LINE: Queueing Analysis Algorithms

Solver Internals

(R Edition)

Last revision: November 28, 2026

Contents

1 Introduction **8**
- 1.1 Purpose of this manual . . . 8
- 1.2 Relation to the existing manuals . . . 8
- 1.3 The common solution pipeline . . . 9
- 1.4 Notation . . . 11
- 1.5 Organization . . . 11
- 1.6 Use of AI assistance . . . 11

2 The R binding **13**
- 2.1 What R owns and what it does not . . . 13
- 2.2 The C application binary interface . . . 13
  - 2.2.1 The surface is the argument vector . . . 14
  - 2.2.2 One document shape . . . 14
  - 2.2.3 Errors carry identity, not prose . . . 14
- 2.3 The seam inside `cpp/` . . . 15
  - 2.3.1 Capture is at the file descriptor, and is not optional . . . 15
  - 2.3.2 Four reentrancy defects, fixed rather than worked around . . . 15
  - 2.3.3 Two gaps closed on the way . . . 16
  - 2.3.4 Reentrancy is bounded and stated . . . 16
  - 2.3.5 The interrupt is partial, and says so . . . 16
- 2.4 The document layer . . . 16
- 2.5 Assembling results . . . 17
- 2.6 What the suite measures . . . 17

I The solver interface **18**

3 SolverAUTO **19**
- 3.1 Overview . . . 19
- 3.2 Selection policy . . . 19
- 3.3 Architectural flow . . . 20
- 3.4 Feature and method support . . . 21

3.5 Method algorithms . . . 21
3.6 Bibliographic notes . . . 21

II Analysis algorithms **22**

4 SolverBA **23**
4.1 Overview . . . 23
4.2 Software architecture . . . 23
4.3 Asymptotic families . . . 24
4.4 Bound hierarchies . . . 25
4.5 Reduction bounds . . . 26
4.6 Single-class bounds . . . 26
4.7 Achievable-region bounds . . . 27
4.8 Stochastic network calculus bounds . . . 28
4.9 Architectural flow . . . 28
4.10 Implementation notes . . . 29
4.11 Feature and method support . . . 30
4.12 Method algorithms . . . 31
4.13 Bibliographic notes . . . 32

5 SolverCTMC **34**
5.1 Overview . . . 34
5.2 State-space generation . . . 34
5.3 Steady-state and transient solution . . . 35
5.4 Analyzers and methods . . . 36
5.5 State-space aggregation via decision diagrams . . . 37
5.6 Symbolic solution and parametric sensitivity . . . 38
5.7 Architectural flow . . . 38
5.8 Implementation notes . . . 39
5.9 Feature and method support . . . 39
5.10 Method algorithms . . . 40
5.11 Bibliographic notes . . . 41

6 SolverFLD **42**
6.1 Overview . . . 42
6.2 Analyzers and methods . . . 42
6.3 Transient analysis and passage times . . . 44
6.4 Architectural flow . . . 44
6.5 Implementation notes . . . 45
6.6 Feature and method support . . . 45
6.7 Method algorithms . . . 47
6.8 Bibliographic notes . . . 47

7 SolverLDES **49**
7.1 Overview . . . 49
7.2 Simulation algorithm . . . 50
7.3 Analyzers, estimation, and rewards . . . 50
7.4 Architectural flow . . . 51
7.5 Implementation notes . . . 52
7.6 Feature and method support . . . 52
7.7 Method algorithms . . . 53
7.8 Bibliographic notes . . . 53

8 SolverMAM **54**
8.1 Overview . . . 54
8.2 Analyzers and methods . . . 54
8.3 Level-dependent and transient QBDs . . . 56
8.4 Architectural flow . . . 56
8.5 Implementation notes . . . 56
8.6 Feature and method support . . . 57
8.7 Method algorithms . . . 58
8.8 Bibliographic notes . . . 59

9 SolverMVA **60**
9.1 Overview . . . 60
9.2 Software architecture . . . 60
9.3 Default method selection . . . 60
9.4 Exact MVA methods . . . 61
9.5 AMVA methods . . . 62
9.6 Bound methods . . . 63
9.7 Specialized analyzers . . . 64
9.8 Fork-join transformations . . . 65
9.9 Architectural flow . . . 66
9.10 Implementation notes . . . 67
9.11 Feature and method support . . . 68
9.12 Method algorithms . . . 69
9.13 Bibliographic notes . . . 70

