Compositional Layered Performance Modeling of Peer-to-Peer Routing Software

Pengfei Wu, Murray Woodside, Chung-Horng Lung
Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada, K1S 5B6
E-mail: {pfwu, cmw, chlung}@sce.carleton.ca

Abstract

Models can help to understand the performance aspects of a computer system from the software architecture and its configurations, but ease of model creation is critical. A compositional model-building approach is described here, in which component submodels are generated from the scenarios they participate in. Submodels classes are derived from an analysis of behaviour patterns as the scenarios traverse the software components. Then submodels are instantiated and combined in the overall system model. The approach is particularly effective in peer-to-peer systems in which subsystems inherit most of their behaviour from a few shared patterns, termed "Behaviour-Inheriting Peer" (BIP) systems. A model-building algorithm is described, and is demonstrated on a prototype emulator for a network of routers. The emulator, called CGNet, can be configured for its deployment and for traffic patterns and routes. An automatic model-generator uses this information to build a model which represents the overall system configuration. The approach is quite general and can be used to model component-based systems in which the components themselves are created in many configurations.

Index Terms—Software performance engineering, Performance modeling, Use Case Maps (UCM), Layered Queueing Network (LQN), Component-path sub model, Path class, Generative modeling, Component sub model, Peer-to-peer systems.

1. Introduction

There can be many advantages to modeling the performance of a software system, as documented by Smith [Smith90, Smith02] and others, however the process of building models may be lengthy and expensive. The present work sets out to automate the building of models, for a class of systems that can be deployed in many different patterns and at large or small scales.

The class of systems has sub-systems which are similar to each other, with similar behaviour, such as peer-to-peer systems or grid applications. An example which is studied here is a network of routers. The class will be called "Behaviour-Inheriting Peer" systems because the node behaviours are inherited from a small set of behaviour classes. Automation is essential for building models of large scale systems, because the effort of creating a model directly is prohibitive. Automation is valuable also for smaller deployments if there are many instances to be modeled and compared.

Performance models for computer system and networks, and for software, can be built using various approaches, as described in texts by Jain [Jain91], Bolch et al [Bolch98] and others. They can be grouped into methods based on queueing networks and their extensions, such as layered queueing networks [Woodside95a] and methods based directly on states and transitions (Markov models) and their extensions such as stochastic Petri nets (e.g. SPNP [Ciardo89]). They are solved by analytic techniques, numerical methods or simulation. Direct simulation without a formal modeling framework is a third category of tools.

Smith advocates an approach to building these models based on system behaviour or scenarios, which she calls execution graphs [Smith90, Smith02]; other approaches are based on structural relationships and summary workload statistics [Jain91]. Cortellessa and Mirandola [Cortellessa00] describe how to use UML sequence diagrams to derive execution graphs and then queueing models. This work follows the scenario approach.

In this paper scenarios expressed by Use Case Maps (UCMs) are used to generate Layered Queueing Network (LQN) models [Rolia95] [Woodside95a]. Model-building directly from UCMs to LQNs was described in [Petriu02], however the present work avoids creating a complete UCM. Instead it exploits repetitive common behaviours, and generates component submodels based on object behaviours in the software and local UCM sub-paths in a component. It extracts advantages from the use of object-oriented design and component-based software.

Innovations in software such as object-oriented design make performance modeling more complex. The interactions between objects are numerous and are obscured by inheritance, polymorphism, and late binding, making them difficult to trace. Also, distributed systems challenge performance intuition by introducing middleware layers, network latencies and message effects such as blocking delays. All these make it difficult to construct performance models especially for a large systems with lots of interaction and communication. However the re-use of behaviour patterns also provides opportunities.

Component based software systems (e.g. [Szyperski98]) are designed to re-use software components, and provide an opportunity to model the components and re-use these performance sub-models. Strategies for doing this are described in [Wu03a] [Bertolino03]. However in many systems the components are configurable and do not have a single structure or a single performance sub-model. To build a performance model from components we must first create suitable component submodels for the particular application, and that is the role of this research. The submodels can be re-used just as the corresponding software components are re-used in the system.

In this paper we begin with the software architecture, and we express the scenarios of the system in term of Use Case Maps (UCMs) [Buhr96] that show paths and responsibilities over components. Then we build sub-models corresponding to scenario fragments. At last we use the compositional strategy to assemble the sub-models into a model for the entire system. A modest-sized deployment of a system called CGNet [Hobbs01] is used as a case study to demonstrate the compositional approach. It reduces the complexity of building the performance model so that the approach can be scaled up to virtually any size of system.

The compositional strategy is defined in Section 3 below, following some background on the models and notation in Section 2, and it is applied to CGNet in Sections 4 and 5.

