Layered Queueing Network Solver and Simulator User Manual

Greg Franks Peter Maly Murray Woodside Dorina C. Petriu Alex Hubbard
Martin Mroz

Department of Systems and Computer Engineering
Carleton University
Ottawa ON K1S 5B6
{cmw,greg}@sce.carleton.ca

January 31, 2023

Revision: 12086

Contents

List of Figures

List of Tables

Abstract

The Layered Queuing Network (LQN) model is a canonical form for extended queueing networks with a layered structure. The layered structure arises from servers at one level making requests to servers at lower levels as a consequence of a request from a higher level. LQN was developed for modeling software systems, but it applies to any extended queueing network with multiple resource possession, in which multiple resources are held in a nested fashion.

This document describes the elements found in Layered Queueing Network Model, the results produced when a LQN model is solved, and the input and output file formats. It also describes the method used to invoke the analytic and simulation solvers, and the possible errors that can arise when solving a model. The reader is referred to "Tutorial Introduction to Layered Modeling of Software Performance" [21] for constructing models.

Chapter 1

The Layered Queueing Network Model

Figure 1.1 illustrates the LQN notation with an example of an on-line e-commerce system. In an LQN, software resources are all called “tasks”, have queues and provide classes of service which are called “entries”. The demand for each class of service can be specified through “phases”, or for more complex interactions, using “activities”. In Figure 1.1, a task is shown as a parallelogram, containing parallelograms for its entries and rectangles for activities. Processor resources are shown as circles, attached to the tasks that use them. Stacked icons represent tasks or processors with multiplicity, making it a multiserver. A multiserver may represent a multi-threaded task, a collection of identical users, or a symmetric multiprocessor with a common scheduler. Multiplicity is shown on the diagram with a label in braces. For example there are five copies of the task ‘Server’ in Figure 1.1.

Entries and activities have directed arcs to other entries at lower layers to represent service requests (or messages)[1]. A request from an entry or an activity to an entry may return a reply to the requester (a synchronous request, or *rendezvous*) indicated in Figure 1.1 by solid arrows with closed arrowheads. For example, task Administrator makes a request to task BackorderMgr who then makes a request to task InventoryMgr. While task InventoryMgr is servicing the request, tasks BackorderMgr and Administrator are blocked. A request may be forwarded to another entry for later reply, such as from InventoryMgr to CustAccMgr. Finally a request may not return any reply at all (an asynchronous request or *send-no-reply*, shown as an arrow with an open arrow head, for example, the request from task ShoppingCart to CustAccMgr.

The first way that the demand at entries can be specified is through phases. The parameters of an entry are the mean number of requests for lower entries (shown as labels in parenthesis on the request arcs), and the mean total host demand for the entry (in units of time, shown as a label on the entry in brackets). An entry may continue to be busy after it sends a reply, in an asynchronous “second phase” of service [7] so each parameter is an array of values for the first and second phase. Second phases are a common performance optimization, for example for transaction cleanup and logging, or delayed write operations.

The second way that demand can be specified is through activities. Activities are the lowest level of granularity in a performance model and are linked together in a directed graph to indicate precedence. When a request arrives at an entry, it triggers the first activity of the activity graph. Subsequent activities may follow sequentially, or may fork into multiple paths which later join. The fork may take the form of an ‘AND’ which means that all the activities on the branch after the fork can run in parallel, or in the form of an ‘OR’, which chooses one of the branches with a specified probability. In Figure 1.1, a request that is received by entry “SCE3” of task “ShoppingCart” is processed using an activity called “SCE3A95” that represents the main thread of control, then the main thread is OR-Forked into two branches, one of which is later AND-forked into three threads. The three threads, starting with activities ‘AFBA109’, ‘AFBA130’ and ‘AFBA133’ respectively, run in parallel. The first thread replies to the entry through activity ‘OJA110’ then ends. The remaining two threads join into one thread at activity ‘AJA131’. When both ‘OJA110’ and ‘AJA131’ terminate, the task can accept a new request.

The holding time for one class of service is the entry service time, which is not a constant parameter but is determined by its lower servers. Thus the essence of layered queuing is a form of simultaneous resource possession. In software systems delays and congestion are heavily influenced by synchronous interactions such as remote procedure

[1] requests may jump over layers, such as the request from the Administrator task to the InventoryMgr task.

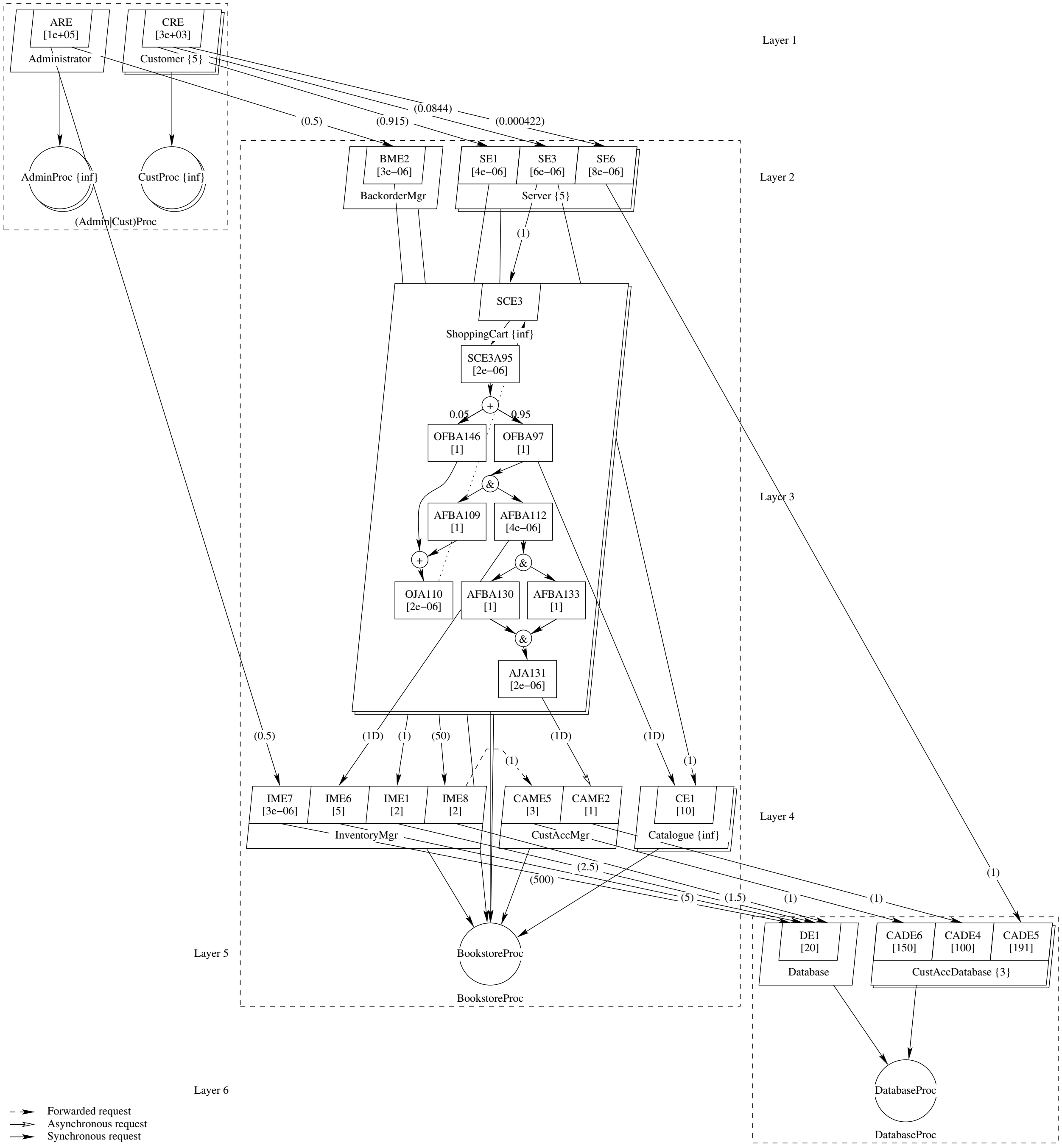


Figure 1.1: Notation

calls (RPCs) or rendezvous, and the LQN model captures these delays by incorporating the lower layer queueing and service into the service time of the upper layer server. This “active server” feature [20] is the key difference between layered and ordinary queueing networks.

1.1 Model Elements

Figure 1.2 shows the *meta-model* used to describe Layered Queueing Networks. This model is unique in that it is more closely aligned with the architecture of a software system that it is with a conventional queueing network model such as Performance Model Interchange Format (PMIF) [17, 19]. The latter consists of stations with queues and visits, whereas a LQN has processors, tasks and requests.

A Layered Queueing Network is a directed graph. Nodes in the graph consist of tasks, processors, entries, activities, and precedence. Arcs in the graph consist of requests from one node to another. The model objects are described below.

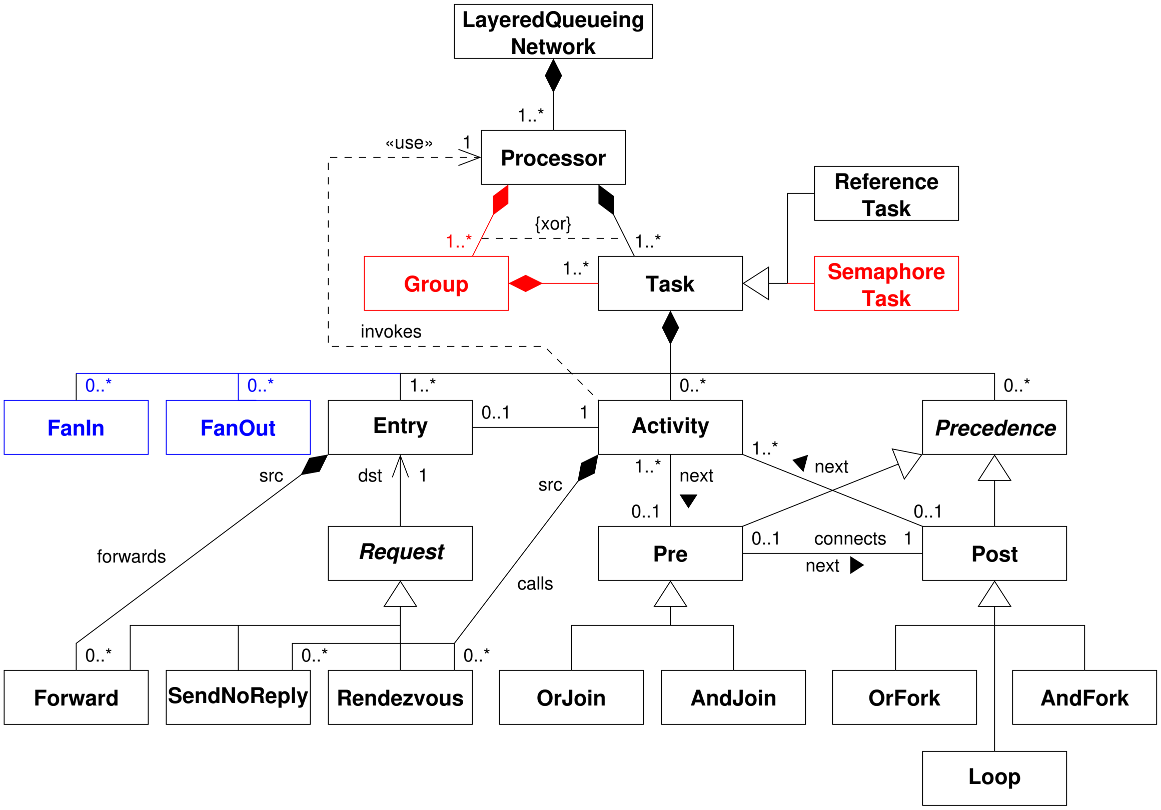


Figure 1.2: LQN Meta Model

1.1.1 Processors

Processors are used by the activities within a performance model to consume *time*. They are *pure servers* in that they only accept requests from other servers and clients. They can be actual processors in the system, or may simply be place holders for tasks representing customers and other logical resources.

Each processor has a single queue for requests. Requests may be scheduled using the following queueing disciplines:

FIFO First-in, first out (first-come, first-served). Tasks are served in the order in which they arrive.

PRI Priority, preemptive resume. Tasks with priorities higher than the task currently running on the processor will preempt the running task. Priorities range from zero to positive infinity, with a priority of zero being the highest. The default priority for all tasks is zero.

HOL Head-of-line priority. Tasks with higher priorities will be served by the processor first. Tasks in the queue will not preempt a task running on the processor even though the running task may have a lower priority.

PS Processor sharing. The processor runs all tasks "simultaneously". The rate of service by the processor is inversely proportional to the number of executing tasks. For *lqsim*, processor sharing is implemented as *round-robin* – a *quantum* must be specified.

INF Infinite (delay).

RAND Random scheduling. The processor selects a task at random.

CFS Completely fair scheduling [9]. Tasks are scheduled within groups using round-robin scheduling and groups are scheduled according to their share. A *quantum* must be specified. This scheduling discipline is implemented on the simulator only at present.

Each processor can have multiple cores, all of which are served by the common queue (see §1.2). The processor multiplicity must be a integer greater than zero or the special constant `@infinity`. If the multiplicity is set to `@infinity`, the processor is coerced to a delay server.

1.1.2 Groups

Groups[9] are used to divide up a processor's execution time up into *shares*. The tasks within a group divide the share up among themselves evenly. Groups can only be created on processors running the scheduling discipline *completely fair scheduling*,. .

Shares may either be *guaranteed* or *capped*. Guarantee shares act as a floor for the share that a group receives. If surplus CPU time is available (i.e., the processor is not fully utilized), tasks in a guaranteed group can exceed their share. Cap shares act as a hard ceiling. Tasks within these groups will never receive more than their share of CPU time.

Note: Completely fair scheduling is a form of priority scheduling. With layered models, calls made by tasks within groups to lower level servers can cause *priority inversion*. Cap scheduling tends to behave better than guaranteed scheduling for these cases.

1.1.3 Tasks

Tasks are used in layered queueing networks to represent resources. Resources include, but are not limited to: actual tasks (or processes) in a computer system, customers, buffers, and hardware devices. In essence, whenever some entity requires some sort of service, requests between tasks involved.

A task has a queue for requests and runs on a processor. Items are served from the queue in a first-come, first-served manner. Different classes of service are specified using *entries* (c.f. §1.1.4). Tasks may also have internal concurrency, specified using *activities* (c.f. §1.1.5).

Requests can be served using the following scheduling methods:

FIFO First-in, first out (first-come, first-served). Requests are served in the order in which they arrive. This scheduling discipline is the default for tasks.

PRI Priority, preemptive resume. Requests arriving at entries with priorities higher than entry that task is currently processing will preempt the execution of the current entry. Priorities range from zero to positive infinity, with a priority of zero being the highest. The default priority for all entries is zero.

INF Infinite (delay).

HOL Head-of-line priority. Requests arriving at entries with higher priorities will be served by the task first. Requests in the queue will not preempt the processing of the current entry by the task.

Each task can have multiple homogenous threads, all of which are served by the common queue (see §1.2). The task multiplicity must be a integer greater than zero or the special constant `@infinity`. If the multiplicity is set to `@infinity`, the task is coerced to a delay server.

The subclasses of *task* are:

Reference Task: Reference tasks are used to represent customers in the layered queueing network. They are like normal tasks in that they have entries and can make requests. However, they can never receive requests and are always found at the top of a call graph. They typically generate traffic in the underlying closed queueing model by making rendezvous requests to lower-level servers. Reference tasks can also generate traffic in the underlying open queueing model by making send-no-reply requests instead of rendezvous requests. However, open class customers are more typically represented using open arrivals which is simply encoded as a parameter to an entry.

Bursty reference tasks are a special case of reference tasks where the service time for the slices are random variables with a *Pareto* distribution (c.f. §1.1.5).

Semaphore Task: Semaphore tasks are used to model passive resources such as buffers. They always have two entries which are used to *signal* and *wait* the semaphore. The wait entry must be called using a synchronous request whereas the signal entry can be called using any type of request. Once a request is accepted by the wait entry, no further requests will be accepted until a request is processed by the signal entry. The signal and wait entries do not have to called from a common task. However, the two entries must share a common call graph, and the call graph must be deterministic. The entries themselves can be defined using phases or activies and can make requests to other tasks. Counting semaphores can be modeled using a multiserver.

Synch Task: Synchronization tasks are used... Cannot be a multiserver.

1.1.4 Entries

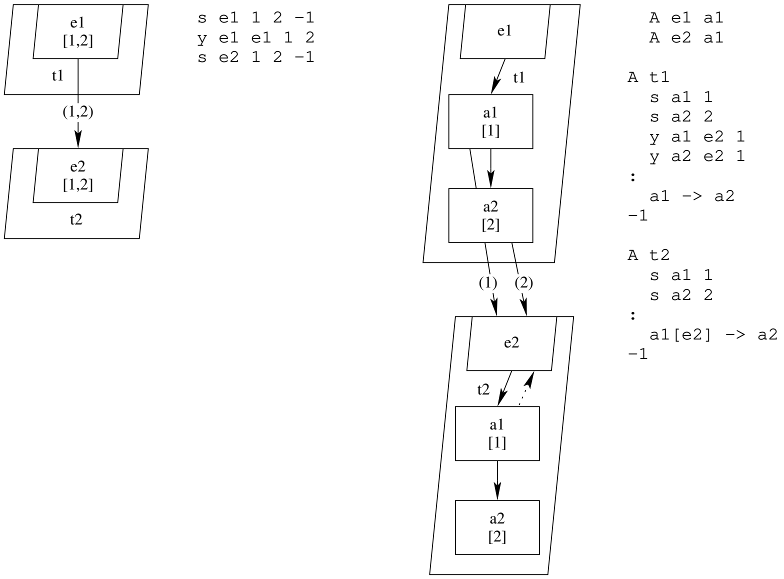
Entries service requests and are used to differentiate the service provided by a task. An entry can accept either synchronous, or asynchronous requests, but not both. Synchronous requests are part of the *closed* queueing model whereas asynchronous requests are part of the *open* model. Message types are described in Section 1.1.7 below.

Entries also generate the replies for synchronous requests. Typically, a reply to a message is returned to the client who originally sent the message. However, entries may also *forward* the reply. The next entry which accepts the forwarded reply may forward the message in turn, or may reply back to the originating client. For example, in Figure 1.1, entry ‘IME8’ on task ‘IventoryMgr’ forwards the request from entry ‘BME2’ on task ‘BackorderMgr’ to entry ‘CAME5’ on task ‘CustAccMgr’. The reply from ‘CAME2’ will be sent directly back to ‘BME2’.

The parameters for an entry can be specified using either phases or activities[2]. The activity method is typically used when a task has complex internal behaviour such as forks and joins, or if its behaviour is specified as an activity graph such as those used by Smith and Williams [18]. The phase method is simply a short hand notation for specifying a sequence of one to three activities, with the reply being generated by the first activity in the sequence. Figure 1.3 shows both methods for specifying a two-phase client calling a two-phase server.

Regardless of the specification method used for an entry, its behaviour as a server to its clients is by *phase*, shown in Figure 1.4. Phases consume time on processors and make requests to entries. Phase one is a *service phase* and is similar to the service given by a station in a queueing network. Phase one ends after the server sends a reply. Subsequent phases are *autonomous* phases which are launched by phase one. These phases operate in parallel with the clients which initiated them. The simulator and analytic solver limit the number of phases to three.

[2] The meta-model in Figure 1.2 only shows activities, phases are a notational short-hand.



```
s e1 1 2 -1
y e1 e1 1 2
s e2 1 2 -1
```

```
  A e1 a1
  A e2 a1

A t1
  s a1 1
  s a2 2
  y a1 e2 1
  y a2 e2 1
:
  a1 -> a2
-1

A t2
  s a1 1
  s a2 2
:
  a1[e2] -> a2
-1
```

(a) Phases (b) Activities

Figure 1.3: Entry Specification

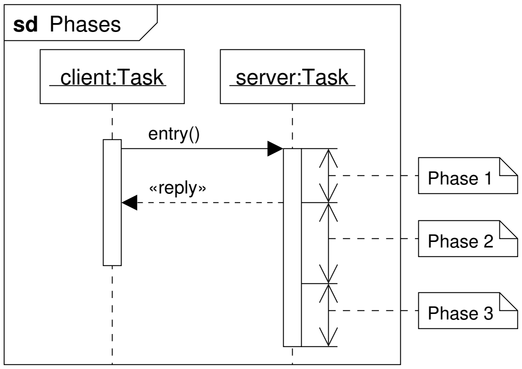


Figure 1.4: Phases for an Entry.

1.1.5 Activities

Activities are the lowest-level of specification in the performance model. They are connected together using "Precedence" (c.f. §1.1.6) to form a directed graph to represent more than just sequential execution scenarios.

Activities consume time on processors. The *service time* is defined by a mean and variance, the latter through *coefficient of variation squared* [3]. The service time between requests to lower level servers is assumed to be exponentially distributed (with the exception of *bursty reference tasks*) so the total service time is the sum of a random number of exponentially distributed random variables.

Activities also make requests to entries on other tasks. The distribution of requests to lower level servers is set by the *call order* for the activity which is either *stochastic* or *deterministic*. If the call order is deterministic, the activity makes the exact number of requests specified to the lower level servers. The number of requests is integral; the order of requests to different entries is not defined. If the call order is stochastic, the activity makes a random number of requests to the lower level servers. The mean number of requests is specified by the value specified. Requests are assumed to be geometrically distributed.

For entries which accept rendezvous requests, replies must be generated. If the entry is specified using phases, the reply is implicit after phase one. However, if the entry is specified using activities, one or more of the activities must explicitly generate the reply. Exactly one reply must be generated for each request.

Slices

Activities consume time by making requests to the processor associated with the task. The service time demand specified for an activity is divided into *slices* between requests to other entries, shown in the UML Sequence Diagram in Figure 1.5. The mean number of slices is always $1 + Y$ where Y is total total number of requests made by the activity.

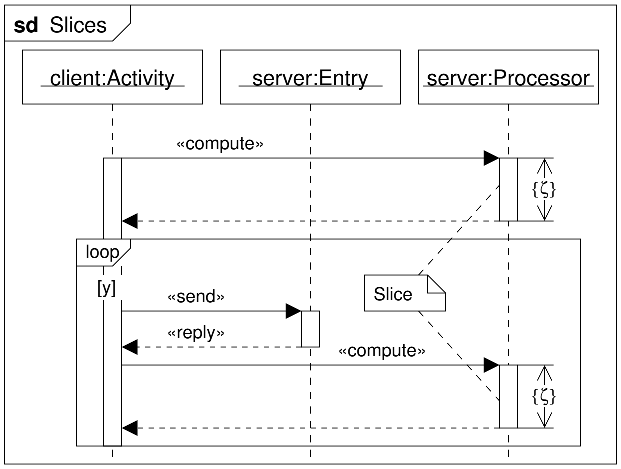


Figure 1.5: Slices. The *slice time* is shown using the label ζ.

By default, the demand of a *slice* is assumed to be exponentially distributed [20] but a variance may be specified through the *coefficient of variation squared* ($cv^2 = \sigma^2/\bar{s}^2$) parameter for the entry or activity. The method used to solve the model depends on the solver being used:

Analytic Solver: All servers with $cv^2 \neq 1$ use the HVFCFS MVA approximation from [13].

Simulator: The simulator uses the following distributions for generating random variates for slice times provided that the task is *not* a bursty reference task.

$\boldsymbol{cv^2 = 0}$**:** deterministic.

[3] The squared coefficient of variation is variance divided by the square of the mean.

$0 < \boldsymbol{cv}^2 < 1$**:** gamma.

$\boldsymbol{cv}^2 = 1$**:** exponential.

$\boldsymbol{cv}^2 > 1$**:** bizarro...

If the task is a bursty reference task, then the simulator generates random variates for slice times according to the Pareto distribution. The scale $x_m > 0$ and shape $k > 0$ parameters for the distribution are derived from the service time s and coefficient of variation squared cv^2 parameters for the corresponding activity (or phase).

$$\begin{aligned} k &= \sqrt{\frac{1}{cv^2} + 1} + 1 \\ x_m &= s \times \frac{(k-1)}{k} \end{aligned}$$

On-off behaviour can simulated by using two or more phases at the client, where on phase corresponds to the on period and makes requests to other servers, while the other phase corresponds to the off period.

1.1.6 Precedence

Precedence is used to connect activities within a task to from an *activity graph*. Referring to Figure 1.2, precedence is subclassed into '**Pre**' (or *'join'*) and '**Post**' (or *'fork'*). To connect one activity to another, the source activity connects to a *pre*-precedence (or a *join*-list). The *pre*-precedence then connects to a *post*-precedence (or a *fork*-list) which, in turn, connects to the destination activity. Table 1.1 summarizes the precedence types.

Name	Icon	Description
Sequence	│	Transfer of control from an activity to a join-list.
And-Join	↘ ↙ &	A Synchronization point for concurrent activities.
Quorum-Join	↘ ↙ n	A Synchronization point for concurrent activities where only n branches must finish.
Or-Join	↘ ↙ +	
Sequence	↓	Transfer of control from fork-list to activity
And-Fork	& ↙ ↘	Start of concurrent execution. There can be any number of forked paths.
Or-Fork	+ p ↙ ↘ $1-p$	A branching point where one of the paths is selected with probability p. There can be any number of branches.
Loop	* n_1 ↙ n_2 ↓ ↘	Repeat the activity an average of n times.

Table 1.1: Activity graph notation.

The semantics of an activity graph are as follows. For AND-forks, AND-joins and QUORUM-joins, each branch of a join must originate from a common fork, and each branch of the join must have a matching branch from the fork. Branches from AND-forks need not necessarily join, either explictly by a "dangling" thread not participating in a join, or implicitly through a quorum join, where only a subset of the branches must join while ignoring the rest. However, all threads started by a fork must terminate before the task will accept a new message (i.e., there is an implied join collecting all threads at the end of a task's cycle). Branches to an AND-join do not necessarily have to originate from a fork – for this case each branch must originate from a unique entry. This case is used to synchronize two or more clients at the server.

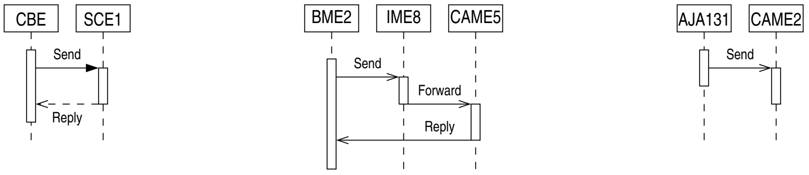
For OR-forks, the sum of the probabilities of the branches must sum to one – there is no "default" operation. AND-forks may join at OR-joins. The threads from the AND-fork implicitly join when the task cycle completes. OR-joins may be called directly from entries. This case is analogous to running common code for different requests to a task.

LOOPs consist of one or more branches, each of which is run a random number of times with the specified mean, followed by an optional deterministic branch exit which is followed after all the looping has completed.

Replies can only occur from activities in *pre*-precedence (*and*-join) lists. Activities cannot reply to entries from a loop branch because the number of times that a branch is executed is a random number.

1.1.7 Requests

Service requests from one task to another can be one of three types: rendezvous, forwarded, and send-no-reply, shown in Figure 1.6. A rendezvous request is a blocking synchronous request – the client is suspended while the server processes the request. A send-no-reply request is an asynchronous request – the client continues execution after the send takes place. A forwarded request results when the reply to a client is redirected to a subsequent server which, may forward the request itself, or may reply to the originating client.

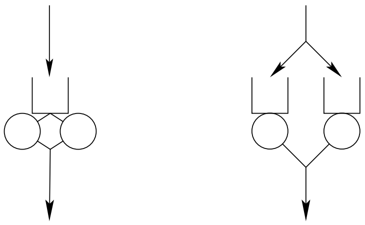


(a) Rendezvous (b) Forwarding (c) Send-no-reply

Figure 1.6: Request Types.

1.2 Multiplicity and Replication

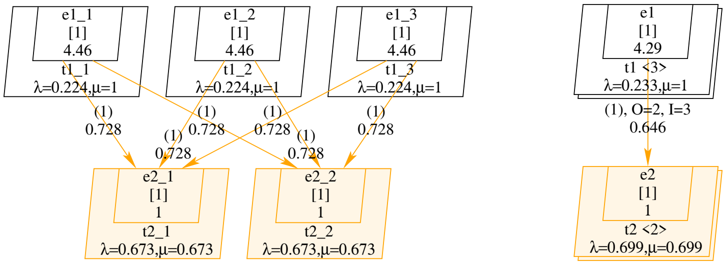
One common technique to improve the performance of a system is to add copies of servers. The performance model supports two techniques: multiplicity and replication. Multiplicity is the simpler technique of the two as a single queue is served by multiple servers. Replication requires a more elaborate specification because the queues of the servers are also copied, so requests must be routed to the various queues. Multi-servers can be replicated. Figure 1.7 shows the underlying queueing models for each technique.



(a) Multi-server (b) Replicated

Figure 1.7: Multiple copies of servers.

Replication reduces the number of nodes in the layered queueing model by combining tasks and processors with identical behaviour into a single object, shown in Figure 1.8. The left figure shows three identical clients making requests to two identical servers. The right figure is the same model, but specified using replication. Labels within angle brackets in Figure 1.8(b) denote the number of replicas.



(a) Flat

(b) Replicated

Figure 1.8: Replicated Model

Replication also introduces the notion of *fan-in* and *fan-in*, denoted using the `O`=*n* and `I`=*n* labels on the request from t1 to t2 in Figure 1.8(b). Fan-out represents the number of replicated servers that a client task calls. Similarly, fan-in represents the number of replicated clients that call a server. The product of the number of clients and the fan-out to a server must be the same as the product of the number of servers and the fan-in to the server. Further, both fan-in and fan-out must be integral and non-zero.

The total number of requests that a client makes to a server is the product of the mean number of requests and the fan-out. If the performance of a system is being evaluated by varying the replication parameter of a server, the number of requests to the server must be varied inversely with the number of server replicas in order to retain a constant number of requests from the client.

1.3 A Brief History

LQN [6] is a combination of Stochastic Rendezvous Networks [20] and the Method of Layers [14].

Chapter 2

Results

Both the analytic solver and the simulator calculate:

- throughput bounds (lqns only),
- mean delay for rendezvous and send-no-reply requests,
- variances for the rendezvous and send-no-reply request delays (lqsim only),
- mean delay for joins,
- entry service times and variances,
- distributions for the service time lqsim
- task throughputs and utilizations,
- processor utilizations and queueing delays.

Figure 2.1 shows some of these results for the model shown in Figure 1.1, after solving the model analytically using *lqns(1)*. The interpretation of these results are describe below in Section 2.1.2.

Results can be saved in three different formats:

1. in a human-readable form.
2. in a "parseable" form suitable for processing by other programs. The grammar for the parseable output is described in Section A on page 99.
3. in XML (again suitable for by processing by other programs). The schema for the XML output is shown in Section 3 on page 21.

If input to the solver is in XML, then output will be in XML. Human-readable output will be produced by default except if output is redirected using the −○*output* flag and either XML or parseable output is being generated. Conversion from parseable output to XML, and from either parseable or XML output to the human-readable form, can be accomplished using *lqn2ps(1)*.

2.1 Header

The human-readable output from the the analytic solver and simulator consists of three parts. Part 1 of the output consists of solution statistics and other header information and is described in detail in Sections 2.1.1 and 2.1.2 below. Part 2 of the output lists the input and is not described further. Part 3 contains the actual results. These results are described in Section 2.1.2, starting on page 13. The section headings here correspond to the section headings in the output file.