10 SolverNC **71**
10.1 Overview . . . 71
10.2 Software architecture . . . 71
10.3 Exact methods . . . 72
10.4 Asymptotic and integral methods . . . 73
10.5 Birman-Kogan methods . . . 75
10.6 Probability and specialized analyzers . . . 76
10.6.1 Matrix permanent library . . . 76

10.6.2 The joint law of the per-station totals . . . 78
10.7 State-dependent routing . . . 79
10.8 Perfect sampling . . . 79
10.9 Architectural flow . . . 80
10.10 Implementation notes . . . 80
10.11 Feature and method support . . . 81
10.12 Method algorithms . . . 82
10.13 Bibliographic notes . . . 83

11 SolverSSA **85**
11.1 Overview . . . 85
11.2 Analyzers and simulation engines . . . 85
11.3 Statistical output analysis . . . 86
11.4 Architectural flow . . . 87
11.5 Implementation notes . . . 87
11.6 Feature and method support . . . 88
11.7 Method algorithms . . . 88
11.8 Bibliographic notes . . . 89

III Compositional solvers **90**

12 SolverENV **91**
12.1 Overview . . . 91
12.2 The blending method . . . 91
12.3 The state-vector method . . . 92
12.4 Architectural flow . . . 92
12.5 Implementation notes . . . 93
12.6 Feature and method support . . . 93
12.7 Method algorithms . . . 93
12.8 Bibliographic notes . . . 94

13 SolverLN **95**
13.1 Overview . . . 95
13.2 Layer construction . . . 95
13.2.1 Two layerings and two encodings . . . 96
13.3 The fixed-point iteration . . . 98
13.4 Architectural flow . . . 99
13.5 Implementation notes . . . 99
13.6 Interlock correction . . . 99
13.7 Feature and method support . . . 100
13.8 Method algorithms . . . 101
13.9 Bibliographic notes . . . 102

14 SolverUQ **103**
14.1 Overview . . . 103
14.2 Method . . . 103
14.3 Architectural flow . . . 104
14.4 Feature and method support . . . 104
14.5 Support-only uncertainty: interval-valued parameters . . . 105
14.6 Method algorithms . . . 105
14.7 Bibliographic notes . . . 106

IV External engines **107**

15 SolverJMT **108**
15.1 Overview . . . 108
15.2 Wrapper architecture . . . 108
15.3 Feature and method support . . . 109
15.4 Architectural flow . . . 109
15.5 Bibliographic notes . . . 109

16 SolverLQNS **111**
16.1 Overview . . . 111
16.2 Wrapper architecture . . . 111
16.3 Feature and method support . . . 112
16.4 Architectural flow . . . 113
16.5 Bibliographic notes . . . 113

V Extending LINE **114**

17 Registering a new solver or method **115**
17.1 Scope . . . 115
17.2 The solver contract . . . 116
17.3 Walkthrough: adding a method to an existing solver . . . 117
17.4 Walkthrough: adding a new solver . . . 118
17.5 Verification obligations . . . 120
17.6 Documentation obligations . . . 121
17.7 Pitfalls . . . 121

VI Reference **123**

A The NetworkStruct reference **124**
A.1 Nodes, stations, and stateful nodes . . . 124
A.2 Static properties . . . 125

A.3 Computed properties 125
A.4 Node parameters 125

B The LayeredNetworkStruct reference **126**
B.1 Representation of the model structure 126
B.2 Decomposition into layers 126

C API Function Reference **128**

Chapter 1

Introduction

1.1 Purpose of this manual

LINE is an open-source software package to analyze queueing models via analytical methods and simulation [30]. The user manuals of the tool (`LINE-user-matlab.pdf`, `LINE-user-java.pdf`, `LINE-user-python.pdf`, `LINE-user-cpp.pdf`) document the modeling language, the solver invocation interfaces, and the model features supported by each solver. They deliberately treat each solver as a black box: the user selects a solver and a method, and the tool returns performance metrics.

This manual takes the complementary viewpoint. It is oriented to researchers who wish to understand, evaluate, extend, or cite the solution algorithms implemented inside LINE. Each chapter maps one-to-one to a solver and presents:

- the solution paradigm the solver encodes, with its theoretical foundations;
- the internal software architecture, i.e., the solver class, its analyzers, its method handlers, and the numerical algorithms in the API layer that they invoke;
- the architectural flow of a solution request, illustrated by UML sequence diagrams;
- the concrete algorithms behind each value of `options.method`, with references to the scientific literature in which they were introduced or analyzed.

