Report SCE-02-02, Feb. 6, 2002. Dept. of Systems and Computer Engineering, Carleton University, Ottawa K1S 5B6, Canada. Accepted for publication in Proc. Performance TOOLS 2002, London, April 2002.

Software Performance Models from System Scenarios in Use Case Maps

Dorin Petriu, Murray Woodside
Dept. of Systems and Computer Engineering
Carleton University, Ottawa K1S 5B6, Canada.
{dorin, cmw}@sce.carleton.ca

Abstract. Software performance concerns begin at the very outset of a new project. The first definition of a software system may be in the form of Use Cases, which may be elaborated as scenarios: this work creates performance models from scenarios. The Use Case Maps notation captures the causal flow of intended execution in terms of responsibilities, which may be allocated to components, and which are annotated with expected resource demands. The SPT algorithm was developed to transform scenario models into performance models. The UCM2LQN tool implements SPT and converts UCM scenario models to layered queueing performance models, allowing rapid evaluation of an evolving scenario definition. The same reasoning can be applied to other scenario models such as Message Sequence Charts, UML Activity Graphs (or Collaboration Diagrams, or Sequence Diagrams), but UCMs are particularly powerful, in that they can combine interacting scenarios and show scenario interactions. Thus a solution for UCMs can be applied to multiple scenarios defined with other notations.

1 Introduction

Software performance analysis often begins from scenario definitions, which describe the system behaviour during a response. For example Smith's Execution Graphs [28][29] can be used to capture a performance analyst's version of system scenarios, and can be used in the earliest stages of planning. Kahkipuro [16] used scenarios expressed as UML collaborations, and this approach has been extended in a proposed UML performance profile [26]. Other authors have used Petri nets to capture the scenarios.

Because software designers are (usually) not also performance analysts, the performance-specific scenario models (such as Execution Graphs) create a conceptual gap between the design and the performance analysis. To avoid this gap, it would be better to use a *software engineering* scenario notation. This work is based on a well-developed scenario language called Use Case Maps (UCMs) [7], but it is extensible to other notations such as UML. UCMs expand Use Cases into scenarios described as causal sequences of responsibilities, either unbound or associated with components. UCMs can be used to reason about architecture and to develop an architecture within a structural notation, possibly based on the UML, such as is described by Hofmeister, Nord and Soni [15].

To provide continuous re-evaluation during the evolution of an architecture there must be automation. This paper describes the UCM2LQN converter, a tool which

automates the conversion of UCM scenario models into Layered Queueing Network (LQN) performance models. It is integrated into the UCM Navigator, which is an editor and repository tool for UCMs, so that the UCMNav can be used as a front-end editor for design models which are also performance models. The concepts of the converter can in principle be extended to other scenario modeling tools such as Stories [8] or UML [5] which provides Activity Diagrams, Collaboration Diagrams and Sequence Diagrams. Conversion of Collaboration models into layered queues has previously been described by Kahkipuro [16]. Eventually the ideas of UCM2LQN may be applied to annotated UML models using a standard profile for performance annotations [26].

The performance model notation used here is Layered Queueing, because this form of model captures logical resource effects and provides good traceability between the performance measures and the emerging software architecture. It incorporates the software components as servers, and logical resources (such as buffers, locks and threads), as well as the hardware resources. Essentially, layered queueing is a canonical form for simultaneous resource queues, that are common in software resources. Simpler queueing models could be used instead, but they would model fewer features of the resource architecture, as described in [32]. Petri net models capture these features very well, as described by Pooley [23], but sometimes have problems with larger system scales.

The difficult problem solved by UCM2LQN applies to any scenario notation which binds actions to software components, and not just to Use Case Maps. The problem is in interpreting paths as interactions between software objects which may have resource attributes. Interactions which imply waiting for logical resources (blocking) have performance effects and are captured by analyzing the entire path. A second kind of difficulty comes from interactions between scenarios, either in competition or in collaboration.

There has been considerable effort expended on methods for Software Performance Engineering (SPE), and an overview can be obtained from the proceedings of the two international Workshops on Software and Performance (WOSP'98 [33] and WOSP2000 [34]). Despite this effort, SPE has proven to be more appealing in concept than in practice. The effort needed to cross into the realm of performance analysis, in the course of any design project, is too high, and the concepts are alien to the developer. Using automated model-building reduces the need for special performance expertise. There is still a requirement to specify the appropriate performance data - such as service demands by responsibilities, arrival rates at start points, branching probabilities, loop repetitions, and device speed factors - as annotations in the UCM in order to get meaningful results. These values can be obtained from known workloads or they can be approximated by using a budgeting approach and supplying values based on an estimate of much time operations have to complete [27][30]. The results may validate the performance aspects of the design by confirming the budgets, or may identify problems such as bottlenecks, and this work can be a partnership between the designer and the performance expert.

