Software Performance Models from System Scenarios

Dorin Bogdan Petriu, Murray Woodside

Dept. of Systems and Computer Engineering

Carleton University, Ottawa K1S 5B6, Canada.

{dorin,cmw}@sce.carleton.ca

Abstract. The earliest definition of a software system may be in the form of Use Cases, which may be elaborated as scenarios. In this work performance models are created from scenarios, to permit the earliest possible analysis of potential performance issues. Suitable forms of scenario models include UML Activity or Sequence Diagrams, and Use Case Maps from the URN standard. They capture the causal flow of intended execution, and the operations, activities or responsibilities which may be allocated to components, with their expected resource demands. The SPT algorithm described here automatically transforms scenario models into performance models, and the LQNGenerator tool implements SPT to convert UCM scenario models into Layered Queueing performance models. SPT can in principle also be applied to other scenario models, including Message Sequence Charts, UML Activity Graphs (or Collaboration Diagrams, or Sequence Diagrams).

1 Introduction

Software performance analysis often begins from scenario definitions, which describe the system behaviour during a response. Many different notations have been used to capture scenarios, and this masks their common features. Scenarios have been developed as an approach to software design relatively recently. Carroll ([8], see Chapter 3) gives a broad discussion of their significance and argues that scenario based techniques avoid premature commitment, contain complexity and maintain focus on the essential problem in a new application. They also naturally address the question of evaluation during design rather than after, which is relevant to our present purpose. Carroll's work is focused on User Interface design, but we have found that his arguments also apply to other software. Carroll's scenarios are expressed as natural language scripts, as a natural extension of Use Cases.

This work is based on a well-developed scenario language for software design, called Use Case Maps (UCMs) [7], but the method applies 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 [17].

To provide continuous re-evaluation during the evolution of an architecture there

must be automation. This paper describes the LQNGenerator, a tool which automatically creates Layered Queueing Network (LQN) performance models from UCMs. It is integrated into the UCM Navigator, which is an editor and repository tool for UCMs, so that the Navigator can be used as a front-end editor for design models which are also performance models.

The concepts can in principle be extended to other scenario modeling tools such as Stories [9], UML Sequence Diagrams or other behavior models [5]. Smith has defined an Execution Graph notation which is converted into queueing models [33][34]. Conversion of UML Collaboration models into layered queues has previously been described by Kahkipuro [20]. Eventually the ideas of LQNGenerator may be applied to annotated UML models using a new standard profile for performance annotations [31].

The performance model formulation used here is Layered Queueing, because it 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 [38]. Petri net models capture these features very well, as described by Pooley [28], but sometimes have problems with larger system scales.

The difficult problem solved by LQNGenerator 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. The algorithm for creating the models is called SPT (Scenario to Performance Model Transformation).

There has been considerable effort expended on methods for Software Performance Engineering (SPE), and an overview can be obtained from the proceedings of the three international Workshops on Software and Performance (WOSP'98 [39], WOSP2000 [40] and WOSP2002 [41]). 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 [32][35]. 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.

An earlier version of this paper [25] gave a first description of the SPT algorithm and UCMs; this expanded version gives a more complete description of the algorithm, adds a much more extensive example related to software for telephony, and shows how SPT can also be applied to UML Sequence Diagrams.

2 Models for Scenarios and Performance

Scenarios were proposed for object-oriented design (e.g. [36]) as plain-language stories, which were analyzed to identify objects (nouns) and activities (verbs). This approach was developed into Use Cases [19], in which the stories have formal relationships to each other, and further developed by Constantine and Lockwood in [9].

Message Sequence Charts (MSCs) were developed to describe communications systems, and evolved from a kind of virtual message trace into a more abstract language which describes operations as well as messages and can combine sub-scenarios [18]. Sequence Diagrams in UML, which are a popular notation for intended behaviour, are a modified version of MSCs. Both of these notations are most effective for expressing single scenarios, although capabilities to describe alternative paths exists; an example of a Sequence Diagram will be shown later. UML also incorporates two other notations to display intended behaviour, the Activity Diagram and the Collaboration Diagram.

The Use Case Map 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. UCMs are being standardized as ITU-T Z.152 as part of the User Requirements Notation (ITU-T Z.150).