2. Models and notation

This work uses Use Case Maps (UCMs) to describe scenarios, and Layered Queueing Networks (LQNs) to model performance. UCMs are a visual notation for use cases [Jacobson92] with additional detail about responsibilities and components. They can be used to reason about architecture related to Use Cases [Buhr96], and to create performance estimates [Petriu02]. An example UCM is shown in Figure 1, with a scenario indicated as a line from a start point (a filled circle) to an end point (a bar). Responsibilities (crosses) indicate operations to be performed, and can be refined by a submap. Responsibilities are contained in and implemented by components, shown as boxes. Components can represent any structural unit, from a subsystem with many processors down to an object or procedure. Components can be nested and in the Figure the inner components with solid lines are called "tasks" T1 to T5, and in this case represent processes, while the outer components C-A, C-B, C-C are just called "components". Each of these outer components is loaded on a different processor P-A, P-B, P-C respectively.

The performance model used here is a kind of extended queueing network called a Layered Queueing Network (LQN), with servers to represent processors and other devices, and also servers to represent software tasks. Layered queueing arises when a software server task has to wait for service at its processor, or at another task. Figure 1 also shows the tasks and the requests for service that are implied by the UCM, with one sub-model for each scenario. Tasks are indicated by rectangles, divided into a field for the task and its properties, and fields for entries which provide classes of service. The entries have been named for the first responsibility they perform. In S1, entry a makes a request to entry b and waits for the reply; this kind of

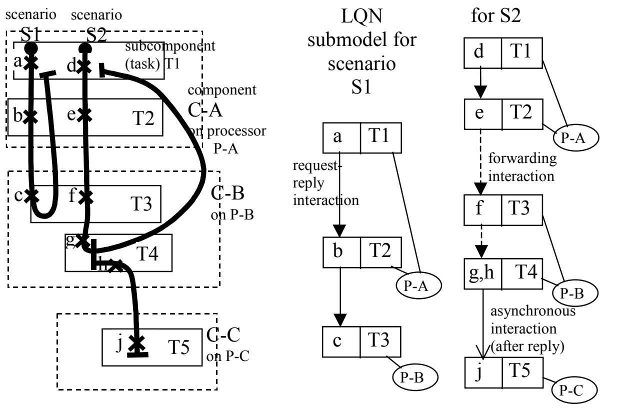


Figure 1. A Use Case Map with two scenarios, and their LQN submodels

"synchronous" request is indicated by an arrow style with a filled head. Entry b then makes a similar request to entry c. In scenario S2, entry d makes a synchronous request to e, which forwards it for processing to f and g (the forwarding task does not wait for a reply; the reply goes to the originator). Forwarding request is indicated by a dashed arrow. After the reply from T4, entry g does a further operation for responsibility h (this kind of delayed operation is called a "second phase"), and then a second phase request to entry j. This final request is asynchronous, with no reply, indicated by an arrow with an open arrowhead.

Tasks and processors are both resources; each resource has a queue and a discipline, and may be multiple (that is, a multiserver such as a multithreaded task or a multiprocessor). Each task has a host processor, which identifies the physical device that carries out its operations. The processors for each of the dashed components in the UCM are indicated by ovals attached to each task. The LQN paradigm can model most of the features such as multi-threaded processors, devices, locks, communication and so on [Franks00]. LQN models can be solved to determine throughputs and delays, and the contention for software and hardware resources, and to identify bottlenecks [Neilson95].

3. Scenario-based generation and composition of submodels

The model-building approach is based on the distributed software objects and on overall scenarios (that is, UCMs) describing the end-to-end system behaviour for different inputs. Scenarios can be constructed based on the system requirements and design. The types of requests can be described as Use Cases, and the processing of each is traced as a sequence of high-level operations, identified in the UCM as its responsibilities. The operations are allocated to software task components, and other resources required are also identified as UCM components (the path enters the component to obtain the resource, and leaves to release it). For early analysis, the scenarios can be based on the documentation and the expertise of the developers [Smith90]; later, they can be based on tracing the execution of the code.

We assume that the system is divided into a set of large grain subsystems, and from here on the term "component" will be reserved for these; each of them will generate a *component submodel* in the overall performance model. They may be subsystems with internal concurrency. The approach is particularly suitable when the system has many of these components and each one includes several tasks and a large number of responsibilities. It is particularly simple and efficient when each component is assembled from tasks and operations which are variations on a small number of basic tasks and operations, as in the router example later.

Object-oriented design promotes re-use not only of data structures, but also of behaviour. This is particularly marked in systems with peer components based on the same set of classes, and with strongly similar functionality; their behaviour is inherited from the common classes. We will call these "Behaviour-Inheriting Peer" (BIP) systems. A set of internet routers with identical code is an example. The objects in each router may be instantiated differently to conform to its configuration, but their behaviour comes from common classes.