This research pins its hopes on embedding most of the description into the software definition as it emerges, and on tracking this definition through its stages into

code. It is important to begin early, and the automatic converter described here captures the first step in design.

2 Models for Scenarios and Performance

2.1 Use Case Maps

UCM notation was invented by Buhr and his co-workers [6][7] to capture designer intentions while reasoning about concurrency and partitioning of a system, in the earliest stages of design. It was derived by watching designers discussing and massaging ideas into architectures, and is intended to be intuitive, and high-level. Details can be represented, but are not the purpose. Compared to the Unified Modeling Language (UML), UCMs fit in between Use Cases and UML behavioural diagrams. In UML Class Diagrams are used to describe how a system is constructed, but do not describe how it works; this task is taken up by UCM's. Collaboration Diagrams do provide a high-level description of how the system works, but only one scenario at a time [3].

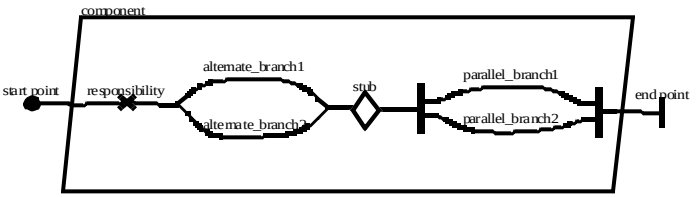


Figure 1: Example of the UCM notation.

A Use Case Map is a collection of elements that describe one or more scenarios unfolding throughout a system [7] [6]. The basic elements of the notation are shown in Figure 1. A scenario is represented by a path, shown as a line from a start point (a filled circle) to an end point (a bar), and traversed by a token from start to end. Paths can be overlaid on components which represent functional or logical entities, which may represent hardware or software resources. Responsibilities, denoted with an X-shaped mark on the path, represent functions to be accomplished. The performance modeling assumes that the computational workload is associated with responsibilities, or is overhead implied by crossings between components. Responsibilities are annotated by service demands (number of CPU or disk operations, or calls to other services) and data store operations.

A path can be traversed by many tokens, and several tokens may occupy a single path at once. The workload of a path is indicated by annotations to its start point (closed or open arrivals, arrival rates and external delays). A path can be refined hierarchically by adding stubs, which represent separately specified maps called plug-ins. There may also be several alternative plug-ins for any stub.

Paths have the usual behaviour constructs of OR fork/joins (representing alternative paths), AND fork/joins (representing parallel paths) and loops. OR forks and loops are annotated by choice probabilities and mean loop counts. AND and OR forks do not have to be nested, that is they do not have to join later. This is realistic for software design, but creates problems for model creation, as the structured workload graph reduction used by Smith ([28], chapter 4) does not always apply.

The UCM Navigator (UCMNav) [17] was developed by Miga as an editor and repository manager, and has been used by our industrial associates to create large, industry-scale scenario specifications. It supports

- drawing and editing UCMs, including multiple scenarios, and storing in an XML format.
- annotations for deployment on system devices and for performance, as well as comments and pseudo code,
- specifying delay requirements along a path,
- generating Message Sequence Charts (MSC) as well as performance models.

The UCM2LQN converter (to be described below) is implemented as an add-on to UCMNav, and generates a file in the LQN language which can then be used (outside the UCMNav) to compute performance measures using either a simulator called LQSim, or an analytic solver LQNS.

2.2 Layered Queueing Networks

Layered Queueing Networks (LQN) model contention for both software and hardware resources, based on requests for services. Entities in the role of clients make service requests and queue at the server. In ordinary queueing networks there is one layer of servers; in LQN, servers may make requests to other servers, with any number of layers [24][48]. An LQN can thus model the performance impact of the software structure and interactions, and be used to detect software bottlenecks as well as hardware performance bottlenecks [19]. There have been many applications [22][31][51].

In an LQN the software resources are called tasks, (representing a software process with its own thread of execution, or some other resource such as a buffer) and the hardware resources are called devices (typical devices are CPUs and disks). Tasks can have priority on their CPU. The workload of a LQN is driven by arrival streams of external requests, and by tasks which cycle and make requests, called *reference tasks*.

An LQN can be represented by a graph with nodes for tasks and devices, and arrows for service requests (labelled by the mean number of messages sent). There are two types of arc to represent asynchronous messages, with no reply, and synchronous messages which block the sender until there is a reply (synchronous messages are also called task calls; the model was created originally for Ada software). Tasks receive either kind of request message at designated interface points called *entries*. A task has a different entry for every kind of service it provides; an entry also represents a class of service. Internally an entry has service demands defined by sequences of smaller computational blocks called *activities*, which are related in sequence, loop, parallel (AND fork/joins) and alternative (OR fork/joins) configurations. Activities have processor service demands and generate calls to entries in other tasks.