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 UCMs. Collaboration Diagrams in UML do provide a high-level description of how the system works, but only one scenario at a time [3]

2.1 Use Case Maps

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

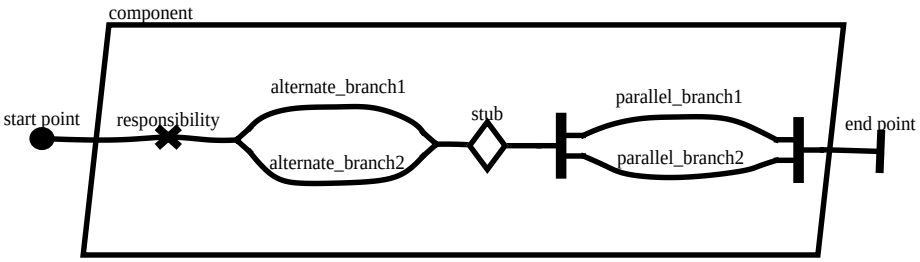


Figure 1: Example of the UCM notation.

filled circle) to an *end point* (a bar), and traversed by a token from start to end. Paths can be overlaid on *components* representing 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 generation of performance models assumes that 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 ([33], chapter 4) does not always apply.

The UCM Navigator (UCMNav) [21] 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 LQNGenerator 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 LQSim, a simulator, or LQNS, an analytic solver.

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 [29]. 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 [23]. There have been many applications [27][37].

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 different priority levels on their CPU. The workload of a LQN is driven by open arrival streams of external requests, or 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 using LQSim, or by analytic approximations using LQNS. LQNS is based on [37] and the Method of Layers [29], with a number of additional approximations [12][13][14]. 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 LQNs can be understood more clearly using UCMs to show the sequences of events. Figure 2 shows a series of UCMs describing the interactions which must be detected when building an LQN model:

1. 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.

2. 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.

3. 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 [33]), 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 [20], require that blocking interactions be explicitly identified.

3.1 Identifying Blocking Interactions

The detection of blocking interactions is important because they are not always obvious in a system but they have a big performance impact. The SPT algorithm takes a conservative approach to performance modeling and assumes that if the path returns to a component, it was waiting for the return and thus was blocked. This maximizes the interpretation of blocking interactions, which yields models that capture all the possible blocking points in a system.

We have found that even when designers use asynchronous messaging, the design may introduce blocking, when a task needs certain information before it can proceed. If there is a truly non-blocking component it is modeled as having multiple threads, one to maintain the context of each uncompleted operation. If the number of opera-

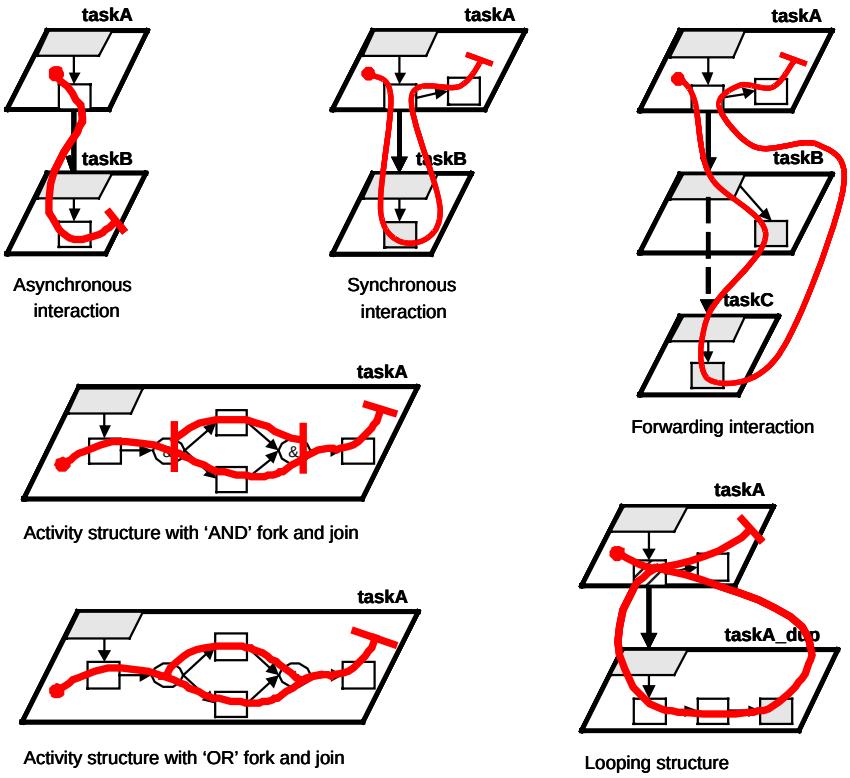


Figure 2: Corresponding interactions and structures in UCM and LQN.

tions that may be carried is unbounded there are infinite threads. In this way blocking behaviour is decoupled from the use of synchronous or asynchronous messages.

There may be additional blocking introduced by the environment, for example an ORB, in behaviour that is not described in the UCM. To discover this blocking, we need more information, for example from completions of the model [32].

3.2 Correspondences between UCMs and LQNs

There are some quite close correspondences between some of the scenario entities, 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 each pair 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.

3.3 Correspondences of Path Structure in LQN

The scenario expression of path structure within a component 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(d) 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(e) 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(f).