Strategy A, for a system defined explicitly by a set of scenarios:
Strategy A does not make any assumption about similar behaviour in different components.

1. Optionally define the system-wide scenario for each request, as just discussed;
2. Decompose each scenario into fragments at the component boundaries, so there is a set of *"component-path"* scenario fragments in each component. For a system with many components, it may be preferable to begin with this step, and avoid creating the very large UCM of step 1.
3. Map each fragment into a partial LQN model called a *"component-path submodel"*, representing the system behaviour for that fragment.
4. For each component, create a *component sub-model*, as follows:
   4.1 Collect the component-path sub-models into one sub-model with all the tasks together, ignoring the fact that a task may be represented more than once;
   4.2 Where a task is represented more than once, unify the instances into a single task with all the entries;
   4.3 In a single task, if there are two entries that make the same demands, and have the same interactions with other entries, these entries can be merged into one entry handling the requests of both. This rule may have to be applied recursively, because when entries are merged then other mergings may become possible.
5. Compose the component sub-models into the *system model*, connecting them by calls (asynchronous by default) where a scenario crosses from one component to another. This gives the performance model for the system.
   5.1 If the system-wide analysis shows that the transfer of control from one component to another is by a synchronous call, then the call between components is synchronous instead. This is the case in Figure 1, for the calls between component C-A and C-B. It must be inferred from the UCM; automated analysis can be applied to the system-wide UCM [Petriu02].

The assembling and composition in steps 4 and 5 can, in principle, be applied recursively at multiple levels of component decomposition.

In the BIP-type systems described above, the components are all closely related to each other and their local component-path scenarios are all variations inherited from a relatively small number of scenario classes. In BIP systems a library of sub-model classes can be created with one for each scenario class. The component-path sub-models then instantiate these classes.

Strategy B, for Behaviour-Inheriting Peer (BIP) systems:
In Strategy A, replace steps 1,2,3 by

1. Analyze the software objects and their behaviour within each component to identify the types of component-path scenarios, and define them as classes of scenarios. For each one, create an LQN sub-model class. Identify parameters that may be different in different instantiations of a component-path scenario. Parameters may include the identity of software elements and system nodes.
2. From the system specification and the role and workload of each component, identify its component-path scenarios by class, and their parameters if any.
3. Generate the set of component sub-models as instances from the sub-model classes.

Then follow steps 4 and 5 for Strategy A.

The case study which follows will be modeled using strategy A for a particular configuration, to explain the ideas, and also discusses how the library of sub-model classes is created. Then the tool created for modeling the family of systems uses Strategy B, to cover the wide range of variations that are possible.

4. Case study on CGNet

In this section we illustrate the approach described above by applying it to CGNet, which is an example of a BIP system. CGNet [Hobbs01] is a network emulator which includes routers (nodes), sources (generators) and destinations (sinks) for the traffic, and is configured with a connection topology with stated link capacities. There is a generator for each node, which sends it packets for different destinations at pre-configured rates. The packets traverse the path specified in the routing table to a traffic sink, which consumes the packets.

Here we treat a router node as a component; the generator is modeled as an arrival process and the sink is modeled by a dummy "user" task.

Every node has the same operations: the main thread receives packets from the incoming sockets, parses packets and switches them to the outgoing queues. The sending or sinking thread sends packets to outgoing sockets; the sending thread also emulates the network interface delay for the link.

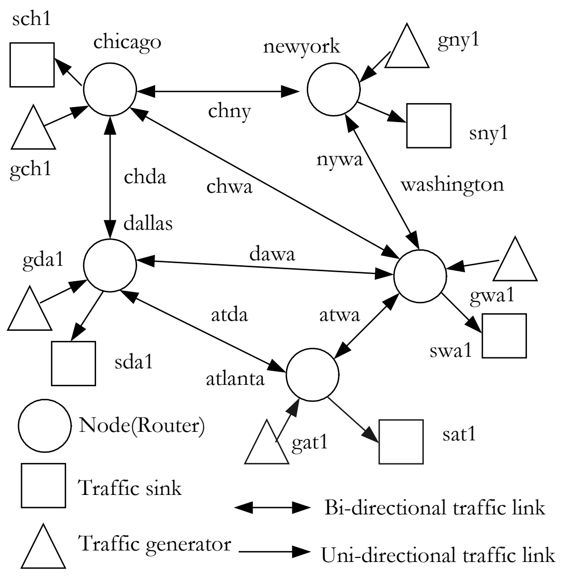


Figure 2. Topology for one configuration with 5 nodes

The case study will consider a configuration of CGNet with five nodes to explain the compositional model-building approach. The topology is shown in Figure 2. Data packets start from a generator (triangle) and end at a sink (square). They traverse the routers (circles) along the paths defined in the routing table.