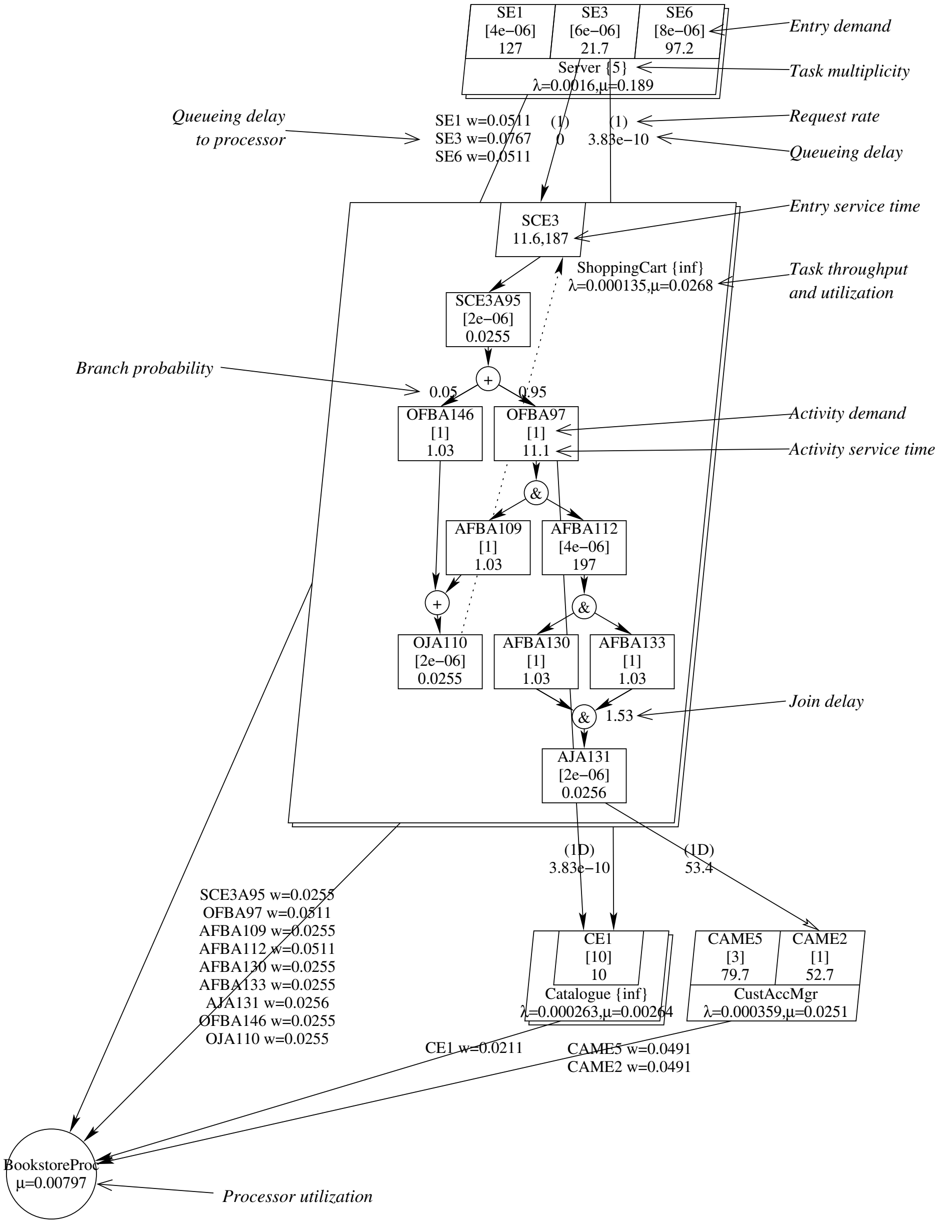


Figure 2.1: Results.

2.1.1 Analytic Solver (lqns)

Figure 2.2 shows the header information output by the analytic solver. The first line of the output shows the version of the solver and where it was run. This information is often useful when reporting problems with the solver. The lines labeled `Input` and `Output` are the input and output file names respectively. The line labelled `Command line` shows all the arguments used to invoke the solver. The `Comment` field contains the information found in the comment field of the general information field of the input file (c.f. §A.1.2, §3.2.1). Next, optionally, the output lists any pragma used. Much of this information is also present if the simulator is used to solve the model. The remainder of the header lists statistics accumulated during the solution of the model and is solver-specific.

`convergence test value:` The `convergence test value` is the root of the mean of the squares of the difference in the utilization of all of the servers from the last two iterations of the solver. If this value is less than the `convergence value` (c.f. §3.2.1, A.1.2) specified in the input file, then the results are considered valid.

`number of iterations:` The `number of iterations` shows the number of times the solver has performed its "outer iteration". If the number of iterations exceeds the iteration limit set by the model file, the results are considered invalid.

`MVA solver information:` This table shows the amount of effort the solver expended solving each submodel. The first column lists the submodel number. Next, the column labelled 'n' indicates the number of times the MVA solver was run on the submodel. The columns labelled 'k' and 'srv' show the number of chains and servers in the submodel respectively. The next three columns show the number of times the core MVA `step()` function was called. The following three columns show the number of time the `wait()` function, responsible for computing the queueing delay at a server, is called. Finally, the last three columns list the time the solver spends solving each submodel.

Finally, the solver lists the name of the machine the it was run on, the time spent executing the solver code, the time spent by the system on behalf of lqns, and the total elapsed time.

2.1.2 Simulator (lqsim)

Figure 2.3 shows the header information output by the simulator after execution is completed. The first line of the output shows the version of the simulator and where it was run. The lines labeled `Input` and `Output` are the input and output file names respectively. The `Comment` field contains the information found in the comment field of the general information field of the input file (c.f. §A.1.2, §3.2.1). Next, optionally, the output lists any pragma used. The remainder of the header lists statistics accumulated during the solution of the model and is specific to the simulator.

`Run time:` The total run time in simulation time units.

`Number of Statistical Blocks:` The number of statistical blocks collected (when producing confidence intervals).

`Run time per block:` The run time in simulation units per block. This value, multiplied by the number of statistical blocks and the initial skip period will total to the run time.

`Seed Value:` The seed used by simulator.

Finally, the simulator lists the name of the machine that it was run on, the time spent executing the simulator code, the time spent by the system on behalf of lqsim, and the total elapsed time.

2.2 Type 1 Throughput Bounds

lqns

The *Type 1 Throughput Bounds* are the "guaranteed not to exceed" throughputs for the entries listed. The value is calculated assuming that there is no contention delay to underlying servers.

```
Generated by lqns, version 3.9 (Darwin 6.8.Darwin Kernel Version 6.8: Wed Sep 10 15:20:55 PDT 2003;  Power Macintosh)

Copyright the Real-Time and Distributed Systems Group,
Department of Systems and Computer Engineering
Carleton University, Ottawa, Ontario, Canada. K1S 5B6

Input:  bookstore.lqn
Output: bookstore.out
Command line: lqns -p
Tue Nov  1 21:37:54 2005

Comment: lqn2fig -Lg bookstore.lqn

    #pragma multiserver          = conway

Convergence test value: 7.51226e-07
Number of iterations:   5

MVA solver information:
Submdl  n   k srv   step()     mean    stddev     wait()     mean    stddev        User      System     Elapsed
1       5   2   4       44      8.8    1.4697       4776    955.2    299.82  0:00:00.01  0:00:00.00  0:00:00.00
2       9   1   1       51   5.6667   0.94281        594       66    22.627  0:00:00.00  0:00:00.00  0:00:00.00
3       9   8   3      240   26.667    9.4751 4.0365e+05    44850     32163  0:00:00.19  0:00:00.00  0:00:00.21
4       9  10   3      271   30.111    7.0623 7.7481e+05    86090     40554  0:00:01.15  0:00:00.00  0:00:01.19
5       9   2   1       70   7.7778    1.6178       3408   378.67    181.73  0:00:00.00  0:00:00.00  0:00:00.00
6       5   0   0        0        0         0          0        0         0  0:00:00.00  0:00:00.00  0:00:00.00
Total  46   0   0      676   14.696    12.464 1.1872e+06    25809     41253  0:00:01.35  0:00:00.00  0:00:01.40

    greg-frankss-Computer.local. Darwin 6.8
    User:     0:00:01.35
    System:   0:00:00.00
    Elapsed:  0:00:01.40
```

Figure 2.2: Analytic Solver Status Output.

```
Generated by lqsim, version 3.9 (Linux 2.4.20-31.9  i686),

Copyright the Real-Time and Distributed Systems Group,
Department of Systems and Computer Engineering,
Carleton University, Ottawa, Ontario, Canada. K1S 5B6

Wed Nov  2 11:42:25 2005

Input: bookstore.lqn
Output: bookstore.out
Comment: lqn2fig -Lg bookstore.lqn


Run time: 4.34765E+09
Number of Statistical Blocks: 15
Run time per block: 2.89651E+08
Max confidence interval: 7.32
Seed Value: 1130948006

    epsilon-13.sce.carleton.ca Linux 2.4.20-31.9
    User:     0:04:47.78
    System:   0:00:00.07
    Elapsed:  0:14:27.66
```

Figure 2.3: Simulator Status Output.

2.3 Mean Delay for a Rendezvous

The *Mean Delay for a Rendezvous* is the queueing time for a request from a client to a server. It does not include the time the customer spends at the server (see Figure 2.4). To find the *residence timeresidence time*, add the queueing time to the *phase one service time* of the request's server.

2.4 Variance of Delay for a Rendezvous

lqsim

The *Variance of Delay for a Rendezvous* is the variance of the queueing time for a request from a client to the server. It does not include the variance of the time the customer spends at the server (see Figure 2.4). This result is only available from the simulator.

2.5 Mean Delay for a Send-No-Reply Request

The *Mean delay for a send-no-reply request* is the time the request spends in queue and in service in phase one at the destination. Phase two is treated as a 'vacation' at the server.

2.6 Variance of Delay for a Send-No-Reply Request

lqsim

2.7 Arrival Loss Probabilities

The *Arrival Loss Probabilities*...

2.8 Mean Delay for a Join

The *Mean Delay for a Join* is the maximum of the sum of the service times for each branch of a fork. The source activity listed in the output file is the first activity prior to the fork (e.g., AFBA112 in Figure 2.1). Similarly, the destination activity listed in the output file is the first activity after the join (AJA131). The variance of the join time is also computed.

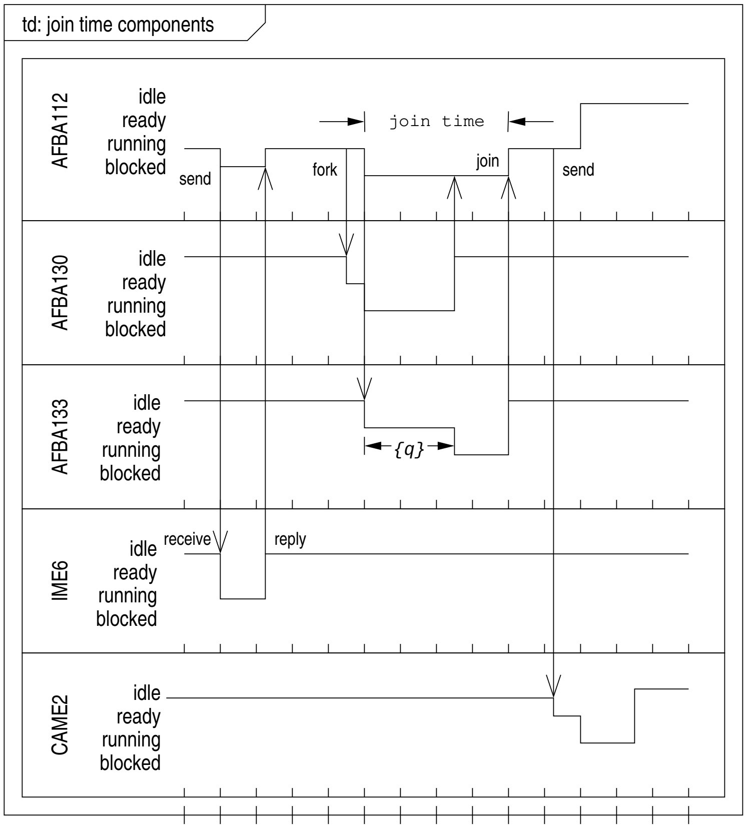


Figure 2.4: Service Time Components for Join.

2.9 Service Times

The *service time* is the total time a phase or activity uses processing a request. The time consists of four components, shown in Figure 2.4:

1. Queueing for the processor (shown as items 1, 4, 6 and 8 in Figure 2.5.(b)).

2. Service at the processor (items 2, 5 and 9)

3. Queueing for serving tasks (item 6), and

4. Phase one service time at serving tasks (items 3 and 7).

Queuing at processors and tasks and can occur because of contention from other tasks (items 1, 6, and 8), or from second phases from previous requests. For example, entry SE3 is queued at the processor because the processor is servicing the second phase of entry SCE3.

Using the results shown in Figure 2.1, the service time for entry SE3 (21.7) is the sum of:

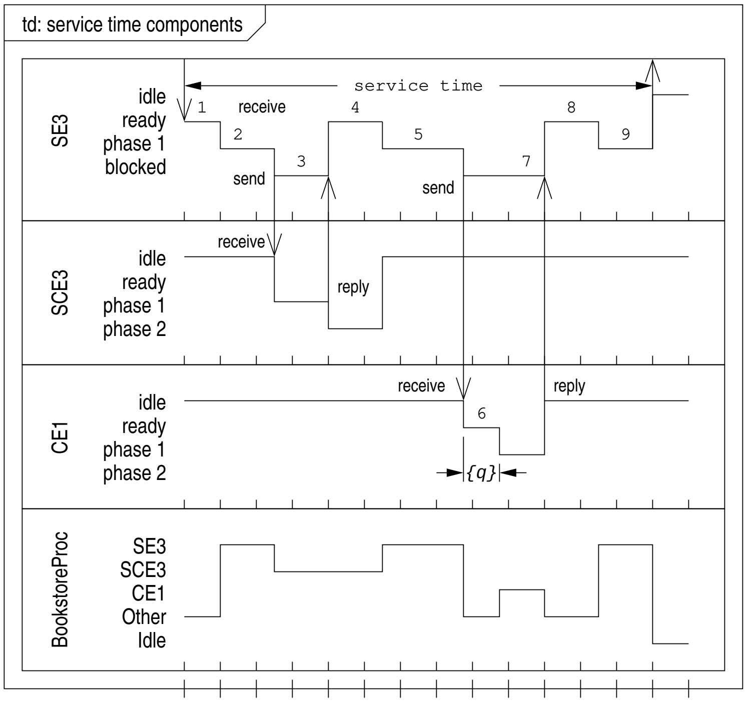


Figure 2.5: Service Time Components for Entry ‘SCE3’.

- the processor wait (0.767),
- it’s own service time (6×10^{-6}),
- the queueing time to entry SCE3 (0),
- the phase one service time at entry SCE3 (11.6),
- the queueing time to entry CE1 (3.83×10^{-10}), and
- the phase one service time at entry CE1 (10)

Queueing time for serving tasks is shown in the *Mean Delay for a Rendezvous* section of the output. (c.f. §2.3). Queueing time for the processor is shown in the *Utilization and Waiting per Phase for Processor* of the output (c.f. §2.16).

2.10 Service Time Variance

The *Service Time Variance* section lists the variance of the service time (c.f. §2.9) for the phases and activities in the model.

2.11 Probability Maximum Service Time Exceeded

lqsim

The *probability maximum service time exceeded* is output by the simulator for all phases and activities with a `max-service-time`. This result is the probability that the service time is greater than the value specified. In effect, it is a histogram with two bins.

2.12 Service Time Distributions for Entries and Activities

lqsim

Service Time Distributions are generated by the simulator by setting the `service-time-distribution` parameter (c.f. §3.2.9, §A.1.6, §A.1.7) for an entry or activity. A histogram of `number-bins` bins between `min` and `max` is generated. Samples that fall either under or over this range are stored in their own under-flow or over-flow bins respectively. The optional `x-samples` parameter can be used to set the sampling behaviour to one of:

linear Each bin is of equal width, found by dividing the histogram range by the number of bins. If the `x-samples` is not set, this behaviour is the default.

log The logarithm of the range specified is divided by `number-bins`. This has the effect of making the width of the bins small near `min`, and large near `max`. A minimum value of zero is **not** allowed.

sqrt The square root of the range specified is divided by `number-bins`. Bins are smallest near `bin` are smaller than those near `max`.

The results of the histogram collection, shown in Figure 2.6, consist of the mean, standard deviation,, skew and kurtosis of the sampled range, followed by the histogram itself. Each entry of the histogram contains the probability of the sample falling within the bucket, and, if available, the confidence intervals of the sample.

The statistics for the histogram are found by multiplying the mid-point of the range defined by `begin` and `end`, not counting either the overflow or underflow bins. If the mean value reported by the histogram is substantially different than the actual service time of the phase or activity, then the range of the histogram is not sufficiently large.

2.13 Semaphore Holding Times

The *Semaphore Holding Times* section lists the average time a semaphore token is held (it's service time), the variance of the holding time, and the utilization of semaphore. Figure 2.7 shows how these values are found.

2.14 Throughputs and Utilizations per Phase

The *Throughputs and Utilizations per Phase* section lists the throughput by entry and activity, and the utilization by phase and activity. The utilization is the *task utilization*, i.e., the reciprocal of the service time for the task (c.f. 2.9). The processor utilization for the task is listed under *Utilization and Waiting per Phase for Processor* (see §2.16).

2.15 Arrival Rates and Waiting Times

The *Arrival Rates and Waiting Times* section is only present in the output when *open arrivals* are present in the input. This section shows the arrival rate (*Lambda*) and the waiting time. The waiting time includes the service time at the task.

2.16 Utilization and Waiting per Phase for Processor

The *Utilization and Waiting per Phase for Processor* lists the processor utilization and the queueing time for every entry and activity running on the processor.

```
Service time distributions for entries and activities:

SCE3             PHASE 1:
    Mean =   11.58, Stddev =   8.457, Skew =  0.8501, Kurtosis = -0.2496
       Begin      End       Prob.      +/-95%     +/-99%
           0        1  0.03355     0.001048   0.001412    |          *
           1        2  0.03786     0.001605   0.002163    |            *
           2        3  0.05406     0.002026   0.002731    |                 *
           3        4  0.06333     0.002031   0.002737    |                   *
           4        5  0.06545     0.001631   0.002199    |                    *
           5        6  0.06369     0.001578   0.002127    |                   *
           6        7  0.06049     0.001692   0.00228     |                  *
           7        8  0.05591     0.001822   0.002456    |                 *
           8        9  0.05133     0.001272   0.001714    |                *
           9       10  0.0472      0.001767   0.002382    |              *
          10       11  0.04318     0.001618   0.002181    |             *
          11       12  0.03931     0.001185   0.001597    |            *
          12       13  0.03579     0.001073   0.001446    |           *
          13       14  0.03231     0.001654   0.002229    |          *
          14       15  0.02952     0.001033   0.001392    |         *
          15       16  0.02677     0.001189   0.001603    |        *
          16       17  0.0243      0.001058   0.001425    |       *
          17       18  0.02214     0.001087   0.001466    |       *
          18       19  0.02001     0.001122   0.001512    |      *
          19       20  0.01806     0.001016   0.001369    |      *
          20       21  0.01653     0.0009079  0.001224    |     *
          21       22  0.01499     0.001018   0.001372    |     *
          22       23  0.01365     0.0007152  0.0009639   |    *
          23       24  0.01229     0.000955   0.001287    |    *
          24       25  0.0112      0.0008691  0.001171    |   *
          25       26  0.009997    0.0006182  0.0008331   |   *
          26       27  0.009227    0.0007344  0.0009898   |   *
          27       28  0.008282    0.0006896  0.0009293   |   *
          28       29  0.007444    0.0005936  0.0007999   |  *
          29       30  0.006802    0.0005752  0.0007751   |  *
             overflow  0.06532     0.001561   0.002104    | *
```

Figure 2.6: Histogram output

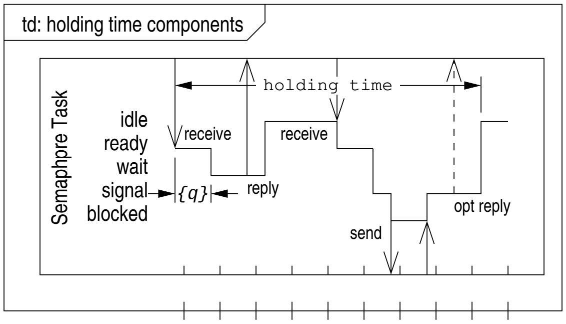


Figure 2.7: Time components of a semaphore task.

Chapter 3

XML Grammar

The definition of LQN models using XML is an evolution of the original SRVN file format (c.f. §5 and Appendix A.1). The new XML format is based on the work done in [22], with further refinement for general usage. There are new features in the XML format to support new concepts for building and assembling models using components. The normal LQN tool suite (like *lqns(1)* and *lqsim(1)*) do not support these new features, however other tools outside the suite are being written to utilize the new parts of the XML format.

3.1 Basic XML File Structure

In XML, layered models are specified in a bottom-up order, which is the reverse of how layered models are typically presented. First, a processor is defined, then within the processor block, all the tasks than run on it are defined. Similarly, within each task block all the entries that are associated with it are defined, etc. A simplified layout of an incomplete LQN model written in XML is shown in Figure 3.1.

Activity graphs (specified by task-activities) belong to a task, and hence are siblings to entry elements. The element entry-activity-graph specifies an activity graph contained within one entry, but is not supported by any of the LQN tools. The concept of phases still exists, but now each phase is an activity, and is defined in the entry-phase-activities element.

3.2 Schema Elements

The XML definition for layered models consists of three files:

`lqn.xsd:` lqn.xsd is the root of the schema.

`lqn-sub.xsd` ...

`lqn-core.xsd` lqn-core is the actual model specfication and is included by lqn.xsd.

All three files should exist in the same location. If the solver cannot located the `lqn.xsd` file, it will emit an error[1] and stop.

Figure 3.1 shows the schema for Layered Queueing Networks using Unified Modeling Language notation. The model is defined starting from `lqn-model`. Unless otherwise specified in the figure, the order of elements in the model is from left to right, i.e., `<solver-params>` always preceeds `<processor>` in the input file. Optional elements are shown using a multiplicity of zero for an association. Note that results (optional, shown in blue) are part of the schema.

[1] See the error message "The primary document entity could not be opened" on 95.

Listing 3.1: XML file layout.

```
<lqn-model>
   <solver-params>
      <pragma/>
   </solver-params>
   <processor>
      <task>
         <entry>
            <entry-phase-activities>
               <activity>
                  <synch-call/>
                  <asynch-call/>
               </activity>
               <activity> ... </activity>
            </entry-phase-activities>
         </entry>
         <entry> ... </entry>
         <task-activities>
            <activity/>
            <precedence/>
         </task-activities>
      </task>
      <task> ... </task>
   </processor>
   <processor> ... </processor>
</lqn-model>
```

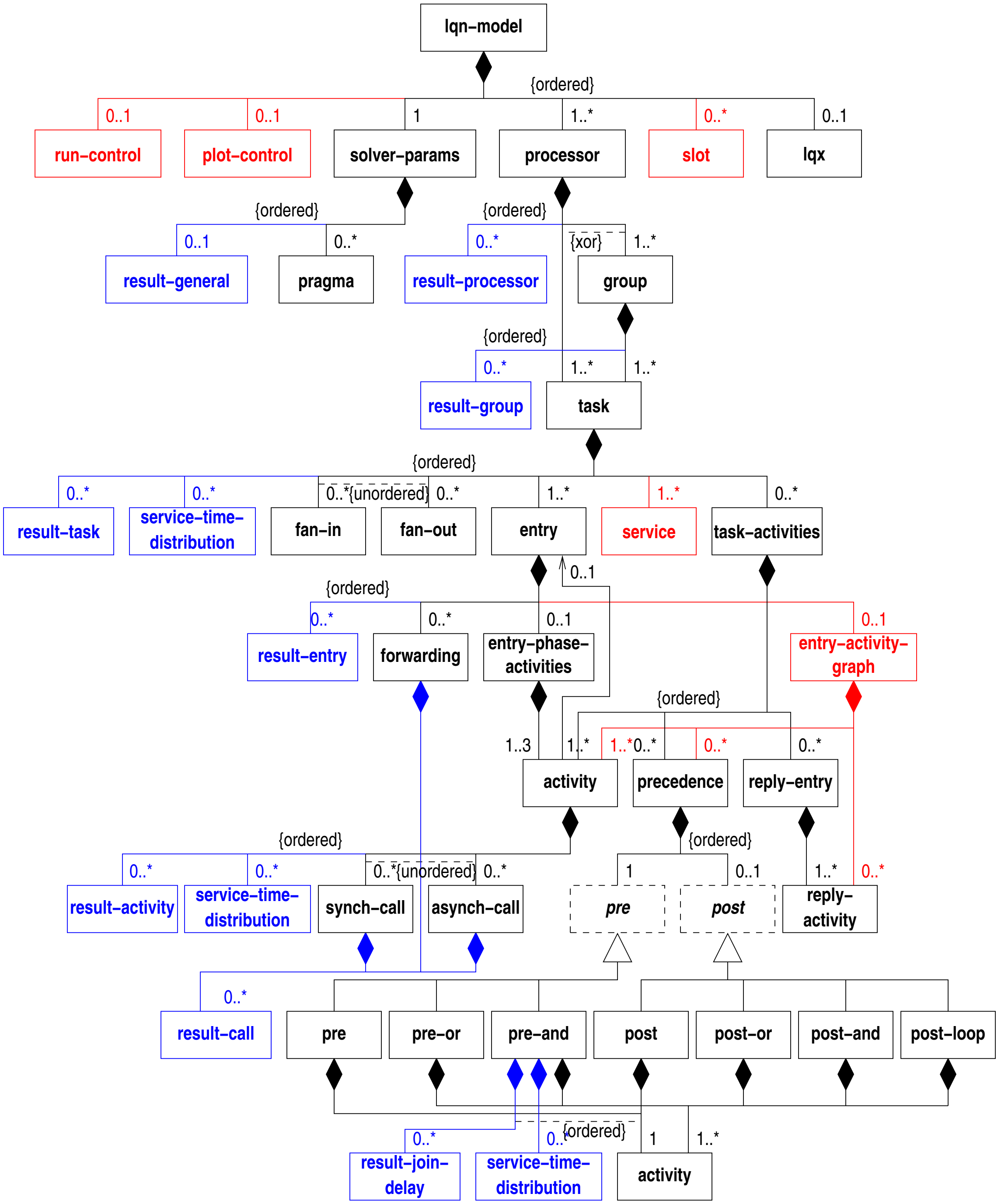


Figure 3.1: LQN Schema. Elements shown in blue are results found in the output. Elements shown in red are not implemented. Unless otherwise indicated, all elements are ordered from left to right.

3.2.1 LqnModelType

The first element in a layered queueing network XML input file is lqn-model, which is of type **LqnModelType** and is shown in Figure 3.2. **LqnModelType** has five elements, namely: run-control, plot-control, solver-params, processor and slot. Run-control and plot-control are not not implemented. Processor is described under Section 3.2.2. Slot is described in [22]. The attributes for **LqnModelType** are shown in Table 3.1.

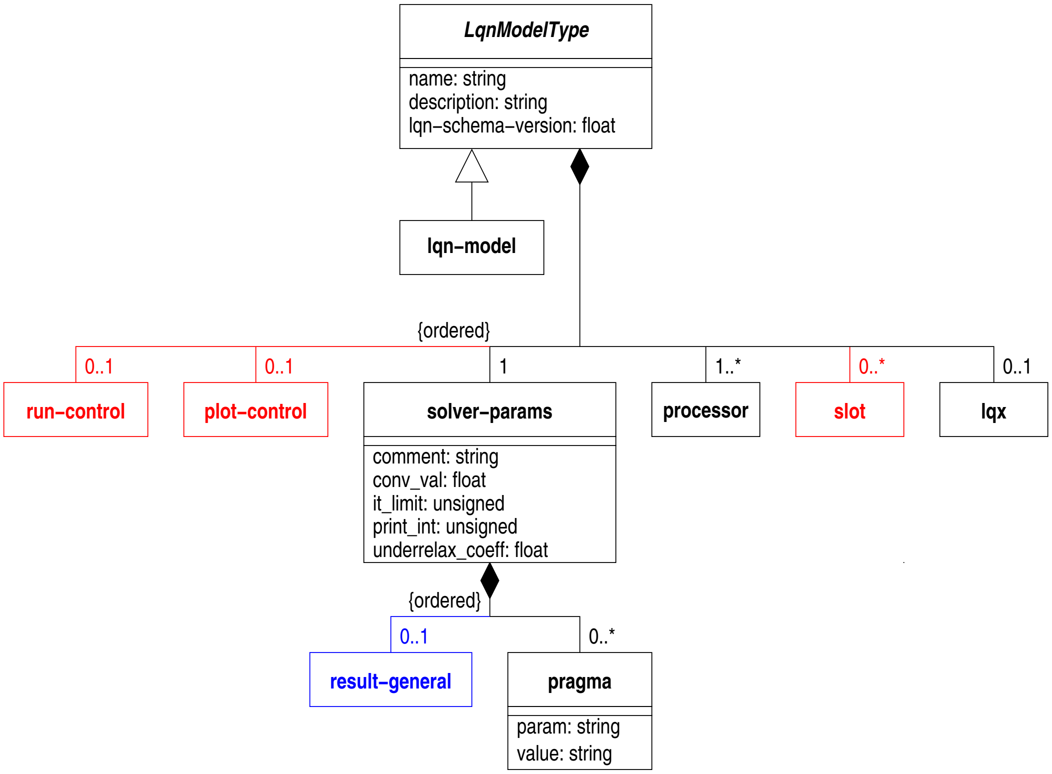


Figure 3.2: Top-level LQN Schema.

Name	Type	Use	Default	Comments
name	string	optional		The name of the model.
description	string	optional		A description of the model.
lqn-schema-version	integer	fixed	1.0	The version of the schema (used by the solver in case of substantial schema changes for model conversion.)
lqncore-schema-version	integer	fixed	1.0	
xml-debug	boolean	optional	false	

Table 3.1: Attributes for elements of type **LqnModelType** from Figure 3.2.

The element solver-params is used to set various operating parameters for the analytic solver, and to record various output statistics after a run completes. It contains the elements result-general and pragma. The attributes for solver-params are shown in Table 3.2. These attributes are mainly used to control the analytic solver. Refer to Section 6.3 for more information. The attributes for result-general are shown in Table 3.3. Refer to Sections 2.1.1 and 2.1.2 for the interpretation of header information. The attributes for pragma are show in Table 3.4. Refer to Section 6.2 for the pragmas supported by lqns and to Section 7.3 for the pragmas supported by lqsim.

Name	Type	Use	Default	Comments
`conv_val`	float	optional	1	Convergence value for lqns (c.f §6.3). Ignored by lqsim.
`it_limit`	integer	optional	50	Iteration limit for lqns (c.f §6.3). Ignored by lqsim.
`print_int`	integer	optional	0	Print interval for intermediate results. The `-t`*print* must be specified to lqns to generate output after *it_limit* iterations. Blocked statistics must be specified to lqsim using the `-A`*n*, `-B`*n*, or `-C`*n* flags.
`underrelax_coeff`	float	optional	0.5	Under-relaxation coefficient for lqns (c.f §6.3). Ignored by lqsim.

Table 3.2: Attributes of element `solver-params` from Figure 3.2.

Name	Type	Use	Default	Comments
`conv-val`	float	required		Convergence value (c.f. 2.1.1)
`valid`	enumeration	required		Either `YES` or `NO`.
`iterations`	float	optional		The number of iterations of the analytic solver or the number of blocks for the simulator.
`elapsed-time`	string	optional		The wall-clock time used by the solver.
`system-cpu-time`	string	optional		The CPU time spent in kernel-mode.
`user-cpu-time`	string	optional		The CPU time spent in user mode.
`platform-info`	string	optional		The operating system and CPU type.
`solver-info`	string	optional		The version of the solver.

Table 3.3: Attributes of element `result-general` from Figure 3.2.

Name	Type	Use	Default	Comments
`param`	string	required		The name of the parameter. (c.f. 6.2, §7.3)
`value`	string	required		the value assigned to the pragma.

Table 3.4: Attributes of element `pragma` from Figure 3.2.

3.2.2 ProcessorType

Elements of type **ProcessorType**, shown in Figure 3.3 are used to define the processors in the model. They contain an optional `result-processor` element and elements of either **GroupType** or **TaskType**. The `scheduling` attribute must by set to `cfs`, for completely fair scheduling, if **GroupType** elements are present and to any other type if **GroupType** are not found. **GroupType** and **TaskType** elements may not be both be defined in a processor.