A third type of interaction called forwarding is a combination of synchronous and

asynchronous behaviour. The sending client task makes a synchronous call and blocks until it receives a reply. The receiving task partially processes the call and then forwards it to another server which becomes responsible for sending a reply to the blocked client task; it can be forwarded with a probability, and any number of times. The intermediate server task can begin a new operation after forwarding the call.

Models are created in a textual language which can be edited as text or with a simple graphical editor, and can be solved either by simulation, or by analytic approximations by the solver LQNS. LQNS is based on [31] and the Method of Layers [24], with a number of additional approximations [10][11][12]. The approximations have limitations in dealing with priorities (poor accuracy) and with AND-joins that do not have an AND-fork in the same task, so simulation is often useful.

The interactions in LQN's can be understood more clearly using UCMs to show the sequences of events. Figure 2 has a series of UCMs describing the interactions which must be detected when building an LQN model:

(a) a basic synchronous interaction between two tasks taskA and taskB has a path launched by an activity (which is an inferred overhead activity for communications); the reply returns the path to the same activity. The interpretation of this message is the same if the path goes on from taskB to other tasks, returning to taskB before returning to taskA.

(b) two activities in taskA send messages, one to taskB and a later one to taskC.

(c) taskA sends an asynchronous message to taskB. The interpretation of the message is the same if the path goes on from there, but never returns to taskA. The LQN notation is an open arrowhead, here shown with one side only.

(d) taskA sends a message to taskB which is forwarded to taskC, before returning directly to taskA. The forwarding path can include any number of intermediate tasks; the assumption is that taskA (or a thread of the task) waits blocked for the return, unless there is a fork in taskA before sending the request. The LQN notation for the forwarding steps is a solid arrow for the original blocking call, and dashed arrows for the other, non-blocking messages.

Figure 2 also shows the LQN notations for forks and joins and for loops.

3 Extracting a Layered Performance Model

The novel contribution in this work is finding disguised synchronous and forwarding interactions. These identify potential software blocking which may have significant performance implications. Compared to many scenario analyses (such as used in [28]), which only determine device demands by class, the layered model also retains the component context of each demand. Other models which retain the software context of demand, e.g. Kahkipuro's AQN [16], require that blocking interactions be explicitly identified.

3.1 Correspondences between UCMs and LQN's

There are some quite close correspondences between some of the scenario enti-

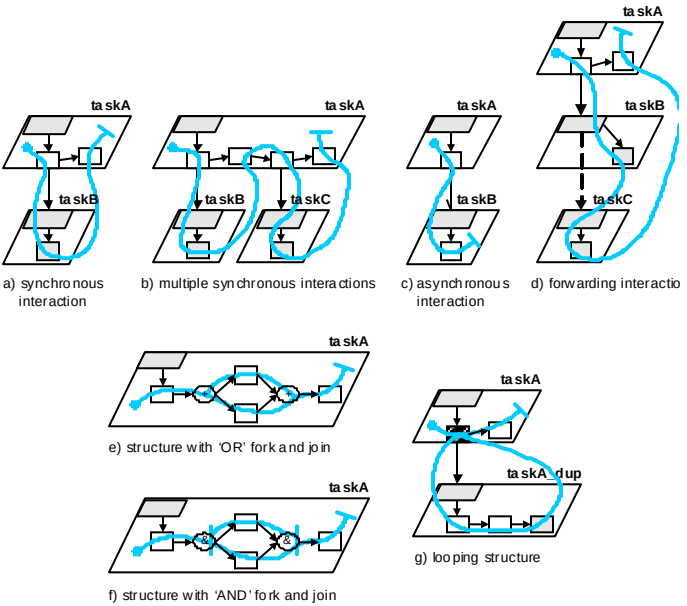


Figure 2: Corresponding interactions and structures in UCM and LQN.

ties, and LQN model entities that can represent them.

UCM Construct	LQN Construct
start point	reference task
responsibility	activity
AND/OR forks and joins	LQN AND/OR forks and joins
component	task
device	device
service	entry in a task (with a dedicated processor)

Table 1: Corresponding UCM and LQN constructs.

Considering these in order,

- A reference task can serve either as an open workload generator inserting asynchronous requests, or a closed workload generator, in which case it has a multiplicity equal to the population, and each task makes synchronous requests (and waits for the response).
- A UCM responsibility can represent an arbitrarily complex set of operations, however here we are restricting its significance to a sequential operation, which can make calls to services. A complex operation can be captured in many cases by these calls, which are mapped to servers and service requests in the LQN.
- A component may represent an operating system process, or an object or module of some kind. An LQN task has a primary meaning as a separate operating system process, but it also represent an object or module executing in the context of some task. A synchronous call to the module is effectively sequential, because of the blocking of the main task, so it is equivalent to including the module inside the main task... modeling a module in this way exposes its contributions to performance.
- A "service" in UCM is an annotation representing a service used by the software but outside the scope of the UCM, such as a file service or a database service. Ultimately a submodel for this subsystem will be added to the model, but as a placeholder, a task with a dedicated processor is inserted to take the calls for the service.