3.3.1 Fork and Join in Separate Components

In a scenario, paths may fork in one component and join in another. Both UCMs and LQNs 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.

3.4 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 Algorithm for Scenario to Performance Model Transformation

The algorithm for Scenario to Performance model 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 Message Stack (MStack)

The Message Stack (MStack) 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 MStack, 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 MStack can be summarized as follows:

- unresolved messages, with the LQN entries and activities involved in sending and receiving, are pushed on the MStack
- when messages are resolved as LQN interactions, they are popped off the MStack and their associated LQN entries and activities are updated
- any messages remaining on the MStack 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.

MStack contents after boundary crossing a:

[1] - message sent by taskA

MStack contents after boundary crossing b:

[1] - call made from taskA to taskB

MStack contents after boundary crossing c:

[2] - message sent by taskB

MStack contents after boundary crossing d:

[2] - call made from taskB to taskC

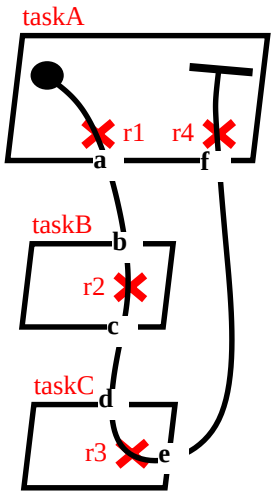
MStack contents after boundary crossing e:

[3] - message sent by taskC

[2] - call made from taskB to taskC

MStack 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 MStack after each of a series of component boundary crossings.

Figure 3 shows a UCM with multiple boundary crossings and the state of the MStack 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 MStack. When a fork is encountered the MStack is also forked by copying, so that there is a separate MStack for each

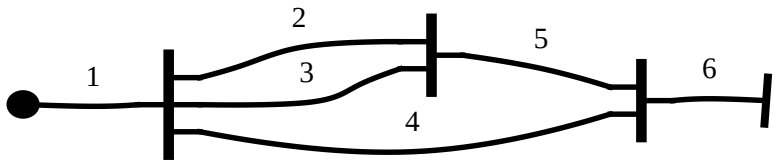


Figure 4: UCM path showing the order in which branches are traversed.

branch of the fork. Branches are explored in an arbitrary order. When exploring a branch, interactions are resolved as they are found. If a branch ends and some messages from before the fork are resolved as being asynchronous, subsequent branches may re-resolve those messages as being synchronous. However in order to maximize the synchronous interpretation of messages, synchronous messages may not be re-resolved as asynchronous by any other branches.

4.2 SPT Algorithm

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

1. 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 an empty stub *then*
     - 1.3.1. create an LQN activity and update it with the service requests of the responsibility or stub
     - 1.3.2. make the activity a candidate activity for use to capture path sequencing detail
   - 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 duplicate MStack 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*
       - 1.5.1.1. proceed past the join and merge the MStack

for the last branch to be traversed with the main path MStack before the branch

1.5.2. *else*

1.5.2.1. go back and traverse the next incoming branch

1.5.2.2. create a duplicate MStack 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

2. look ahead to the next point on the path

3. 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*

3.1.1.1. create an unresolved message

3.1.1.1.1. *if* there is a candidate activity in the task *then*

3.1.1.1.1.1. use the candidate activity to send the message

3.1.1.1.2. *else*

3.1.1.1.2.1. create a new default activity to send the message

3.1.1.2. push the message on the MStack

3.1.2. *else if* the next point resides in a different component that has a message pending on the MStack *then*

3.1.2.1. 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 MStack *then*

3.1.3.1. identify a call of unknown type (may later turn out to be a synchronous, forwarding, or asynchronous call)

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

nent

3.2.1. *if* the next point resides in a component that does not have any messages pending on the MStack *then*

3.2.1.1. 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 MStack *then*

3.2.2.1. identify a synchronous or forwarding interaction and resolve the appropriate message

4. *if* the current point is an end point *then*

4.1. any unresolved interactions are asynchronous

5. *else*

5.1. set the next point as the current path point and go to step 1

The SPT 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 [24].