Element `result-processor` is of type **OutputResultType** and is described in Section 3.2.12. Element `task` is described in Section 3.2.4. The attributes of **ProcessorType**, described in A.1.3, are shown in Table 3.5.

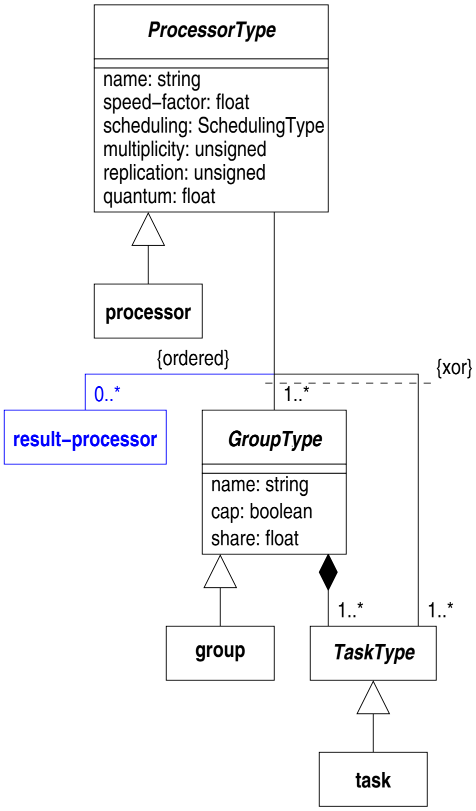


Figure 3.3: Processor Schema.

Name	Type	Use	Default	Comments
`name`	string	required		
`multiplicity`	integer	optional	1	See §1.2
`speed-factor`	float	optional	1.0	Scaling factor for the processor.
`scheduling`	enumeration	optional	fcfs	The allowed scheduling types are `fcfs`, `hol`, `pp`, `rand`, `inf`, `ps-hol`, `ps-pp` and `cfs`. See §1.1.1.
`replication`	integer	optional	1	See §1.2
`quantum`	float	optional	0.0	Mandatory for processor sharing scheduling when using lqsim.

Table 3.5: Attributes for elements of type **ProcessorType**.

3.2.3 GroupType

Optional elements of type **GroupType**, shown in Figure 3.3, are used to define groups of tasks for processors running completely fair scheduling. Each group must contain a minimum of one task. The attributes of **GroupType** are shown in Table 3.6.

Name	Type	Use	Default	Comments
`name`	string	required		
`share`	float	required		The fraction of the processor allocated to this group.
`cap`	boolean	optional	false	If true, shares are *caps* (ceilings). Otherwise, shares are guarantees (floors)

Table 3.6: Attributes for elements of type **GroupType**

3.2.4 TaskType

Elements of type **TaskType**, shown in Figure 3.4, are used to define the tasks in the model. These elements contain an optional `result-task` element, one or more elements of **EntryType**, and optionally, elements of `service` and `task-activities`. Element `result-task` is of type **OutputResultType**, and is described in Section 3.2.12. Element `entry` is described in Section 3.2.6. The attributes of **TaskType**, described in Section A.1.5, are shown in Table 3.7.

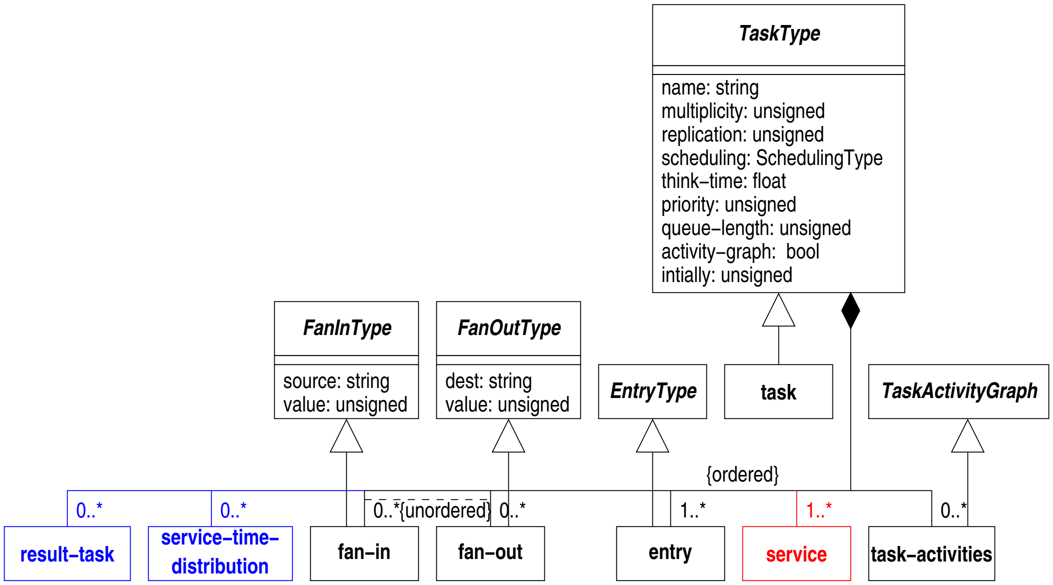


Figure 3.4: TaskType

3.2.5 FanInType and FanOutType

3.2.6 EntryType

Elements of type **EntryType**, shown in Figure 3.5, are used to define the entries of tasks. Entries can be specified one of three ways, based on the attribute `type` of an `entry` element, namely:

Name	Type	Use	Default	Comments
`name`	string	required		
`multiplicity`	integer	optional	1	See §1.2.
`priority`	integer	optional	0	The priority used by the processor for scheduling. See §1.1.1.
`queue-length`	integer	optional	0	Maximum queue size (for open-class requests only). See §1.1.3.
`replication`	integer	optional	1	See §1.2
`scheduling`	enumeration	optional	FCFS	The scheduling of requests at the task. The allowed scheduling types are `ref`, `fcfs`, `hol`, `pri`, `inf`, `burst`, and `poll` and `semaphore`. See §1.1.3.
`activity-graph`	enumeration	required		`yes` or `no`
`think-time`	float	optional	0	Reference tasks only. Customer think time.
`initially`	integer	optional	*multiplicity*	Semaphore tasks only. Set the initial number of semaphore tokens to zero. By default, the number of tokens is set to the multiplicity of the task.

Table 3.7: Attributes for elements of type **TaskType**

Name	Type	Use	Default	Comments
`source`	integer	required		(See §1.2)
`value`	integer	required		(See §1.2)

Table 3.8: Attributes for elements of type **FanInType**.

Name	Type	Use	Default	Comments
`dest`	integer	required		(See §1.2)
`value`	integer	required		(See §1.2)

Table 3.9: Attributes for elements of type **FanOutType**.

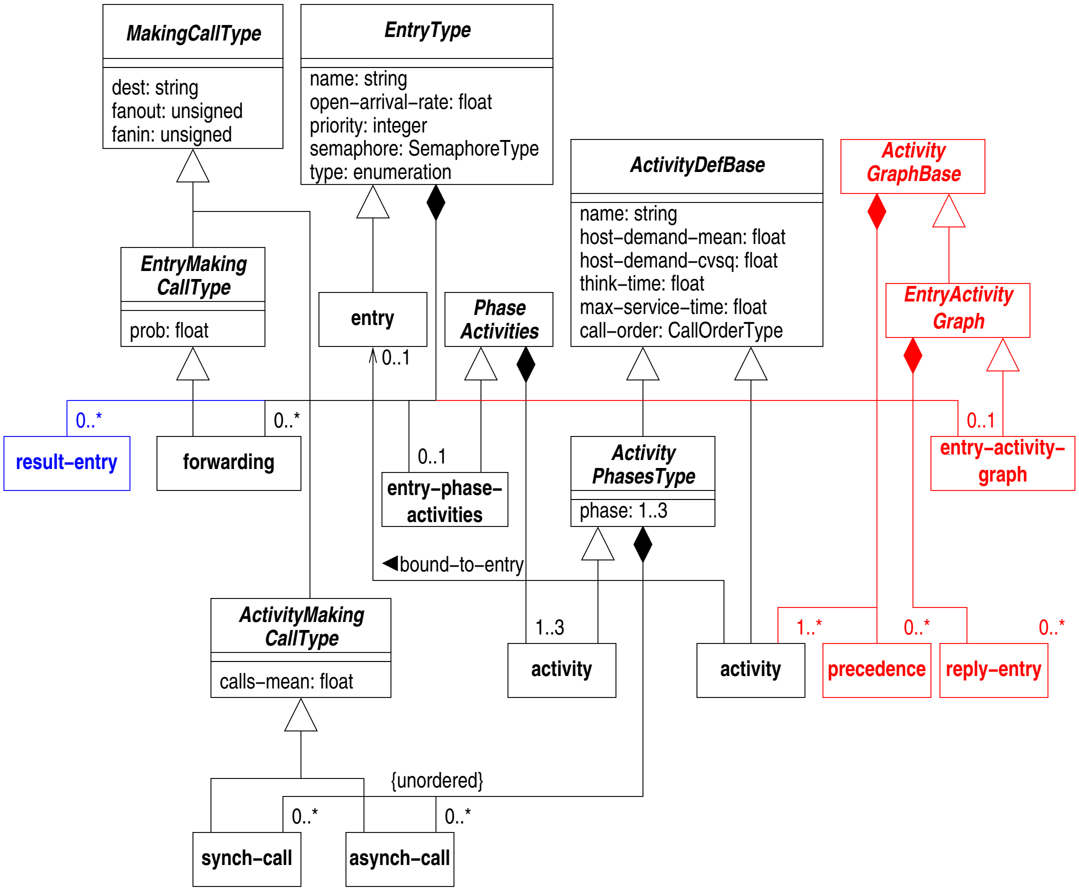


Figure 3.5: Schema for type **EntryType**.

ph1ph2 The entry is specified using phases. The phases are specified using an `entry-phase-activities` element which is of the **ActivityPhasesType** type. Activities defined within this element must have a unique `phase` attribute.

graph The entry is specified as an activity graph defined within the entry. The demand is specified using elements of type **ActivityEntryDefType**. This method of defining an entry is not supported currently.

none The entry is specified using an activity graph defined within the task. A `task-activities` element of type **ActivtyDefType** must be present and one of the activities defined within this element must have a `bound-to-entry` attribute. The **TaskActivityGraph** type is defined in Section 3.2.8.

ActivityPhasesType, **ActivityEntryDefType** and **ActivtyDefType** are all based on **ActivityDefBase**, described in Section 3.2.9. They only differ in the way the start of the graph is identified, and in the case of **ActivityPhasesType**, the way the activities are connected.

The attributes for **EntryType**, described in Section A.1.6, are shown in Table 3.10. The optional element `result-entry` is of type **OutputResultType**, and is described in Section 3.2.12. The optional element `forwarding` is used to describe the probability of forwarding a request to another entry; it is described in Section 3.2.10.

Name	Type	Use	Default	Comments
`name`	string	required		The entry name
`type`	enumeration	required		`PH1PH2`, `GRAPH`, or `NONE`
`open-arrival-rate`	float	optional		
`priority`	integer	optional		(c.f. 1.1.3)
`sempahore`	enumeration	optional		`signal` or `wait` (c.f. 1.1.3)

Table 3.10: Attributes for elements of type **EntryType**.

3.2.7 ActivityGraphBase

Elements of type **ActivityGraphBase**, shown in Figure 3.6, are used to define activities (c.f. 1.1.5) and their relationships to each other. They are used by elements of both **EntryType** and **TaskActivityGraph** types.

Elements of the **ActivityGraphBase** consist of a sequence of one or more `activity` elements followed by a sequence of `precedence` elements. `Activity` elements are used to store the demand for an activity and requests to other servers (through the **ActivityDefType**) and, optionally, results through elements of **ActivityDefType**. `Precedence` elements are defined by the **PrecedenceType** in Section 3.2.11.

3.2.8 TaskActivityGraph

Task Activity Graphs, defined using elements of type **TaskActivityGraph** and shown in Figure 3.6, are used to specify the behaviour of a task using activities. This type is almost the same as **EntryActivityGraph**, except that the activity that replies to an entry must explicitly specify the entry for which the reply is being generated. The actual activity graph is defined using elements of type **ActivityGraphBase**, described in Section 3.2.7. The attributes for elements `reply-entry` and `reply-activity` are shown in Tables 3.11 and 3.12 respectively.

Name	Type	Use	Default	Comments
`name`	string	required		The name of the entry for which the list of `reply-activity` elements generate replies.

Table 3.11: Attributes of element `reply-entry` from Figure 3.6.

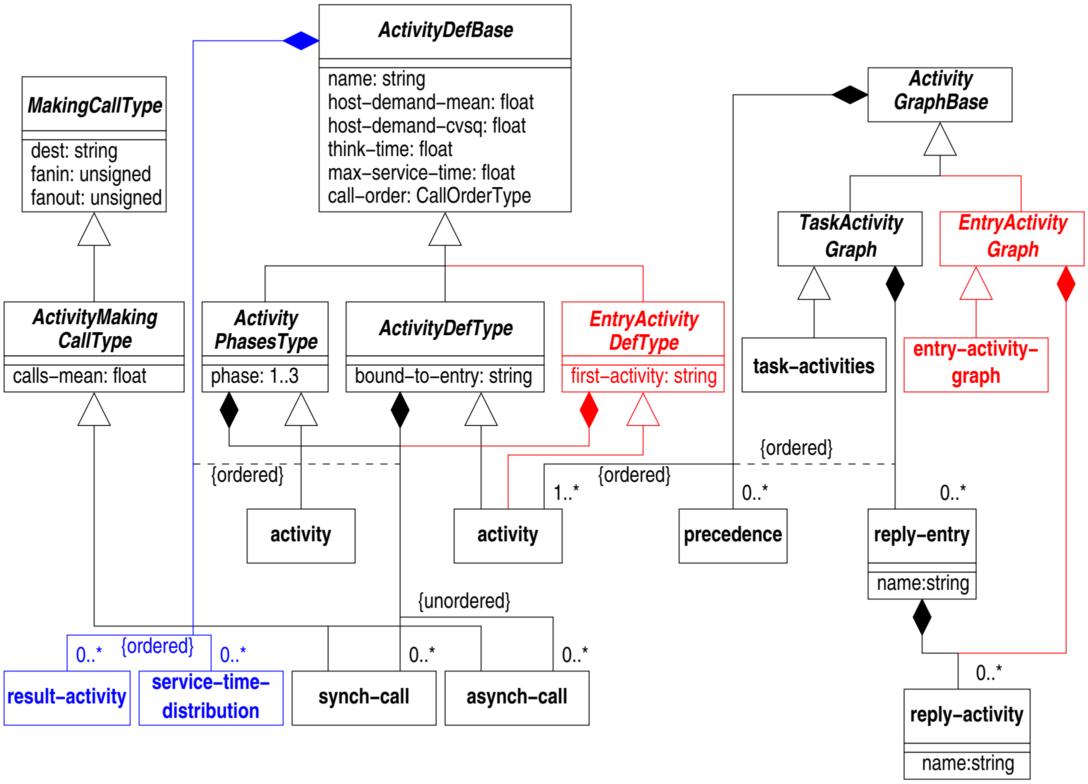


Figure 3.6: Schema diagram for the type **ActivityGraphBase**

Name	Type	Use	Default	Comments
`name`	string	required		The name of the activity which generates a reply. The entry is either implicitly defined if this element is defined within an **EntryType**, or part of list defined within a `reply-element`.

Table 3.12: Attributes of element `reply-activity` from Figure 3.6.

3.2.9 ActivityDefBase

The type **ActivityDefBase**, shown in Figure 3.6, is used to define the parameters for an activity, such as demand and call-order. This type is extended by **ActivityPhasesType**, **EntryActivityDefType**, and **ActivityDefType** to define the requests from an activity to an entry, and to connect the activity graph to the requesting entry. Table 3.13 lists the parameters used as attributes and the attributes used by the three sub-types. Refer to Section A.1.7 for more information on these parameters. Refer to **MakingCallType** (§3.2.10) for the `Activity-CallGroup` used to make requests to other entries[2]. Refer to **OutputResultForwardingANDJoinDelay** (§3.2.13) for `result-join-delay` and `result-forwarding` for join-delay and forwarding results respectively. Refer to **OutputDistributionType** (§3.2.14) for `service-time-distribtion`. Finally, refer to **OutputResultType** (§3.2.12) for `result-activity`. This element contains most of the results for an activity or phase.

Name	**Type**	**Use**	**Default**	**Comments**
`name`	string	required		
`host-demand-mean`	float	required		The mean service time demand for the activity.
`host-demand-cvsq`	float	optional	1.0	The squared coefficient of variation for the activity.
`think-time`	float	optional	0.0	
`max-service-time`	float	optional	0.0	
`call-order`	enumeration	optional	STOCHASTIC	`STOCHASTIC` or `DETERMINISTIC`
		ActivtyPhasesType		
`phase`	integer	required		1, 2, or 3
		ActivtyEntryDefType		
`first-activity`	string	required		
		ActivtyDefType		
`bound-to-entry`	string	optional		If set, this activity is the start of an activity graph.

Table 3.13: Attributes for elements of type **ActivityDefBase**.

3.2.10 MakingCallType

The type **MakingCallType**, shown in Figure 3.7, is used to define the parameters for requests to entries. This type is extended by **ActivityMakingCallType** and **EntryMakingCallType** to defined requests from activities to entries and for forwarding requests from entry to entry respectively. Requests from activities to entries can be either synchronous, (i.e., a *rendezvous*), through a `sync-call` element, or asynchronous (i.e., a *send-no-reply*), through a `async-call` element. Section 1.1.7 defines the parameters for a request. Table 3.14 lists the attributes for the types.

Name	**Type**	**Use**	**Default**	**Comments**
`dest`	string	required		The name of the entry to which the requests are made.
		ActivityMakingCallType		
`calls-mean`	float	required		The mean number of requests.
		EntryMakingCallType		
`prob`	float	required		The probability of forwarding requests.

Table 3.14: Attributes for elements of type **MakingCallType**.

[2] `Call-List-Group` is not defined at present.

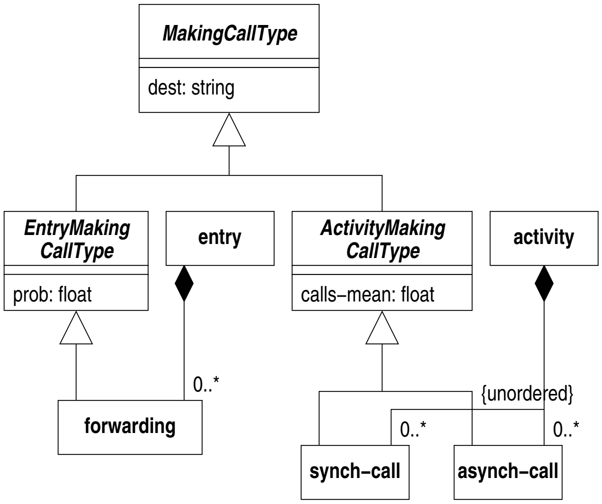


Figure 3.7: Schema diagram for the group **MakingCallType**.

3.2.11 PrecedenceType

The type **PrecedenceType**, shown in Figure 3.8, is used to connect one activity to another within an activity graph. Each element of this type contains exactly one `pre` element and, optionally, one `post` element. The pre elements are referred to as *join*-lists as all of the branches associated with the activities in the join-list must finish (i.e. "join") before the activities in the subsequent post element can start. The post element itself is referred to as a *fork*-list.

Elements of **PrecedenceType** can be of one of five types:

SingleActivityListType: Elements of this type have no attributes and a sequence of exactly one `activity` element of **ActivityType**.

ActivityListType: Elements of this type have no attributes and a sequence one or more `activity` elements of **ActivityType**.

AndJoinListType: Elements of this type have an optional `quorum` element and a sequence of one or more or more `activity` elements of **ActivityType**. Table 3.15 show the attributes of **AndJoinListType**.

OrListType: Elements of this type have no attributes and a sequence one or more `activity` elements of **ActivityOrType**. These elements specify an activity name and a branch probability. Table 3.16 show the attributes of **ActivityOrType**.

ActivityLoopListType: Elements of this type have one optional attribute and a sequence one or more `activity` elements of **ActivityLoopType**. These elements specify an activity name and a loop count. The optional attribute is used to specify the activity that is executed after all the loop branches complete. Tables 3.17 and 3.18 show the attributes of **ActivityLoopListType** and **ActivityLoopType** respectively.

Name	Type	Use	Default	Comments
`name`	string	required		
`quorum`	integer	optional	0	The number of branches which must complete for the join to finish. If this attribute is not specified, then all of the branches must finish, which makes this object an AND-Join

Table 3.15: Attributes for elements of type **AndJoinListType**.

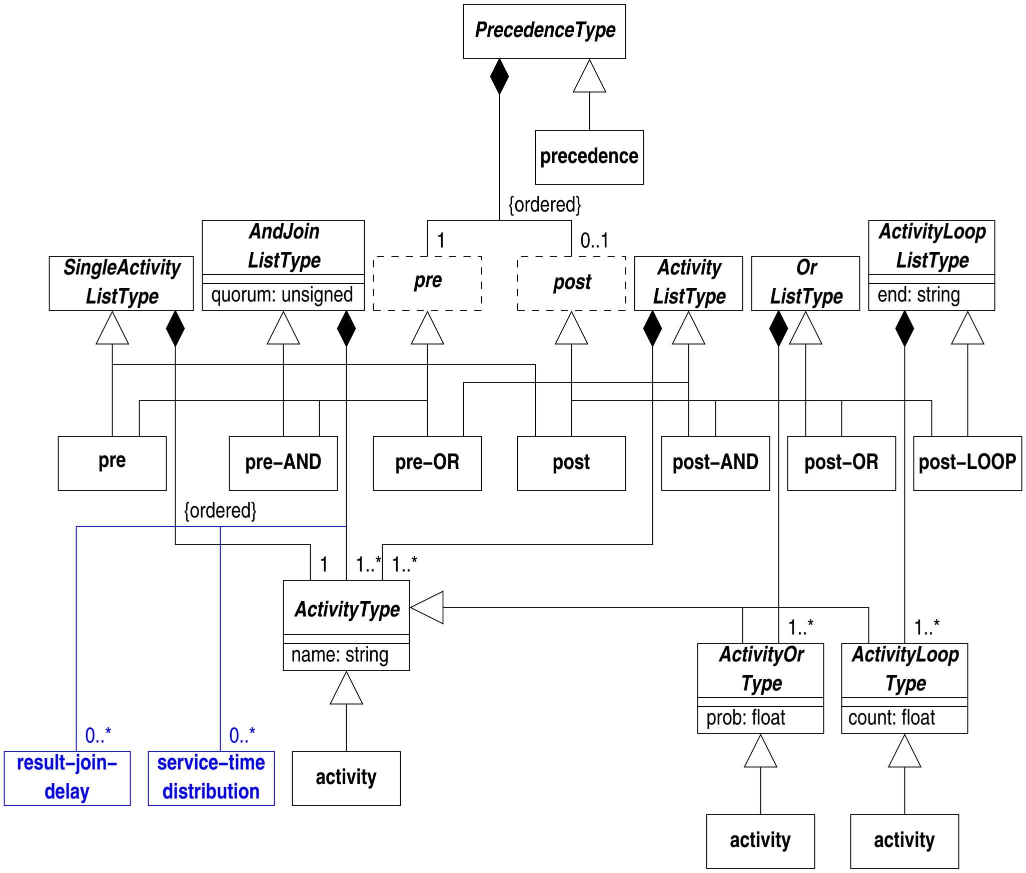


Figure 3.8: Schema diagram for the type **PrecedenceType**.

Name	Type	Use	Default	Comments
`name`	string	required		
`prob`	float	optional	1.0	The probability that the branch is taken, on average (c.f. §1.1.6)

Table 3.16: Attributes for elements of type **ActivityOrType**.

Name	Type	Use	Default	Comments
`end`	string	required		

Table 3.17: Attributes for elements of type **ActivityLoopListType**.

Name	Type	Use	Default	Comments
`count`	float	optional	1.0	The number of times the loop is executed, on average (c.f. §1.1.6)

Table 3.18: Attributes for elements of type **ActivityLoopType**.

3.2.12 OutputResultType

The type **OutputResultType**, shown in Figure 3.9, is used to create elements that store results described earlier in Section 2. **OutputResultType** is a subtype of **ResultContentType**. This latter type defines the result element's attributes. Elements of this **OutputResultType** can contain two elements of type **ResultContentType**, which contain the $\pm 95\%$ and $\pm 99\%$ confidence intervals, provided that these results are available. The attributes for elements of **ResultContentType** are listed in Table 3.19 and are used to store the actual results produced by the solver. Note that all the attributes are optional: elements of this type will only have those attributes which are relevant.

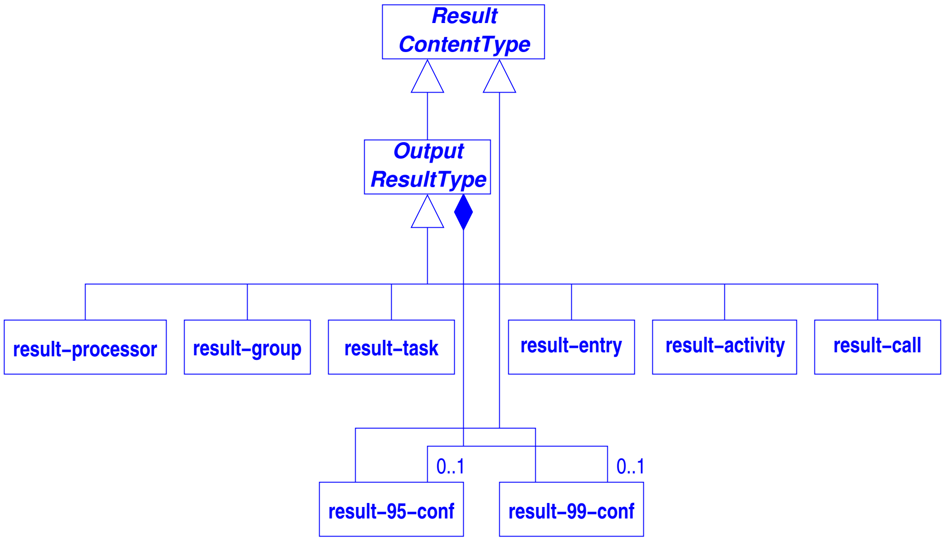


Figure 3.9: Schema diagram for type **OutputResultType**

3.2.13 OutputResultJoinDelayType

The type **OutputResultJoinDelayType** is similar to **OutputResultType**. The attributes of this type are shown in Table 3.20.

3.2.14 OutputDistributionType

Elements of type **OutputDistributionType**, shown in Figure 3.11, are used to define and store histograms of phase and activity service times. The optional `underflow-bin`, `overflow-bin` and `histogram-bin` elements, all the elements are of type **HistogramBinType**, are used to store results.

The attributes of **OutputDistributionType** elements are used to both store the parameters for the histogram, and output statistics. Refer to Table 3.21

3.2.15 HistogramBinType

3.3 Schema Constraints

The schema contains a set of constraints that are checked by the Xerces XML parser [1] to ensure that the model file is valid. XML editors can also enforce these constraints so that the model is somewhat correct before being passed to the simulator or analytic solver. The constraints are as follow:

- All processor must have a unique name.
- All tasks must have a unique name.

Name	Type	Comments	(xref)
proc-utilization	float	Processor utilization for a task, entry, or activity.	§2.16
proc-waiting	float	Waiting time at a processor for an activity.	§2.16
phaseX-proc-waiting	float	Waiting time at a processor for phase *X* of an entry.	§2.16)
open-wait-time	float	Waiting time for open arrivals.	§2.15
service-time	float	Activity service time.	§2.9
loss-probability	float	Probability of dropping an asynchronous message.	§2.3
phaseX-service-time	float	Service time for phase X of an entry.	§2.9
service-time-variance	float	Variance for an activity.	§2.10
phaseX-service-time-variance	float	Variance for phase *X* of an entry.	§2.10
phaseX-utilization	float	Utilization for phase *X* of an entry.	§2.14
prob-exceed-max-service-time	float		§2.12
squared-coeff-variation	float	Squared coefficient of variation over all phases of an entry	§2.10
throughput-bound	float	Throughput bound for an entry.	§2.2
throughput	float	Throughput for a task, entry or activity.	§2.14
utilization	float	Utilization for a task, entry, activity.	§2.14
waiting	float	Rendezvous delay	§2.3
waiting-variance	float	Variance of delay for a rendezvous	§2.4

Table 3.19: Attributes for elements of type **ResultContentType**.

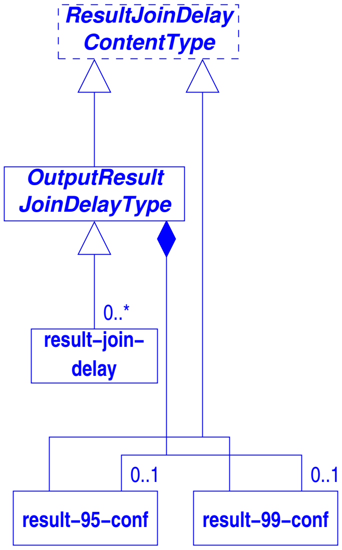


Figure 3.10: Schema diagram for type **OutputResultJoinDelayType**

Name	Type	Comments	(xref)
join-waiting	float	Join delay	§2.8
join-variance	float	Join delay variance	§2.8

Table 3.20: Attributes for elements of type **OutputResultJoinDelayType**.

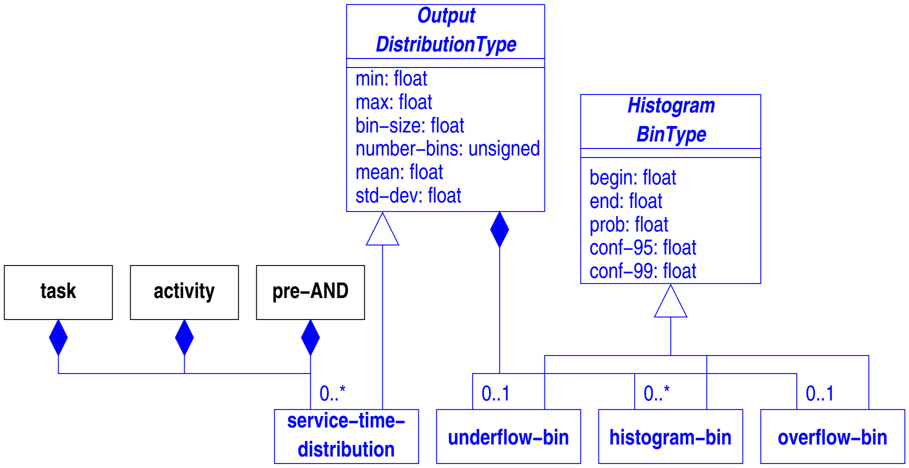


Figure 3.11: Schema for type **OutputDistributionType**.

Name	Type	Use	Default	Comments
min	float	required		The lower bound of the collected histogram data.
max	float	required		The upper bound of the collected histogram data.
number-bins	integer	optional	20	The number of bins in the distribution.
mid-point	float	optional		
bin-size	float	optional		

Table 3.21: Attributes for elements of type **OutputDistributionType**.

Name	Type	Comments	(xref)
begin	float	Lower limit of the bin.	
end	float	Upper limit of the bin.	
prob	float	The probability that the measured value lies within begin and end.	
conf-95	float		
conf-99	float		

Table 3.22: Attributes for elements of type **HistogramBinType**.

- All entries must have a unique name.
- All activities must have a unique name within a given task.
- All synchronous requests must have a valid destination.
- All asynchronous requests must have a valid destination.
- All forwarding requests must have a valid destination.
- All activity connections (in precedence blocks) must refer to valid activities.
- All activity replies must refer to a valid entry.
- All activity loops must refer to a valid activities.
- Each entry has only one activity bound to it.
- Phases are restricted to values one through three.
- All phase attributes within an entry must be unique.

Further validation is performed by the solver itself. Refer to Section 8 for the error messages generated.

One downside of using the Xerces XML parser library is that the Xerces tends to give rather cryptic error messages when compared to other tools. If an XML file fails to pass the validation phase, and the error looks cryptic, chances are very good that there is a genuine problem with the XML input file. Xerces has a bad habit of coming back with cryptic errors when constraint checking fails, and only gives you the general area in the file where the actual problem is.

