Systems Journal*.
2. Balsamo, S. and Simeoni, M. (2001) On transforming UML models into performance models. *Workshop on Transformations in the Unified Modeling Language*, Genova, Italy, April 2001.
3. Buhr, R.J.A. (1998) Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*. Vol. 24, No. 12, December 1998, 1131-1155.
4. Cortellessa, V., Di Marco, A., and Inverardi, P. (2003) *Comparing Performance Models from a Software Designer Perspective*, TR SAH/042, Universita di L'Aquila, Italy, `http://sahara.di.univaq.it/tech.php?id_tech=42`
5. Dulz, W., Gruhl, S., Lambert, L., and Sollner, M. (1999) Early performance prediction of SDL/MSC specified systems by automated synthetic code generation. *Proc. of the Ninth SDL Forum (SDL'99)*, Montréal, Canada. Elsevier.
6. ITU-T (2000) *Recommendation Z.100, Specification and Description Language (SDL)*. Geneva.
7. ITU-T (2001) *Recommendation Z. 120, Message Sequence Chart (MSC)*. Geneva.
8. ITU-T (2003), *Recommendation Z.150, User Requirements Notation (URN) - Language Requirements and Framework*. Geneva. `http://www.UseCaseMaps.org/urn/`
9. ITU-T, URN Focus Group (2002), *Draft Rec. Z.152 - UCM: Use Case Map Notation (UCM)*. Geneva.
10. P. Kähkipuro (2001) UML-Based Performance Modeling Framework for Component-Based Distributed Systems. *Performance Engineering*, LNCS 2047, Springer, pp. 167-184.
11. Kerber, L. (2001) Scenario-based Performance Evaluation of SDL/MSC-Specified Systems. *Performance Engineering*, LNCS 2047, Springer, pp. 185-201.
12. Layered Queues for Software and Hardware Performance Modeling: Resource Page `http://www.layeredqueues.org/`
13. Miga, A. (1998) *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada.
14. OMG (2003) *Unified Modeling Language Specification, Version 1.5*. March 2003. `http://www.omg.org`
15. OMG (2001), UML Profile for Scheduling, Performance and Time. Document ad/2001-06-14, `http://www.omg.org/cgi-bin/doc?ad/2001-06-14`, June 2001.
16. Petriu, D. and Woodside, C.M. (2002) Software Performance Models from System Scenarios in Use Case Maps. *12th Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation*, London, U.K., April. `http://www.UseCaseMaps.org/pub/tools02.pdf`

17. Petriu, D. and Woodside, C.M. (2002) Analysing Software Requirements Specifications for Performance. *Third International Workshop on Software and Performance (WOSP 2002)*, Rome, Italy.
18. Pooley, R. (2000) Software Engineering and Performance: a Roadmap. In: *The Future of Software Engineering, ICSE'2000*, Limerick, Ireland, pp. 189-200.
19. Scratchley, W.C. (2000) *Evaluation and Diagnosis of Concurrency Architectures*. Ph.D. thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada.
20. Siddiqui, K. and Woodside, C.M. (2002) Performance-Aware Software Development (PASD) Using Resource Demand Budgets. *Third International Workshop on Software and Performance (WOSP 2002)*, Rome, Italy.
21. Smith, C.U., Williams, L.G. (2001) *Performance Solutions*. Addison-Wesley.
22. Woodside, C.M., Hrischuk, C., Selic, B., and Bayarov, S. (2001) Automated Performance Modeling of Software Generated by a Design Environment. *Performance Evaluation*, vol. 45, pp 107-124, July 2001.
23. Woodside, C.M. (2002) *Tutorial Introduction to Layered Performance Modeling of Software Performance*, on-line, May 2002.
`http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialf.pdf`