The traffic is made up of classes, according to the route followed. For the packet class named as (XX, ZZ), node XX is the *source* router that is connected to its generator; node ZZ is the *destination* router that is connected to the traffic sink. XX and ZZ will be replaced by the two-letter names of the router nodes shown in Figure 2. There may also be *forwarding* routers along the path between XX and ZZ; we will designate such a router YY. Each packet class has its own scenario; Figure 3 shows the path view [Woodside95b] of the scenario for class (AT, CH), showing the path and responsibilities described above overlaid on the components shown in Figure 2.

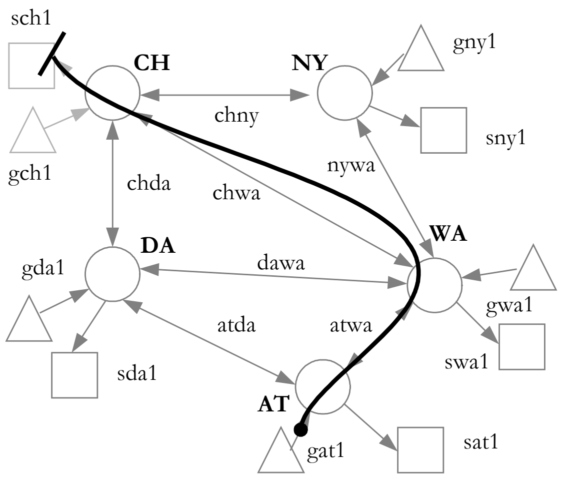


Figure 3. Path of scenario for packet class (AT, CH)

The components in the analysis are the router nodes, and the path fragments will describe the handling of each packet class (with a different fragment for each packet class traversing the node). Figure 4 shows the three fragments of the system-wide scenario for packet class (CH, AT), as it traverses nodes CH, WA and AT. It also shows the responsibilities *receive, switch, send* (or send-to-sink) and *delay* (for emulating network delay) within each scenario fragment. They are labelled *rcv, sw, snd, del* respectively.

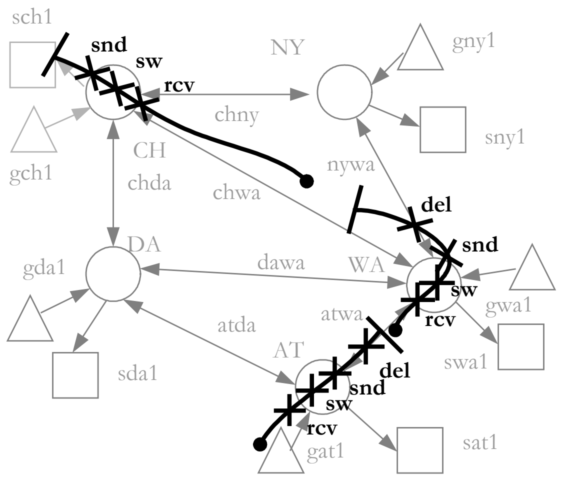
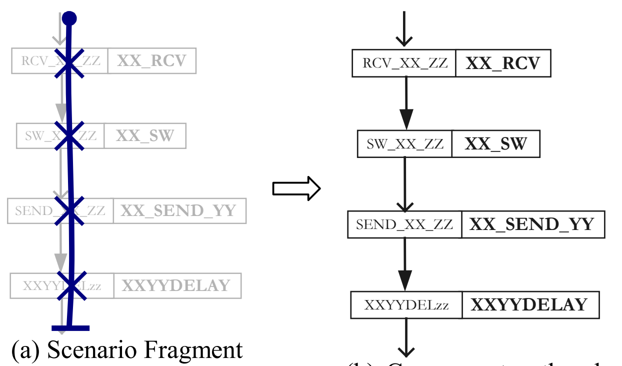


Figure 4. Scenario fragments for packet class (AT, CH)

For further analysis we have defined three different roles for the component-path scenario fragments. They are *source, forwarding,* and *destination* roles. The role will affect which sub-model class is instantiated, for the component sub-path.

Every component sub-path does the same overall operation, to handle a packet. There are three variations, which we will call *handleXX, handleYY and handleZZ*, for the handling done by a source, forwarding and destination node respectively. Each of these is a component-path class, and they give the sub-model classes *smXX, smYY,* and *smZZ*.

They can be instantiated by providing node names to replace XX, YY and ZZ. As a first example, Figure 5 shows *handleXX* as a UCM fragment and *smXX* as a LQN submodel. This class has a parameter YY, which is the name of the next node in the route followed by packet class (XX, ZZ) from node XX. Notice that one node has component-path fragments with different roles, for classes that are originated, forwarded or have their final destination at that node.