One easy and convenient solution around this problem is to validate the XML file using another XML tool. Tools that have been found to give more user friendly feedback are XMLSpy (any edition), and XSDvalid (Java based, freely available). Another solution is to check if a particular tool can de-activate schema validation and rely on the actual tool to do its own internal error checking. Currently this is not supported in any of the LQN tools which are XML enabled, but it maybe implemented later on.

If the XML file validates using other tools, but fails validation with Xerces, or if the XML file fails validation on other tools, but passes with Xerces then please report the problem. The likelihood of validation passing with Xerces and not other tools will be much higher then the reverse scenario, because Xerces does not rigorously apply the XML Schema standard as other tools. Other sources of problems could be errors in the XML schema itself, or some unknown bug in the Xerces library.

Chapter 4

LQX Users Guide

4.1 Introduction to LQX

The LQX programming language is a general purpose programming language used for the control of input parameters to the Layer Queueing Network Solversystem for the purposes of sensitivity analysis. This language allows a user to perform a wide range of different actions on a variety of different input sources, and to subsequently solve the model and control the output of the resulting data.

4.1.1 Input File Format

The LQX programming language follows grammar rules which are very similar to those of ANSI C and PHP. The main difference between these languages and LQX is that LQX is a loosely typed language with strict runtime type-checking and a lack of variable coercion ("type casting"). Additionally, variables need not be declared before their first use. They do, however, have to be initialized. If they are un-initialized prior to their first use, the program will fail.

Comment Style

LQX supports two of the most common commenting syntaxes, "C-style" and "C++-style." Any time the scanner discovers two forward slashes side-by-side (//), it skips any remaining text on that line (until it reaches a newline). These are "C++-style" comments. The other rule that the scanner uses is that should it encounter a forward slash followed by an asterisk ("/*"), it will ignore any text it finds up until a terminating asterisk followed by a slash ("*/"). The preferred commenting style in LQX programs is to use "C++-style" comments for single-line comments and to use "C-style" comments where they span multiple lines. This is a matter of style.

Intrinsic Types

There are five intrinsic types in the LQX programming languages:

- **Number**: All numbers are stored in IEEE double-precision floating point format.
- **String**: Any literal values between (") and (") in the input.
- **Null**: This is a special type used to refer to an "empty" variable.
- **Boolean**: A type whose value is limited to either "true" or "false."
- **Object**: An semi-opaque type used for storing complex objects. See "Objects."
- **File Handle** File handles to open files for writing/appending or reading. See "File Handles."

LQX also supports a pseudo-intrinsic "Array" type. Whereas for any other object types, the only way to interact with them is to explicitly invoke a method on them, objects of type Array may be accessed with `operator []` and with `operator []=`, in a familiar C- and C++-style syntax.

The Object type also allows certain attributes to be exposed as "properties." These values are accessed with the traditional C-style `object.property` syntax. An example property is the `size` property for an object of type Array, accessed as `array.size` Only instances of type Object or its derivatives have properties. Number, String, Null and Boolean instances all have no properties.

Built-in Operators

There are eight built in arithmetic operators in the LQX programming language:

- **«**: Shift the left operand *left* by the amount specified by the right operand. Both operands must be non-negative integers.
- **»**: Shift the left operand *right* by the amount specified by the right operand. Both operands must be non-negative integers.
- `+`: Add the left operand to the right operand.
- `-`: Subtract the right operand from the left operand.
- `*`: Multiply the left operand by the right operand.
- `/`: Divide the left operand by the right operand.
- `%`: Take the modulus of the left operand by the right operand. This operation is implemented using the `fmod()` function, so both operands can be real numbers.
- `**`: Raise the left operand by the right operand. This operation is implemented using the `power()` function.

All operands must be numeric.

There are six built in comparison operators in the LQX programming language:

- `==`: Return *true* if the left operand equals the right operand.
- `!=`: Return *true* if the left operand is not equal to the right operand.
- `<=`: Return *true* if the left operand is less than or equal to the right operand.
- `>=`: Return *true* if the left operand is greater than or equal to the right operand.
- `>`: Return *true* if the left operand is greater than the right operand.
- `<`: Return *true* if the left operand is less than the right operand.

All operands must be numeric.

There are three built in logical operators in the LQX programming language:

- `!`: Return *true* if the operand is false.
- `&&`: Return *true* if the left and right operands are true, otherwise return false. Short-circuit evaluation is used so if the left operand evaluates to false, the right operand is not evaluated.
- `||`: Return *true* if either the left or right operand is true, otherwise return false. Short-circuit evaluation is used so if the left operand evaluates to true, the right operand is not evaluated.

All operands must be boolean.

There are nine built in assignement operators in the LQX programming language:

- **=**: Set the value of the right operand to the value of the left operand.

- **+=**: Equivalent to: `a = a + (b).`

- **-=**: Equivalent to: `a = a - (b).`

- ***=**: Equivalent to: `a = a * (b).` Note that `a *= b + c` is not necessarily the same as `a = a * b + c` because `*=` has lower precedence than `+`.

- **/=**: Equivalent to: `a = a / (b).`

- ****=**: Equivalent to: `a = a ** (b).`

- **«=**: Equivalent to: `a = a « (b).` The left and right operands must both be non-negative integers.

- **»=**: Equivalent to: `a = a » (b).` The left and right operands must both be

All operands must be numeric.

Operator Precedence and Associativity

pre	operator	associativity
1	`()`	left
2	`[]`	left
3	`!`	left
4	`**`	right
5	`*`, `/`, `%`	left
6	`+`, `-`	left
7	«, »	left
8	>=, <=<, >	left
9	`== !=`	left
10	`&&`	left
11	`||`	left
12	=, +=, -=, *=, /=, %=, **=, «=, »=	none

Arrays and Iteration

The built-in Array type is very similar to that used by PHP. It is actually a hash table, also known as a “Dictionary” or a “Map” for which you may use any object as a key, and any object as a value. It is important to realize that different types of keys will reference different entries. That is to say that `integer 0` and `string "0"` will not yield the same value from the Array when used as a key.

The Array object exposes a couple of convenience APIs, as detailed in Section 4.2. These methods are simply short-hand notation for the full function calls they replace, and provide no additional functionality. Arrays may be created in three different ways:

- `array_create(...)` and `array_create_map(key,value,...)`:
  The explicit, but long and wordy way of creating an array of objects or a map is by using the standard functional API. `array_create(...)` takes an arbitrary number of parameters (from 0 up to the maximum specified, for all practical purposes infinity), and returns a new Array instance consisting of `[0=>arg1, 1=>arg2, 2=>arg3, ...].`

  The other function, `array_create_map(key,value,...)` takes an even number of arguments, from 0 to 2n. The first argument is used as the key, and the second argument used as the value for that key, and so on. The resulting Array instance consists of `[arg1=>arg2, arg3=>arg4, ...].` Both of these methods are documented in Section 4.2.

- `[arg1, arg2, ...]`: Shorthand notation for `array_create(...)`

- `{k1=>v1, k2=>v2, ...}`: Shorthand notation for `array_create_map(...)`

The LQX language supports two different methods of iterating over the contents of an Array. The first involves knowing what the keys in the array actually are. This is a “traditional” iteration.

```
/* Traditional Array Iteration */
for (idx = 0; key < array.size; idx=idx+1) {
  print("Key ", idx, " => ", array[idx]);
}
```

In the above code snippet, we assume there exists an array which contains `n` values, stored at indexes 0 through `n-1`, continuously. However, the language provides a more elegant method for iterating over the contents of an array which does not require prior knowledge of the contents of the array. This is known as a “`foreach`” loop. The statement above can be rewritten as follows:

```
/* More modern array itteration */
foreach (key, value in array) {
  print("Key ", key, " => ", value);
}
```

This method of iteration is much cleaner and is the recommended way of iterating over the contents of an array. However, there is little guarantee of the order of the results in a `foreach` loop, especially when keys of multiple different types are used.

Type Casting

The LQX programming language provides a number of built-in methods for converting between variables of different types. Any of these methods support any input value type except for the Object type. The following is a non-extensive list of use cases for each of the different type casting methods and the results. Complete documentation is provided in Section 4.2.

str(...)	
`str()`	“”
`str(1.0)`	“1”
`str(1.0, "+", true)`	“1+true”
`str([1.0, "t"])`	“[0=>1, 1=>t]”
`str(null)`	“(null)”

double(?)	
`double(1.0)`	1.0
`double(null)`	0.0
`double("9")`	9.0
`double(true)`	1.0
`double([0])`	`null`

boolean(?)	
`boolean(1.0)`	`true`
`boolean(17.0)`	`true`
`boolean(-9.0)`	`true`
`boolean(0.0)`	`false`
`boolean(null)`	`false`
`boolean("yes")`	`true`
`boolean(true)`	`true`
`boolean([0])`	`null`

Keywords

The following strings are keywords in the language:

Control Flow : `break`, `else`, `foreach`, `for`, `function`, `if`, `in`, `return`, `while`.

Constants : `NULL`, `false`, `null`, `true`.

File Input/Output : `append`, `file_close`, `file_open`, `print_spaced`, `println_spaced`, `println`, `print`, `read_data`, `read_loop`, `read`, `write`.

User-Defined Functions

The LQX programming language has support for user-defined functions. When defined in the language, functions do not check their arguments types so every effort must be taken to ensure that arguments are the type that you expect them to be. The number of arguments will be checked. Variable-length argument lists are also supported with the use of the ellipsis (`...`) notation. Any arguments given that fall into the ellipsis are converted into an array named (`_va_list`) in the functions' scope. This is a regular instance of Array consisting of 0 or more items and can be operated on using any of the standard operators.

User-defined functions do **not** have access to any variables except their arguments and External ($-prefixed) and Constant (@-prefixed) variables. Any additional variables must be passed in as arguments, and all values must be returned. All arguments are in **only**. There are no out or inout arguments supported. All arguments are copied, pass-by-value. The basic syntax for declaring functions is as follows:

```
function <name>(<arg1>, <arg2>, ...) {
  <body>
  return (value);
}
```

You can return a value from a function anywhere in the body using the `return` function. A function which reaches the end of its body without a call to return will automatically return NULL. `return()` is a function, not a language construct, and as such the brackets are required. The number of arguments is not limited, so long as each one has a unique name there are no other constraints.

4.1.2 Program Input/Output and External Control

The LQX language allows users to write formatted output to external files and standard output and to read input data from external files/pipes and standard input. These features may be combined to allow LQNX to be controlled by a parent process as a child process providing model solving functionality. These capabilities will be described in the following sections.

File Handles

The LQX language allows users to open files for program input and output. Handles to these open files are stored in the symbol table for use by the print() functions for file output and the read_data() function for data input. Files may be opened for writing/appending or for reading. The LQX interpreter keeps track of which file handles were opened for writing and which were opened for reading.

The following command opens a file for writing. If it exists it is overwritten. It is also possible to append to an existing file. The three options for the third parameter are `write`, `append`, and `read`.

```
file_open( output_file1, "test_output_99-peva.txt", write );
```

To close an open file handle the following command is used:

```
file_close( output\_file1 );
```

File Output

Program output to both files and standard output is possible with the print functions. If the first parameter to the functions is an existing file handle opened for writing output is directed to that file. If the first parameter is not a file handle output is sent to standard output. Standard output is useful when it is desired to control LQNX execution from a parent process using pipes. If the given file handle has been opened for reading instead of writing a runtime error results.

There are four variations of print commands with two options. One option is a newline at the end of the line. It is possible to specify additional newlines with the `endl` parameter. The second option is controlling the spacing between columns either by specifying column widths in integers or supplying a text string to be placed between columns.

The basic print functions are `print()` and `println()` with the `ln` specifying a newline at the end.

```
println( output_file1, "Model run #: ", i, " t1.throughput: ", t1.throughput );

print( output_file1, "Model run #: ", i, " t1.throughput: ", t1.throughput, endl );
```

It should be noted that with the extra `endl` parameter both of these calls will produce the same output. The acceptable inputs to all print functions are valid file handles, quoted strings, LQX variables that evaluate to numerical or boolean values (or expressions that evaluate to numerical/boolean values) as well as the newline specifier `endl`. Parameters should be separated by commas.

To print to standard output no file handle is specified as follows:

```
println( "subprocess lqns run #: ", i, " t1.throughput: ", t1.throughput );
```

To specify the content between columns the print functions `print_spaced()` and `println_spaced()` are used. The first parameter after the file handle (the second parameter when a file handle is specified) is used to specify either column widths or a text string to be placed between columns. If no file handle is specified as when printing to standard output then the first parameter is expected to be the spacing specifier. The specifier must be either an integer or a string.

The following `println_spaced()` command specifies the string `", "` to be placed between columns. It could be used to create comma separated value (csv) files.

```
println_spaced( output_file2, ", ", $p1, $p2, $y1, $y2, t1.throughput );
```

Example output: 0, 2, 0.1, 0.05, 0.0907554

The following `println_spaced()` command specifies the integer 12 as the column width.

```
println_spaced( output_file3, 12, $p1, $p2, $y1, $y2, t1.throughput );
```

Reading Input Data from Files/Pipes

Reading data from input files/pipes is done with the `read_data()` function. Data can either be read from a valid file handle that has been opened for reading or from standard input. Reading data from standard input is useful when is useful when it is desired to control LQNX execution from a parent process using pipes. If the given file handle has been opened for writing rather than reading a runtime error results. The first parameter is either a valid file handle for reading or the strings `stdout` or `-` specifying standard input. The data that can be read can be either numerical values or boolean values.

There are two forms in which the `read_data()` function can be used. The first is by specifying a list of LQX variables which correspond to the expected inputs from the file/pipe. This requires the data inputs from the pipe to be in the expected order.

```
read_data( input_file, y, p, keep_running );
```

The second form in which the `read_data()` function can be used is much more robust. It can go into a loop attempting to read string/value pairs from the input pipe until a termination string `STOP_READ` is encountered. The string must corespond to an existing LQX variable (either numeric or boolean) and the corresponding value must be of the same type.

```
read_data( stdin, read_loop );
```

Sample input:

```
y 10.0 p 1.0 STOP_READ
continue_processing false STOP_READ
```

Controlling LQNX from a Parent Process

The file output and data reading functions can be combined to allow an LQNX process to be created and controlled by a parent process through pipes. Input data can be read in from pipes, be used to solve a model with those parameters and the output of the solve can be sent back through the pipes to the parent process for analysis. A LQX program can easily be written to contain a main loop that reads input, solves the model, and returns output for analysis. The termination of the loop can be controlled by a boolean flag that can be set from the parent process.

This section describes an example of how to control LQNX execution from a parent process, in this case a `perl` script which uses the `open2()` function to create a child process with both the standard input and output mapped to file handles in the `perl` parent process. This allows data sent from the parent to be read with `read_data( stdin, ...)` and output from the LQX print statements sent to standard output to be received for analysis in the parent.

This also provides synchronization between the parent and the child LQNX processes. The `read_data()` function blocks the LQNX process until it has received its expected data. Similarly the parent process can be programmed to wait for feedback from the child LQNX process before it continues.

The following is an example perl script that can be used to control a LQNX child process.

```
1   #!/usr/bin/perl -w
2   # script to test the creation and control of an lqns solver subprocess
3   # using the LQX language with synchronization
4
5   use FileHandle;
6   use IPC::Open2;
7
8   @phases = ( 0.0, 0.25, 0.5, 0.75, 1.0 );
9   @calls = ( 0.1, 3.0, 10.0 );
10
11  # run lqnx as subprocess receiving data from standard input
12  open2( *lqnxOutput, *lqnxInput, "lqnx 99-peva-pipe.lqnx" );
13
14  for $call (@calls) {
15    for $phase (@phases) {
16      print( lqnxInput "y ", $call, " p ", $phase, " STOP_READ " );
17      while( $response = <lqnxOutput>) !~ m/subprocess lqns run/ ){}
18      print( "Response from lqnx subprocess: ", $response );
19    }
20  }
21
22  # send data to terminate lqnx process
23  print( lqnxInput "continue_processing false STOP_READ" );
```

The above program invokes the lqnx program with its input file as a child process with `open2()`. Two file handles are passed as parameters. These will be used to send data over the pipe to the LQNX process to be received as standard input and to receive feedback from the LQX program which it sends as standard output.

The while loop at line 17 waits for the desired feedback from the model solve before continuing. This example uses stored data but a real application such as optimization would need to analyze the feedback data to decide which data to send back in the next iteration therefore this synchronization is important.

When the data is exhausted the LQNX process needs to be told to quit. This is done with the final print statement which sets the continue_processing flag to false. This causes the main loop in the LQX program which follows to quit.

```
<lqx><![CDATA[

i = 1;
p = 0.0;
y = 0.0;
continue_processing = true;

while ( continue_processing ) {

  read_data( stdin, read_loop ); /* read data from input pipe */

  if( continue_processing ) {

    $p1 = 2.0 * p;
    $p2 = 2.0 * (1 - p);
    $y1 = y;
    $y2 = 0.5 * y;
    solve();

    /* send output of solve through stdout through pipe */
    println( "subprocess lqns run #: ", i, " t1.throughput: ", t1.throughput );
    i = i + 1;
  }
}
]]></lqx>
```

The variables `p`, `y`, and `continue_processing` all need to be initialized to their correct types before the loop begins as they need to exist when the `read_data()` function searches for them in the symbol table. This is necessary as they are all local variables. External variables that exist in the LQN model such as `$p` and `$y` don't need initialization.

4.1.3 Writing Programs in LQX

Hello, World Program

A good place to start learning how to write programs in LQX is of course the traditional Hello World program. This would actually be a single line, and is not particularly interesting. This would be as follows:

```
println("Hello, World!");
```

The "`println()`" function takes an arbitrary number of arguments of any type and will output them (barring a file handle as the first parameter) to standard output, followed by a newline.

Fibonacci Sequence

This particular program is a great example of how to perform flow control using the LQX programming language. The Fibonacci sequence is an extremely simple infinite sequence which is defined as the following piecewise function:

$$\mathrm{fib}(X) = \begin{cases} 1 & x = 0, 1 \\ \mathrm{fib}(x-1) + \mathrm{fib}(x-2) & \text{otherwise} \end{cases} \tag{4.1}$$

Thus we can see that the Fibonacci sequence is defined as a recursive sequence. The naive approach would be to write this code as a recursive function. However, this is extremely inefficient as the overhead of even simple recursion in LQX can be substantial. The best way is to roll the algorithm into into a loop of some type. In this case, the loop is terminated when we have reached a target number in the Fibonacci sequence { 1, 1, 2, 3, 5, 8, 13, 21, ...}.

```
/* Initial Values */
fib_n_minus_two = 1;
fib_n_minus_one = 1;
fib_n = 0;

```

```
/* Loop until we reach 21 */
while (fib_n < 21) {
  fib_n = fib_n_minus_one + fib_n_minus_two;
  fib_n_minus_two = fib_n_minus_one;
  fib_n_minus_one = fib_n;
  println("Currently: ", fib_n);
}
```

As you can see, this language is extremely similar to C or PHP. One of the few differences as far as expressions are concerned is that pre-increment/decrement and post-increment/decrement are not supported. Neither are short form expressions such as `+=, -=, *=, /=`, etc.

Re-using Code Sections

Many times, there will be code in your LQX programs that you would like to invoke in many places, varying only the parameters. The LQX programming language does provide a pretty standard functions system as described earlier. Bearing in mind the caveats (some degree of overhead in function calls, plus the inability to see global variables without having them passed in), we can make pretty ingenious use of user-defined functions within LQX code.

When defining functions, you can specify only the number of arguments, not their types, so you need to make sure things are what you expect them to be, or your code may not perform as you expect. We will begin by demonstrating a substantially shorter (but as described earlier) much less efficient implementation of the Fibonacci Sequence using functions and recursion.

```
function fib(n) {
  if (n == 0 || n == 1) { return (1); }
  return (fib(n-2) + fib(n-1));
}
```

Once defined, a function may be used anywhere in your code, even in other user defined functions (and itself | recursively). This particular example functions very well for the first 10-11 fibonacci numbers but becomes substantially slower due to the increased number of relatively expensive function invocations. *Remember*, `return()` is a function, not a language construct. The brackets are required.

A much more interesting use of functions, specifically those with variable length argument lists, is an implementation of the formula for standard deviation of a set of values:

```
function average(/*Array<double>*/ inputs) {
  double sum = 0.0;
  foreach (v in inputs) { sum = sum + v; }
  return (sum / inputs.size);
}

function stdev(/*boolean*/ sample, ...) {
  x_bar = average(_va_list);
  sum_of_diff = 0.0;

  /* Figure out the divisor */
  divisor = _va_list.size;
  if (sample == true) {
    divisor = divisor - 1;
  }

  /* Compute sum of difference */
  foreach (v in _va_list) {
    sum_of_diff = sum_of_diff + pow(v - x_bar, 2);
  }
```

```

    return (pow(sum_of_diff / divisor, 0.5));
  }
```

You can then proceed to compute the standard deviation of the variable length of arguments for either sample or non-sample values as follows, from anywhere in your program after it has been defined:

```
  stdev(true,  1, 2, 5, 7, 9, 11);
  stdev(false, 2, 9, 3, 4, 2);
```

Using and Iterating over Arrays

As mentioned in the "Arrays and Iteration" under section 1.1 of the Manual, LQX supports intrinsic arrays and `foreach` iteration. Additionally, any type of object may be used as either a key or a value in the array. The following example illustrates how values may be added to an array, and how you can iterate over its contents and print it out. The following snippet creates an array, stores some key-value pairs with different types of keys and values, looks up a couple of them and then iterates over all of them.

```
  /* Create an Array */
  array = array\_create();

  /* Store some key-value pairs */
  array[0] = "Slappy";
  array[1] = "Skippy";
  array[2] = "Jimmy";

  /* Iterate over the names */
  foreach ( index,name in array ) {
    print("Chipmunk #", index, " = ", name);
  }

  /* Store variables of different types, shorthand */
  array = {true => 1.0, false => 3.0, "one" => true, "three" => false}

  /* Shorthand indexed creation with iteration */
  foreach (value in [1,1,2,3,5,8,13]) {
    print ("Next fibonacci is ", value);
  }
```

4.1.4 Actual Example of an LQX Model Program

The following LQX code is the complete LQX program for the model designated `peva-99`. The model itself contains a few model parameters which the LQX code configures, notably `$p1`, `$p2`, `$y1` and `$y2`. The LQX program is responsible for setting the values of all model parameters at least once, invoking solve and optionally printing out certain result values. Accessing of result values is done via the LQNS bindings API documented in Section 4.2.

The program begins by defining an array of values that it will be setting for each of the external variables. By enumerating as follows, the program will set the variables for the cross product of `phase` and `calls`.

```
  phase = [ 0.0, 0.25, 0.5, 0.75, 1.0 ];
  calls = [ 0.1, 3.0, 10.0 ];
  foreach ( idx,p in phase ) {
    foreach ( idx,y in calls ) {
```

Next, the program uses the input values `p` and `y` to compute the values of `$p1`, `$p2`, `$y1` and `$y2`. Any assignment to a variable beginning with a `$` requires that variable to have been defined externally, within the model definition. When such an assignment is made the value of the right-hand side is effectively put everywhere the left-hand side is found within the model.

```
    $p1 = 2.0 * p;
    $p2 = 2.0 * (1 - p);
    $y1 = y;
    $y2 = 0.5 * y;
```

Since all variables have now been set, the program invokes the solve function with its optional parameter, the suffix to use for the output file of the current run. This particular program outputs `in.out-$p1-$p2-$y1-$y2` files, so that results for a given set of input values can easily be found. As shown in the documentation in Section 3, `solve(<opt> suffix)` will return a boolean indicating whether or not the solution converged, and this program will abort when that happens, although that is certainly not a requirement.

```
    if (solve(str($p1,"-",$p2,"-",$y1,"-",$y2)) == false) {
      println("peva-99.xml:LQX: Failed to solve the model properly.");
      abort(1, "Failed to solve the model.");
    } else {
```

The remainder of the program outputs a small table of results for certain key values of interest to the person running the solution using the APIs in Section 3.

```
      t0 = task("t0");
      p0 = processor("p0");
      e0 = entry("e0");
      ph1 = phase(e0, 1);
      ctoe1 = call(ph1, "e1");
      println("+-------------------------------------+");
      println("t0 Throughput:  ", t0.throughput        );
      println("t0 Utilization: ", t0.utilization       );
      println("+                -----                +");
      println("e0 Throughput:  ", e0.throughput        );
      println("e0 TP Bound:    ", e0.throughput_bound  );
      println("e0 Utilization: ", e0.utilization       );
      println("+                -----                +");
      println("ph Utilization: ", ph1.utilization      );
      println("ph Svt Variance:", ph1.service_time_variance );
      println("ph Service Time:", ph1.service_time     );
      println("ph Proc Waiting:", ph1.proc_waiting     );
      println("+                -----                +");
      println("call Wait Time: ", ctoe1.waiting_time   );
      println("+-------------------------------------+");
    }
  }
}
```

4.2 API Documentation

4.2.1 Built-in Class: Array

Summary of Attributes		
numeric	`size`	The number of key-value pairs stored in the array.

Summary of Constructors		
object[Array]	`array_create(...)`	This method returns a new instance of the Array class, where each the first argument to the method is mapped to index numeric(0), the second one to numeric(1) and so on, yielding `[0=>arg0, 1=>arg1, ...]`
object[Array]	`array_create_map(k,v,...)`	This method returns a new instance of the Array class where the first argument to the constructor is used as the key, and the second is used as the value, and so on. The result is a n array `[arg0=>arg1, arg2=>arg3,...]`

Summary of Methods		
null	`array_set(object[Array] a, ? key, ? value)`	This method sets the value `value` of any type for the key `key` of any type, for array `a`. The shorthand notation for this operation is to use the `operator []`.
ref<?>	`array_get(object[Array] a, ? key)`	This method obtains a reference to the slot in the array `a` for the key `key`. If there is no value defined in the array yet for the given key, a new slot is created for that key, assigned to NULL, and a reference returned.
boolean	`array_has(object[Array] a, ? key)`	Returns whether or not there is a value defined on array `a` for the given key, `key`.

4.2.2 Built-in Global Methods and Constants

Intrinsic Constants

Summary of Constants		
double	`@infinity`	IEEE floating-point numeric infinity.
double	`@type_un`	The type_id for an Undefined Variable.
double	`@type_boolean`	The type_id for a Boolean Variable.
double	`@type_double`	The type_id for a Numeric Variable.
double	`@type_string`	The type_id for a String Variable.
double	`@type_null`	The type_id for a Null Variable.

General Utility Functions

Summary of Methods		
null	`abort(numeric n, string r)`	This call will immediately halt the flow of the program, with failure code `n` and description string `r`. This cannot be "caught" in any way by the program and will result in the interpreter not executing any more of the program.
null	`copyright()`	Displays the LQX copyright message.
null	`print_symbol_table()`	This is a very useful debugging tool which output the name and value of all variables in the current interpreter scope.
null	`print_special_table()`	This is also a useful debugging tool which outputs the name and value of all special (External and Constant) variables in the interpreter scope.
numeric	`type_id(? any)`	This method returns the Type ID of any variable, including intrinsic types (numeric, boolean, null, etc.) and the result can be matched to the constants prefixed with @type (@type_null, @type_un, @type_double, etc.)
null	`return(? any)`	This method will return any value from a user-defined function. This method cannot be used in global scope.

Numeric/Floating-Point Utility Functions

Summary of Methods		
numeric	`abs(numeric n)`	Returns the absolute value of the argument `n`
numeric	`ceil(numeric n)`	Returns the value of `n` rounded up.
numeric	`exp(numeric n)`	Returns e^n.
numeric	`floor(numeric n)`	Returns the value of `n` rounded down.
numeric	`log(numeric n)`	Returns $\log(n), n > 0$ (natural log).
numeric	`max(array a)`	Returns the largest value found in the array `a`.
numeric	`max(numeric a, numeric b, ...)`	Returns the largest value among the numeric args.
numeric	`min(array a)`	Returns the smallest value found in the array `a`.
numeric	`min(numeric a, numeric b, ...)`	Returns the smallest value among the numeric args.
numeric	`pow(numeric bas, numeric x)`	Returns `bas` to the power `x`.
numeric	`rand()`	Returns a random number between 0 and 1.
numeric	`round(numeric n)`	Returns the value of `n` rounded to the nearest integer.
numeric	`sqrt(numeric n)`	Returns $\sqrt{n}, n \geq 0$.
numeric	`normal(numeric a, numeric b)`	Returns a normally distributed random number with mean of `a` and a standard deviation of `b`.
numeric	`gamma(numeric a, numeric b)`	Returns a Gamma distributed random number with a mean of `a` and a shape of `b`.
numeric	`uniform(numeric a, numeric b)`	Returns a uniformily distributed random number between `a` and `b`.
numeric	`poisson(numeric a)`	Returns a Poisson distributed random number with a mean of `a`.

Type-casting Functions

Summary of Methods		
string	`str(...)`	This method will return the same value as the function `print(...)` would have displayed on the screen. Each argument is coerced to a string and then adjacent values are concatenated.
numeric	`double(? x)`	This method will return 1.0 or 0.0 if provided a boolean of `true` or `false` respectively. It will return the passed value for a double, 0.0 for a null and fail (NULL) for an object. If it was passed a string, it will attempt to convert it to a double. If the whole string was not numeric, it will return NULL, otherwise it will return the decoded numeric value.
boolean	`bool(? x)`	This method will return `true` for a numeric value of (not 0.0), a boolean `true` or a string "true" or "yes". It will return `false` for a numeric value 0.0, a NULL or a string "false" or "no", or a boolean `false`. It will return NULL otherwise.

4.3 API Documentation for the LQN Bindings

4.3.1 LQN Class: Document

Summary of Attributes		
Read-Write Attributes		
string	`comment`	The model comment.
double	`conv_val`	The model convergence value for lqns.
double	`it_limit`	The iteration limit for lqns.
double	`print_int`	Iteration numbers where intermediate results are generated.
double	`underrelax_coeff`	The underrelaxation coefficient for lqns.
double	`seed_value`	The initial seed value for the random number generator for lqsim.
double	`number_of_blocks`	
double	`block_time`	
double	`precision`	
double	`warm_up_loops`	
double	`warm_up_time`	
Read-Only Attributes		
double	`iterations`	The number of solver iterations/simulation blocks.
double	`invocation`	The solution invocation number.
double	`system_cpu_time`	Total system time for this invocation.
double	`user_cpu_time`	Total user time for this invocation.
double	`elapsed_time`	Total elapsed time for this invocation.
boolean	`valid`	True if the results are valid.
double	`waits`	The number of times `wait()` was called.

Summary of Constructors		
Document	`document()`	Returns the Document object

4.3.2 LQN Class: Processor

Summary of Attributes		
double	`utilization`	The utilization of the Processor

Summary of Constructors		
Processor	`processor(string name)`	Returns an instance of Processor from the current LQN model with the given name.

4.3.3 LQN Class: Group

Summary of Attributes		
double	`utilization`	The utilization of the Group

Summary of Constructors		
Group	`processor(string name)`	Returns an instance of Group from the current LQN model with the given name.

4.3.4 LQN Class: Task

Summary of Attributes		
double	`throughput`	The throughput of the Task
double	`utilization`	The utilization of the Task
double	`proc_utilization`	This Task's processor utilization
Array	`phase_utilizations`	Individual phase utilizations

Summary of Constructors		
Task	`task(string name)`	Returns an instance of Task from the current LQN model with the given name.