3.2 Correspondences of Path Structure in LQN

Within a component the scenario expression of path structure translates directly to the LQN activity sequence notation, with the usual constructs of alternate and parallel branching (and joining), as well as looping. The LQN notation supports the same constructs. Figure 2(e) shows a UCM interpretation of an LQN task with an OR fork and join (in the LQN model the OR is indicated by a '+' connector). Figure 2(f) shows an AND fork and join (the LQN model uses '&' in the connector). A UCM loop point is indicated by an OR join followed immediately by an OR fork; the LQN notation has a loop traversal count. A complex loop body can be represented in the LQN by a pseudo-task which is "called" by the loop controller and executes the activities of the body, as indicated in Figure 2(g).

3.2.1 Fork and Join in Separate Components

In a scenario, paths may fork in one component and join in another. Both UCMs and LQN's support this feature; the path is conveyed from the first component to the second by asynchronous or forwarding interactions. Simulation evaluation in our tools assumes that any token on the joining paths is a candidate, but applications may require that only tokens that are siblings from the fork should be allowed to join. If the scenario is such that tokens cannot pass each other this is no problem, otherwise it is a headache both to model and (indeed) to implement.

3.3 Performance Annotations in UCMs

The performance annotations on UCM elements were mentioned in the description of the UCM notation above, but it is worth summarizing them more formally since they provide the parameters and some of the elements of the performance model. Table 2 shows the annotations and their default values.

UCM Element	Performance Annotation	Default
responsibility	number of calls	1.0 calls
component	associated devices	one infinite processor
device	speed-up factor	1.0
OR fork	probability of each branch (as a weight)	equal probability for each branch
loop	number of loop iterations	1.0 iterations
start point	open system arrival rate and distribution	1 arrival/sec, with deterministic delay
	OR closed system population and delay	10 jobs with deterministic delay of 1 sec.

Table 2: UCM constructs, the necessary performance data needed to create meaningful LQNs, UCMNav support for entering the data, and default values used if the data is not specified.

4 Scenario to Performance Model Transformation Algorithm

The algorithm for scenario to performance transformation (SPT) must do the following:

- identify when the path crosses component boundaries
- determine the type of messages sent or received when crossing component boundaries
- capture the path structure and the correct sequence of path elements
    - create the LQN objects that correspond directly to UCM elements
    - handle forks, joins, and loops

The UCM is transformed into an LQN on a path by path basis. Each start point is assumed to begin an independent path, and as such is assigned to its own reference task. Reference tasks act as the work generators for the LQN model. Each reference tasks is assigned arrival rates and distributions as specified by the start points in the UCM. Similarly, LQN activities are assigned workload demands as specified in the corresponding UCM responsibility and OR branches are assigned probabilities set in the UCM. If any performance data is missing from the UCM, default values are assigned as noted in Table 2.

The SPT algorithm follows a UCM path from its start point. Each element along the path is checked for its enclosing component, and if the enclosing component has changed then a boundary has been crossed. Each boundary crossing corresponds to a message between components. The message may be a synchronous call, a reply, an asynchronous call, or a forwarding; to resolve its role in an interaction requires examining a portion of the history of the path. This is called *resolving* the interaction. Therefore there is a need to keep track of all messages that have been discovered, but not yet resolved.

4.1 Call and Reply Stack (CRS)

The Call and Reply Stack (CRS) is the mechanism that stores the unresolved message history as the path is traversed. Whenever a component boundary is crossed, the message event is pushed onto the stack and then the pattern of messages in the stack is analyzed to see if they satisfy one of the interaction patterns illustrated in Figure 2. For example, if the most recent message can be interpreted as a reply to a previous message on the CRS, the interpretation is performed and a synchronous interaction is generated and attached to the LQN elements. The priority in resolving interactions is to interpret them first as synchronous, and then as forwarding; interactions are interpreted as asynchronous only as a last resort. The operations of the CRS can be summarized as follows:

- unresolved messages, with the LQN entries and activities involved in sending and receiving, are pushed on the CRS
- when messages are resolved as LQN interactions, the associated LQN entries and activities are popped off the CRS
- any messages remaining on the CRS when the end of the path is reached are resolved as being involved in asynchronous calls.

A result of this approach is that no message (with its associated workload) is ever lost.

Figure 3 shows a UCM with multiple boundary crossings and the state of the CRS after each of those crossings.

The path traversal is made more complicated by the presence of forks and joins. If a fork is encountered along the path, then the outgoing branches are followed one by one, until either a join or an end point is reached. Figure 4 shows the order in which a set of path segments with forks and joins are traversed, starting from the start point on the left. When a join is encountered the traversal proceeds past it only after all the incoming branches that can be reached from the current start point have been traversed.