(a) Scenario Fragment *handleXX* for a source fragment

(b) Component path sub model class *smXX*

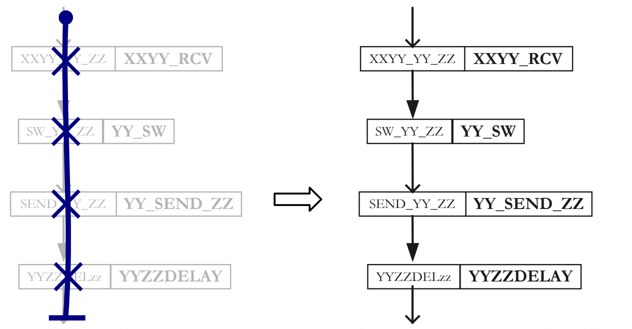
Figure 5. Scenario fragment and sub-model class for a source node XX and a packet class with next node YY.

The sub-model structure is obtained partly from analysis of the code. There is a thread for receiving packets, and a thread for each outgoing interface, and these threads are modeled by LQN tasks. The switching and routing is also modeled by a task, which later will be aggregated into Receive. When it receives a packet the software reads it from a socket, so in *smXX*, the receiving task responds to an asynchronous call. Once the packets are received to the workspace, the switching procedure performs switching/routing according to the routing table before another packet can be received. Thus the call from receiving task to the switching task in *smXX* is synchronous. Once the packet is stored in the outgoing queue, the receiving and switching tasks are no longer blocked, so the entry in the switching task makes an asynchronous call to the entry of the sending task. In the program, the sending thread triggers an emulation of the network delay and is blocked during this time, so a synchronous call is used from the sending task to the delay task, which represents an operating system timer. The sending thread writes the packets to the outgoing socket and does not wait for a reply from the next hop. So from the network delay task to the receiving task of the next component the call is asynchronous. Thus the requests between components (nodes) are all asynchronous.

For a component-path fragment with a forwarding role the operations are identical, as shown in Figure 6 for operations at node YY to forward packet class (XX, ZZ). The only difference is that on instantiation, the input is bound to a predecessor node in the path instead of to the local packet generator. For a destination role, there is a difference shown in Figure 7. The call from Switch goes to Sink rather than to Send, and there is no network delay emulation.

Applying the steps for node AT

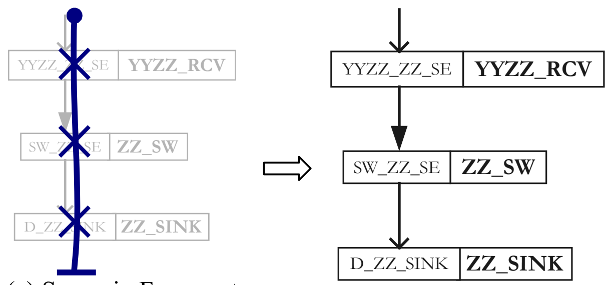
Consider the node Atlanta (AT) and the steps for deriving its component sub-model from the LQN submodel classes. Necessary information is retrieved from the routing tables in node AT and connected nodes DA and WA.



(a) Scenario Fragment *handleYY* for a forwarding fragment

(b) Component path sub model class *smYY*

Figure 6. Scenario fragment and sub-model class for a forwarding node YY and a packet class with next node ZZ



(a) Scenario Fragment *handleZZ* for a destination fragment

(b) Component path sub model class *smZZ*

Figure 7. Scenario fragment sub-model class for a destination node ZZ and packet class with previous node YY.

First we consider packet classes that originate at AT. From the routing table in node AT we find it is the source of 4 packet classes (AT, CH), (AT, DA), (AT, NY) and (AT, WA). Class (AT, DA) chooses node DA as its next hop, while all the others choose node WA as next hop. We create four instances of the component-path sub-model class *smXX* in Figure 5, replacing XX by AT, ZZ by the destination of the class, and YY by the node which is next in each route. This gives four component-path sub-models for AT.

Second, we consider packet classes that terminate at AT. The nodes connected to AT are DA and WA. From the *handleZZ* and *smZZ* class definitions in Figure 7 it is clear that the source of the packet class has no effect on the processing of a packet coming from either one, so just two