4.3.5 LQN Class: Entry

Summary of Attributes		
boolean	`has_phase_1`	Whether the entry has a phase 1 result
boolean	`has_phase_2`	Whether the entry has a phase 2 result
boolean	`has_phase_3`	Whether the entry has a phase 3 result
boolean	`has_open_wait_time`	Whether the entry has an open wait time
double	`phase1_proc_waiting`	Phase 1 Processor Wait Time
double	`phase1_service_time_variance`	Phase 1 Service Time Variance
double	`phase1_service_time`	Phase 1 Service Time
double	`phase1_utilization`	Phase 1 (task) Utilization
double	`phase1_pr_time_exceeded`	Phase 1 Max Service Time Exceeded
double	`phase2_proc_waiting`	Phase 2 Processor Wait Time
double	`phase2_service_time_variance`	Phase 2 Service Time Variance
double	`phase2_service_time`	Phase 2 Service Time
double	`phase2_utilization`	Phase 2 (task) Utilization
double	`phase2_pr_time_exceeded`	Phase 2 Max Service Time Exceeded
double	`phase3_proc_waiting`	Phase 3 Processor Wait Time
double	`phase3_service_time_variance`	Phase 3 Service Time Variance
double	`phase3_service_time`	Phase 3 Service Time
double	`phase3_utilization`	Phase 3 (task) Utilization
double	`phase3_pr_time_exceeded`	Phase 3 Max Service Time Exceeded
double	`proc_utilization`	Entry processor utilization
double	`squared_coeff_variation`	Squared coefficient of variation
double	`throughput_bound`	Entry throughput bound
double	`throughput`	Entry throughput
double	`utilization`	Entry utilization
double	`waiting`	Entry open wait time

Summary of Constructors		
Entry	`entry(string name)`	Returns the Entry object for the model entry whose name is given as name

4.3.6 LQN Class: Phase

Summary of Attributes		
double	`service_time`	Phase service time
double	`service_time_variation`	Phase service time variance
double	`utilization`	Phase utilization
double	`proc_waiting`	Phase processor waiting time
double	`pr_time_exceeded`	Phase Max Service Time Exceeded

Summary of Constructors		
Phase	`phase(object entry, numeric_int nr)`	Returns the Phase object for a given entry's phase number specified as nr

4.3.7 LQN Class: Activity

Summary of Attributes		
double	`proc_utilization`	The activities' share of the processor utilization
double	`proc_waiting`	Activities' processor waiting time
double	`service_time_variance`	Activity service time variance
double	`service_time`	Activity service time
1091 double	`pr_time_exceeded`	Activity Max Service Time Exceeded
double	`squared_coeff_variation`	The square of the coefficient of variation
double	`throughput`	The activity throughput
double	`utilization`	Activity utilization

Summary of Constructors		
Activity	`activity(object task, string name)`	Returns an instance of Activity from the current LQN model, whose name corresponds to an activity in the given task.

4.3.8 LQN Class: Call

Summary of Attributes		
double	`waiting`	Call waiting time
double	`waiting_variance`	Call waiting time
double	`loss_probability`	Message loss probability for asynchronous messages

Summary of Constructors		
Call	`call(object phase, string destinationEntry)`	Returns the call from an entry's phase (phase) to the destination entry whose name is (dest).
Call	`call(object activity, string destinationEntry)`	Returns the call from a task's activity (activity) to the destination entry whose name is (dest)

4.3.9 Pragmas

Summary of Attributes		
string	`value`	Value of pramga.

Summary of Constructors		
Pragma	`pragma(string pragma)`	Returns the value for the pragma supplied as an argument.

4.3.10 Confidence Intervals

Summary of Constructors		
conf_int	`conf_int(object, int level)`	Returns the ± (level) for the attribute for the object

Chapter 5

LQN Input File Format

This Chapter describes the original ‘SRVN’ input file format, augmented with the Software Performance EXperiment driver (SPEX) grammar. In this model format models are specified breadth-first, in contrast to the XML format described in §3 where models are specified depth-first. This specification means that all resources such as processors, tasks and entries, are defined before they are referenced. Furthermore, each resource is grouped into its own section in the input file. Figure 5.1 shows the basic schema and Listing 5.1 shows the basic layout of the model file.

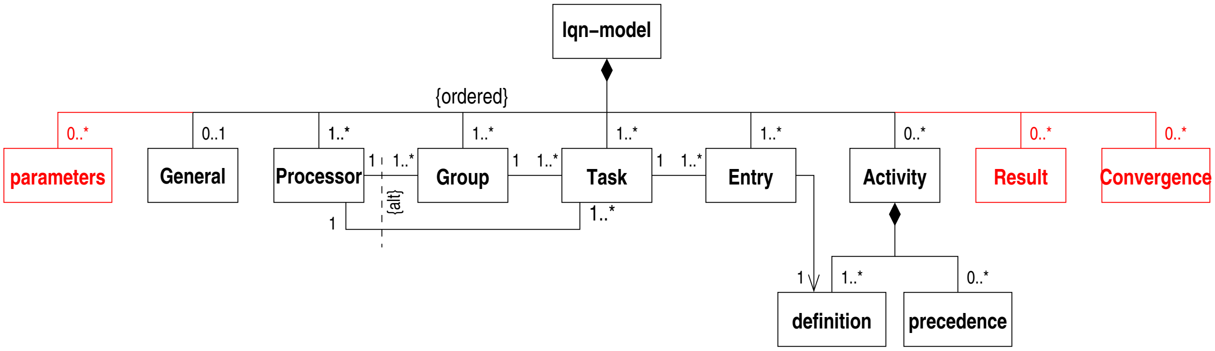


Figure 5.1: SRVN input schema

Each of the sections within the input file begins with a key-letter, as follows:

$ SPEX parameters (optional).

G General solver parameters (optional).

P Processor definitions.

U Processor group definitions (optional).

T Task definitions.

E Entry definitions.

A Task activity definitions (optional).

R SPEX result definitions (optional).

C SPEX convergence (optional).

Section 5.2 describes the input sections necessary to solve a model, i.e. `P`, `U` `T`, `E`, and `A`. Section 5.3 describes the additional input sections for solving multiple models using SPEX, i.e. `$`, `R`, and `C`. The complete input grammar is listed in Appendix A.

Listing 5.1: LQN file layout

```
# Pragmas
#pragma <param>=<value>

# Parameters (SPEX)
$var = <expression>
$var = [ <expression-list> ]

# General Information
G "<string>" <real> <int> <int> <real> -1

# Processor definitions
P 0
  p <proc-id> <sched> <opt-mult> <opt-repl> <opt-obs>
-1

# Group definitions
U 0
  g <group-id> <real> <opt-cap> <proc-id>
-1

# Task definitions
T 0
  t <task-id> <sched> <entry-list> -1 <proc-id> <opt-pri> <opt-think-time>
              <opt-mult> <opt-repl> <opt-grp> <opt-obs>
-1

# Entry definitions
E 0
  A <activity-id>
  s <entry-id> <real> ... -1 <opt-obs>
  y <entry-id> <entry-id> <real> ... -1 <opt-obs>
-1

# Activity definitions
A <task-id>
  s <activity-id> <real> <opt-obs>
  y <activity-id> <entry-id> <real> <opt-obs>
:
  <activity-list> -> <activity-list>
-1

# Result defintions (SPEX)
R 0
  plot( $var, $var,... )
  $var = <expression>
-1

# Convergence defintions (SPEX)
C 0
  $var = <expression>
-1
```

5.1 Lexical Conventions

The section describes the lexical conventions of the SRVN input file format.

5.1.1 White Space

White space, such as spaces, tabs and new-lines, is ignored except within strings. Object definitions can span multiple lines.

5.1.2 Comments

Any characters following a hash mark (#) through to the end of the line are considered to be a comment and are generally ignored. However, should a line begin with optional whitespace followed by '`#pragma`', then the remainder of the line will be treated by the solver as a pragma (more on pragmas below).

5.1.3 Identifiers

Identifiers are used to name the objects in the model. They consist of zero or more leading underscores ('`_`'), followed by a character, followed by any number of characters, numbers or underscores. Punctuation characters and other special characters such as the dollar-sign ('`$`') are not permitted. Non-numeric identifiers must be a minimum of two characters in length[1] The following, `1`, `p1`, `p_1`, and `__P_21_proc` are valid identifiers, while `$proc` and `$1` are not.

5.1.4 Variables

Variables are used to set values of various objects such as the multiplicity of tasks and the service times of the phases of entries. Variables are modifed by SPEX (see §5.3) to run multiple experiments. Variables start with a dollar-sign ('`$`') followed by any number of characters, numbers or underscores. `$var` and `$1` are valid variables while `$$` is not.

5.2 LQN Model Specification

This section describes the mandatory and option input for a basic LQN model file. SPEX information, namely *Variables*, (§5.2.7), *Report Information* (§5.3.2) and *Convergence Information* (§5.3.3) are described in the section that follows. All input files are composed of three mandatory sections: *Processor Information* (§5.2.3), *Task Information* (§5.2.5) and *Entry Information* (§5.2.6), which define the processors, tasks and entries respectively in the model. All of the other sections for a basic model file are optional. They are: *Pragmas*, *General Information* (§5.2.2), *Group Information* (§5.2.4), and *Activity Information*. The syntax of these specifications are described next in the order in which they appear in the input model.

5.2.1 Pragmas

Any line beginning with optional whitespace followed by the word '`#pragma`'efines a pragma which is used by either the analytic solver or the simulator to change its behaviour. The syntax for a pramga directive is shown in line 2 in Listing 5.1. Pragma's which are not handled by either the simulator or the analytic solver are ignored. Pragma's can appear anywhere in the input file[2] though they typically appear first.

[1] Single characters are used for section and record keys.

[2] Pragma's are processed during lexical analysis.

5.2.2 General Information

The optional general information section is used to set various control parameters for the analytic solver LQNS. These parameters, with the exception of the model comment, are ignored by the simulator, lqsim. Listing 5.2 shows the format of this section. Note that these parameters can also be set using SPEX variables, described below in §5.3.1.

Listing 5.2: General Information

```
G "<string>"     # Model title.
  <real>         # convergence value
  <int>          # iteration limit
  <int>          # Optional print interval.
  <real>         # Optional under-relaxation.
-1
```

5.2.3 Processor Information

Processors are specified in the processor information section of the input file using the syntax shown in Listing 5.3. The start of the section is identified using "`P` <*int*>" and ends with "`-1`". The <*int*> parameter is either the number of processor definitions in this section, or zero[3].

Listing 5.3: Processor Information

```
1  P <int>
2    p <proc-id> <sched> <opt-mult>
3  -1
```

Each processor in the model is defined using the syntax shown in line 2 in Listing 5.3. Each record in this section beginning with a '`p`' defines a processor. <*proc-id*> is either an integer or an identifier (defined earlier in §5.1.3). <*sched*> is used to define the scheduling discipline for the processor and is one of the code letters listed in Table 5.1. The scheduling disciplines supported by the model are described in Section 1.1.1. Finally, the optional <*opt-mult*> specifies the number of copies of this processor serving a common queue. Multiplicity is specified using the syntax shown in Table 5.2. By default, a single copy of a processor is used for the model.

<***sched***>	**Scheduling Discipline**
`f`	First-come, first served.
`p`	Priority-preemptive resume.
`r`	Random.
`i`	Delay (infinite server).
`h`	Head-of-Line.
`c` <*real*>	Completely fair share with time quantum <*real*>.
`s` <*real*>	Round Robin with time quantum <*real*>.

Table 5.1: Processor Scheduling Disciplines (see §1.1.1).

5.2.4 Group Information

Groups are specified in the group information section of

Listing 5.4: Group Information

```
U <int>
  g <group-id> <real> <opt-cap> <proc-id>
```

[3] The number of processors, <*int*>, is ignored with all current solvers.

<opt-mult>	**Multiplicity**
m *<int>* i	*<int>* identical copies with a common queue. Infinite (or delay).

<opt-repl>	**Repliplication**
r *<int>*	*<int>* replicated copies with separate queues.

Table 5.2: Multiplicity and Replication (see §1.2).

```
-1
```

5.2.5 Task Information

Tasks are specified in the task information section of the input file using the syntax shown in Listing 5.5. The start of the task section is identified using "T *<int>*" and ends with "-1". The *<int>* parameter is either the number of task definitions in this section, or zero.

Listing 5.5: Task Information

```
1 T <int>
2   t <task-id> <sched> <entry-list> -1 <opt-queue-length> <opt-tokens>
3               <proc-id> <opt-pri> <opt-think-time>
4               <opt-mult> <opt-repl> <opt-grp>
5   I <task-id> <task-id> <int>    # fan-in for replication
6   O <task-id> <task-id> <int>    # fan-out for replication
7 -1
```

Each task definition within this section starts with a 't' and is is defined using the syntax shown in lines 2 and 3 of Listing 5.5[4]. *<task-id>* is an identifier which names the task. *<sched>* is used to define the request distribution for reference tasks, or the scheduling discipline for non-reference tasks. The scheduling and distribution code letters are shown in Table 5.3. Some disciplines are only supported by the simulator; these are identified using "†". *<entry-list>* is a list of idententifiers naming the entries of the task. The optional *<opt-pri>* is used to set the priority for the task provided that the processor running the task is scheduled using a priority discipline. The optional *<opt-think-time>* specifies a think time for a reference task. The optional *<opt-mult>* specifies the number of copies of this task serving a common queue. Multiplicity is specified using the syntax shown in Table 5.2. By default, a single copy of a task is used for the model. Finally, the optional *<opt-grp>* is used to identify the group that this task belongs to provided that the task's processor is using fair-share scheduling

5.2.6 Entry Information

Entries are specified in the entry information section starting from "E *<int>*" and ending with "-1". The *<int>* parameter is either the number of entry definitions in this section, or zero. Each record in the entry section defines a single parameter for an entry, such as its priority, or a single parameter for the phases of the entry, such as service time. Listing 5.6 shows the syntax for the most commonly used parameters.

All entry records start with a key letter, followed by an *<entry-id>*, followed by from one to up to five arguments. Table 5.5 lists all the possible entry specifiers. The table is split into six classes, based on the arguments to the parameter. Records used to specifiy service time and call rate parameters for phases take a list of from one to three arguments and terminated with a '-1'. All other entry records, with the exception of histogram information, take a fixed number of arguments. Records which only apply to the simulator are marked with a '†'.

[4] Line 3 is a continuation of line 2.

Reference tasks (customers).	
<sched>	**Request Distribution**
r	Poisson.
b	Bursty†.
u	Uniform†.

Non-Reference tasks (servers).	
<sched>	**Queueing Discipline**
n	First come, first served.
P	Polled service at entries†.
h	Head-of-line priority.
f	First come, first served.
i	Infinite (delay) server.
w	Read-Write lock task†.
S	Semaphore task†.

Table 5.3: Task Scheduling Disciplines (see §1.1.3).

Option	Parameter
<integer>	Task priority for tasks running on processors supporting priorities.
`z` *<real>*	Think Time for reference tasks.
`q` *<real>*	Maximum queue length for asynchronous requests.
`T` *<integer>*	Initial tokens at semaphore task†.
`m` *<integer>*	Task multiplicity.
`r` *<integer>*	Task replication.
`g` *<identifier>*	Group identifier for tasks running of processors with fair share scheduling.

Table 5.4: Optional parameters for tasks (see §1.1.3).

5.2.7 Activity Information

Activity information sections are required to specify the parameters and connectivity of the activities for a task. Note that unlike all other sections, each task with activities has its own activity information section.

An activity information section starts with "`A` *<task-id>*" and ends with "`-1`". The data within an activity information section is partitioned into two parts. The first part lists the parameter data for an activity in a fashion similar to the parameter data for an entry; the second section defines the connectivity of the activities. Listing 5.7 shows the basic syntax.

Listing 5.6: Entry Information

```
E <int>
  A <entry-id> <activity-id>            # Start activity.
  F <entry-id> <entry-id> <real>        # forward.
  s <entry-id> <real> ... -1            # Service time by phase.
  y <entry-id> <entry-id> <real> ... -1 # Synchronous request by phase.
-1
```

Key	Paramater	Arguments
One argument		
a *<entry-id>* *<real>*	Arrival Rate	
A *<entry-id>* *<activity-id>*	Start activity	
p *<entry-id>* *<int>*	Entry priority	
One to three phase arguments		
s *<entry-id>* *<real>* ... -1	Service Time.	The entry's *<entry-id>* and mean service time value per phase.
c *<entry-id>* *<real>* ... -1	Coefficient of Variation Squared.	The entry's *<entry-id>* and CV^2 value for each phase.
f *<entry-id>* *<int>* ... -1	Call Order	STOCHASTIC or DETERMINISTIC
M *<entry-id>* *<real>* ... -1	Max Service Time†	Output probability that the service time result exceeds the *<real>* parameter, per phase.
Arguments for a single phase		
H *<int>* *<entry-id>* *<real>* : *<real>* *<opt-int>*		Histogram†: An *<int>* phase, followed by a range from *<real>* to *<real>*, and an optional *<int>* buckets.
Destination and one argument		
F *<entry-id>* *<real>* -1	Forwarding Probability	Source and Destination entries, and probability reply is forwarded.
Destination and one to three phase arguments		
y *<entry-id>* *<entry-id>* *<real>* ... -1	Rendevous Rate	Source and Destination entries, and rate per phase.
z *<entry-id>* *<entry-id>* *<real>* ... -1	Send-no-Reply Rate	Source and Destination entries, and rate per phase.
Semaphores and Locks†		
P *<entry-id>*	Signal†	Entry *<entry-id>* is used to *signal* a semaphore task.
V *<entry-id>*	Wait†	
R *<entry-id>*	Read lock†	
U *<entry-id>*	Read unlock†	
W *<entry-id>*	Write lock†	
X *<entry-id>*	Write unlock†	

Table 5.5: Entry Specifiers

Listing 5.7: Activity Information

```
A <task-id>
  s <activity-id> <real>
  c <activity-id> <real>
  f <activity-id> <int>
  y <activity-id> <entry-id> <real>
  z <activity-id> <entry-id> <real>
:
  <activity-list> -> <activity-list>
-1
```

Key	Paramater	Arguments
one to three phase arguments		
`s`	Service Time.	The entry's <*entry-id*> and mean service time value per phase.
`c`	Coefficient of Variation Squared.	The entry's <*entry-id*> and CV^2 value for each phase.
`f`	Call Order	`STOCHASTIC` or `DETERMINISTIC`
Destination and one to three phase arguments		
`y`	Rendevous Rate	Source and Destination entries, and rate per phase.
`z`	Send-no-Reply Rate	Source and Destination entries, and rate per phase.

Table 5.6: Activity Specifiers

Post (or Join) lists	
<*activity-id*>	
<*activity-id*> + <*activity-id*> +...	
<*activity-id*> & <*activity-id*> &...	

Pre (or Fork) lists	
<*activity-id*>	
<*activity-id*> + <*activity-id*> +...	
<*activity-id*> & <*activity-id*> &...	

Table 5.7: Activity Lists

5.3 SPEX: Software Performance Experiment Driver

SPEX, the **S**oftware **P**erformance **Ex**P**e**riment driver, was originally a Perl program used to generate and solve multiple layered queueing network models. With version 5 of the solvers this functionality has been incorporated into the `lqiolib` and `lqx` libraries used by the simulator and analytic solver. The primary benefit of this change is that analytic solutions can run faster for reasons described in [11].

SPEX augments the input file described in §5.2 by adding *variables* for setting input values, a *Report Information* (§5.3.2) used to format output, and an optional *Convergence Information* (5.3.3) for feeding result values back into input variables. Listing 5.1 shows these sections starting with comments in red. The syntax of these sections are described next in the order in which they appear in the input model.

5.3.1 Variables

SPEX variables are used to set and possibly vary various input values to the model, and to record results from the solution of the model. There are four types of variables: control, scalar, array and observation. Control variables are used to define parameters that control the execution of the solver. Scalar and array variables are used to set input parameters to the model. Finally, observation variables are used to record results such as throughputs and utilizations.

Control Variables

Control variables are used to set parameters that are used to control either the analytic solver *lqns*, or the simulator *lqsim*. With the exception of `$comment`, all of these variables can be changed as SPEX executes, though this behaviour may not be appropriate in many cases. Table 5.8 lists all of the control variables accepted by SPEX. See Table 5.10 for control variables that are no longer recognized.

Scalar Variables

Scalar variables are used to set input values for the model and are initialized using any <*ternary-expression*> (`?:`) using this syntax:

Control Variable	**Type of Value**	**Default Value**	**Program**
`$model_comment`	*<string>*	""	
`$solver`	*<string>*	*deprecated*	
`$convergence_limit`	*<real>*	0.00001	lqns
`$iteration_limit`	*<int>*	100	lqns
`$print_interval`	*<int>*	1	lqns
`$underrelaxation`	*<real>*	0.9	lqns
`$block_time`	*<int>*	50000	lqsim
`$number_of_blocks`	*<int>*	1	lqsim
`$result_precision`	*<real>*	–	lqsim
`$seed_value`	*<int>*	–	lqsim
`$warm_up_loops`	*<int>*	–	lqsim
`$convergence_iters`	*<int>*	–	spex
`$convergence_under_relax`	*<real>*	–	spex

Table 5.8: Spex Control Variables

$name = <*ternary-expression*>

The <*ternary-expression*> may contain any variables defined previously or subsequently in the input file; order does not matter. However, undefined variables and observation variables (more on these below) are not permitted. If <*ternary-expression*> is an actual ternary expression, the test part must evaluate to a boolean. Refer to Appendix A, §A.1.10 for the complete grammar for <*ternary-expression*>.

Array Variables

Array variables are used to specify a range of values that an input parameter may take on. There are two ways to specify this information:

```
$name = [x, y, z, ...]
$name = [a : b, c]
```

The first form is used to set the variable `$name` to the values in the list, `x`, `y`, `z`, `...`. The second form is used the set the variable `$name` from the value `a` to `b` using a step size of `c`. The value of `b` must be greater that `a`, and the step size must be positive. Regardless of the format, the values for array variables must be constants.

During the execution of the solver, SPEX iterates over all of the values defined for each array variable. If multiple arrays are defined, then SPEX generates the cross-product of all possible parameter values. Note that if a scalar variable is defined in terms of an array variable, then the scalar variable will be recomputed for each model generated by SPEX.

Observation Variables

There is a set of special symbols that are used to indicate to SPEX which result values from the solution of the model are of interest. This result indication has the following form:

%<*key*><*phase*> $var

where <*key*> is a one or two letter key indicating the type of data to be observed and <*phase*> is an optional integer indicating the phase of the data to be observed. The data, once obtained from the results of the model, is placed into the variable $var where it may be used in the Result Information section described below.

To obtain confidence interval information, the format is

%<*key*><*phase*>[confidence] $var1 $var2

Key	Phase	Description	Location	
`%u`	no	Utilization	processor declaration	(p info)
	yes		task declaration	(t info)
	yes		entry service declaration	(s info)
`%f`	no	Throughput	task declaration	(t info)
	no		entry service declaration	(s info)
`%pu`	no	Processor Utilization	task declaration	(t info)
	no		entry service declaration	(s info)
`%s`	yes	Service Time	entry service declaration	(s info)
`%v`	yes	Service Time Variance	entry service declaration	(s info)
`%fb`	no	Throughput Bound	entry service declaration	(s info)
`%pw`	yes	Processor waiting time by task	entry service declaration	(s info)
`%w`	yes	Call waiting time	entry call declaration	(y info)
	no		entry open arrival declaration	(a info)
`%x`	yes	Max Service Time Exceeded	entry service declaration	(s info)

Table 5.9: Observation Key location

Listing 5.8: Report Information

```
R <int>
  plot( <var-list> )
  splot( <var-list> )
  $var = <ternary-expression>
  <expression>
-1
```

where confidence can be 95 or 99, $var1 is the mean and $var2 is the half-width of the confidence interval.

The location of a result indication determines the entity to be observed. Table 5.9 describes each of the keys and where they may be used.

For any key/location combination that takes a phase argument, if none is supplied then the sum of the values for all phases is reported. This also happens if a phase of zero is given.

5.3.2 Report Information

The purpose of the report information section of the input file is to specify which variable values (including result indications) are to be printed in the SPEX result file. The format of this section is shown in Listing 5.8 and consists of either a single plot function, or a list of variables (with possible computed results).

There may be any number of report declarations, however, the integer parameter to `R` must either be the number of report declarations present or zero[5].

The <*ternary-expression*> may be any valid ternaray expression as discussed above. The <*expression*> can be any expression, including a lone variable or a function call. Report indication variables and the parameter variables may both be used together, but may not be mixed with either `plot()` or `splot()`.

The values of the variables listed in this section are printed from left to right in the order that they appear in the input file separated by commas. This output can then be used as input to Gnuplot or a spreadsheet such as Excel. Output is normally sent to the terminal, but can be redirected using `-ooutput` *filename*.

If the `plot()` or `splot()` function is used, output will formatted in such a way that it can be used as input into gnuplot. The first argument to the `plot` function call is the x variable for the graph. The first two arguments to the `splot` function call are the x and y variables for the graph. Multiple additional arguments are permitted but should

[5] The number is ignored; it is present in the syntax so that the report section matches the other sections.

```
$array = 1 2 3
$array = 1:10,2
```

(a) Spex 1

```
$array = [1, 2, 3]
$array = [1:10,2]
```

(b) Spex 2

Figure 5.2: x

be grouped by result type (ie., throughputs, utilizations). When using `plot`, up to two groups can be plotted with results in first group being plotted against the left 'y' axis, and results from the second on the right 'y' axis. When using `splot`, only one group of results can be used. Only one plot function can be specified in the report section at this time. The actual output is self-contained gnuplot input.

There is a special variable called `$0` which represents the independent variable in the results tables (the x-axis in plots). The variable `$0` may be set to any expression allowing for flexibility in producing result tables. This variable cannot be used as a parameter in the model.

5.3.3 Convergence Information

Spex allows a parameter value to be modified at the end of a model solution and then fed back in to the model. The model is solved repeatedly until the parameter value converges. The convergence section is declared in a manner similar to the result section:

Listing 5.9: Convergence Information

```
C <int>
   $var = <ternary-expression>
-1
```

Convergence variables must be parameters. They cannot be result variables.

5.3.4 Differeneces to SPEX 1

This section outlines differences in the syntax between SPEX version 1 and version 2. SPEX version 1 was processed by a Perl program to convert the model into a conventional LQN model file. SPEX version 2 is now parsed directly and converted into LQX internally.

Array Initialization

Lists used for array initialization must now be enclosed within square brackets ('[]'). Further, the items must be separated using commas. Figure 5.2 shows the old and new syntax.

Perl Expressions

Perl Expressions are no longer supported in SPEX 2.0. Rather, a subset of LQX expressions are used instead. For SPEX convergence expressions, Perl `if then else` statements must be converted to use the ternary `?:` operator. SPEX 2 cannot invoke Perl functions.

Line Continuation

Line continuation, where a line is terminated by a backslash ('\'), is not needed with Version 2 SPEX. All whitespace, including newlines, is ignored.

Control Variable
$coefficient_of_variation
$hosts

Table 5.10: Obsolete SPEX Control Variables.

Listing 5.10: SPEX random parameter generation

```
$experiments = [1:10,1]             # 10 experiments.
$experiments, $parameter1 = rand()  # $experiments is ignored.
$experiments, $parameter2 = rand()  # $experiments is ignored.
```

Comments

In Version 1 of SPEX, all text before a dollar sign (‘$’), or either an upper case ‘G’ or ‘P’ at the start of a line, was treated as a comment. Since Version 2 SPEX is parsed directly, all comments must start with the hash symbol (‘#’).

String Substitution

Version 2 SPEX does not support variable substitution of string parameters such as pragmas, and scheduling types. This restriction may be lifted in future versions.

Pragmas

Version 1 SPEX did not require the hash symbol (‘#’) for setting pragmas. Version 2 SPEX does.

SPEX AGR

SPEX AGR is no longer supported.

Control Variables

Version 2 SPEX does not support the control variables shown in Table 5.10.

Random Numbers

Version 2 SPEX introduces the function `rand()` to generate random numbers in the range of $[0..1)$. To generate a set of experiments with random parameters, an array (or set of arrays) with the number of elements corresponding to the number of experiments is required to cause SPEX to iterate (see Section 5.3.5). Listing 5.10 shows the syntax to generate random values for ten experiments.

5.3.5 SPEX and LQX

SPEX uses LQX to generate individual model files. All scalar paramaters are treated as globally scoped variables in LQX and can be used to set parameters in the model. If the assignement expression for a scalar variable does not reference any array variables, it is set prior to the iteration of any loop. Otherwise, the scalar variable is set during the execution of the innermost loop of the program.

Array variables are used to generate `foreach` loops in the LQX program. The variable defining the array is local (i.e. without the ‘$’) with the name of the SPEX parameter. Each array variable generates a for loop; the loops are nested in the order of the definition of the array variable. The value variable for the `foreach` loop is global (i.e. with the ‘$’) with the name of the SPEX parameter and can be used as a parameter in the model.

If SPEX convergence is used, a final innermost loop is created. This loop tests the variables defined in the convergence section for change, and if any of the variables changes by a non-trivial amount, the loop repeats.

Listing 5.11: SPEX file layout.

```
$m_client = [1, 2, 3]
$m_server = [1: 3, 1]
$s_server = $m_server / 2

P 2
  p client i
  p server s 0.1
-1

T 2
  t client r client -1 client m $m_client %f $f_client
  t server n server -1 server m $m_server %u $u_server
-1

E 2
  s client 1 -1
  y client server $s_server -1
  s server 1 -1
-1

R 3
  $0
  $f_client
  $u_server
-1
```

Listing 5.11 shows a model defined defined using SPEX syntax. Listing 5.12 shows the corresponding LQX program generated by the model file.

Listing 5.12: LQX Program for SPEX input.

```
m_client = array_create(1, 2, 3);
_0 = 0;
_f_client = 0;
_u_server = 0;
println_spaced(", ", "$0", "$f_client", "$u_server");
foreach( $m_client in m_client ) {
  for ( $m_server = 1; $m_server <= 3; $m_server = ($m_server + 1)) {
    $s_server = ($m_server / 2);
    _0 = (_0 + 1);
    if (solve()) {
      _f_client = task("client").throughput;
      _u_server = task("server").utilization;
      println_spaced(", ", _0, _f_client, _u_server);
    } else {
      println("solver failed: $0=", _0);
    }
  }
}
```

Chapter 6

Invoking the Analytic Solver “lqns”

The Layered Queueing Network Solver (LQNS) is used to solved Layered Queueing Network models analytically. **Lqns** reads its input from `filename`, specified at the command line if present, or from the standard input otherwise. By default, output for an input file `filename` specified on the command line will be placed in the file `filename.out`. If the `-p` switch is used, parseable output will also be written into `filename.p`. If XML input or the `-x` switch is used, XML output will be written to `filename.lqxo`. This behaviour can be changed using the `-ooutput` switch, described below. If several files are named, then each is treated as a separate model and output will be placed in separate output files. If input is from the standard input, output will be directed to the standard output. The file name ‘`-`’ is used to specify standard input.

The `-ooutput` option can be used to direct output to the file `output` regardless of the source of input. Output will be XML if XML input or if the `-x` switch is used, parseable output if the `-p` switch is used, and normal output otherwise. Multiple input files cannot be specified when using this option. Output can be directed to standard output by using `-o-` (i.e., the output file name is ‘`-`’.)

6.1 Command Line Options

`-a`, `-no-advisories`
Ignore advisories. The default is to print out all advisories.

`-b`, `-bounds-only`
This option is used to compute the “Type 1 throughput bounds” only. These bounds are computed assuming no contention anywhere in the model and represent the guaranteed not to exceed values.

`-c`, `-convergence-value=`*arg*
Set the convergence value to *arg*. *Arg* must be a number between 0.0 and 1.0.

`-d`, `-debug=`*arg*
This option is used to enable debug output. *Arg* can be one of:

all Enable all debug output.

forks Print out the fork-join matching process.

interlock Print out the interlocking table and the interlocking between all tasks and processors.

lqx Print out the actions the LQX parser while reading an LQX program.

overtaking Print the overtaking probabilities in the output file.

submodels Print out the contents of all of the submodels found in the model.

variance Print out variance calculation.

xml Print out the actions of the Expat parser while reading XML input.

-e, -error=*arg*
This option is to enable floating point exception handling. *Arg* must be one of the following:

1. **a** Abort immediately on a floating point error (provided the floating point unit can do so).
2. **d** Abort on floating point errors. (default)
3. **i** Ignore floating point errors.
4. **w** Warn on floating point errors.

The solver checks for floating point overflow, division by zero and invalid operations. Underflow and inexact result exceptions are always ignored.

In some instances, infinities will be propogated within the solver. Please refer to the **stop-on-message-loss** pragma below.

-f, -fast-linearizer
This option is used to set options for quick solution of a model using One-Step (Bard-Schweitzer) MVA. It is equivalent to setting **pragma** *mva=one-step*, *layering=batched*, *multiserver=conway*

-h, -huge
Solve using one-step-schweitzer, no interlocking, and Rolia multiserver.

-H, -help=*arg*
Print a summary of the command line options. The optional argument shows help for -d, -t, -z, and -P respectively.