Branching can also affect the structure of the CRS. When a fork is encountered the CRS is also forked, so that there is a separate CRS sub-stack for each branch of the fork. Branches are explored in an arbitrary order. When exploring a branch, interactions are resolved as they are found. If, after an OR fork, the resolution of messages on a branch involves messages sent before the fork (and therefore in a previous CRS sub-stack), then the fork is moved back to just before them. The CRS elements back to the new fork are duplicated for all branches.

CRS contents after boundary crossing a:
[1] - message sent by taskA

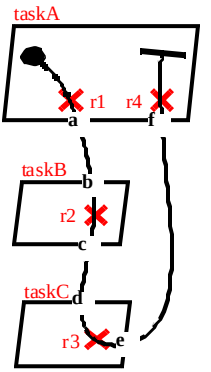
CRS contents after boundary crossing b:
[1] - call made from taskA to taskB

CRS contents after boundary crossing c:
[2] - message sent by taskB
[1] - call made from taskA to taskB

CRS contents after boundary crossing d:
[2] - call made from taskB to taskC
[1] - call made from taskA to taskB

CRS contents after boundary crossing e:
[3] - message sent by taskC
[2] - call made from taskB to taskC
[1] - call made from taskA to taskB

CRS contents after boundary crossing f:
(empty)



Final resolution of messages after boundary crossing f:
...taskA makes a synchronous call to taskB
...taskB makes a forwarding call to taskC
...taskC sends a reply to taskA

Figure 3: UCM showing the contents of the CRS after each of a series of component boundary crossings.

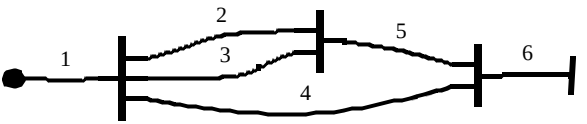


Figure 4: UCM path showing the order in which branches are traversed.

4.2 SPT Algorithm

The following high-level description of the algorithm describes the operations carried out at each point along the path:

(a) create appropriate LQN objects for the current path point

1.1. *if* the current point is a start point *then*

1.1.1. create a reference task for the start point

1.2. *if* the current point is an end point *then* go to step 4

1.3. *if* the current point is a responsibility or a stub with no plugin *then*

1.3.1. create an LQN activity and update it with the service requests of the responsibility or stub

1.4. *if* the current point is a fork *then*

1.4.1. create an LQN fork of an appropriate type

1.4.2. create a branch CRS for the next branch path to be traversed

1.5. *if* the current point is a join *then*

1.5.1. *if* all the incoming branches have been traversed *then* proceed past the join and merge the CRS for the last branch to be traversed with the main path CRS before the branch

1.5.2. *else* go back and traverse the next incoming branch

1.5.2.1. create a branch CRS for the next branch path to be traversed

1.6. *if* the current point is a loop head *then*

1.6.1. create a repeated LQN activity to be the loop control activity

1.6.2. create an LQN task to handle the loop body

1.6.3. add a synchronous call from the loop control activity to the loop body

(b) look ahead to the next point on the path

(c) analyze inter-component interactions (identify any component boundary crossings and resolve the nature of the inter-component messages)

3.1. *if* the current point resides in a component *then*

3.1.1. *if* the next point does not reside in a component *then* create an unresolved message, with an activity to send it, and push them on the CRS

3.1.2. *else if* the next point resides in a different component that has a message pending on the CRS *then* identify a synchronous or forwarding interaction and resolve it

3.1.3. *else if* the next point resides in a different component that does not have any message pending on the CRS *then* identify a call of unknown type (synchronous, forwarding, or asynchronous).

3.2. *else* the current point does not reside in a component

3.2.1. *if* the next point resides in a component that does not have any message pending on the CRS *then* identify the reception of a call

3.2.2. *else if* the next point resides in a component that has a message pending on the CRS *then* identify a synchronous or forwarding interaction and resolve it

(d) *if* the current point is an end point *then* any unresolved interactions are asynchronous

(e) *else* set the next point as the current path point and go to step 1

The algorithm ensures that every responsibility in the scenario is traversed and that a corresponding LQN activity is generated with the specified service demands. A more detailed description of the algorithm can be found in [20].