5 Examples

5.1 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 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*

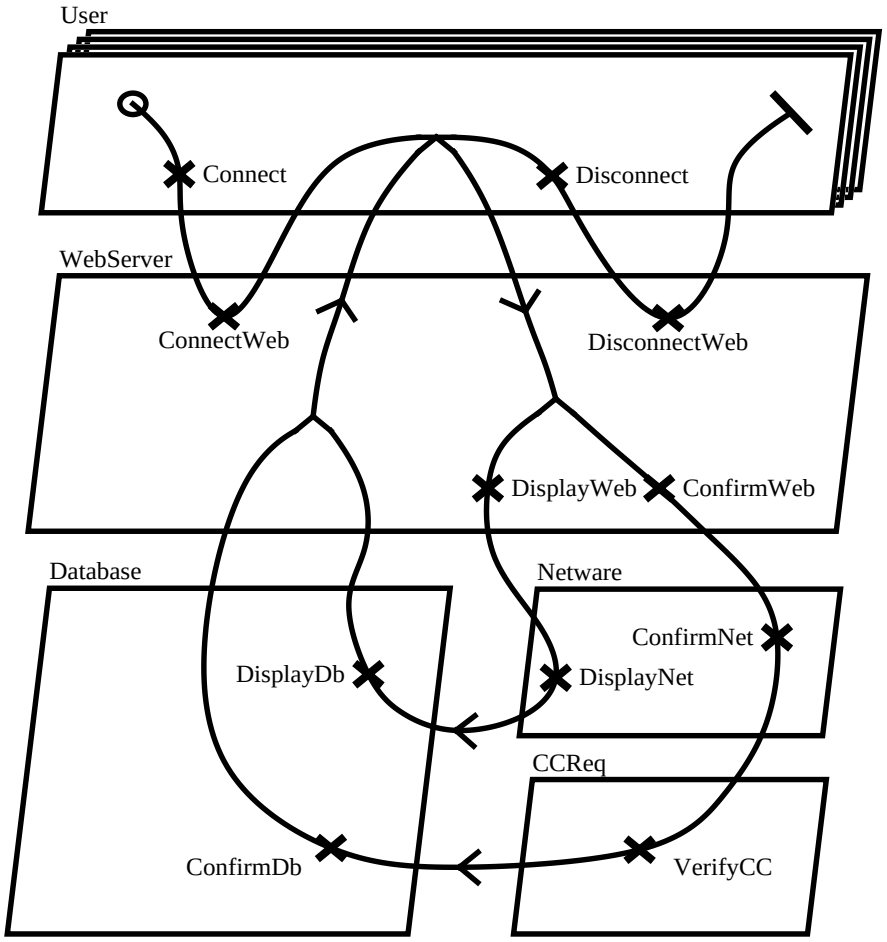


Figure 5: Ticket Reservation System Use Case Map model.

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.

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

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_loop1* task. The rest of the loop body is taken care of by the activities in *User_loop1*. 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 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.

5.2 POTS

This section analyzes a Plain Old Telephone System (POTS) call connection scenario. Figure 7 and Figure 8 describe the scenario, including stubs for incorporating enhanced features which are not elaborated in this analysis. Features can be described by separate plug-in maps or, for the purposes of performance analysis, their workload can be expressed as an average workload for the stub.

POTS has two possible scenarios that can happen when attempting to make a call: the call can either be set up successfully, or the callee is busy. If the call is placed successfully, the scenario unfolds as follows:

- the originator (caller) picks up the receiver
- the switch notes that the originator is now busy
- the originator gets a dial tone
- the originator dials the desired terminator's number (callee)
- the dial tone stops
- the switch checks and finds that the terminator is currently idle
- the switch stores the originator's number as the terminator's last incoming number
- the terminator gets a ring and the originator gets a remote ringing tone
- the terminator picks up the receiver
- the terminator's ring stops
- the originator's remote ringing tone stops and the billing details are recorded by the operations system

- the connection is now made

Otherwise, an unsuccessful call connection scenario unfolds as follows:

- step 1 through step 5 are the same
- the switch checks and finds that the terminator is currently busy
- the originator gets a busy tone
- the connection is not made

Figure 7 shows the root UCM for the POTS system. The components for all the POTS maps are as follows:

- *Orig*: the caller's telephone set
- *Term*: the callee's telephone set
- *Switch*: the telephone company's switch gear
- *SCP*: the Service Control Point that processes IN features (not used in the POTS scenario)
- *OS*: the Operations System that does the billing

The POTS root map features the following stubs:

- PreDial: features that are activated before the number is dialed
- PostDial: features that are activated after the number is dialed
- Billing: different billing schemes depending on the kind of connection and which features are invoked

For POTS operation the PreDial stub can either be left empty or it may have a default plug-in that merely connects the input and output paths. Similarly, the Billing stub has a straight path connecting its input and output. This billing path has a single responsibility to log the start time of the connection between the caller and the callee.

The PostDial stub has more functionality and the plug-in is shown in Figure 8. The PostDial plug-in describes the behaviour of contacting the called party and establishing a connection, or the notification of the caller that the called party is busy. The PostDial map features the following stubs:

- ProcessCall: features dealing with making a connection
- ProcessBusy: features associated with the callee being busy
- NumberDisplay: feature displaying the caller's number

For POTS operation both the ProcessBusy and NumberDisplay stubs can either be left empty or have default plug-ins without any responsibilities

The ProcessCall plug-in for normal POTS operation is shown in Figure 9 and checks whether the called party is idle or busy. If the called party is idle then its status is changed to busy and the process of making the connection is started. Otherwise the process of notifying the caller that the called party is busy begins.