-i, -iteration-limit=*arg*
Set the maximum number of iterations to *arg*. *Arg* must be an integer greater than 0. The default value is 50.

-I, -input-format=*arg*
This option is used to force the input file format to either *xml* or *lqn*. By default, if the suffix of the input filename is one of: *.in*, *.lqn* or *.xlqn* , then the LQN parser will be used. Otherwise, input is assumed to be XML.

-j, -json
Generate JSON output regardless of input format.

-M, -mol-underrelaxation=*arg*
Set the under-relaxation factor to *arg* for the MOL (Rolia) multiserver approximation. If the approximation is failing, lower this value. *Arg* must be a number between 0.0 and 1.0. The default value is 0.5.

-n, -no-execute
Build the model, but do not solve.The input is checked for validity but no output is generated.

-o, -output=*arg*
Direct analysis results to *output*. A filename of '-' directs output to standard output. If `output` is a directory, all output is saved in `output/input.out`. If the input model contains a SPEX program with loops, the SPEX output is sent to `output`; the individual model output files are found in the directory `output.d`. If **lqns** is invoked with this option, only one input file can be specified.

-p, -parseable
Generate parseable output suitable as input to other programs such as **lqn2ps(1)** and **srvndiff(1)**. If input is from `filename`, parseable output is directed to `filename.p`. If standard input is used for input, then the parseable output is sent to the standard output device. If the `-ooutput` option is used, the parseable output is sent to the file name `output`. (In this case, only parseable output is emitted.)

-P, -pragma=*arg*
Change the default solution strategy. Refer to the PRAGMAS section below for more information.

-r, -rtf
Output results using Rich Text Format instead of plain text. Processors, entries and tasks with high utilizations are coloured in red.

-t, -trace=*arg*
This option is used to set tracing options which are used to print out various intermediate results while a model is being solved. *arg* can be any combination of the following:

activities Print out results of activity aggregation.

cfs Print out CFS computation before each submodel is solved.

convergence=arg Print out convergence value after each submodel is solved. This option is useful for tracking the rate of convergence for a model. The optional numeric argument supplied to this option will print out the convergence value for the specified MVA submodel, otherwise, the convergence value for all submodels will be printed.

delta-wait Print out difference in entry service time after each submodel is solved.

forks Print out overlap table for forks prior to submodel solution.

idle-time Print out computed idle time after each submodel is solved.

interlock Print out interlocking adjustment before each submodel is solved.

intermediate Print out intermediate solutions at the print interval specified in the model. The print interval field in the input is ignored otherwise.

mva=arg Output the inputs and results of each MVA submodel for every iteration of the solver. The optional argument is a bit set of the submodels to output. Submodel 1 is 0x1, submodel 2 is 0x2, submodel 3 is 0x4, etc. By default all submodels are traced.

overtaking Print out overtaking calculations.

quorum Print quorum traces.

replication

throughput Print throughput's values.

variance Print out the variances calculated after each submodel is solved.

virtual-entry Print waiting time for each rendezvous in the model after it has been computed; include virtual entries.

wait Print waiting time for each rendezvous in the model after it has been computed.

-u, -underrelaxation=*arg*
Set the underrelaxation to *arg*. *Arg* must be a number between 0.0 and 1.0. The default value is 0.9.

-v, -verbose
Generate output after each iteration of the MVA solver and the convergence value at the end of each outer iteration of the solver.

-V, -version
Print out version and copyright information.

-w, -no-warnings
Ignore warnings. The default is to print out all warnings.

-x, -xml
Generate XML output regardless of input format.

-z, -special=*arg*
This option is used to select special options. Arguments of the form *nn* are integers while arguments of the form *nn.n* are real numbers. *Arg* can be any of the following:

full-reinitialize For multiple runs, reinitialize all service times at processors.

generate=arg This option is used to generate a queueing model for solver in the directory *arg*. A directory named *arg* will be created containing source code for invoking the MVA solver directly.

man=arg Output this manual page. If an optional *arg* is supplied, output will be written to the file named *arg*. Otherwise, output is sent to stdout.

min-steps=arg Force the solver to iterate min-steps times.

overtaking Print out overtaking probabilities.

print-interval=arg Set the printing interval to *arg*. The `-d` or `-v` options must also be selected to display intermediate results. The default value is 10.

single-step Stop after each MVA submodel is solved. Any character typed at the terminal except end-of-file will resume the calculation. End-of-file will cancel single-stepping altogether.

tex=arg Output this manual page in LaTeX format. If an optional *arg* is supplied, output will be written to the file named *arg*. Otherwise, output is sent to stdout.

If any one of *convergence*, *iteration-limit*, or*print-interval* are used as arguments, the corresponding value specified in the input file for general information, 'G', is ignored.

`-exact-mva`
Use exact MVA instead of Linearizer for solving submodels.

`-schweitzer`
Use Bard-Schweitzer approximate MVA to solve all submodels.

`-batch-layering`
Default layering strategy where a submodel consists of the servers at a layer and each server's clients.

`-hwsw-layering`
Use HW/SW layering instead of batched layering.

`-method-of-layers`
This option is to use the Method Of Layers solution approach to solving the layer submodels.

`-squashed-layering`
Use only one submodel to solve the model.

`-srvn-layering`
Use one server per layer instead of batched layering.

`-processor-sharing`
Use Processor Sharing scheduling at all fixed-rate processors.

`-no-stop-on-message-loss`
Do not stop the solver on overflow (infinities) for open arrivals or send-no-reply messages to entries. The default is to stop with an error message indicating that the arrival rate is too high for the service time of the entry

`-no-variance`
Do not use variances in the waiting time calculations. The variance of an entry is used with fixed-rate servers. Ignorning variance will help with convergence problems with some models. .

`-reload-lqx`
Re-run the LQX/SPEX program without re-solving the models. Results must exist from a previous solution run. This option is useful if LQX print statements or SPEX results are changed.

-restart

Re-run the LQX/SPEX program without re-solving models which were solved successfully. Models which were not solved because of early termination, or which were not solved successfully because of convergence problems, will be solved. This option is useful for running a second pass with a new convergnece value and/or iteration limit.

-no-header

Do not output the variable name header on SPEX results.This option can be also be set by using **pragma** *spex-header=no*.This option has no effect if SPEX is not used.

-print-comment

Add the model comment as the first line of output when running with SPEX input.

-print-interval=*arg*

Output the intermediate solution of the model after <n> iterations.

-reset-mva

Reset the MVA calculation prior to solving a submodel.

-trace-mva=*arg*

Output the inputs and results of each MVA submodel for every iteration of the solver. The optional argument is a bit set of the submodels to output. Submodel 1 is 0x1, submodel 2 is 0x2, submodel 3 is 0x4, etc. By default all submodels are traced.

-debug-submodels

Print out submodels. <n> is a 64 bit number where the bit position is the submodel output.The output for each submodel consists of the number of customers for closed classes, closed class clients, closed class servers, open class servers, and the calls from clients to servers in the submodel. Calls are shown from entries to entries, or from tasks to processors. Synchronous calls are shown using ->, asynchronous calls are shownn using ˜>, and processor calls are shown using *>.

-debug-json

Output JSON elements and attributes as they are being parsed. Since the JSON parser usually stops when it encounters an error, this option can be used to localize the error.

-debug-lqx

Output debugging information as an LQX program is being parsed.

-debug-spex

Output LQX progam corresponding to SPEX input.

-debug-srvn

Output debugging information while parsing SRVN input.This is the output of the Bison LALR parser.

-debug-xml

Output XML elements and attributes as they are being parsed. Since the XML parser usually stops when it encounters an error, this option can be used to localize the error.

Lqns exits with 0 on success, 1 if the model failed to converge, 2 if the input was invalid, 4 if a command line argument was incorrect, 8 for file read/write problems and -1 for fatal errors. If multiple input files are being processed, the exit code is the bit-wise OR of the above conditions.

6.2 Pragmas

Pragmas are used to alter the behaviour of the solver in a variety of ways. They can be specified in the input file with "#pragma", on the command line with the `-P` option, or through the environment variable *LQNS_PRAGMAS*. Command line specification of pragmas overrides those defined in the environment variable which in turn override those defined in the input file. The following pragmas are supported. Invalid pragma specification at the command line will stop the solver. Invalid pragmas defined in the environment variable or in the input file are ignored as they might be used by other solvers.

convergence-value=arg
Set the convergence value to *arg*. *Arg* must be a number between 0.0 and 1.0.

cycles=arg
This pragma is used to enable or disable cycle detection in the call graph. Cycles may indicate the presence of deadlocks. *Arg* must be one of:

no Disallow cycles in the call graph.

yes Allow cycles in the call graph. The interlock adjustment is disabled.

The default is no.

force-infinite=arg
This pragma is used to force the use of an infinite server instead of a fixed-rate server and/or multiserver for all the tasks in the model. *Arg* must be one of:

all Change all tasks to infinite servers.

fixed-rate Change all fixed-rate tasks to infinite servers.

multiservers Change all multiserver tasks to infinite servers.

none Do not change and fixed-rate or multiserver task to an infinite server.

The default is none.

force-multiserver=arg
This pragma is used to force the use of a multiserver instead of a fixed-rate server whenever the multiplicity of a server is one. *Arg* must be one of:

all Always use a multiserver solution for non-delay servers (tasks and processors) even if the number of servers is one (1). The Rolia multiserver approximation is known to fail for this case.

none Use fixed-rate servers whenever a server multiplicity is one (1). Note that fixed-rateservers with variance may have results that differ from fixed-rate servers that don't and that the multiserver servers never take variance into consideration.

processors Always use a multiserver solution for non-delay processors even if the number of servers is one (1). The Rolia multiserver approximation is known to fail for this case.

tasks Always use a multiserver solution for non-delay server tasks even if the number of servers is one (1). The Rolia multiserver approximation is known to fail for this case.

The default is none.

interlocking=arg
The interlocking is used to correct the throughputs at stations as a result of solving the model using layers [5]. This pragma is used to choose the algorithm used. *Arg* must be one of:

no Do not perform interlock adjustment.

yes Perform interlocking by adjusting throughputs.

The default is yes.

iteration-limit=arg
Set the maximum number of iterations to *arg*. *Arg* must be an integer greater than 0. The default value is 50.

layering=arg
This pragma is used to select the layering strategy used by the solver. *Arg* must be one of:

batched Batched layering – solve layers composed of as many servers as possible from top to bottom.

batched-back Batched layering with back propagation – solve layers composed of as many servers as possible from top to bottom, then from bottom to top to improve solution speed.

hwsw Hardware/software layers – The model is solved using two submodels: One consisting solely of the tasks in the model, and the other with the tasks calling the processors.

mol Method Of layers – solve layers using the Method of Layers [14]. Layer spanning is performed by allowing clients to appear in more than one layer.

mol-back Method Of layers – solve layers using the Method of Layers. Software submodels are solved top-down then bottom up to improve solution speed.

squashed Squashed layers – All the tasks and processors are placed into one submodel. Solution speed may suffer because this method generates the most number of chains in the MVA solution. See also `-P`*mva*.

srvn SRVN layers – solve layers composed of only one server. This method of solution is comparable to the technique used by the **srvn** solver. See also `-P`*mva*.

The default is batched.

mol-underrelaxation=arg
Set the under-relaxation factor to *arg* for the MOL (Rolia) multiserver approximation. If the approximation is failing, lower this value. *Arg* must be a number between 0.0 and 1.0. The default value is 0.5.

multiserver=arg
This pragma is used to choose the algorithm for solving multiservers. *Arg* must be one of:

bruell Use the Bruell multiserver [2] calculation for all multiservers.

conway Use the Conway multiserver [4, 3] calculation for all multiservers.

reiser Use the Reiser multiserver [13] calculation for all multiservers.

reiser-ps Use the Reiser multiserver calculation for all multiservers. For multiservers with multiple entries, scheduling is processor sharing, not FIFO.

rolia Use the Rolia [15, 14] multiserver calculation for all multiservers.

rolia-ps Use the Rolia multiserver calculation for all multiservers. For multiservers with multiple entries, scheduling is processor sharing, not FIFO.

schmidt Use the Schmidt multiserver [16] calculation for all multiservers.

suri experimental.

The default multiserver calculation uses the the Conway multiserver for multiservers with less than five servers, and the Rolia multiserver otherwise.

mva=arg
This pragma is used to choose the MVA algorithm used to solve the submodels. *Arg* must be one of:

exact-mva Exact MVA. Not suitable for large systems.

fast-linearizer Fast Linearizer

linearizer Linearizer.

one-step Perform one step of Bard Schweitzer approximate MVA for each iteration of a submodel. The default is to perform Bard Schweitzer approximate MVA until convergence for each submodel. This option, combined with `-P`*layering=srvn* most closely approximates the solution technique used by the **srvn** solver.

one-step-linearizer Perform one step of Linearizer approximate MVA for each iteration of a submodel. The default is to perform Linearizer approximate MVA until convergence for each submodel.

schweitzer Bard-Schweitzer approximate MVA.

The default is linearizer.

overtaking=arg
This pragma is usesd to choose the overtaking approximation. *Arg* must be one of:

markov Markov phase 2 calculation.

none Disable all second phase servers. All stations are modeled as having a single phase by summing the phase information.

rolia Use the method from the Method of Layers.

simple Simpler, but faster approximation.

special ?

The default is markov.

processor-scheduling=arg
Force the scheduling type of all uni-processors to the type specfied.

fcfs All uni-processors are scheduled first-come, first-served.

hol All uni-processors are scheduled using head-of-line priority.

ppr All uni-processors are scheduled using priority, pre-emptive resume.

ps All uni-processors are scheduled using processor sharing.

The default is to use the processor scheduling specified in the model.

save-marginal-probabilities=arg
This pragma is used to enable or disable saving the marginal queue probabilities for multiservers in the results.

severity-level=arg
This pragma is used to enable or disable warning messages.

advisory

all

run-time

warning

spex-comment=arg
This pragma is used to enable or disable the comment line of SPEX output. *Arg* must be one of:

false Do not output a comment line (the output can then be fed into gnuplot easily).

true Output the model comment in the SPEX output.

The default is false.

spex-header=arg
This pragma is used to enable or disable the header line of SPEX output. *Arg* must be one of:

false Do not output a header line (the output can then be fed into gnuplot easily).

true Output a header line consisting of the names of all of the variables used in the Result section on the input file.

The default is false.

stop-on-message-loss=arg
This pragma is used to control the operation of the solver when the arrival rate exceeds the service rate of a server. *Arg* must be one of:

no Stop if messages are lost.

yes Ignore queue overflows for open arrivals and send-no-reply requests. If a queue overflows, its waiting times is reported as infinite.

The default is no.

tau=arg
Set the tau adjustment factor to *arg*. *Arg* must be an integer between 0 and 25. A value of *zero* disables the adjustment.

threads=arg
This pragma is used to change the behaviour of the solver when solving models with fork-join interactions.

exponential Use exponential values instead of three-point approximations in all approximations [8].

hyper Inflate overlap probabilities based on arrival instant estimates.

mak Use Mak-Lundstrom [10] approximations for join delays.

none Do not perform overlap calculation for forks.

The default is hyper.

variance=arg
This pragma is used to choose the variance calculation used by the solver.

init-only Initialize the variances, but don't recompute as the model is solved.

mol Use the MOL variance calculation.

no-entry By default, any task with more than one entry will use the variance calculation. This pragma will switch off the variance calculation for tasks with only one entry.

none Disable variance adjustment. All stations in the MVA submodels are either delay- or FIFO-servers.

stochastic ?

6.3 Stopping Criteria

Lqns computes the model results by iterating through a set of submodels until either convergence is achieved, or the iteration limit is hit. Convergence is determined by taking the root of the mean of the squares of the difference in the utilization of all of the servers from the last two iterations of the MVA solver over the all of the submodels then comparing the result to the convergence value specified in the input file. If the RMS change in utilization is less than convergence value, then the results are considered valid.

If the model fails to converge, three options are available:

1. reduce the under-relaxation coefficient. Waiting and idle times are propogated between submodels during each iteration. The under-relaxation coefficient determines the amount a service time is changed between each iteration. A typical value is 0.7 - 0.9; reducing it to 0.1 may help.

2. increase the iteration limit. The iteration limit sets the upper bound on the number of times all of the submodels are solved. This value may have to be increased, especially if the under-relaxation coefficient is small, or if the model is deeply nested. The default value is 50 iterations.

3. increase the convergence test value. Note that the convergence value is the standard deviation in the change in the utilization of the servers, so a value greater than 1.0 makes no sense.

The convergence value can be observed using `-t`*convergence* flag.

6.4 Model Limits

The following table lists the acceptable parameter types for **lqns**. An error will be reported if an unsupported parameter is supplied except when the value supplied is the same as the default.

Parameter	lqns
Phases	3
Scheduling	FIFO, HOL, PPR
Open arrivals	yes
Phase type	stochastic, deterministic
Think Time	yes
Coefficient of variation	yes
Interprocessor-delay	yes
Asynchronous connections	yes
Forwarding	yes
Multi-servers	yes
Infinite-servers	yes
Max Entries	unlimited
Max Tasks	unlimited
Max Processors	unlimited
Max Entries per Task	unlimited

Table 6.1: LQNS Model Limits.

6.5 Diagnostics

Most diagnostic messages result from errors in the input file. If the solver reports errors, then no solution will be generated for the model being solved. Models which generate warnings may not be correct. However, the solver will generate output.

Sometimes the model fails to converge, particularly if there are several heavily utilized servers in a submodel. Sometimes, this problem can be solved by reducing the value of the under-relaxation coefficient. It may also be necessary to increase the iteration-limit, particularly if there are many submodels. With replicated models, it may be necessary to use 'srvn' layering to get the model to converge. Convergence can be tracked using the `-t`*convergence* option.

The solver will sometimes report some servers with 'high' utilization. This problem is the result of some of the approximations used, in particular, two-phase servers. Utilizations in excess of 10% are likely the result of failures in the solver. Please send us the model file so that we can improve the algorithms.

Chapter 7

Invoking the Simulator "lqsim"

Lqsim is used to simulate layered queueing networks using the PARASOL [12] simulation system. Lqsim reads its input from files specified at the command line if present, or from the standard input otherwise. By default, output for an input file `filename` specified on the command line will be placed in the file `filename.out`. If the `-p` switch is used, parseable output will also be written into `filename.p`. If XML input is used, results will be written back to the original input file. This behaviour can be changed using the `-o`*output* switch, described below. If several files are named, then each is treated as a separate model and output will be placed in separate output files. If input is from the standard input, output will be directed to the standard output. The file name '`-`' is used to specify standard input.

The `-o`*output* option can be used to direct output to the file or directory named *output* regardless of the source of input. Output will be XML if XML input is used, parseable output if the `-p` switch is used, and normal output otherwise; multiple input files cannot be specified. If *output* is a directory, results will be written in the directory named `output`. Output can be directed to standard output by using `-o-` (i.e., the output file name is '`-`'.)

7.1 Command Line Options

`-A`, `-automatic`=*run-time[,precision[,skip]]*
Use automatic blocking with a simulation block size of *run-time*. The *precision* argument specifies the desired mean 95% confidence level. By default, precision is 1.0%. The simulator will stop when this value is reached, or when 30 blocks have run. *Skip* specifies the time value of the initial skip period. Statistics gathered during the skip period are discarded. By default, its value is 0. When the run completes, the results reported will be the average value of the data collected in all of the blocks. If the `-R` flag is used, the confidence intervals will for the raw statistics will be included in the monitor file.

`-B`, `-blocks`=*blocks[,run-time[,skip]]*
Use manual blocking with *blocks* blocks. The value of *blocks* must be less than or equal to 30. The run time for each block is specified with *run-time*. *Skip* specifies the time value of the initial skip period.

`-C`, `-confidence`=*precision[,initial-loops[,run-time]]*
Use automatic blocking, stopping when the specified precision is met. The run time of each block is estimated, based on *initial-loops* running on each reference task. The default value for *initial-loops* is 500. The *run-time* argument specifies the maximum total run time.

`-d`, `-debug`
This option is used to dump task and entry information showing internal index numbers. This option is useful for determining the names of the servers and tasks when tracing the execution of the simulator since the Parasol output routines do no emit this information at present. Output is directed to stdout unless redirected using `-m`*file*.

`-e`, `-error`=*error*
This option is to enable floating point exception handling.

a Abort immediately on a floating point error (provided the floating point unit can do so).

b Abort on floating point errors. (default)

i Ignore floating point errors.

w Warn on floating point errors.

The solver checks for floating point overflow, division by zero and invalid operations. Underflow and inexact result exceptions are always ignored.

In some instances, infinities will be propogated within the solver. Please refer to the *stop-on-message-loss* pragma below.

-h*output*
Generate comma separated values for the service time distribution data. If *output* is a directory, the output file name will be the name of a the input file with a `.csv` extension. Otherwise, the output will be written to the named file.

-m*file*
Direct all output generated by the various debugging and tracing options to the monitor file *file*, rather than to standard output. A filename of '`-`' directs output to standard output.

-n, -no-execute
Read input, but do not solve. The input is checked for validity. No output is generated.

-o, -output=*output*
Direct analysis results to output. A file name of '`-`' directs output to standard output. If *output* is a directory, all output from the simulator will be placed there with filenames based on the name of the input files processed. Otherwise, only one input file can be processed; its output will be placed in *output*.

-p, -parseable
Generate parseable output suitable as input to other programs such as *MultiSRVN(1)* and *srvndiff(1)*. If input is from `filename`, parseable output is directed to `filename.p`. If standard input is used for input, then the parseable output is sent to the standard output device. If the `-o`*output* option is used, the parseable output is sent to the file name output. (In this case, only parseable output is emitted.)

-P, -pragma=*pragma*
Change the default solution strategy. Refer to the PRAGMAS chapter (§7.3) below for more information.

-R, -raw-statistics
Print the values of the statistical counters to the monitor file. If the `-A`, `-B` or `-C` option was used, the mean value, 95th and 99th percentile are reported. At present, statistics are gathered for the task and entry, cycle time task, processor and entry utilization, and waiting time for messages.

-S, -seed=*seed*
Set the initial seed value for the random number generator. By default, the system time from time *time(3)* is used. The same seed value is used to initialize the random number generator for each file when multiple input files are specified.

-t, -trace=*traceopts*
This option is used to set tracing options which are used to print out various steps of the simulation while it is executing. *Traceopts* is any combination of the following:

driver Print out the underlying tracing information from the Parasol simulation engine.

***processor*=regex** Trace activity for processors whose name match *regex*. If *regex*is not specified, activity on all processors is reported. *Regex* is regular expression of the type accepted by *egrep(1)*.

***task*=regex** Trace activity for tasks whose name match *regex*. If *regex* is not specified, activity on all tasks is reported. pattern is regular expression of the type accepted by *egrep(1)*.

***events*regex[:regex]** Display only events matching pattern. The events are: msg-async, msg-send, msg-receive, msg-reply, msg-done, msg-abort, msg-forward, worker-dispatch, worker- idle, task-created, task-ready, task-running, task-computing, task-waiting, thread-start, thread-enqueue, thread-dequeue, thread-idle, thread-create, thread-reap, thread-stop, activity-start, activity-execute, activity-fork, and activity-join.

msgbuf Show msgbuf allocation and deallocation.

timeline Generate events for the timeline tool.

-T, -run-time=*run-time*

Set the run time for the simulation. The default is 10,000 units. Specifying `-T` after either `-A` or `-B` changes the simulation block size, but does not turn off blocked statistics collection.

-v, -verbose

Print out statistics about the solution on the standard output device.

-V, -version

Print out version and copyright information.

-w, -no-warnings

Ignore warnings. The default is to print out all warnings.

-x, -xml

Generate XML output regardless of input format.

-z*specialopts*

This flag is used to select special options. Arguments of the form n are integers while arguments of the form $n.n$ are real numbers. *Specialopts* is any combination of the following:

***print-interval*=nn** Set the printing interval to n. Results are printed after nn blocks have run. The default value is 10.

***global-delay*=n.n** Set the interprocessor delay to nn.n for all tasks. Delays specified in the input file will override the global value.

-global-delay

Set the inter-processor communication delay to n.n.

-print-interval

Ouptut results after n iterations.

-restart

Re-run the LQX program without re-solving the models unless a valid solution does not exist. This option is useful if LQX print statements are changed, or if a subset of simulations has to be re-run.

-debug-lqx

Output debugging informtion as an LQX program is being parsed.

-debug-xml

Output XML elements and attributes as they are being parsed. Since the XML parser usually stops when it encounters an error, this option can be used to localize the error.

7.2 Return Status

Lqsim exits 0 on success, 1 if the simulation failed to meet the convergence criteria, 2 if the input was invalid, 4 if a command line argument was incorrect, 8 for file read/write problems and -1 for fatal errors. If multiple input files are being processed, the exit code is the bit-wise OR of the above conditions.

7.3 Pragmas

Pragmas are used to alter the behaviour of the simulator in a variety of ways. They can be specified in the input file with "#pragma", on the command line with the `-P` option, or through the environment variable `LQSIM_PRAGMAS`. Command line specification of pragmas overrides those defined in the environment variable which in turn override those defined in the input file.

The following pragmas are supported. An invalid pragma specification at the command line will stop the solver. Invalid pragmas defined in the environment variable or in the input file are ignored as they might be used by other solvers.

***block-period*=real**

Set the block period to *real*. This value is used in conjuction with *max-blocks* or *precision*.

***initial-delay*=real**

Set the initial warmup period to *real*.

***initial-loops*=real**

Run reference tasks *int* times before recording data.

***max-blocks*=int**

Set the maximum number of blocks to *int*. *Int* must be no more than 30.

***precision*=real**

Set the precision of the simulation results, based on the confidence intervals of the utilizations of all of the tasks and processors, to *real*.

***run-time*=real**

Set the run-time of the simulations to *real*. If used by itself, the simulation will use one block and not report confidence intervals.

***seed-value*=int**

Set the seed for the random number generator to *int*

***nice*=int**

Set the "nice" value (i.e, lower the priority) when runninng the simulation.

***severity-level*=enum**

Suppress messages with a severity-level lower than *enum*. *Enum* is any one of the following:

all Show all messages.

warning-only Suppress warnings.

advisory Suppress warnings and advisorys.

runtime-error Suppress runtime errors, warnings and advisorys.

The default is to report all messages.

***scheduling-model*=enum**

This pragma is used to select the scheduler used for processors. *Enum* is any one of the following:

default Use the scheduler built into parasol for processor scheduling. (faster)

custom Use the custom scheduler for scheduling which permits more statistics to be gathered about processor utilization and waiting times. However, this option invokes more internal tasks, so simulations are slower than when using the default scheduler.

default-natural Use the parasol scheduler, but don't reschedule after the end of each phase or activity. This action more closely resembles the scheduling of real applications.

custom-natural Use the custom scheduler; don't reschedule after the end of each phase or activity.

reschedule-on-async-send=bool
In models with send-no-reply messages, the simulator does not reschedule the processor after an asynchronous message is sent (unlike the case with synchronous messages). The meanings of *bool* are:

true reschedule after each asynchronous message.

false reschedule after each asynchronous message.

stop-on-message-loss=bool
This pragma is used to control the operation of the solver when the arrival rate exceeds the service rate of a server. The simulator can either discard the arrival, or it can halt. The meanings of *bool* are:

false Ignore queue overflows for open arrivals and send-no-reply requests. If a queue overflows, its waiting times is reported as infinite.

true Stop if messages are lost.

7.4 Stopping Criteria

It is important that the length of the simulation be chosen properly. Results may be inaccurate if the simulation run is too short. Simulations that run too long waste time and resources.

Lqsim uses *batch means* (or the method of samples) to generate confidence intervals. With automatic blocking, the confidence intervals are computed after the simulations runs for three blocks plus the initial skip period If the root or the mean of the squares of the confidence intervals for the entry service times is within the specified precision, the simulation stops. Otherwise, the simulation runs for another block and repeats the test. With manual blocking, lqsim runs the number of blocks specified then stops. In either case, the simulator will stop after 30 blocks.

Confidence intervals can be tightened by either running additional blocks or by increasing the block size. A rule of thumb is the block size should be 10,000 times larger than the largest service time demand in the input model.

7.5 Model Limits

The following table lists the acceptable parameter types and limits for lqsim. An error will be reported if an unsupported parameter is supplied except when the value supplied is the same as the default.

Parameter	lqsim
Phases	3
Scheduling	FIFO, HOL, PRI, RAND
Open arrivals	yes
Phase type	stochastic, deterministic
Think Time	yes
Coefficient of variation	yes
Interprocessor-delay	yes
Asynchronous connections	yes
Forwarding	yes
Multi-servers	yes
Infinite-servers	yes
Max Entries	unlimited
Max Tasks	1000
Max Processors	1000
Max Entries per Task	unlimited

Table 7.1: Lqsim Model Limits

Chapter 8

Error Messages

Error messages are classified into four categories ranging from the most severe to the least, they are: fatal, error, advisory and warning. Fatal errors will cause the program to exit immediately. All other error messages will stop the solution of the current model and suppress output generation. However, subsequent input files will be processed. Advisory messages occur when the model has been solved, but the results may not be correct. Finally, warnings indicate possible problems with the model which the solver has ignored.

8.1 Fatal Error Messages

- `Internal error`

  Something bad happened...

- `No more memory`

  A request for memory failed.

- `Model has no` *(activity|entry|task|processor)*

  This should not happen.

- `Activity stack for "`*identifier*`" is full.`

  The stack size limit for task *identifier* has been exceeded.

- `Message pool is empty.  Sending from "`*identifier*`" to "`*identifier*`".`

  Message buffers are used when sending asynchronous send-no-reply messages. All the buffers have been used.

8.2 Error Messages

- *(task|processor)* `"`*identifier*`":  Replication not supported.` lqsim

  The simulator does not support replication. The model can be “flattened” using *rep2flat(1)*.

- *n.n* `Replies generated by Entry "`*identifier*`".`

  This error occurs when an entry is supposed to generate a reply because it accepts rendezvous requests, but the activity graph does not generate exactly one reply. Common causes of this error are replies being generated by two or more branches of an AND-fork, or replies being generated as part of a LOOP[1].

[1] Replies cannot be generated by branches of loops because the number of iterations of the loop is random, not deterministic

- `Activity "`*identifier*`" is a start activity.`

  The activity named *identifier* is the first activity in an activity graph. It cannot be used in a *post*-precedence (*fork*-list).

- `Activity "`*identifier*`" previously used in a fork."`

  The activity *identifier* has already been used as part of a fork expression. Fork lists are on the right hand side of the -> operator in the old grammar, and are the *post*-precedence expressions in the XML grammar. This error will cause a loop in the activity graph.

- `Activity "`*identifier*`" previously used in a join."`

  The activity *identifier* has already been used as part of a join list. Join lists are on the left hand side of the -> operator in the old grammar, and are the *pre*-precedence expressions in the XML grammar. This error will cause a loop in the activity graph.

- `Activity "`*identifier*`" requests reply for entry "`*identifier*`" but none pending.` lqsim

  The simulator is trying to generate a reply from entry *identifier*, but there are no messages queued at the entry. This error usually means there is a logic error in the simulator.

- `An error occured while compiling the LQX program found in file:` *filename*`'.` lqx

  A syntax error was found in the LQX program found in the file *filename*. Refer to earlier error messages.

- `An error occured executing the LQX program found in file:` *filename*`.` lqx

  A error occured while executing the the LQX program found in the file *filename*. Refer to earlier error messages.

- `Attribute "`*attribute*`" is missing from "`*type*`" element.`

  The attribute named *attribute* for the `type`-element is missing.

- `Attribute '`*attribute*`' is not declared for element '`*element*`'`

  The attribute named *attribute* for *element* is not defined in the schema..

- `"Both LQX and SPEX found in file` *filename* `. Use one or the other."`

  XML input allows for the use of LQX or SPEX, but not both at the same time.

- `Cannot create` *(processor|processor for task|task)* `"`*identifier*`".` lqsim

  Parasol could not create an object such as a task or processor.

- `Cycle in activity graph for task "`*identifier*`", back trace is "`*list*`".`

  There is a cycle in the activity graph for the task named *identifier*. Activity graphs must be acyclic. *List* identifies the activities found in the cycle.