5 Example - Ticket Reservation System

The Ticket Reservation System (TRS) allows users to browse through a catalogue of events and seat availability, and to buy tickets using a credit card. The UCM design for the TRS is shown in Figure 5, with the components being as follows:

- *User*: TRS customer
- *WebServer*: web interface to the TRS, executes CGI scripts
- *Netware*: the underlying network software and the network itself
- *CCReq*: credit card verification and authorization server
- *Database*: database server

A *User* can access the TRS can be used either to browse events by displaying an event schedule and seating availability, or to buy tickets using a credit card. A typical scenario involves the *User* logging on to the system by requesting a connection. The *WebServer* then logs the user on and opens a session, then confirms that the connection was made. Once she is connected to the system, the *User* enters a loop where she has two options. She can either choose to browse and check information about an event, or she can buy a ticket. If the browsing option is chosen, the *WebServer* sends an event information request to the *Database*, through the *Netware*. The *Database* is responsible for retrieving the data requested and send it to the *WebServer*. The information can

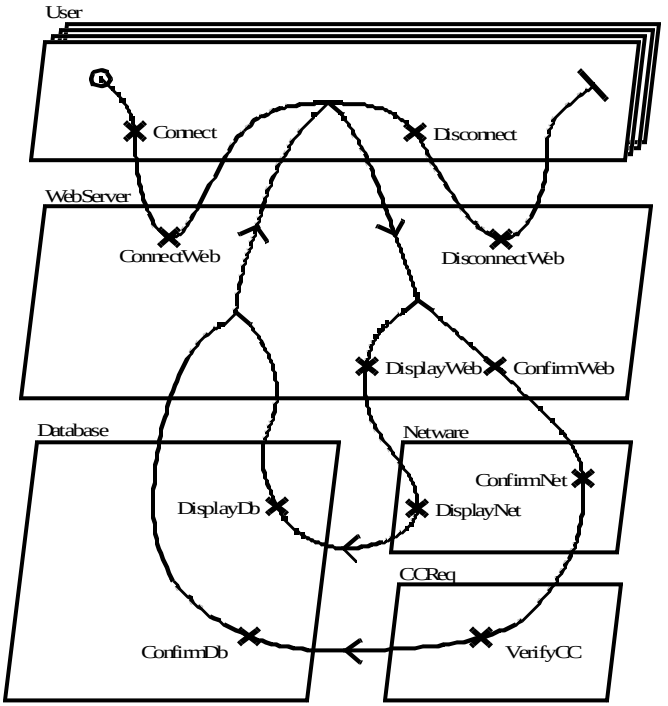


Figure 5: Ticket Reservation System Use Case Map model.

then be displayed back to the *User*, who can now choose whether to continue browsing, purchase tickets or disconnect. If the ticket purchasing option is chosen, the *User* must supply a credit card number to which the purchase price can be billed. The *WebServer* then begins to confirm the transaction by contacting the *CCReq* through the *Netware* and requesting that the credit card be verified. Once the credit card is checks out, *CCReq* forwards the purchase request to the *Database* so it may update its records. The transaction is now completed and a confirmation is sent to the *WebServer*, which in turn relays it back to the *User*. The *User* may continue to browse or purchase more tickets as she wishes. Once the *User* is done, she can make a disconnection

request and the *WebServer* closes the session and confirms that the she has been logged out.

The TRS LQN is shown in Figure 6. The LQN shows an initial asynchronous call from the reference task to the *User*, due to the open nature of the model's arrivals. This example requires the conversion of a complex loop, the body of which features forking and joining and makes service requests of other tasks.

Examining the flow of activities and messages in the LQN, the UCM path is readily identifiable. The loop control activity is shown as the diagonally shaded activity marked with an asterisk in the *User* task. The loop body was abstracted away from the loop head and is represented by the *User_dup1* task. The rest of the loop body is taken care of by the activities in *User_dup1*. The activities for the *WebServer* task incorporate the OR fork and join necessary to separate the sequence of actions for browsing or buying. Calls from the *WebServer* are forwarded by the *Netware*, and by the *CCReq* if a reservation was made, before being replied to by the *Database*.

The resulting LQN file has been solved by the solver LQNS, to demonstrate that it is a correctly formed model definition. However the model results are not critical to the present discussion and will not be presented here. With the model, one could address such issues as

- the CPU load imposed by the servers
- the levels of concurrency needed in the servers,
- the impact on capacity, if there are longer sessions or longer internet delays for each interaction.

6 Transforming Other Scenario Models (e.g. UML Sequence Diagrams)

The SPT algorithm can work with any scenario notation which is based on the sequence connectors described here (sequence, alternative paths, AND forks and joins, and loops), and which binds elements to components. Thus any of the three UML diagrams which show scenarios (Sequence Diagrams, Collaboration Diagrams, and Activity Diagrams) can be transformed by the SPT algorithm. Other scenario notations that can be handled include the ITU standards, Message Sequence charts and High-Level Message Sequence Charts.