Readers are assumed to be familiar with queueing network theory at the level of standard textbooks [20, 114, 118] and with the modeling abstractions of LINE as described in the user manuals. Whenever this manual mentions user-facing behavior, such as option names or supported model features, the user manuals remain the authoritative reference; conversely, for the internal behavior of the algorithms, this manual supersedes the brief descriptions given there.

1.2 Relation to the existing manuals

The user manuals are organized by modeling concern: network specification, solver invocation, and output interpretation. This manual is organized by solver, mirroring the source tree. The two documents share

terminology:

- A *solver* encodes a general solution paradigm, e.g., mean-value analysis or continuous-time Markov chain analysis. Solvers are exposed to users as classes named `Solver[Name]`.
- An *analyzer* is an internal component that transforms the model representation and dispatches it to a concrete solution method. Analyzers are selected automatically based on the model class (e.g., a network containing a cache node routes to a cache analyzer).
- A *method* is a concrete algorithm selected via `options.method`, e.g., `exact`, `amva`, or `comom`. Methods are implemented in *handler* classes and often delegate the numerical core to *API algorithms*.
- The *API layer* collects reusable numerical functions organized by domain: `PFQN` (product-form queueing networks), `NPFQN` (non-product-form approximations), `MC` (Markov chains), `MAM` (matrix-analytic methods), `MAP` (Markovian arrival processes), `CACHE`, `QSYS` (single queueing systems), `POLLING`, `SN` (network structure transformations), and others.

LINE is a multi-language project. The canonical implementation is the Java JAR (`jar/` subfolder), which is also the backend called by MATLAB; MATLAB (`matlab/`) is the reference implementation used as ground truth in parity testing; a native Python implementation (`python/`) mirrors both without any JVM dependency; and a header-only C++ port (`cpp/`) carries the same algorithms in templated form, addressable from its `line-cli` front end. The MATLAB, JAR, and Python implementations follow the same object-oriented architecture and naming conventions: for example, the MATLAB function `solver_mva_analyzer.m` corresponds to the Java class `Solver_mva_analyzer` and to the Python module `solver_mva_analyzer.py`. In those editions, Java class names provide the canonical identifiers used in language-independent discussion; language-specific blocks give the concrete names of the selected codebase.

1.3 The common solution pipeline

All network solvers share the same solution pipeline, inherited from the abstract class `NetworkSolver`. A user constructs a `Network` model, instantiates a solver over it, and requests metrics, typically through `getAvgTable` or one of the finer-grained handles (`getAvg`, `getAvgQLen`, `getAvgRespT`, `getTranAvg`, and so forth). The request triggers the following steps:

1. *Structural compilation.* The object-oriented model is compiled by `Network.getStruct()` into a `NetworkStruct` instance, denoted `sn` throughout this manual. This is a flat, matrix-oriented description of the extended multiclass queueing network: stations, classes, chains, routing tables, phase-type service representations $(\boldsymbol{D}_0, \boldsymbol{D}_1)$, scheduling identifiers, capacities, and the auxiliary parameter blocks documented in the user manuals. All analyzers operate exclusively on `sn`, never on the object model; this is the model-to-model transformation boundary that decouples specification from solution.
2. *Feature admission.* The solver checks the model features present in `sn` against its feature set (`getFeatureSet`); unsupported models are rejected before any computation.