- `Cycle in call graph, backtrace is "`*list*`".`

  There is a cycle in the call graph indicating either a possible deadlock or livelock condition. A deadlock can occur if the same task, but via a different entry, is called in the cycle of rendezvous indentified by *list*. A livelock can occur if the same task and entry are found in the cycle.

  In general, call graphs must be acyclic. If a deadlock condition is identified, the *cycles=allow* pragma can be used to suppress the error. Livelock conditions cannot be suppressed as these indicate an infinite loop in the call graph.

- `Data for` *n* `phases specified. Maximum permitted is` *m*`.`

  The solver only supports *m* phases (typically 3); data for *n* phases was specified. If more than *m* phases need to be specified, use activities to define the phases.

- `Datatype error: Type:InvalidDatatypeValueException, Message:`*message*
- `Delay from processor "`*identifier*`" to processor "`*identifier*`" previously specified.` lqsim

  Inter-processor delay...
- `Derived population of `*n.n*` for task "`*identifier*`" is not valid."` lqns

  The solver finds populations for the clients in a given submodel by traversing up the call graphs from all the servers in the submodel. If the derived population is infinite, the submodel cannot be solved. This error usually arises when open arrivals are accepted by infinite servers.
- `Destination entry "`*dst-identifier*`" must be different from source entry "`*src-identifier*`".`

  This error occurs when *src-identifier* and *dst-identifier* specify the same entry.
- `Deterministic phase "`*src-identifier*`" makes a non-integral number of calls (`*n.n*`) to entry `*dst-identifier*`.`

  This error occurs when a deterministic phase or activity makes a non-integral number of calls to some other entry.
- `Duplicate unique value declared for identity constraint of element '`*task*`'.`

  One or more activities are being bound to the same entry. This is not allowed, as an entry is only allowed to be bound to one activity. Check all `bound-to-entry` attributes for all activities to ensure this constraint is being met.
- `Duplicate unique value declared for identity constraint of element '`*lqn-model*`'.`

  This error indicated that an element has a duplicate name – the parser gives the line number to the start of the second instance of duplicate element. The following elements must have unique name attributes, but the uniqueness does not span elements. Therefore a processor and task element can have the same name attribute, but two processor elements cannot have the same name attribute.

  The following elements must have a unique `name` attribute:

  – processor

  – task

  – entry
- `Empty content not valid for content model:'`*element*`'`

  (result-processor,task)
- `Entry "`*identifier*`" accepts both rendezvous and send-no-reply messages.`

  An entry can either accept synchronous messages (to which it generates replies), or asynchronous messages (to which no reply is needed), but not both. Send the requests to two separate entries.
- `Entry "`*identifier*`" has invalid forwarding probability of `*n.n*`.`

  This error occurs when the sum of all forwarding probabilities from the entry *identifier* is greater than 1.
- `Entry "`*entry-identifier*`" is not part of task "`*task-identifier*`".`

  An activity graph part of task *task-identifer* replies to *entry-identifier*. However, *entry-identifier* belongs to another task.
- `Entry "`*identifier*`" is not specified.`

  An entry is declared but not defined, either using phases or activities. An entry is “defined” when some parameter such as service time is specified.

- `Entry "`*identifier*`" must reply; the reply is not specified in the activity graph.`

  The entry *identifier* accepts rendezvous requests. However, no reply is specified in the activity graph.

- `Entry "`*identifier*`" specified using both activity and phase methods.`

  Entries can be specified either using phases, or using activities, but not both..

- `Entry "`*identifier*`" specified as both a signal and wait.`

  A semaphore task must have exactly one signal and one wait entry. Both entries have the same type..

- `Expected end of tag '`*element*`'`

  The closing tag for *element* was not found in the input file.

- `External synchronization not supported for task "`*identifier*`" at join "`*join-list*`".` lqns

  The analytic solver does not implement external synchronization.

- `External variables are present in file "`*filename*`, but there is no LQX program to resolve them.` lqx

  The input model contains a variable of the form “`$var`” as a parameter such as a service time, multiplicty, or rate. The variables are only assigned values when an LQX program executes. Since no LQX program was present in the model file, the model cannot be solved.

- `Fan-ins from task "`*from-identifier*`" to task "`*to-identifier*`" are not identical for all calls.` lqns

  All requests made from task *from-identifier* to task *to-identifier* must have the same fan-in and fan-out values.

- `Fan-out from `*(activity|entry|task)*` "`*src-identifier*`" (`*n*` * `*n*` replicas) does not match fan-in to `*(entry|processor)*` "`*dst-identifier*`" (`*n*` * `*n*`).` lqns

  This error occurs when the number of replicas at *src-identifier* multiplied by the fan-out for the request to *dst-identifier* does not match the number of replicas at *dst-identifier* multiplied by the fan-in for the request from *src-identifier*. A fan-in or fan-out of zero (a common error case) can arise when the ratios of tasks to processors is non-integral.

- `Fewer entries defined (`*n*`) than tasks (`*m*`).`

  A model was specified with more tasks than entries. Since each task must have at least one entry, this model is invalid.

- `Group "`*identifier*`" has no tasks.`

  The group named by *identifier* has no tasks assigned to it. A group requires a minimum of one task.

- `Group "`*identifier*`" has invalid share of `*n.n*`.`

  The share value of *n.n* for group *identifier* is not between the range of $0 < n.n <= 1.0$.

- `Infinite throughput for task "`*identifier*`". Model specification error.` lqns

  The response time for the task *identifier* is zero. The likely cause is zero service time for all calls made by the task.

- `Initial delay of `*n.n*` is too small, `*n*` client(s) still running.` lqsim

  This error occurs when the *initial-loops* parameter for automatic blocking is too small.

- `Invalid fan-in of `*n*`: source task "`*identifier*`" is not replicated.` lqns

  The fan-in value for a request specifies the number of replicated source tasks making a call to the destination. To correct this error, the source task needs to be replicated by a multiple of n.

- `Invalid fan-out of` *n*`: destination task "`*identifier*`" has only` *m* `replicas.` lqns

  The fan-out value n is larger than the number of destination tasks m. In effect, the source will have more than one request arc to the destination.

- `Invalid path to join "`*join-list*`" for task "`*identifier*`": backtrace is "`*list*`".`

  The activity graph for task *identifer* is invalid because the branches to the join *join-list* do not all originate from the same fork. *List* is a dump of the activity stack when the error occurred.

- `Invalid probability of` *n.n*`.`

  The probability of *n.n* is not between the range of zero to one inclusive. The likely cause for this error is the sum of the probabilities either from an OR-fork, or from forwarding from an entry, is greater than one.

- `Name "`*identifier*`" previously defined.`

  The symbol *identifer* was previously defined. Tasks, processors and entries must all be named uniquely. Activities must be named uniquely within a task.

- `Model has no reference tasks.`

  There are no reference tasks nor are there any tasks with open arrivals specified in the model. Reference tasks serve as customers for closed queueing models. Open-arrivals serve as sources for open queueing models.

- `No calls from` *(entry|activity)* `"`*from-identifier*`" to entry "`*to-identifier*`".` lqns

  This error occurs when the fan-in or fan-out parameter for a request are specified *before* the actual request type. Switch the order in the input file.

- `No group specified for task "`*task_identifier*`" running on processor "`*proc_identifier*`" using fair share scheduling.`

  Task *task_identifier* has no group specified, yet it is running on processor *proc_identifier* which is using completely fair scheduling.

- `No signal or wait specified for semaphore task "`*identifier*`".`

  Task *identifier* has been identified as a semaphore task, but neither of its entries has been designated as a signal or a wait.

- `Non-reference task "`*identifier*`" cannot have think time.`

  A think time is specified for a non-reference task. Think times for non-reference tasks can only be specified by entry.

- `Non-semaphore task "`*identifer*`" cannot have a` *(signal|wait)* `for entry "`*entry*`".`

  The *entry* is designated as either a signal or a wait. However, *identifier* is not a semaphore task.

- `Number of` *(entries|tasks|processors)* `is outside of program limits of (1,`*n*`).`

  An internal program limit has been exceeded. Reduce the number of objects in the model.

- `Number of paths found to AND-Join "`*join-list*`" for task "`*identifier*`" does not match join list."`

  During activity graph traversal, one or more of the branches to the join *join-list* either originate from different forks, or do not originate from a fork at all.

- `Open arrival rate of` *n.n* `to task "`*identifier*`" is too high. Service time is` *n.n*`.` lqns

  The open arrival rate of $n.n$ to entry *identifier* is too high, so the input queue to the task has overflowed. This error may be the result of a transient condition, so the *stop-on-message-loss* pragma (c.f. §6.2) may be used to suppress this error. If the arrival rate exceeds the service time at the time the model converges, then the waiting time results for the entry will show infinity. Note that if a task accepts both open and closed classes, an overflow in the open class will result in zero throughput for the closed classes.

- `OR branch probabilities for OR-Fork "`*list*`" for task "`*identifier*`" do not sum to 1.0; sum is `*n.n*`.`

  All branches from an or-fork must be specified so that the sum of the probabilities equals one.

- `Processor "`*identifier*`" has invalid rate of `*n.n*`.`

  The processor rate parameter is used to scale the speed of the processor. A value greater than zero must be used.

- `Processor "`*identifier*`" using CFS scheduling has no group."`

  If the completely fair share scheduler is being used, there must be at least one group defined for the processor.

- `Reference task "`*identifier*`" cannot forward requests.`

  Reference tasks cannot accept messages, so they cannot forward.

- `Reference task "`*task-identifier*`", entry "`*entry-identifier*`" cannot have open arrival stream.`

  Reference tasks cannot accept messages.

- `Reference task "`*task-identifier*`", entry "`*entry-identifier*`" receives requests.`

  Reference tasks cannot accept messages.

- `Reference task "`*task-identifier*`", replies to entry "`*entry-identifier*`" from activity "`*activity-identifier*`)".`

  Reference tasks cannot accept messages, so they cannot generate replies. The activity *activity-identifier* replies to entry *entry-identifier*.

- `Required attribute '`*attribute*`' was not provided`

  The attribute named *attribute* is missing for the element.

- `Semaphore "wait" entry "`*entry-identifier*`" cannot accept send-no-reply requests.`

  An entry designated as the semaphore "wait" can only accept rendezvous-type messages because send-no-reply messages and open arrivals cannot block the caller if the semaphore is busy.

- `Start activity for entry "`*entry-identifier*`" is already defined.  Activity "`*activity-identifier*`" is a duplicate.`

  A start activity has already been defined. This one is a duplicate.

- `Symbol "`*identifier*`" not previously defined.`

  All identifiers must be declared before they can be used.

- `Task "`*identifier*`" cannot be an infinite server."`

  This error occurs whenever a reference task or a semaphore task is designated as an infinite server. Reference tasks are the customers in the model so an infinite reference task would imply an infinite number of customers[2]. An infinite semaphore task implies an infinite number of buffers – no blocking at the wait entry would ever occur.

- `Task "`*identifier*`" has activities but none are reachable.`

  None of the activities for *identifier* is reachable. The most likely cause is that the start activity is missing.

- `Task "`*identifier*`" has no entries.`

  No entries were defined for *identifier*.

[2] An infinite source of customers should be represented by open arrivals instead.

- `"Task "`*identifier*`" has `*n*` entries defined, exactly `*m*` are required.`

  The task *identifier* has *n* entries, *m* are required. This error typically occurs with semaphore tasks which must have exactly two entries.

- `Task "`*task-identifier*`", Activity "`*activity-identifer*`" is not specified.`

  An activity is declared but not defined.. An activity is "defined" when some parameter such as service time is specified.

- `Task "`*task-identifier*`", Activity "`*activity-identifer*`" makes a duplicate reply for Entry "`*entry-identifier*`".`

  An activity graph is making a reply to entry *entry-identifier* even though the entry is already in phase two. This error usually occurs when more than one reply to *entry-identifier* is specified in a sequence of activities.

- `Task "`*task-identifier*`", Activity "`*activity-identifer*`" makes invalid reply for Entry "`*entry-identifier*`".`

  An activity graph is making a reply to entry *entry-identifier* even though the activity is not reachable..

- `Task "`*task-identifier*`", Activity "`*activity-identifer*`" replies to Entry "`*entry-identifier*`" which does not accept rendezvous requests.`

  The activity graph specifies a reply to entry *entry-identifier* even though the entry does not accept rendezvous requests. (The entry either accepts send-no-reply requests or open arrivals).

- `Unknown element '`*element*`'`

  The *element* is not expected at this point in the input file. *Element* may not be spelled incorrectly, or if not, in an incorrect location in the input file.

8.3 Advisory Messages

- `Invalid convergence value of `*n.n*`, using `*m.m*`.` lqns

  The convergence value specified in the input file is not valid. The analytic solver is using $m.m$ instead.

- `Invalid standard deviation: sum=`*n.n*`, sum_sqr=`*n.n*`, n=`*n.n*`.`

  When calculating a standard deviation, the difference of the sum of the squares and the mean of the square of the sum was negative. This usually implies an internal error in the simulator.

- `Iteration limit of `*n*` is too small, using `*m*`.` lqns

  The iteration limit specified in the input file is not valid. The analytic solver is using m instead.

- `Messages dropped at task `*identifier*` for open-class queues.`

  Asynchronous send-no-reply messages were *lost* at the task *task*. This message will occur when the *stop-on-message-loss* pragma (c.f. §6.2) is set to ignore open class overflows. Note that if a task accepts both open and closed classes, an overflow in the open class will result in zero throughput for the closed classes.

- `Model failed to converge after `*n*` iterations (convergence test is `*n.n*`, limit is `*n.n*`).` lqns

  Sometimes the model fails to converge, particularly if there are several heavily utilized servers in a submodel. Sometimes, this problem can be solved by reducing the value of the under-relaxation coefficient. It may also be necessary to increase the iteration-limit, particularly if there are many submodels. With replicated models, it may be necessary to use 'loose' layering to get the model to converge. Convergence can be tracked using `-t`*convergence*.

- `No solve() call found in the lqx program in file:` *filename*`. solve() was invoked implicitly.`

  An LQX program was found in file *filename*. However, the function `solve()` was not invoked explictity. The program was executed to completion, after which `solve()` was called using the final value of all the variables found in the program.

- `Replicated Submodel` *n* `failed to converge after` *n* `iterations (convergence test is` *n.n*`, limit is` *m.m*`).` lqns

  The inner "replication" iteration failed to converge....

- `Service times for` *(processor) identifier* `have a range of` *n.n* `-` *n.n*`. Results may not be valid.` lqns

  The range of values of service times for a processor using processor sharing scheduling is over two orders of magnitude. The results may not be valid.

- `Specified confidence interval of` *n.n*`% not met after run time of` *n.n*`. Actual value is` *n.n*`%.` lqsim

- `Submodel` *n* `is empty.` lqns

  The call graph is interesting, to say the least.

- `Underrelaxation ignored.` *n.n* `outside range [0-2), using` *m.m*`.` lqns

  The under-relaxation coefficient specified in the input file is not valid. The solver is using $m.m$ instead[3].

- `The utilization of` *n.n* `at` *(task|processor) identifier* `with multiplicity` *m* is too high.

  This problem is the result of some of the approximations used by the analytic solver. The common causes are two-phase servers and the Rolia multiserver. If *identifer* is a multiserver, switching to the Conway approximation will often help. Values of $n.n$ in excess of 10% are likely the result of failures in the solver. Please send us the model file so that we can improve the algorithms.

8.4 Warning Messages

- *(activity|entry|task|processor)* `"`*identifier*`" is not used.`

  The object is not reachable. This may indicate an error in the specification of the model.

- *(Processor|Task)* `"`*identifier*`" is an infinite server with a multiplicity of` n`.`

  Infinite servers must always have a multiplicty of one. This error is caused by specifying both *delay* scheduling and a multiplicity for the named task or processor. The multiplicity attribute is ignored.

- *sched* `scheduling specified for` *(processor|task)* `"`*identifier*`" is not supported.`

  The solver does not support the specified scheduling type. First-in, first-out scheduling will be used instead.

- `Activity "`*identifier*`" has no service time specified.`

  No service time is specified for *identifier*.

- `Coefficient of variation is incompatible with phase type at` *(entry|task)* `"`*identifier*`"` *(phase|activity)* `"`*identifier*`".` lqns

  A coefficient of variation is specified at a using stochastic phase or activity.

[3] Values of under-relaxation from $1 < n \leq 2$ are more correctly called over-relaxation.

- `Entry "`*identifier*`" does not receive any requests.`

  Entry *identifier* is part of a non-reference task. However, no requests are made to this entry.

- `Entry "`*identifier*`" has no service time specified for any phase.`

  No service time is specified for entry *identifier*.

- `Entry "`*identifier*`" has no service time specified for phase` *n*`.`

  No service time is specified for entry *identifier*, phase n.

- `Infinite server "`*identifier*" accepts either asynchronous messages or open arrivals.

  The task or processor, *identifier*, is an infinite server. It processes either asynchronous messages or open arrivals. If the arrival rate exceeds the service rate of the infinite server, the number of instances of the infinite server will grow to infinity.

- `Histogram requested for entry "`*identifier*`", phase` *n* `- phase is not present.` lqsim

  A histogram for the service time of phase *n* of entry *identifier* was requested. This entry has no corresponding phase.

- `Priority specified (`*n*`) is outside of range (`*n*`,`*n*`). (Value has been adjusted to` *n*`).` lqsim

  The priority n is outside of the range specified.

- `No quantum specified for PS scheduling discipline. FIFO used."` lqsim

  A processor using processor sharing scheduling needs a quantum value when running on the simulator.

- `No requests made from` *from-identifier* `to` *to-identifier*`.` lqns

  The input file has a rendezvous or send-no-reply request with a value of zero.

- `Number of` *(processors|tasks|entries)* `defined (`*n*`) does not match number specified (`*m*`).`

  The processor task and entry chapters of the original input grammar can specify the number of objects that follow. The number specified does not match the actual number of objects. Specifying *zero* as a record count is valid.

- `Parameter is specified multiple times.`

  A parameter is specified more than one time. The first occurance is used.

- `Processor "`*identifier*`" is not running fair share scheduling."`

  A group was defined in the model and associated with a processor using a scheduling discipline other than completely fair scheduling.

- `Processor "`*identifier*`" has no tasks.`

  A processor was defined in the model, but it is not used by any tasks. This can occur if none of the entries or phases has any service time.

- `Queue Length is incompatible with task type at task` *identifier*`.` lqns

  A queue length parameter was specified at a task which does not support bounded queues.

- `Reference task "`*identifier*`" does not send any messages."`

  Reference tasks are customers in the model. This reference task does not visit any servers, so it serves no purpose.

- `Reference task "`*identifier*`" has more than one entry defined.`

  Reference tasks typically only have one entry. The named reference task has more than one. Requests are generated in proportion to the service times of the entries.

- `Task "`*task-identifier*`" with priority is running on processor "`*processor-identifier*`" which does not have priority scheduling.`

  Processors running with FCFS scheduling ignore priorities.

- `Value specified for `*(fanin|fanout)*`, `*n*`, is invalid.` lqns

  The value specified for a fan-in or fan-out is not valid and will be ignored.

- `The `*x*` feature is not supported in this version.`

  Feature *x* is not supported in this release.

8.5 Input File Parser Error Messages

- `error: not well-formed (invalid token)`

  This error occurs when an XML input file is expected, but some other input file type was given.

- `Parse error.`

  An error was detected while processing the XML input file. See the list below for more explantion:

  - `The primary document entity could not be opened. Id=`***URI***` while parsing `***filename***`.`

    This error is generated by the Xerces when the Uniform resource indicator *(URI)* specified as the argument to the `xsi:noNamespaceSchemaLocation` attribute of the `lqn-model` element cannot be opened. This argument must refer to a valid location containing the LQN schema files.

  - `The key for identity constraint of element '`*lqn-model*`' is not found.`

    When this message appears, Xerces does **not** provide many hints on where the actual error occurs because it always gives a line number which points to the end of the file (i.e. where the terminating tag `</lqn-model>` is).

    In this case, the following three points should be inspected to ensure validity of the model:

    1. All synchronous calls have a `dest` attribute which refers to a valid entry.
    2. All asynchronous calls have a `dest` attribute which refers to a valid entry.
    3. All forwarding calls have a `dest` attribute which refers to a valid entry.

    If it is not practical to manually inspect the model, run the XML file through another tool like XMLSpy or XSDvalid which will report more descriptive errors.

  - `The key for identity constraint of element '`*task*`' is not found.`

    When this error appears, it means there is something wrong within the `task` element indicated by the line number. Check that:

    ∗ The name `attribute` of all `reply-entry` elements refers to a valid entry name, which exists within the same task as the task activity graph.

    ∗ All activities which contain the attribute `bound-to-entry` have a valid entry name that exists within the same task as the task activity graph.

  - `The key for identity constraint of element '`*task-activities*`' is not found.`

    When this error appears, it means there is something wrong within the `task-activities` element indicated by the line number.

    Check that:

- ∗ All activities referenced within the `precedence` elements refer to activities which are defined for that particular task activity graph.
- ∗ The `name` attribute of all `reply-activity` elements refers to an activity defined within the mentioned `task-activities` element.
- ∗ The head attribute of all `post-loop` elements refers to an activity defined within the mentioned `task-activities` element.
- ∗ All post-LOOP elements which contain the optional attribute `end`, refers to an activity defined within the mentioned `task-activities` element.

– `Not enough elements to match content model :`*elements* ((run-control,plot-control,solver-params,processor),slot)

8.6 LQX Error messages

- `Runtime Exception Occured: Unable to Convert From:` *'«uninitialized»'* `To:` *'Array'*
  An unitialized variable is used where an array is expected (like in a foreach loop).

Chapter 9

Known Defects

9.1 MOL Multiserver Approximation Failure

The MOL multiserver approximation sometimes fails when the service time of the clients to the multiserver are significantly smaller than the service time of the server itself. The utilization of the multiserver will be too high. Sometimes, the problem can be solved by changing the mol-underrelaxation. Otherwise, switch to the more-expensive Conway multiserver approximation.

9.2 Chain construction for models with multi- and infinite-servers

9.3 No algorithm for phased multiservers OPEN class.

9.4 Overtaking probabilities are calculated using CV=1

9.5 Need to implement queue lengths for open classes.

f

Appendix A

Traditional Grammar

This chapter gives the formal description of Layered Queueing Network input file and parseable output file grammars in extended BNF form. For the nonterminals the notation $\langle \textit{nonterminal\_id} \rangle$ is used, while the terminals are written without brackets as they appear in the input text. The notation $\{\cdots\}_n^m$, where $n \leq m$ means that the part inside the curly brackets is repeated at least n times and at most m times. If $n = 0$, then the part may be missing in the input text. The notation $\langle\cdots\rangle_{\mathrm{opt}}$ means that the non-terminal is optional.

A.1 Input File Grammar

$$
\begin{array}{lcl}
\langle \textit{LQN\_input\_file} \rangle & \rightarrow & \langle \textit{general\_info} \rangle\ \langle \textit{processor\_info} \rangle\ \langle \textit{group\_info} \rangle_{\mathrm{opt}}\ \langle \textit{task\_info} \rangle\ \langle \textit{entry\_info} \rangle \\
 & & \{\langle \textit{activity\_info} \rangle\}_0 \\
 & | & \langle \textit{parameter\_list} \rangle\ \langle \textit{processor\_info} \rangle\ \langle \textit{group\_info} \rangle_{\mathrm{opt}}\ \langle \textit{task\_info} \rangle\ \langle \textit{entry\_info} \rangle \\
 & & \{\langle \textit{activity\_info} \rangle\}_0\ \langle \textit{report\_info} \rangle_{\mathrm{opt}}\ \langle \textit{convergence\_info} \rangle_{\mathrm{opt}}
\end{array}
$$

A.1.1 SPEX Parameters

$$
\begin{array}{lcl}
\langle \textit{parameter\_list} \rangle & \rightarrow & \{\langle \textit{comma\_expr} \rangle\}_1^{np} \\
\langle \textit{comma\_expr} \rangle & \rightarrow & \langle \textit{variable\_def} \rangle \\
 & | & \texttt{[}\ \langle \textit{expression} \rangle\ \texttt{,}\ \langle \textit{variable\_def} \rangle\ \texttt{]} \\
\langle \textit{variable\_def} \rangle & \rightarrow & \langle \textit{variable} \rangle\ \texttt{=}\ \langle \textit{ternary\_expr} \rangle \\
 & | & \texttt{[}\ \langle\ \textit{expression\_list} \rangle\ \texttt{]} \\
 & | & \texttt{[}\ \langle \textit{real} \rangle\ \texttt{:}\ \langle \textit{real} \rangle\ \texttt{,}\ \langle \textit{real} \rangle\ \texttt{]}
\end{array}
$$

A.1.2 General Information

$$
\begin{array}{lcllr}
\langle \textit{general\_info} \rangle & \rightarrow & \texttt{G}\ \langle \textit{comment} \rangle\ \langle \textit{conv\_val} \rangle\ \langle \textit{it\_limit} \rangle\ \langle \textit{print\_int} \rangle_{\mathrm{opt}}\ \langle \textit{underrelax\_coeff} \rangle_{\mathrm{opt}}\ \langle \textit{end\_list} \rangle & & \\
\langle \textit{comment} \rangle & \rightarrow & \langle \textit{string} \rangle & /*\ \textit{comment on the model}\ */ & \\
\langle \textit{conv\_val} \rangle & \rightarrow & \langle \textit{real} \rangle & /*\ \textit{convergence value}\ */ & \ddagger \\
\langle \textit{it\_limit} \rangle & \rightarrow & \langle \textit{integer} \rangle & /*\ \textit{max. nb. of iterations}\ */ & \ddagger \\
\langle \textit{print\_int} \rangle & \rightarrow & \langle \textit{integer} \rangle & & \ddagger \\
 & & & /*\ \textit{intermed. res. print interval}\ */ & \\
\langle \textit{underrelax\_coeff} \rangle & \rightarrow & \langle \textit{real} \rangle & /*\ \textit{under\_relaxation coefficient}\ */ & \ddagger \\
\langle \textit{end\_list} \rangle & \rightarrow & \texttt{-1} & /*\ \textit{end\_of\_list mark}\ */ & \\
\langle \textit{string} \rangle & \rightarrow & \texttt{"}\ \langle \textit{text} \rangle\ \texttt{"} & &
\end{array}
$$

A.1.3 Processor Information

⟨*processor_info*⟩	→	`P` ⟨*np*⟩ ⟨*p_decl_list*⟩	
⟨*np*⟩	→	⟨*integer*⟩	/∗ *total number of processors* ∗/
⟨*p_decl_list*⟩	→	$\{\langle p\_decl \rangle\}_1^{np}$ ⟨*end_list*⟩	
⟨*p_decl*⟩	→	`p` ⟨*proc_id*⟩ ⟨*scheduling_flag*⟩ ⟨*quantum*⟩$_{\mathrm{opt}}$ ⟨*multi_server_flag*⟩$_{\mathrm{opt}}$ ⟨*replication_flag*⟩$_{\mathrm{opt}}$ ⟨*proc_rate*⟩$_{\mathrm{opt}}$	
⟨*proc_id*⟩	→	⟨*integer*⟩ $\mid$ ⟨*identifier*⟩	/∗ *processor identifier* ∗/
⟨*scheduling_flag*⟩	→	`f`	/∗ *First come, first served* ∗/
	$\mid$	`h`	/∗ *Head Of Line* ∗/
	$\mid$	`p`	/∗ *Priority, preemeptive* ∗/
	$\mid$	`c` ⟨*real*⟩	/∗ *completely fair scheduling* ∗/
	$\mid$	`s` ⟨*real*⟩	/∗ *processor sharing* ∗/
	$\mid$	`i`	/∗ *Infinite or delay* ∗/
	$\mid$	`r`	/∗ *Random* ∗/
⟨*quantum*⟩	→	⟨*real*⟩ $\mid$ ⟨*variable*⟩	
⟨*multi_server_flag*⟩	→	`m` ⟨*copies*⟩	/∗ *number of duplicates* ∗/
	$\mid$	`i`	/∗ *Infinite server* ∗/
⟨*replication_flag*⟩	→	`r` ⟨*copies*⟩	/∗ *number of replicas* ∗/
⟨*proc_rate*⟩	→	`R` ⟨*ratio*⟩ $\mid$ ⟨*variable*⟩	/∗ *Relative proc. speed* ∗/
⟨*copies*⟩	→	⟨*integer*⟩ $\mid$ ⟨*variable*⟩	
⟨*ratio*⟩	→	⟨*real*⟩ $\mid$ ⟨*variable*⟩	

A.1.4 Group Information

⟨*group_info*⟩	→	`U` ⟨*ng*⟩ ⟨*g_decl_list*⟩ ⟨*end_list*⟩	
⟨*ng*⟩	→	⟨*integer*⟩	/∗ *total number of groups* ∗/
⟨*g_decl_list*⟩	→	$\{\langle g\_decl \rangle\}_1^{ng}$ ⟨*end_list*⟩	
⟨*g_decl*⟩	→	`g` ⟨*group_id*⟩ ⟨*group_share*⟩ ⟨*cap_flag*⟩$_{\mathrm{opt}}$ ⟨*proc_id*⟩	
⟨*group_id*⟩	→	⟨*identifier*⟩	
⟨*group_share*⟩	→	⟨*real*⟩ $\mid$ ⟨*variable*⟩	
⟨*cap_flag*⟩	→	`c`	

A.1.5 Task Information

⟨*task_info*⟩	→	`T` ⟨*nt*⟩ ⟨*t_decl_list*⟩	
⟨*nt*⟩	→	⟨*integer*⟩	/∗ *total number of tasks* ∗/
⟨*t_decl_list*⟩	→	$\{\langle t\_decl \rangle\}_1^{nt}$ ⟨*end_list*⟩	
⟨*t_decl*⟩	→	`t` ⟨*task_id*⟩ ⟨*task_sched_type*⟩ ⟨*entry_list*⟩ ⟨*queue_length*⟩$_{\mathrm{opt}}$ ⟨*proc_id*⟩ ⟨*task_pri*⟩$_{\mathrm{opt}}$ ⟨*think_time_flag*⟩$_{\mathrm{opt}}$ ⟨*tokens*⟩$_{\mathrm{opt}}$ ⟨*multi_server_flag*⟩$_{\mathrm{opt}}$ ⟨*replication_flag*⟩$_{\mathrm{opt}}$ ⟨*group_flag*⟩$_{\mathrm{opt}}$	
	$\mid$	`I` ⟨*from_task*⟩ ⟨*to_task*⟩ ⟨*fan_in*⟩	
	$\mid$	`O` ⟨*from_task*⟩ ⟨*to_task*⟩ ⟨*fan_out*⟩	
⟨*task_id*⟩	→	⟨*integer*⟩ $\mid$ ⟨*identifier*⟩	/∗ *task identifier* ∗/
⟨*task_sched_type*⟩	→	`r`	/∗ *reference task* ∗/
	$\mid$	`n`	/∗ *non-reference task* ∗/
	$\mid$	`h`	/∗ *Head of line* ∗/

	|	f	/∗ *FIFO Scheduling* ∗/
	|	i	/∗ *Infinite or delay server* ∗/
	|	p	/∗ *Polled scheduling at entries* ∗/
	|	b	/∗ *Bursty Reference task* ∗/
	|	S	/∗ *Semaphore* ∗/
⟨entry_list⟩	→	{⟨entry_id⟩}$_1^{ne_t}$ ⟨end_list⟩	
			/∗ *task t has* ne_t *entries* ∗/
⟨entry_id⟩	→	⟨integer⟩ | ⟨identifier⟩	
			/∗ *entry identifier* ∗/
⟨task_pri⟩	→	⟨integer⟩	/∗ *task priority, optional* ∗/
⟨queue_length⟩	→	q ⟨integer⟩	/∗ *open class queue length* ∗/
⟨group_flag⟩	→	g ⟨identfier⟩	/∗ *Group for scheduling* ∗/
⟨tokens⟩	→	t ⟨integer⟩	/∗ *Initial tokens* ∗/
⟨from_task⟩	→	⟨task_id⟩	/∗ *Source task* ∗/
⟨to_task⟩	→	⟨task_id⟩	/∗ *Destination task* ∗/
⟨fan_in⟩	→	⟨integer⟩	/∗ *fan in to this task* ∗/
⟨fan_out⟩	→	⟨integer⟩	/∗ *fan out from this task* ∗/