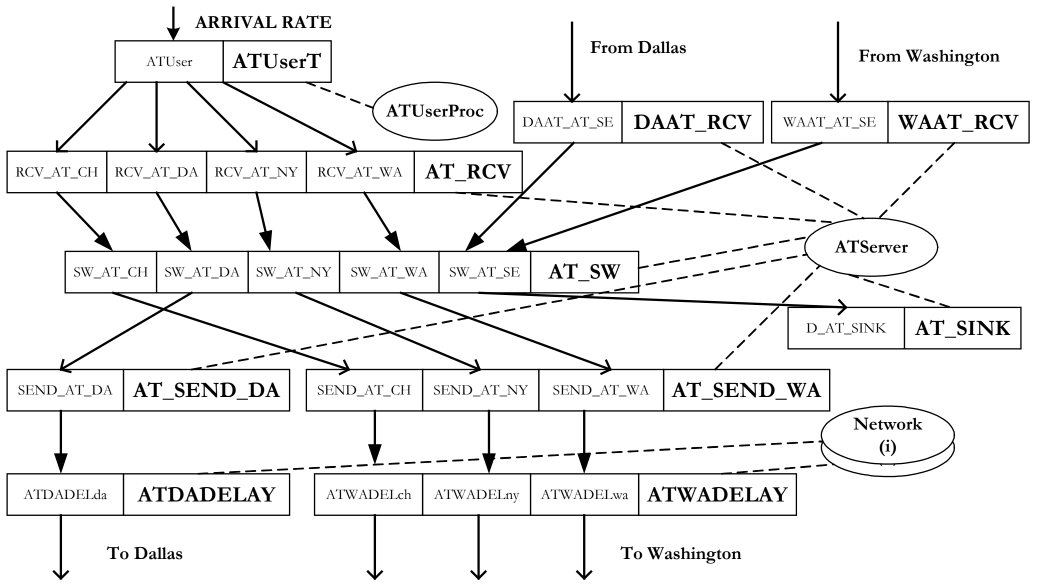


Figure 8. The component sub-model *nodeAT* for node Atlanta

component-path submodels are created from the class in Figure 7, one with YY replaced by DA, and one with YY replaced by WA.

Third, we can consider classes which are forwarded by AT, however in this configuration there are none. If there were, the LQN sub-model class in Figure 6 would be instantiated.

The resulting collection of sub-models are then merged together using step 4 of Strategy A, to give the component sub-model for the Atlanta node shown in Figure 8. The merging of tasks and entries is done as follows. (1) All packet classes coming from the generator wait in a queue located in the same incoming socket. Step 4.2 merges all the entries of the receiving task coming from the generator. (2) One switching task is created for each incoming socket, combining the entries that handle its packet classes, in step 4.2. (3) Because node AT does not take account of the source node origins of packet classes (XX, AT), the entries for all classes ending at AT are merged into one entry in the switching task by step 4.3. The same reason gives one entry in the sinking task that sends them to the sink at AT. (4) The routing table in AT indicates that packet classes with the destinations CH, NY, and WA should go through the link ATWA. All these packet classes should wait in the same outgoing queue for the outgoing socket. Step 4.2 creates one sending and network delay task for these classes.

This gives the sub-model nodeAT for the Atlanta node, shown in Figure 8. The complete network node includes the router node component nodeAT, the processor ATServer and a user pseudo task ATuserTask to receive packets from local generator. The next step is to combine the node sub-models into a system model.

To create a complete system model, the component sub-models for the nodes must be joined together. For this step it is useful to define a high-level view of the components seen from outside, showing their interfaces. Figure 9 shows nodeAT as a box with plug-in points (circles in boxes) for the interfaces (input interfaces at the top, and output at the bottom). The interfaces are labelled as <link, list of classes>, where the link is named for the two nodes that it connects, in the order source-destination. The link gAT is the link from the generator to the node, and its classes are all traffic originating at AT. The input and output interfaces connected to another node are shown separately, even though they are provided by the same socket in the prototype.

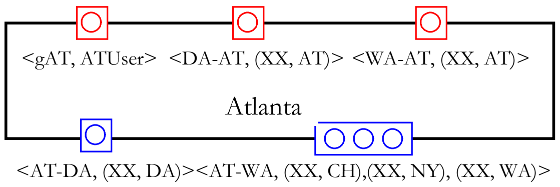


Figure 9. The labeled high-level component for *nodeAT* for performance model

By the same procedures we can obtain component sub-models for all nodes, and they can be shown in the same way. Then their interconnection is illustrated in Figure 10.

To construct the overall LQN model, calls are inserted between the component sub-models. Where an output interface is connected to an input interface, the call made from the sending object is merged with the call received by the receiving object. The interactions that are merged must be of the same type (here, we have seen that they are all asynchronous). The system-wide LQN model is too complex to show here.