5.2.1 POTS Path Traversal

The POTS model incorporates multiple levels of abstraction in the form of multiple levels of stubs. The SPT algorithm handles the abstraction by flattening out the UCM as the stubs are traversed. The path traversal for a stub is done as follows:

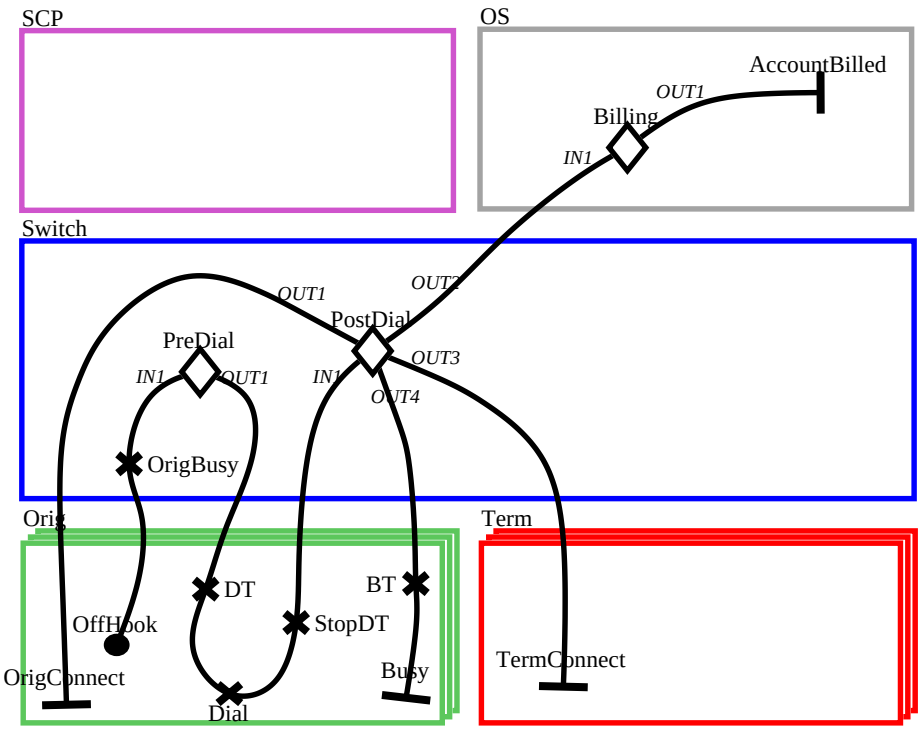


Figure 7: Top level UCM map for the POTS example.

1. *if* the stub has multiple plug-ins *then*
   1.1. identify the stub as a dynamic stub
   1.2. create an OR fork for each stub input point
   1.3. create an OR join for each stub output point
   1.4. *for* each plug-in
      1.4.1. traverse the path from each bound start point as if it were a branch on the stub input OR fork and join each bound end point as if were a branch on the stub output OR fork
2. *else if* the stub only has one plug-in *then*
   2.1. identify the stub as a static stub
   2.2. traverse the path from each bound start point as if it were joined to the stub input and join each bound end point to the stub output
3. *else*