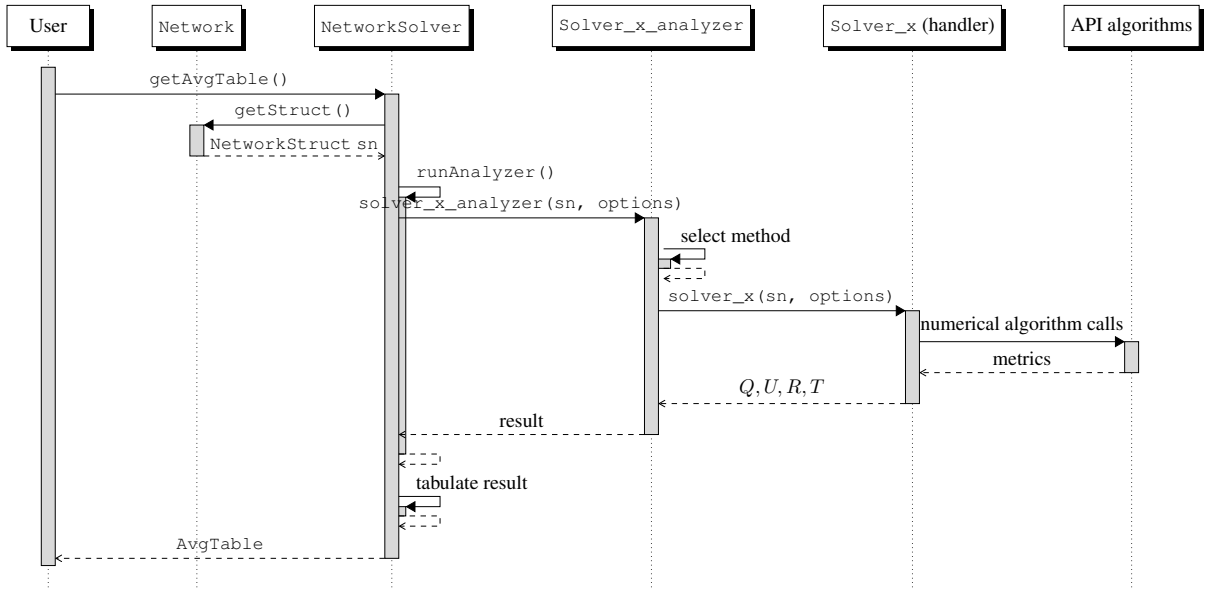


Figure 1.1: Generic solution pipeline shared by all LINE network solvers. The placeholder `x` stands for the solver name (`mva`, `nc`, `ctmc`, ...).

3. *Analyzer dispatch.* The solver method `runAnalyzer()` selects an analyzer based on model characteristics: the presence of cache nodes, polling stations, load dependence, an isolated open queue, or bound-computation requests each route to a specialized analyzer, otherwise the default analyzer runs.

4. *Method dispatch.* The analyzer normalizes `options.method`, applies default-selection policies, and invokes the handler implementing the chosen algorithm.

5. *Numerical solution.* The handler evaluates the model by calls into the API layer and returns station-class matrices of mean queue lengths Q_{ir}, utilizations U_{ir}, response times R_{ir}, throughputs T_{ir}, arrival rates, and residence times.

6. *Tabulation.* The solver stores the result and materializes it into the requested table object (e.g., `NetworkAvgTable`), applying chain aggregation and visit scaling as needed.

Figure 1.1 depicts this pipeline as a UML sequence diagram; every chapter of this manual contains an analogous diagram specialized to the internal components of the corresponding solver.

The compositional solvers (`AUTO`, `ENV`, `LN`, `UQ`) deviate from this pipeline in that they construct an ensemble of network submodels and coordinate other solvers over it rather than evaluating one `sn` directly. They typically do so through the abstract class `EnsembleSolver`. LINE also includes wrappers around external tools (`JMT` [13], `LQNS` [67], the latter also driving `qnsolver` on a flat network), which replace steps 4–5 with a marshalling stage that exports `sn` to the input format of the external tool, executes it as a subprocess, and parses the results back. Since their solution algorithms are external to LINE, Part IV documents only the wrapper architecture: model marshalling, method and feature mapping, and result parsing. The external algorithms themselves remain documented in the user manuals and in the cited literature.

1.4 Notation

We use the index conventions of the `NetworkStruct` reference of Appendix A: i, j range over stations (M stations), r, s over classes (R classes), c over chains, and k over phases. The population vector of a closed model is $\boldsymbol{N} = (N_1, \ldots, N_R)$ with total population N; service rates are μ_{ir}, mean service demands $D_{ir} = V_{ir}/\mu_{ir}$ where V_{ir} is the visit ratio; c_i denotes the number of servers of station i; and Z_r is the class-r think time, i.e., the aggregate demand at delay (infinite-server) stations. We write $\mathbf{1}_r$ for the r-th unit vector, so that $\boldsymbol{N} - \mathbf{1}_r$ is the population with one class-r job removed, and $\boldsymbol{n}$ for a generic population state with $\mathbf{0} \leq \boldsymbol{n} \leq \boldsymbol{N}$, whose components are n_r and whose restriction to station i is $\boldsymbol{n}_i$ (so $\sum_i \boldsymbol{n}_i = \boldsymbol{n}$). Phase-type and Markovian arrival processes are written in $(\boldsymbol{D}_0, \boldsymbol{D}_1)$ notation [25, 140]. For a CTMC we write $\boldsymbol{Q}$ for the infinitesimal generator and $\boldsymbol{\pi}$ for its stationary probability vector, $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$, and $\mathbf{0}$, $\mathbf{1}$ for the all-zeros and all-ones vectors.