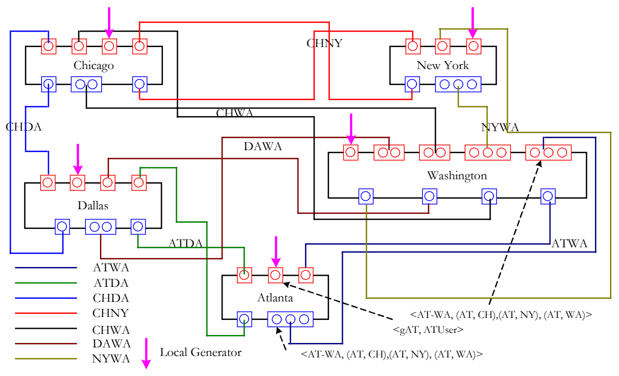


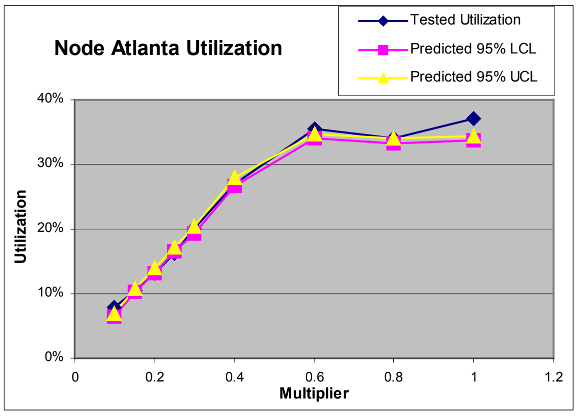
Figure 10. A high-level model for the network

An LQN model is incomplete without processor execution demand parameters, giving the demand for each call to an entry. In this case we used the fact that each entry call is associated with operations on one data packet. It is straightforward to measure the total execution demand at a node and allocate it per packet. However each packet is handled by two concurrent threads, one to receive and switch, and one to send and emulate network delay. It would be better to estimate these separately, to see if the nodes are bottlenecked at receive or at send; this can affect the location of buffer overflow.

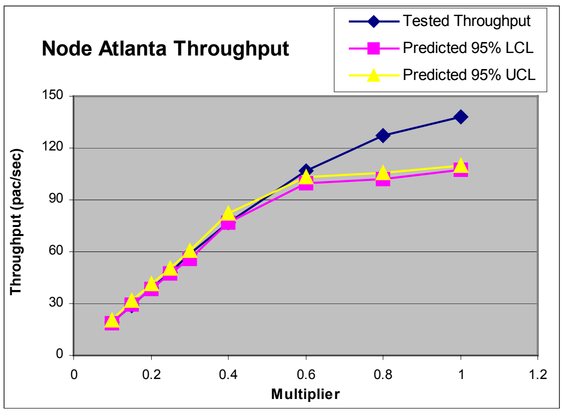
5. Experiments on the CGNet model

Experiments were performed with each node running on a separate Unix workstation, all of the same type. The total CPU time for all packets handled at each node was recorded, and the number of packets that arrived and were sent (they were not the same when buffers overflowed). The CPU time for each node was split into amounts for sending and receiving by fitting parameters, using least squares regression (see e.g. [Scheaffer86] for a discussion of regression, and [Wu03b] for details of its application). The resulting execution demand of receiving and switching is 0.0016 sec, and for sending or sinking it is 0.0018 sec. The network delay emulation derived from configuration files was taken to place execution demand on the network processor. Overhead and message handling execution are included in the two parameters.

When the model is compared to measurements, it gives results for utilization shown in Figure 11(a) and for throughput, shown in Figure 11(b). For the model the confidence interval is plotted as (LCL, UCL) and all the predicted values lies in this confidence limits. The horizontal axis shows a multiplier on the workload intensity, which was defined by a profile of arrival rates for different classes. The agreement is good when the workload is low (no packet loss), and less good for the



(a) Utilization of node Atlanta by prediction and measurement against *multiplier* of workload



(b) Throughput of node Atlanta by prediction and measurement against *multiplier* of workload

Figure 11. Predicted vs measured performance for node AT

cases where there is packet loss. The model solutions bysimulation appear to drop packets a bit differently from the real system CGNet; this is being investigated. The full results are beyond the scope of this paper, but this sample indicates that (with this limitation in representing loss) the strategy generates a realistic model. Also, in cases that were simple enough to generate models by hand, the strategy gave the same model.

CGNet can be configured in many different ways, with different numbers of nodes and different connections and emulated link capacities. The configuration and its workload and routing tables are all driven by data which has been used as input to a Converter Tool, implemented to automate the process of model generation. As illustrated in Figure 12, it uses the configuration data to create the model structure with the compositional approach, and then adds the (assumed) known parameters for the execution demands.

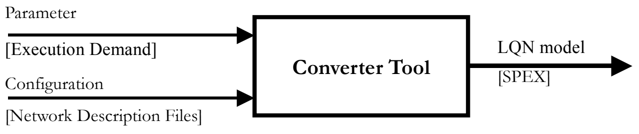


Figure 12. The overall approach to building the LQN model from the CGNet configuration

6. Conclusions

An approach has been defined for building performance models of complex systems with non-repetitive structure, but with components based on the same software objects and executing a few strongly related behaviours. These were called BIP (Behaviour-Inheriting Peer) systems. The model is created by instantiating sub-models that represent classes of behaviour, and then by composing them in two stages, first to create a submodel for each component, and then to create the overall system model. This approach can model arbitrarily large configurations without the need to program the simulations. For CGNet, a prototype tool to emulate a network of routers, a Converter tool was developed to automate this process. From the data required to configure CGNet, a performance model can be created also. The accuracy of the CGNet model appears to be good, except when it is limited by the modeling platform's handling of packet losses.