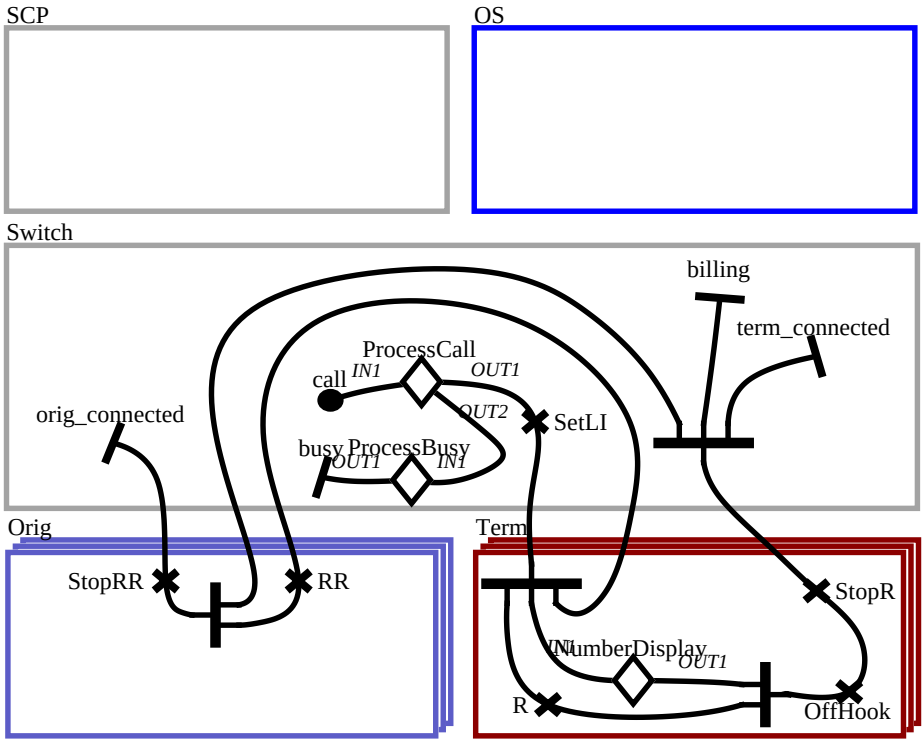


Figure 8: Plug-in UCM for the PostDial stub in Figure 7.

3.1. `identify the stub as an empty stub`

3.2. `create an LQN activity and update it with the service requests of the stub`

3.3. `make the activity a candidate activity for use to capture path sequencing detail`

4. `continue past the stub`

Notice that this traversal of the paths “flattens” the plug-ins so they are incorporated seamlessly in the system behavior.

5.2.2 POTS Performance Analysis Results

Figure 10 shows the POTS LQN model generated by the LQNGenerator from the UCMs described above. The layered architecture of the system is now evident, with the *Orig* (caller) generating the requests for the *Switch* which in turn calls the *Term* (callee) and sends billing messages to the OS

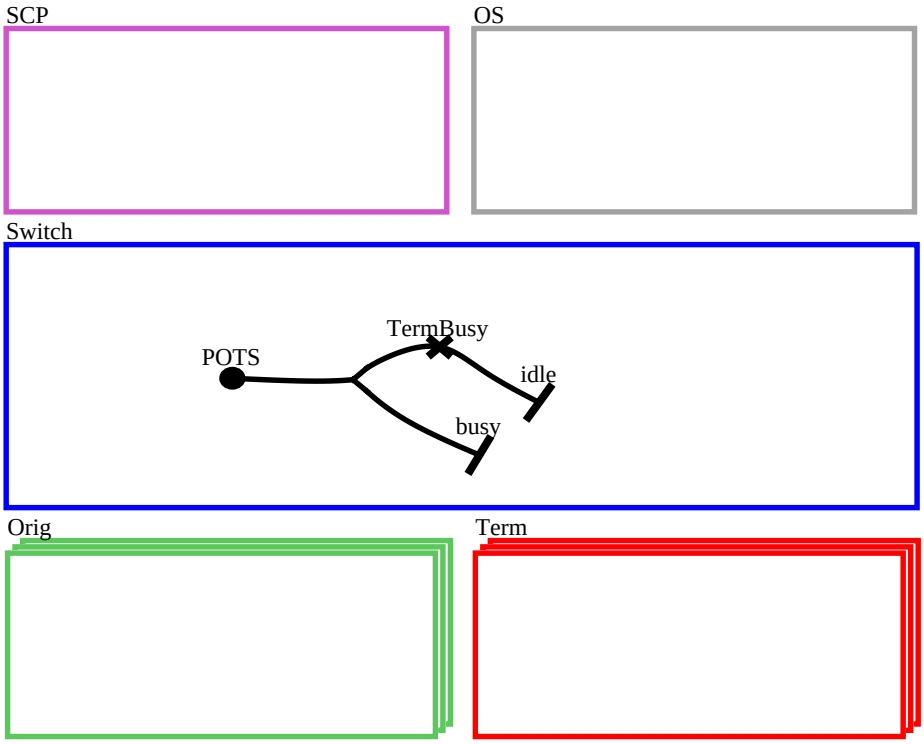


Figure 9: POTS call processing plug-in for the ProcessCall stub in Figure 8.

Figure 11 shows the connection delay vs. call volume results obtained by simulating the POTS LQN using LQSim. The simulations were run using either a single thread, five threads, or ten threads for the *Switch* task. Assuming that an acceptable threshold for the connection delay is 100 ms, then the system can handle up to about 210000 calls per hour in the single-threaded case, up to about 250000 calls per hour in the case with five threads, and up to a little more than 250000 calls per hour in the case with ten threads. The results show that there is practically no difference, and thus no benefit, to having ten *Switch* threads as opposed to five *Switch* threads.

6 Obtaining Performance Models from 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. Recently a new standard UML Profile has been defined for Schedulability, Performance and Time [31], which defines performance annotations in the form of stereotypes and tags that can be added

Figure 10: POTS LQN generated by the LQNGenerator implementing the SPT algorithm.

to these diagrams. The Profile interprets the sequence of messages that invoke operations, as Steps in a Scenario, which includes all the necessary precedence relationships.