A.1.6 Entry Information

⟨entry_info⟩	→	E ⟨ne⟩ ⟨entry_decl_list⟩	
⟨ne⟩	→	⟨integer⟩	/∗ *total number of entries* ∗/
⟨entry_decl_list⟩	→	{⟨entry_decl⟩}$_1$ ⟨end_list⟩	
			/∗ *k = maximum number of phases* ∗/
⟨entry_decl⟩	→	a ⟨entry_id⟩ ⟨arrival_rate⟩	
	|	A ⟨entry_id⟩ ⟨activity_id⟩	
	|	F ⟨from_entry⟩ ⟨to_entry⟩ ⟨p_forward⟩	
	|	H ⟨entry_id⟩ ⟨phase⟩ ⟨hist_min⟩ ':' ⟨hist_max⟩ ⟨hist_bins⟩ ⟨hist_type⟩	
	|	M ⟨entry_id⟩ {⟨max_service_time⟩}$_1^k$ ⟨end_list⟩	
	|	P ⟨entry_id⟩	/∗ *Signal Semaphore* ∗/
	|	V ⟨entry_id⟩	/∗ *Wait Semaphore* ∗/
	|	Z ⟨entry_id⟩ {⟨think_time⟩}$_1^k$ ⟨end_list⟩	
	|	c ⟨entry_id⟩ {⟨coeff_of_variation⟩}$_1^k$ ⟨end_list⟩	
	|	f ⟨entry_id⟩ {⟨ph_type_flag⟩}$_1^k$ ⟨end_list⟩	
	|	p ⟨entry_id⟩ ⟨entry_priority⟩	
	|	s ⟨entry_id⟩ {⟨service_time⟩}$_1^k$ ⟨end_list⟩	
	|	y ⟨from_entry⟩ ⟨to_entry⟩ {⟨rendezvous⟩}$_1^k$ ⟨end_list⟩	
	|	z ⟨from_entry⟩ ⟨to_entry⟩ {⟨send_no_reply⟩}$_1^k$ ⟨end_list⟩	
⟨arrival_rate⟩	→	⟨real⟩ | ⟨variable⟩	/∗ *open arrival rate to entry* ∗/
⟨coeff_of_variation⟩	→	⟨real⟩ | ⟨variable⟩	/∗ *squared service time coefficient of variation* ∗/
⟨from_entry⟩	→	⟨entry_id⟩	/∗ *Source of a message* ∗/
⟨hist_bins⟩	→	⟨integer⟩	/∗ *Number of bins in histogram.* ∗/
⟨hist_max⟩	→	⟨real⟩	/∗ *Median service time.* ∗/
⟨hist_min⟩	→	⟨real⟩	/∗ *Median service time.* ∗/
⟨hist_type⟩	→	log | linear | sqrt	/∗ *bin type.* ∗/
⟨max_service_time⟩	→	⟨real⟩	/∗ *Median service time.* ∗/
⟨p_forward⟩	→	⟨real⟩	/∗ *probability of forwarding* ∗/
⟨phase⟩	→	1 | 2 | 3	/∗ *phase of entry* ∗/

```
⟨ph_type_flag⟩        →  0                                               /* stochastic phase */
                      |  1                                               /* deterministic phase */
⟨rate⟩                →  ⟨real⟩ | ⟨variable⟩                             /* nb. of calls per arrival */
⟨rendezvous⟩          →  ⟨real⟩ | ⟨variable⟩                             /* mean number of RNVs/ph */
⟨send_no_reply⟩       →  ⟨real⟩ | ⟨variable⟩                             /* mean nb.of non-blck.sends/ph */
⟨service_time⟩        →  ⟨real⟩ | ⟨variable⟩                             /* mean phase service time */
⟨think_time⟩          →  ⟨real⟩ | ⟨variable⟩                             /* Think time for phase. */
⟨to_entry⟩            →  ⟨entry_id⟩                                      /* Destination of a message */
```

A.1.7 Activity Information

```
⟨activity_info⟩       →  ⟨activity_defn_list⟩ ⟨activity_connections⟩_opt ⟨end_list⟩
                                                                         /* Activity definition. */
⟨activity_defn_list⟩  →  {⟨activity_defn⟩}_1^na
⟨activity_defn⟩       →  c ⟨activity_id⟩ ⟨coeff_of_variation⟩            /* Sqr. Coeff. of Var. */
                      |  f ⟨activity_id⟩ ⟨ph_type_flag⟩                  /* Phase type */
                      |  H ⟨entry_id⟩ ⟨hist_min⟩ ':' ⟨hist_max⟩ ⟨hist_bins⟩ ⟨hist_type⟩
                      |  M ⟨activity_id⟩ ⟨max_service_time⟩
                      |  s ⟨activity_id⟩ ⟨ph_serv_time⟩                  /* Service time */
                      |  Z ⟨activity_id⟩ ⟨think_time⟩                    /* Think time */
                      |  y ⟨activity_id⟩ ⟨to_entry⟩ ⟨rendezvous⟩         /* Rendezvous */
                      |  z ⟨activity_id⟩ ⟨to_entry⟩ ⟨send_no_reply⟩      /* Send-no-reply */
                                                                         /* Activity Connections. */
⟨activity_connections⟩  →  : ⟨activity_conn_list⟩
⟨activity_conn_list⟩  →  ⟨activity_conn⟩ {; ⟨activity_conn⟩}_1^na
⟨activity_conn⟩       →  ⟨join_list⟩
                      |  ⟨join_list⟩ -> ⟨fork_list⟩
⟨join_list⟩           →  ⟨reply_activity⟩
                      |  ⟨and_join_list⟩
                      |  ⟨or_join_list⟩
⟨fork_list⟩           →  ⟨activity_id⟩
                      |  ⟨and_fork_list⟩
                      |  ⟨or_fork_list⟩
                      |  ⟨loop_list⟩
⟨and_join_list⟩       →  ⟨reply_activity⟩ {& ⟨reply_activity⟩}_1^na ⟨quorum_count⟩_opt
⟨or_join_list⟩        →  ⟨reply_activity⟩ {+ ⟨reply_activity⟩}_1^na
⟨and_fork_list⟩       →  ⟨activity_id⟩ {& ⟨activity_id⟩}_1^na
⟨or_fork_list⟩        →  ⟨prob_activity⟩ {+ ⟨prob_activity⟩}_1^na
⟨loop_list⟩           →  ⟨loop_activity⟩ {, ⟨loop_activity⟩}_0^na ⟨end_activity⟩_opt
⟨prob_activity⟩       →  ( ⟨real⟩ ) ⟨activity_id⟩
⟨loop_activity⟩       →  ⟨real⟩ * ⟨activity_id⟩
⟨end_activity⟩        →  , ⟨activity_id⟩
⟨reply_activity⟩      →  ⟨activity_id⟩ ⟨reply_list⟩_opt
⟨reply_list⟩          →  [ ⟨entry_id⟩ {, ⟨entry_id⟩ }_0^ne ]
⟨quorum_count⟩        →  ( ⟨integer⟩ )                                   /* Quorum */
```

A.1.8 SPEX Report Information

```
⟨report_info⟩           →  R ⟨nr⟩ ⟨report_decl_list⟩ ⟨end_list⟩
                        |  R ⟨nr⟩ ⟨identifier⟩ ( ⟨expression_list⟩ )
⟨report_decl_list⟩      →  {⟨r_decl⟩}_1^nr
⟨r_decl⟩                →  ⟨variable⟩ = ⟨ternary_expr⟩
                        |  ⟨expression⟩
```

A.1.9 SPEX Convergence Information

```
⟨convergence_info⟩       →  C ⟨nc⟩ ⟨convergence_decl_list⟩ ⟨end_list⟩
⟨convergence_decl_list⟩  →  {⟨c_decl⟩}_1^nr
⟨c_decl⟩                 →  ⟨variable⟩ = ⟨ternary_expr⟩
```

A.1.10 Expressions

```
⟨ternary_expression⟩  →  ⟨or_expression⟩ ? ⟨or_expression⟩ : ⟨or_expression⟩
                      |  ⟨or_expression⟩
⟨or_expression⟩       →  ⟨or_expression⟩ | ⟨and_expression⟩                /* Logical OR */
                      |  ⟨and_expression⟩
⟨and_expression⟩      →  ⟨and_epxression⟩ & ⟨compare_expression⟩          /* Logical AND */
                      |  ⟨compare_expression⟩
⟨compare_expression⟩  →  ⟨compare_expression⟩ == ⟨expression⟩
                      |  ⟨compare_expression⟩ != ⟨expression⟩
                      |  ⟨compare_expression⟩ < ⟨expression⟩
                      |  ⟨compare_expression⟩ <= ⟨expression⟩
                      |  ⟨compare_expression⟩ > ⟨expression⟩
                      |  ⟨compare_expression⟩ >= ⟨expression⟩
                      |  ⟨expression⟩
⟨expression⟩          →  ⟨expression⟩ + ⟨term⟩
                      |  ⟨expression⟩ − ⟨term⟩
                      |  ⟨term⟩
⟨term⟩                →  ⟨term⟩ * ⟨power⟩
                      |  ⟨term⟩ / ⟨power⟩
                      |  ⟨term⟩ % ⟨power⟩                                 /* Modulus */
                      |  ⟨power⟩
⟨power⟩               →  ⟨prefix⟩ ** ⟨power⟩               /* Exponentiation, right associative */
                      |  ⟨prefix⟩
⟨prefix⟩              →  ! ⟨factor⟩                                       /* Logical NOT */
                      |  ⟨factor⟩
⟨factor⟩              →  ( ⟨expression⟩ )
                      |  ⟨identifier⟩ ( ⟨expression_list⟩ )
                      |  ⟨variable⟩ [ ⟨expression_list⟩ ]
                      |  ⟨variable⟩
                      |  ⟨double⟩
⟨expression_list⟩     →  ⟨expression⟩ {, ⟨expression⟩ }_0
⟨int⟩                 →                                            /* Non negative integer */
⟨double⟩              →                              /* Non negative double precision number */
```

A.1.11 Identifiers

Identifiers may be zero or more leading underscores (‘_’), followed by a character, followed by any number of characters, numbers or underscores. Punctuation characters and other special characters such as the dollar-sign (‘$’) are not permitted. The following, `_p1`, `foo_bar`, and `__P_21_proc` are valid identifiers, while `_21` and `$proc` are not.

A.1.12 Variables

SPEX variables all start with a dollar-sign (‘$’) followed by any number of characters, numbers or underscores (‘_’). The following, `$s1`, `$1`, and `$_x` are all valid variables. SPEX variables are treated as global symbols by the underlying LQX program. Variables used to store arrays will also generate a *local* variable of the same name, except without the leading dollar-sign.

A.2 Output File Grammar

⟨*LQN_output_file*⟩	→	⟨*general*⟩ ⟨*bound*⟩$_{\text{opt}}$ ⟨*waiting*⟩$_{\text{opt}}$ ⟨*wait_var*⟩$_{\text{opt}}$ ⟨*snr_waiting*⟩$_{\text{opt}}$ ⟨*snr_wait_var*⟩$_{\text{opt}}$ ⟨*drop_prob*⟩$_{\text{opt}}$ ⟨*join*⟩$_{\text{opt}}$ ⟨*service*⟩$_{\text{opt}}$ ⟨*variance*⟩$_{\text{opt}}$ `<exceeded>`$_{\text{opt}}$ {`<distribution>`}$_0$ ⟨*thpt_ut*⟩ ⟨*open_arrivals*⟩$_{\text{opt}}$ ⟨*processor*⟩	
⟨*from_entry*⟩	→	⟨*entry_name*⟩	/∗ *Source entry id.* ∗/
⟨*to_entry*⟩	→	⟨*entry_name*⟩	/∗ *Destination entry id.* ∗/
⟨*entry_name*⟩	→	⟨*identifier*⟩	
⟨*task_name*⟩	→	⟨*identifier*⟩	
⟨*proc_name*⟩	→	⟨*identifier*⟩	
⟨*float_phase_list*⟩	→	{⟨*real*⟩} ⟨*end_list*⟩	
⟨*real*⟩	→	⟨*float*⟩ | ⟨*integer*⟩	

A.2.1 General Information

⟨*general*⟩	→	⟨*valid*⟩ ⟨*convergence*⟩ ⟨*iterations*⟩ ⟨*n_processors*⟩ ⟨*n_phases*⟩
⟨*valid*⟩	→	`V` ⟨*yes_or_no*⟩
⟨*yes_or_no*⟩	→	`y` | `Y` | `n` | `N`
⟨*convergence*⟩	→	`C` ⟨*real*⟩
⟨*iterations*⟩	→	`I` ⟨*integer*⟩
⟨*n_processors*⟩	→	`PP` ⟨*integer*⟩
⟨*n_phases*⟩	→	`NP` ⟨*integer*⟩

A.2.2 Throughput Bounds

⟨*bound*⟩	→	`B` ⟨*nt*⟩ {⟨*bounds_entry*⟩}$_1^{nt}$ ⟨*end_list*⟩	
⟨*bounds_entry*⟩	→	⟨*task_name*⟩ ⟨*real*⟩	
⟨*nt*⟩	→	⟨*integer*⟩	/∗ *Total number of tasks* ∗/

A.2.3 Waiting Times

⟨*waiting*⟩	→	`W` ⟨*ne*⟩ {⟨*waiting_t_tbl*⟩}$_1^{nt}$ ⟨*end_list*⟩
⟨*waiting_t_tbl*⟩	→	⟨*task_name*⟩ `:` ⟨*waiting_e_tbl*⟩ ⟨*end_list*⟩ ⟨*waiting_a_tbl*⟩$_{\text{opt}}$
⟨*waiting_e_tbl*⟩	→	{⟨*waiting_entry*⟩}$_0^{ne}$
⟨*waiting_entry*⟩	→	⟨*from_entry*⟩ ⟨*to_entry*⟩ ⟨*float_phase_list*⟩

$\langle \textit{ne} \rangle \rightarrow \langle \textit{integer} \rangle$ /* *Number of Entries* */

$\langle \textit{waiting\_a\_tbl} \rangle \rightarrow \texttt{:}\ \{\langle \textit{waiting\_activity} \rangle\}_0^{na}\ \langle \textit{end\_list} \rangle$

$\langle \textit{waiting\_activity} \rangle \rightarrow \langle \textit{from\_activity} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

$\langle \textit{na} \rangle \rightarrow \langle \textit{integer} \rangle$ /* *Number of Activities* */

A.2.4 Waiting Time Variance

$\langle \textit{wait\_var} \rangle \rightarrow \texttt{VARW}\ \langle \textit{ne} \rangle\ \{\langle \textit{wait\_var\_t\_tbl} \rangle\}_1^{nt}\ \langle \textit{end\_list} \rangle$

$\langle \textit{wait\_var\_t\_tbl} \rangle \rightarrow \langle \textit{task\_name} \rangle\ \texttt{:}\ \langle \textit{wait\_var\_e\_tbl} \rangle\ \langle \textit{end\_list} \rangle\ \langle \textit{wait\_var\_a\_tbl} \rangle_{\mathrm{opt}}$

$\langle \textit{wait\_var\_e\_tbl} \rangle \rightarrow \{\langle \textit{wait\_var\_entry} \rangle\}_0^{ne}$

$\langle \textit{wait\_var\_entry} \rangle \rightarrow \langle \textit{from\_entry} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

$\langle \textit{wait\_var\_a\_tbl} \rangle \rightarrow \texttt{:}\ \{\langle \textit{wait\_var\_activity} \rangle\}_0^{na}\ \langle \textit{end\_list} \rangle$

$\langle \textit{wait\_var\_activity} \rangle \rightarrow \langle \textit{from\_activity} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

A.2.5 Send-No-Reply Waiting Time

$\langle \textit{snr\_waiting} \rangle \rightarrow \texttt{Z}\ \langle \textit{ne} \rangle\ \{\langle \textit{snr\_waiting\_t\_tbl} \rangle\}_1^{nt}\ \langle \textit{end\_list} \rangle$

$\langle \textit{snr\_waiting\_t\_tbl} \rangle \rightarrow \langle \textit{task\_name} \rangle\ \texttt{:}\ \langle \textit{snr\_waiting\_e\_tbl} \rangle\ \langle \textit{end\_list} \rangle\ \langle \textit{snr\_waiting\_a\_tbl} \rangle_{\mathrm{opt}}$

$\langle \textit{snr\_waiting\_e\_tbl} \rangle \rightarrow \{\langle \textit{snr\_waiting\_entry} \rangle\}_0^{ne}$

$\langle \textit{snr\_waiting\_entry} \rangle \rightarrow \langle \textit{from\_entry} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

$\langle \textit{snr\_waiting\_a\_tbl} \rangle \rightarrow \texttt{:}\ \{\langle \textit{snr\_waiting\_activity} \rangle\}_0^{na}\ \langle \textit{end\_list} \rangle$

$\langle \textit{snr\_waiting\_activity} \rangle \rightarrow \langle \textit{from\_activity} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

A.2.6 Send-No-Reply Wait Variance

$\langle \textit{snr\_wait\_var} \rangle \rightarrow \texttt{VARZ}\ \langle \textit{ne} \rangle\ \{\langle \textit{snr\_wait\_var\_t\_tbl} \rangle\}_1^{nt}\ \langle \textit{end\_list} \rangle$

$\langle \textit{snr\_wait\_var\_t\_tbl} \rangle \rightarrow \langle \textit{task\_name} \rangle\ \texttt{:}\ \langle \textit{snr\_wait\_var\_e\_tbl} \rangle\ \langle \textit{end\_list} \rangle\ \langle \textit{snr\_wait\_var\_a\_tbl} \rangle_{\mathrm{opt}}$

$\langle \textit{snr\_wait\_var\_e\_tbl} \rangle \rightarrow \{\langle \textit{snr\_wait\_var\_entry} \rangle\}_0^{ne}$

$\langle \textit{snr\_wait\_var\_entry} \rangle \rightarrow \langle \textit{from\_entry} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

$\langle \textit{snr\_wait\_var\_a\_tbl} \rangle \rightarrow \texttt{:}\ \{\langle \textit{snr\_wait\_var\_activity} \rangle\}_0^{na}\ \langle \textit{end\_list} \rangle$

$\langle \textit{snr\_wait\_var\_activity} \rangle \rightarrow \langle \textit{from\_activity} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

A.2.7 Arrival Loss Probabilities

$\langle \textit{drop\_prob} \rangle \rightarrow \texttt{DP}\ \langle \textit{ne} \rangle\ \{\langle \textit{drop\_prob\_t\_tbl} \rangle\}_1^{nt}\ \langle \textit{end\_list} \rangle$

$\langle \textit{drop\_prob\_t\_tbl} \rangle \rightarrow \langle \textit{task\_name} \rangle\ \texttt{:}\ \langle \textit{drop\_prob\_e\_tbl} \rangle\ \langle \textit{end\_list} \rangle\ \langle \textit{drop\_prob\_a\_tbl} \rangle_{\mathrm{opt}}$

$\langle \textit{drop\_prob\_e\_tbl} \rangle \rightarrow \{\langle \textit{drop\_prob\_entry} \rangle\}_0^{ne}$

$\langle \textit{drop\_prob\_entry} \rangle \rightarrow \langle \textit{from\_entry} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

$\langle \textit{drop\_prob\_a\_tbl} \rangle \rightarrow \texttt{:}\ \{\langle \textit{drop\_prob\_activity} \rangle\}_0^{na}\ \langle \textit{end\_list} \rangle$

$\langle \textit{drop\_prob\_activity} \rangle \rightarrow \langle \textit{from\_activity} \rangle\ \langle \textit{to\_entry} \rangle\ \langle \textit{float\_phase\_list} \rangle$

A.2.8 Join Delays

$\langle \textit{join} \rangle \rightarrow \texttt{J}\ \langle \textit{ne} \rangle\ \{\langle \textit{join\_t\_tbl} \rangle\}_1^{nt}\ \langle \textit{end\_list} \rangle$

$\langle \textit{join\_t\_tbl} \rangle \rightarrow \langle \textit{task\_name} \rangle\ \texttt{:}\ \langle \textit{join\_a\_tbl} \rangle\ \langle \textit{end\_list} \rangle$

$\langle \textit{join\_a\_tbl} \rangle \rightarrow \{\langle \textit{join\_entry} \rangle\}_0^{na}$

$\langle \textit{join\_entry} \rangle \rightarrow \langle \textit{from\_activity} \rangle\ \langle \textit{to\_activity} \rangle\ \langle \textit{real} \rangle\ \langle \textit{real} \rangle$

A.2.9 Service Time

⟨*service*⟩	→	`X` ⟨*ne*⟩ $\{\langle\textit{service\_t\_tbl}\rangle\}_{1}^{nt}$ ⟨*end_list*⟩	
⟨*service_t_tbl*⟩	→	⟨*task_name*⟩ **:** ⟨*service_e_tbl*⟩ ⟨*end_list*⟩ ⟨*service_a_tbl*⟩$_{\text{opt}}$	
⟨*service_e_tbl*⟩	→	$\{\langle\textit{service\_entry}\rangle\}_{0}^{ne}$	
⟨*service_entry*⟩	→	⟨*entry_name*⟩ ⟨*float_phase_list*⟩	
⟨*service_a_tbl*⟩	→	**:** $\{\langle\textit{service\_activity}\rangle\}_{0}^{na}$ ⟨*end_list*⟩	
⟨*service_activity*⟩	→	⟨*activity_name*⟩ ⟨*float_phase_list*⟩	

A.2.10 Service Time Variance

⟨*variance*⟩	→	`VAR` ⟨*ne*⟩ $\{\langle\textit{variance\_t\_tbl}\rangle\}_{1}^{nt}$ ⟨*end_list*⟩	
⟨*variance_t_tbl*⟩	→	⟨*task_name*⟩ **:** ⟨*variance_e_tbl*⟩ ⟨*end_list*⟩ ⟨*variance_a_tbl*⟩$_{\text{opt}}$	
⟨*variance_e_tbl*⟩	→	$\{\langle\textit{variance\_entry}\rangle\}_{0}^{ne}$	
⟨*variance_entry*⟩	→	⟨*entry_name*⟩ ⟨*float_phase_list*⟩	
⟨*variance_a_tbl*⟩	→	**:** $\{\langle\textit{variance\_activity}\rangle\}_{0}^{na}$ ⟨*end_list*⟩	
⟨*variance_activity*⟩	→	⟨*activity_name*⟩ ⟨*float_phase_list*⟩	

A.2.11 Probability Service Time Exceeded

⟨*variance*⟩	→	`VAR` ⟨*ne*⟩ $\{\langle\textit{variance\_t\_tbl}\rangle\}_{1}^{nt}$ ⟨*end_list*⟩	

A.2.12 Service Time Distribution

⟨*distribution*⟩	→	`D` ⟨*entry_name*⟩ ⟨*statistics*⟩ $\{\langle\textit{hist\_bin}\rangle\}_{0}$ ⟨*end_list*⟩	
	|	`D` ⟨*task_name*⟩ ⟨*activity_name*⟩ ⟨*statistics*⟩ $\{\langle\textit{hist\_bin}\rangle\}_{0}$ ⟨*end_list*⟩	
⟨*statistics*⟩	→	⟨*phase*⟩ ⟨*mean*⟩ ⟨*stddev*⟩ ⟨*skew*⟩ ⟨*kurtosis*⟩	
⟨*hist_bin*⟩	→	⟨*begin*⟩ ⟨*end*⟩ ⟨*probability*⟩ $\{\langle\textit{bin\_conf}\rangle\}_{0}^{2}$	
⟨*mean*⟩	→	⟨*real*⟩	/∗ *Distribution mean* ∗/
⟨*stddev*⟩	→	⟨*real*⟩	/∗ *Distribution standard deviation* ∗/
⟨*skew*⟩	→	⟨*real*⟩	/∗ *Distribution skew* ∗/
⟨*kurtosis*⟩	→	⟨*real*⟩	/∗ *Distribution kurtosis* ∗/
⟨*probability*⟩	→	⟨*real*⟩	/∗ *0.0 - 1.0* ∗/
⟨*bin_conf*⟩	→	`%` ⟨*conf_level*⟩ ⟨*real*⟩	

A.2.13 Throughputs and Utilizations

⟨*thpt_ut*⟩	→	`FQ` ⟨*nt*⟩ $\{\langle\textit{thpt\_ut\_task}\rangle\}_{1}^{nt}$ ⟨*end_list*⟩	
⟨*thpt_ut_task*⟩	→	⟨*task_name*⟩ ⟨*net*⟩ {`<thpt_ut_entry>`}$_{1}^{net}$ ⟨*end_list*⟩ ⟨*thpt_ut_task_total*⟩$_{\text{opt}}$	
⟨*thpt_ut_entry*⟩	→	⟨*entry_name*⟩ ⟨*entry_info*⟩ $\{\langle\textit{thpt\_ut\_confidence}\rangle\}_{0}$	
⟨*entry_info*⟩	→	⟨*throughput*⟩ ⟨*utilization*⟩ ⟨*end_list*⟩ ⟨*total_util*⟩	
⟨*throughput*⟩	→	⟨*real*⟩	/∗ *Task Throughput* ∗/
⟨*utilization*⟩	→	⟨*float_phase_list*⟩	/∗ *Per phase task util.* ∗/
⟨*total_util*⟩	→	⟨*real*⟩	
⟨*thpt_ut_task_total*⟩	→	⟨*entry_info*⟩ $\{\langle\textit{thpt\_ut\_conf}\rangle\}_{0}$	
⟨*thpt_ut_conf*⟩	→	`%` ⟨*conf_level*⟩ ⟨*entry_info*⟩	

⟨*conf_level*⟩ → ⟨*integer*⟩

A.2.14 Arrival Rates and Waiting Times

⟨*open_arrivals*⟩ → `R` ⟨*no*⟩ {⟨*open_arvl_entry*⟩}$_1^{no}$ ⟨*end_list*⟩

⟨*no*⟩ → ⟨*integer*⟩ /* *Number of Open Arrivals* */

⟨*open_arvl_entry*⟩ → ⟨*from_entry*⟩ ⟨*to_entry*⟩ ⟨*real*⟩ ⟨*real*⟩
| ⟨*from_entry*⟩ ⟨*to_entry*⟩ ⟨*real*⟩ `Inf`

A.2.15 Utilization and Waiting per Phase for Processor

⟨*processor*⟩ → {⟨*proc_group*⟩}$_1^{n\_processors}$ ⟨*end_list*⟩

⟨*proc_group*⟩ → `P` ⟨*proc_name*⟩ ⟨*nt*⟩ {⟨*proc_task*⟩}$_1^{nt}$ ⟨*end_list*⟩ ⟨*proc_total*⟩$_{\mathrm{opt}}$

⟨*proc_task*⟩ → ⟨*task_name*⟩ ⟨*proc_task_info*⟩ {⟨*proc_entry_info*⟩}$_1^{ne}$ ⟨*end_list*⟩ ⟨*task_total*⟩$_{\mathrm{opt}}$

⟨*proc_task_info*⟩ → ⟨*ne*⟩ ⟨*priority*⟩ ⟨*multiplicity*⟩$_{\mathrm{opt}}$

⟨*priority*⟩ → ⟨*integer*⟩ /* *Prio. of task* */

⟨*multiplicity*⟩ → ⟨*integer*⟩ /* *Number of task instances* */

⟨*proc_info*⟩ → ⟨*entry_name*⟩ ⟨*proc_entry_info*⟩ {⟨*proc_entry_conf*⟩}$_0$

⟨*proc_entry_info*⟩ → ⟨*utilization*⟩ ⟨*sched_delay*⟩ ⟨*end_list*⟩

⟨*sched_delay*⟩ → ⟨*float_phase_list*⟩ /* *Scheduling delay* */

⟨*proc_entry_conf*⟩ → `%` ⟨*integer*⟩ ⟨*proc_entry_info*⟩

⟨*task_total*⟩ → ⟨*real*⟩ {⟨*proc_total_conf*⟩}$_0$

⟨*proc_total*⟩ → ⟨*real*⟩ {⟨*proc_total_conf*⟩}$_0$ ⟨*end_list*⟩

⟨*proc_total_conf*⟩ → `%` ⟨*integer*⟩ ⟨*real*⟩

Bibliography

[1] The Apache Software Foundation. *Xerces C++ Documentation*.

[2] S. C. Bruell, G. Balbo, and P. V. Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service centers. *Performance Evaluation*, 4(4):241–260, 1984. `doi:10.1016/0166-5316(84)90010-5`.

[3] Adrian E. Conway. Fast approximate solution of queueing networks with multi-server chain-dependent FCFS queues. In Ramon Puigjaner and Dominique Potier, editors, *Modeling Techniques and Tools for Computer Performance Evaluation*, pages 385–396. Plenum, New York, 1989.

[4] Edmundo de Souza e Silva and Richard R. Muntz. Approximate solutions for a class of non-product form queueing network models. *Performance Evaluation*, 7(3):221–242, 1987. `doi:10.1016/0166-5316(87)90042-3`.

[5] Greg Franks. Traffic dependencies in client-server systems and their effect on performance prediction. In *IEEE International Computer Performance and Dependability Symposium*, pages 24–33, Erlangen, Germany, April 1995. IEEE Computer Society Press. `doi:10.1109/IPDS.1995.395840`.

[6] Greg Franks, Tariq Al-Omari, Murray Woodside, Olivia Das, and Salem Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Transactions on Software Engineering*, 35(2):148–161, March–April 2009. `doi:10.1109/TSE.2008.74`.

[7] Roy Gregory Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.

[8] Xianghong Jiang. Evaluation of approximation for response time of parallel task graph model. Master's thesis, Department of Systems and Computer Engineering, Carleton University, Canada, April 1996.

[9] Lianhua Li and Greg Franks. Performance modeling of systems using fair share scheduling with layered queueing networks. In *Proceedings of the Seventeenth IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS 2009)*, pages 1–10, London, September 21–23 2009. `doi:10.1109/MASCOT.2009.5366689`.

[10] Victor W. Mak and Stephen F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990. `doi:10.1109/71.80155`.

[11] Martin Mroz and Greg Franks. A performance experiment system supporting fast mapping of system issues. In *Fourth International Conference on Performance Evaluation Methodologies and Tools*, Pisa, Italy, October 20–22 2009. `doi:10.4108/ICST.VALUETOOLS2009.7807`.

[12] John E. Neilson. PARASOL: A simulator for distributed and/or parallel systems. Technical Report SCS TR-192, School of Computer Science, Carleton University, Ottawa, Ontario, Canada, May 1991.

[13] Martin Reiser. A queueing network analysis of computer communication networks with window flow control. *IEEE Transactions on Communications*, 27(8):1199 – 1209, August 1979. `doi:10.1109/TCOM.1979.1094531`.

[14] J. A. Rolia and K. A. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, August 1995. doi:10.1109/32.403785.

[15] Jerome Alexander Rolia. *Predicting the Performance of Software Systems*. PhD thesis, Univerisity of Toronto, Toronto, Ontario, Canada, January 1992.

[16] Rainer Schmidt. An approximate MVA algorithm for exponential, class-dependent multiple servers. *Performance Evaluation*, 29(4):245–254, 1997. doi:10.1016/S0166-5316(96)00048-X.

[17] C. U. Smith and L. G. Williams. A performance model interchange format. *Journal of Systems and Software*, 49(1):63–80, 1999. doi:10.1016/S0164-1212(99)00067-9.

[18] C. U. Smith and L. G Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Object Technology Series. Addison Wesley, 2002.

[19] Connie U. Smith and Catalina M. Lladó. Performance model interchange format (PMIF 2.0): XML definition and implementation. In *Proceedings of the First International Conference on the Quantative Evaluation of Systems (QEST)*, pages 38–47, Enschede, the Netherlands, September 27–30 2004. IEEE Computer Society Press. doi:10.1109/QEST.2004.1348017.

[20] C. Murray Woodside, John E. Neilson, Dorina C. Petriu, and Shikharesh Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *IEEE Transactions on Computers*, 44(8):20–34, August 1995. doi:10.1109/12.368012.

[21] Murray Woodside and Greg Franks. Tutorial introduction to layered modeling of software performance. Revision 6554.

[22] Xiuping Wu. An approach to predicting peformance for component based systems. Master's thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, August 2003. Available from: ftp://ftp.sce.carleton.ca/pub/cmw/xpwu-mthesis.pdf.

Index