The Converter tool has achieved, for models of systems generated from CGNet, the goal of configuration-driven modeling described in [Wu03a] [Bertolino03]. The Converter tool can build a performance model for a CGNet configuration of any size, for essentially zero additional effort by the analyst. The effort necessary to support this capability is to create the behaviour and path sub-model classes, and to calibrate the demand parameters. If the software evolves, these model aspects must be maintained.

The same strategy should apply equally well to model other kinds of systems with strongly related components, such as peer-to-peer application systems, and grid systems. Submodels created by the BIP strategy described here can be freely combined with other submodels that are created by other means, to give a powerful and unconstrained modeling capability.

Acknowledgement

This research was supported by NSERC, the Natural Sciences and Engineering Research Council of Canada, and Nortel Networks, through the OCRI-NSERC Industrial Research Chair in Performance of Real-time Software.

References

[Bertolino03] A. Bertolino and R. Mirandola "*Towards Component-Based Software Performance Engineering*", Proc CBSE6 - 6th Workshop on Component-Based Software Engineering, part of the Int. Conf. on Software Engineering (ICSE 2003), Portland, Oregon, May 3- 4, 2003.

[Bolch98] G. Bolch, S. Greiner, H. de Meer, K.S. Trivedi "*Queueing Networks and Markov Chains*" Wiley, 1998.

[Buhr96] R. J. A. Buhr, R. S. Casselman "*Use Case Maps for Object-oriented Systems*" Prentice Hall, 1996.

[Ciardo89] G. Ciardo, K. S. Trivedi, and J. Muppala, "*SPNP: stochastic Petri net package*", Proc. Third Int. Workshop on Petri Nets and Performance Models (PNPM'89) Kyoto, Japan, 1989.

[Cortellessa00] V. Cortellessa and R. Mirandola, "*Deriving a Queueing Network Based Performance Model from UML Diagrams*" Proc. 2nd Int. Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000.

[Franks00] G. Franks, "*Performance Analysis of Distributed Server Systems*", Report OCIEE-00-01, Ph. D. Thesis, Carleton University, Ottawa, Canada, Jan 2000.

[Hobbs01] C. Hobbs, G. Young, "*CGNet: A User's guide & designer's manual*", Private communication, Jun 2001

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. "*Object-Oriented Software Engineering: A Use Case Driven Approach*" Addison-Wesley, 1993

[Jain91] R. Jain, "*The Art of Computer Systems Performance Modeling Analysis*", John wiley & Sons, 1991.

[Neilson95] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, "*Software Bottlenecking in Client-Server Systems and Rendezvous Networks*", IEEE Trans. on Software Engineering, v. 21, n 9, Sep 1995.

[Petriu02] D. Petriu, M. Woodside, "*Software Performance Models from System Scenarios in Use Case Maps*", Proc. 12 Int. Conf. on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002), London, UK, April 2002

[Rolia95] J.A. Rolia, K.C. Sevcik,"*The Method of Layers*", IEEE Trans. on Software Engineering, Vol. 21 No. 8, Aug 1995

[Scheaffer86] R.L. Scheaffer, J.T. McClave "*Probability and Statistics for Engineers*" 2nd Edition, Duxbury Press, 1986.

[Szyperski98] C. Szyperski, "*Component Software; Beyond Object-Oriented Programming*", Addison-Wesley, 1998

[Smith90] C.U. Smith, "*Performance Engineering of Software Systems*", Addison-Wesley, 1990.

[Smith02] C. U. Smith and L. G. Williams, "*Performance Solutions*", Addison-Wesley, 2002.

[Woodside95a] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "*The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software*", *IEEE Trans. on Computers*, v 44, n 1, Jan 1995

[Woodside95b] C.M. Woodside, "*A Three-View Model for Performance Engineering of Concurrent Software*", *IEEE Trans. on Software Engineering*, Vol. 21, No. 9, Sept. 1995.

[Wu03a] X. Wu, D. McMullan, M. Woodside, "*Component Based Performance Prediction*", Proc CBSE6 - 6th Workshop on Component-Based Software Engineering, part of the Int. Conf. on Software Engineering (ICSE 2003), Portland, Oregon, May 3- 4, 2003.

[Wu03b] P. Wu, "*A Performance Model for a Network of Prototype Software Routers*", Master's Thesis, Carleton University, Ottawa, Canada, August 2003.