Figure 12 shows a Sequence Diagram for the Ticket Reservation scenario, to illustrate these ideas. The vertical lines (“lifelines”) are associated with instances (typically objects or subsystems) which represent components, at the level of abstraction of the diagram. Operations analogous to responsibilities in UCMs are shown by the rectangles (“focus of control”) over the vertical lines, and causal sequences are indicated by arrows representing messages from one component to another. A sequence of Steps in the same component is indicated by a series of nested focus of control rectangles. OR forks may be shown in two ways; by multiple messages leaving the same point, or by a

Figure 11: POTS simulation results for call connection delay vs. call volume with 1, 5, and 10 Switch threads.

split in the instance lifeline (the vertical timeline associated with the instance). An OR join may be shown by a merging of the split lifelines, or by messages arriving at the same point. An AND fork is indicated where two or more messages leave the same lifeline with no message received between them, and the join, where two or more messages arrive at a lifeline with no message sent between them. A loop can indicated by a message with a repeat count (e.g. *N), which shows that the focus of control that receives it will be repeated.

The annotations shown as tagged values in the performance-related stereotypes are related in obvious ways to the performance annotations in a UCM, covering the load intensity, processor deployment, and demands made by a Step.

Sequence Diagrams are more limited than UCMs for expressing scenarios. The workload intensity and deployment descriptions are similar. Both describe operations carried out by components. The essential structure of paths can be represented in both, as described above, although the SD representation requires making assumptions. Apart from these points,

- A UCM can include operations not assigned to a component (which we interpret

Figure 12: SD for the Ticket Reservation System superimposed with the equivalent UCM.

as executed by a task and processor created for that purpose)

- One UCM can represent several Sequence Diagrams with different scenarios.
- UCM stubs are more descriptive than hiding of a sub-scenario in a SD. This can be done only at the subsystem level, by creating a lifeline to represent a sub-system with a separate diagram to show its internals; the inner diagram is restricted to the internals. A UCM stub can show any behavior involving any components

A general approach to automatically converting a Sequence Diagram to a LQN model is, to first generate a UCM from it, and then to convert the UCM with the SPT algorithm. This can always be done, because the Sequence Diagram has more limited semantics than the UCM in those aspects that influence performance modeling, as described briefly above.

A two-step approach can be taken to show the application of the SPT algorithm to this scenario. First, it is transformed to an equivalent UCM, then SPT is applied to the UCM. Here, the UCM is indicated by drawing the paths and path connectors over the Sequence Diagram, directly in Figure 12. The instances are interpreted as UCM components, and will be transformed into LQN tasks. Messages are clearly identified and do not need to be inferred. Each message reception or focus of control is interpreted as a UCM responsibility. The annotations for workloads, resource demands and resource parameters, provided by the UML Profile for performance [31], are approximately equivalent to those of a UCM that were described above. In general, a scenario expressed in a UML Sequence Diagram can be interpreted without ambiguity.

Where a system is defined by multiple scenarios, several Sequence Diagrams may be required. In this case each one is converted separately and the resulting submodels can be composed together in a straightforward way, by composing the entries of a given task from all scenarios, into a single task based on the task instance name.

Other Scenario Models.

Other kinds of scenario models are transformable in the same way. For example, the closely related notation of Message Sequence Charts was used for performance modeling in [4], [10] and [34]. However, the procedures used for creating a performance model in those works either required user intervention, and were restricted to simpler behaviour. They could not, for instance, recognize blocking interaction patterns and software task resources. SPT could be applied to them also, provided they are used to express models of similar semantics to the UCM models.

In general, any scenario notation that can in principle be transformed to a UCM, can be transformed to a performance model by SPT. Either it is actually first transformed to a UCM and processed by LQNGenerator, or SPT is directly applied to the elements of the notation.

7 Conclusions

The SPT algorithm and the tool based on it address the problem of capturing performance issues in the earliest software design efforts. The LQNGenerator presently 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).

These tools have been used in several projects to model substantial specifications, including an e-commerce site with a dozen scenarios, and a presence service. They provide rapid tracking of the specifier's evolving ideas. One hazard is that it is so easy to create a new UCM, that the performance annotations may be lost! A facility for loading performance annotations from a table such as a spreadsheet has been designed, to overcome this and to give a focussed access to the parameter values.

The two examples described here cover most of the kinds of behaviour and interactions that may occur in UCMs and LQNs. The LQN components correspond one-to-

one with the elements of the UCM specification, traceable by name, so the performance results can be correlated instantly with the scenario specification.

The key difficulty addressed by SPT 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 LQNGenerator model-building tool is currently 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).