For example, in Sequence Diagrams, each participating instance will be treated as a component and transformed to a LQN task. If necessary these can be aggregated later on the basis of concurrent processes. Messages are clearly identified and do not need to be inferred. Looping may be represented by an enclosed set of interactions, which is identifiable as a loop body for the transformation. AND forks (and joins) have to be recognized implictly by multiple message send events (or receive events). OR forks may occur in two ways; by multiple messages leaving the same point, or by a split in the instance lifeline (the vertical timeline associated witht he instance). An OR join may be shown by a merging of the split lifelines, or by messages arriving at the same point. Operations can be inferred for handling the receipt of messages (which activate methods) and for the purposes of the transformation can be treated similarly to UCM responsibilities. This covers all the key elements of the scenario, so the interaction types can be determined by the SPT algorithm. The problem of parameter annotations

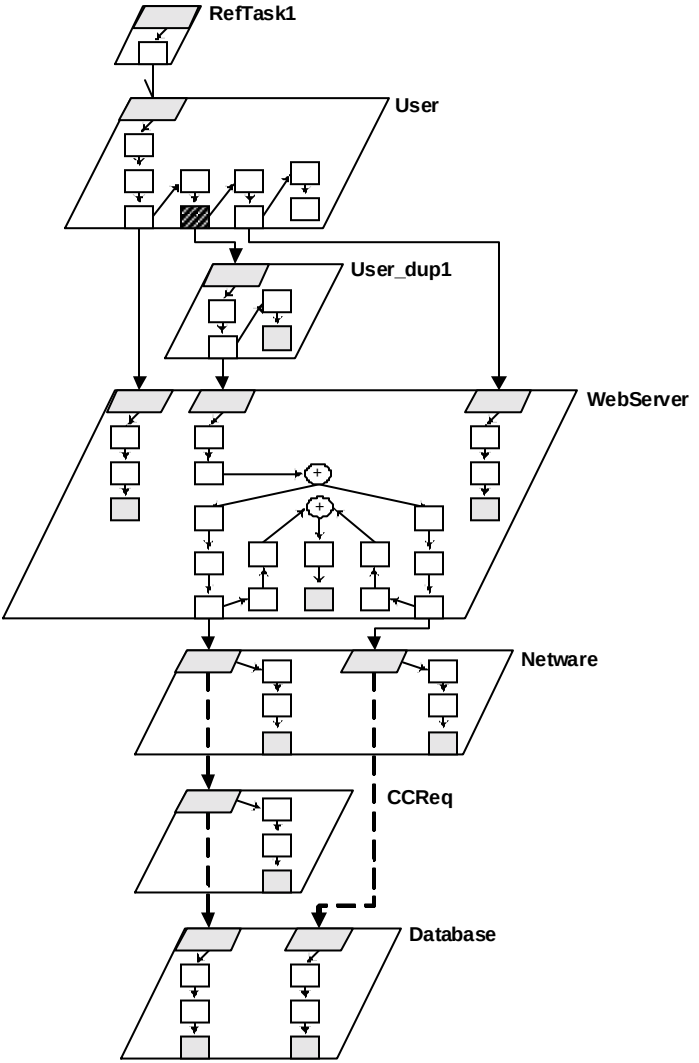


Figure 6: TRS LQN showing activity connections based on the output generated by the UCM2LQN converter from the UCM shown in Figure 5.

may be resolved by a proposed standard profile for this purpose [26].

7 Conclusions

The tool described here addresses the problem of capturing performance issues in the earliest software design efforts. The UCM2LQN converter connects high level design in the form of Use Case Maps with performance analysis using Layered Queueing Networks. It demonstrates close integration between the software specification tool (the UCMNav editor) and the performance analysis programs (the LQNS analytic solver and the LQSim simulator).

The SPT algorithm used in UCM2LQN can be applied equally to scenario specifications in other languages, such as sequence diagrams in UML. Some interpretation of the sequence diagram is needed to establish the corresponding constructs for the purposes of the algorithm. Only a sketch of the interpretation of sequence is given here, but it does not appear to be difficult to fill in the details. Since one UCM may present many paths, the equivalent conversion may involve many sequence diagrams.

The key difficulty in the conversion is in identifying blocking interactions between software entities, and potential contention for servers and other logical resources. This involves matching patterns for two kinds of synchronous interactions (synchronous and forwarding), delaying the matching to obtain sufficient information from the path traversal, and careful handling of forks in the path that occur during one of these interactions.

The model-building tool is integrated into the UCM Navigator, which is freely distributed and has over a hundred users (see the web site `www.usecasemaps.org` for the UCM User Group).

Some improvements to the model building are still needed in the use of data conditions to define which paths are part of a given scenario, and the handling of paths which are parts of larger paths.

Acknowledgements

This research was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC), through its program of Research Grants, and by Communications and Information Technology Ontario (CITO).

References.