The SPT algorithm used in the LQNGenerator can be applied equally to scenario specifications in other languages, such as Sequence Diagrams conforming to the UML performance profile [31]. A procedure for doing this was described, and tools for UML are a subject of current research.

Acknowledgments

Conversations with Daniel Amyot, Andrew Miga, Luigi Logrippo, Don Cameron and Dorina Petriu were helpful in this research. Funding is gratefully acknowledged from the Natural Sciences and Engineering Research Council of Canada (NSERC), through its program of Research Grants and its University Industry Program, from Nortel Networks, and from 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 (UML2000), 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] J. M. Carroll, “Making Use: Scenario-based Design of Human-Computer Interactions”, MIT Press, Cambridge, MA, 2000

[9] L. Constantine, L. Lockwood, “Software for Use”, Addison Wesley 1999

[10] 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

[11] A. Engels, and S. Graf, C. Jard and Y. Lahav (editors), “Design Decisions on Data and Guards in MSC2000”, 2nd Workshop on SDL and MSC (SAM2000), Col de Porte, Grenoble, June 2000, pp. 33-46

[12] G. Franks and M. Woodside, “Effectiveness of early replies in client-server systems”, Performance Evaluation, vol. 36--37, pp. 165-183, August 1999.

[13] G. Franks and M. Woodside, “Performance of Multi-level Client-Server Systems with Parallel Service Operations”, in Proc. of Workshop on Software Performance (WOSP98), Santa Fe, October 1998, pp. 120-130.

[14] Greg Franks, “Performance Analysis of Distributed Server Systems”, Report OCIEE-00-01, Ph.D. thesis, Carleton University, Ottawa, Jan. 2000

[15] 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

[16] J. Hodges and J. Visser, “Accelerating Wireless Intelligent Network Standards Through Formal Techniques”, IEEE 1999 Vehicular Technology Conference (VTC’99), Houston, 1999

[17] C. Hofmeister, R. Nord, and D. Soni, “Applied Software Architecture”. Addison-Wesley, 1999.

[18] ITU-T, “Message Sequence Charts”, ITU-T Recommendation Z.120, International Telecommunications Union, Geneva, November 1999

[19] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, “Object-Oriented Software Engineering: A Use Case Driven Approach”, Addison-Wesley Publishing Co., 1992

[20] 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.

[21] 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

[22] O. Monkewich, “NEW QUESTION 12: URN: User Requirements Notation”, ITU-T Study Group 10, Temporary Document 99, November 1999

[23] 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

[24] Dorin Petriu, “Layered Software Performance Models Constructed from Use Case Map Specifications”, M.A.Sc thesis, Carleton University, May 2001.

[25] Dorin Petriu, Murray Woodside, “Software Performance Models from System Scenarios in Use Case Maps”, Proc. 12 Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002), London, April 2002.

[26] Dorin Petriu and C. M. Woodside, “Evaluating the Performance of Software Architectures”, The 5th Mitel Workshop (MICON2000), Mitel Networks, Ottawa, August 2000

[27] 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

[28] 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.

[29] J. A. Rolia, K. C. Sevcik, “The Method of Layers”, IEEE Transactions on Software Engineering, Vol. 21, No. 8, 1995, pp. 682-688

[30] 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

[31] Object Management Group (OMG), “UML Profile for Schedulability, Performance, and Time Specification”, OMG document ptc/02-03-02 (adopted standard), Mar. 2002.

[32] K. H. Siddiqui, C. M. Woodside, “Performance-Aware Software Development (PASD) Using Resource Demand Budgets”, Proc Third Int. Workshop on Software and Performance, Rome, July 2002, pp 275 - 285.

[33] C. U. Smith, “Performance Engineering of Software Systems”, Addison-Wesley, 1990

[34] C. U. Smith, L.G. Williams, “Performance Solutions”, Addison-Wesley, 2000

[35] C. U. Smith, Murray Woodside, “Performance Validation at Early Stages of Development”, Position paper, Performance 99, Istanbul, Turkey, October 99

[36] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, “Designing Object-Oriented Software”, Prentice-Hall, Englewood Cliffs, N.J., 1990.

[37] 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

[38] Murray Woodside, “Software Resource Architecture”, Int. Journal on Software Engineering and Knowledge Engineering (IJSEKE), v 11, no 4, pp 407-429, 2001

[39] ACM Proceedings of the First International Workshop on Software and Performance (WOSP 98), Santa Fe, USA, Oct. 1998

[40] ACM Proceedings of the Second International Workshop on Software and Performance (WOSP2000), Ottawa, Canada, Sept. 2000

[41] ACM Proceedings of the Third International Workshop on Software and Performance (WOSP2002), Rome, Italy, July 2002