1. D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware, “Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS”, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00), Glasgow, Scotland, May 2000

2. D. Amyot, L. Logrippo, R.J.A. Buhr, and T. Gray, “Use Case Maps for the Capture and Validation of Distributed Systems Requirements”, Fourth International Symposium on Requirements Engineering (RE'99), Limerick, Ireland, June 1999
3. D. Amyot and G. Mussbacher, “On the Extension of UML with Use Case Maps Concepts”, The 3rd International Conference on the Unified Modeling Language (UML 2000), York, UK, October 2000.
4. S. Balsamo and M. Simeoni, “Deriving Performance Models from Software Architecture Specifications”, European Simulation Multiconference 2001 (ESM 2001), Prague, June 2001
5. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide. Addison-Wesley, 1998.
6. R. J. A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems," IEEE Transactions on Software Engineering, vol. 24, no. 12 pp. 1131 - 1155, 1998.
7. R.J.A. Buhr and R.S. Casselman, “Use Case Maps for Object-Oriented Systems”, Prentice Hall, 1996
8. L. Constantine, L. Lockwood , "Software for Use", Addison Wesley 1999
9. V. Cortellessa and R. Mirandola, “Deriving a Queueing Network-based Performance Model from UML Diagrams”, ACM Proceedings of the Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000, pp. 58-70
10. G. Franks and M. Woodside, "Effectiveness of early replies in client-server systems," Performance Evaluation, vol. 36--37, pp. 165-183, August 1999.
11. G. Franks and M. Woodside, "Performance of Multi-level Client-Server Systems with Parallel Service Operations," in Proc. of Workshop on Software Performance (WOSP98), October 1998, pp. 120-130.
12. Greg Franks, “Performance Analysis of Distributed Server Systems”, Report OCIEE-00-01, Ph.D. thesis, Carleton University, Ottawa, Jan. 2000
13. H. Gomaa and D. A. Menasce, “Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures”, ACM Proceedings of the Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000, pp. 117-126
14. J. Hodges and J. Visser, “Accelerating Wireless Intelligent Network Standards Through Formal Techniques”, IEEE 1999 Vehicular Technology Conference (VTC’99), Houston, 1999
15. C. Hofmeister, R. Nord, and D. Soni, Applied Software Architecture. Addison-Wesley, 1999.
16. P. Kahkipuro, "UML Based Performance Modeling Framework for Object Oriented Systems," in UML99, The Unified Modeling Language, Beyond the Standard, LNCS 1723, Springer-Verlag, Berlin, 1999, pp. 356-371.
17. A. Miga, “Application of Use Case Maps to System Design With Tool Support”, M.Eng. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998
18. O. Monkewich, “NEW QUESTION 12: URN: User Requirements Notation”, ITU-T Study Group 10, Temporary Document 99, November 1999

19. J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, “Software Bottlenecking in Client-Server Systems and Rendez-vous Networks”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995
20. Dorin Petriu, “Layered Software Performance Models Constructed from Use Case Map Specifications”, M.A.Sc thesis, Carleton University, May 2001.
21. Dorin Petriu and C. M. Woodside, “Evaluating the Performance of Software Architectures”, The 5th Mitel Workshop (MICON2000), Mitel Networks, Ottawa, August 2000
22. D. C. Petriu, C. Shousha, and A. Jalnapurkar, “Architecture-Based Performance Analysis Applied to a Telecommunication System”, IEEE Transactions on Software Engineering, Vol. 26, No. 11, Nov 2000, pp. 1049-1065
23. R. Pooley, "Software Engineering and Performance: a Roadmap," in The Future of Software Engineering, part of the 22nd Int. Conf. on Software Engineering (ICSE2000), Limerick, Ireland, June 2000, pp. 189-200.
24. J. A. Rolia, K. C. Sevcik, “The Method of Layers”, IEEE Transactions on Software Engineering, Vol. 21, No. 8, 1995, pp. 682-688
25. C. Scratchley, C. M. Woodside, “Evaluating Concurrency Options in Software Specifications”, Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecomm Systems (MASCOTS99), College Park, Md., October 1999, pp 330 - 338
26. B. Selic, A. Moore, M. Bjorkander, M. Gerhardt, B. Watson, and M. Woodside, "Response to the OMG RFP for Schedulability, Performance and Time," Obect Management Group, OMG document ad/01-06-14, June 12,2001.
27. K. H. Siddiqui, C. M. Woodside, “A description of Time/Performance Budgeting for UCM Designs”, The 5th Mitel Workshop (MICON2000), Mitel Networks, Ottawa, August 2000
28. C. U. Smith, “Performance Engineering of Software Systems”, Addison-Wesley, 1990
29. C. U. Smith, L.G. Williams, “Performance Engineering Evaluation of Object Oriented Systems with SPE-ED”, Computer Performance Evaluation: Modeling Techniques and Tools (LNCS 1245), Springer-Verlag, 1997
30. C. U. Smith, Murray Woodside, “Performance Validation at Early Stages of Development”, Position paper, Performance 99, Istanbul, Turkey, October 99
31. 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 Transactions on Computers, Vol. 44, No. 1, Jan 1995, pp. 20-34
32. Murray Woodside, "Software Resource Architecture", to appear in Int. Journal on Software Engineering and Knowledge Engineering (IJSEKE), 2001
33. ACM Proceedings of the Workshop on Software and Performance (WOSP’98), Santa Fe, USA, 1998
34. ACM Proceedings of the Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000