Mean performance metrics follow the tool-wide convention: `QLen` Q_{ir} (mean number of class-r jobs at station i), `Util` U_{ir} (utilization, in $[0, 1]$ for single-server stations and reported as mean busy servers otherwise), `RespT` R_{ir} (response time per visit at the station), `ResidT` (residence time scaled by visits relative to the reference station), `ArvR` (arrival rate), and `Tput` T_{ir} (departure rate). These satisfy Little's law $Q_{ir} = T_{ir} R_{ir}$ at each station [123] and the utilization law $U_{ir} = T_{ir} D_{ir}/c_i$. We write $Q_i = \sum_r Q_{ir}$ for the station-aggregate mean queue length, and c^2 (with the appropriate station, class, and arrival/service subscripts, e.g., c^2_{ir}) for a squared coefficient of variation (SCV).

1.5 Organization

Part I covers the solver interface, that is the selection of an algorithm for a model: `AUTO` (Chapter 3). Part II covers the analysis algorithms: `BA` (Chapter 4), `MVA` (Chapter 9), `NC` (Chapter 10), `CTMC` (Chapter 5), `SSA` (Chapter 11), `FLD` (Chapter 6), `MAM` (Chapter 8), and `LDES` (Chapter 7). Part III covers the compositional solvers, which coordinate other solvers over an ensemble of submodels: `ENV` (Chapter 12), `LN` (Chapter 13), and `UQ` (Chapter 14). Part IV covers the external engines: `JMT` (Chapter 15) and `LQNS` (Chapter 16, including its `qnsolver` path for flat networks). The final part explains how to register a new solver or method (Chapter 17). The reference part collects the field-by-field description of `sn` (Appendix A) and the codebase-specific API reference.

Each chapter closes with bibliographic notes that position the implemented algorithms in the literature. To cite the LINE solver itself, we recommend referencing [30].

1.6 Use of AI assistance

During the development of LINE, the authors used AI-assisted coding tools based on Anthropic Claude models for implementation support, cross-codebase porting, test authoring, refactoring, and documentation drafting. The authors directed this work and validated it mainly through automated testing: cross-language consistency checks, comparison against simulation and external tools, and, where the corresponding papers report them, the published numerical examples. Given the volume of AI-generated code, not every change has been individually inspected by a human, and the extent of validation varies across features and codebases. The methods and their cited sources remain the work of their respective authors; AI assistance

concerned their realization in software and documentation, not their derivation. AI systems are neither authors nor copyright holders of any part of LINE. The last stable release developed without such assistance is version 2.0.39.

Chapter 2

The R binding

This chapter appears only in the R edition of this manual. The other four editions *are* the implementation it describes calling.

Every other chapter here documents an algorithm that LINE implements four times over. This one documents the machinery that lets R reach those implementations without owning any of them, because that is what the R edition has instead of internals of its own. Read it first: it explains why the remaining chapters describe C++ and not R.

2.1 What R owns and what it does not

R owns	the model description, the `line-model` document, the argument vector, the presentation of results
R does not own	visit ratios, chains, the station partition, moment fitting, any solver, or model validation

Table 2.1: The division of labour in the R edition

The second row is a design rule and not an accident of effort. Each of those quantities already has four implementations that must agree; a fifth, written in R, would be a fifth thing to keep in step and a fifth place for a divergence to originate. So the R package computes none of them. Where it needs one, it asks the engine: `model_index()` returns the node and class lists, the station partition and the two index maps, rather than deriving "which node is a station" in R.

2.2 The C application binary interface

The boundary is `r/capi/include/line/capi/line_capi.h`, a C99 header with nine functions. It is C and not C++ throughout: no default arguments, no `bool`, no namespace, no overload. R compiles `src/` with the C driver, as does every other foreign-function interface that will ever reach this library, so a header that merely happens to parse as C++ is a header none of them can use. `r/tests/capi_smoke.c` compiles it with `-std=c99 -Wall -Wextra -Werror` for exactly this reason: the C++ test suite cannot catch a C++-ism here, because to it there is nothing to catch.

2.2.1 The surface is the argument vector

```
int line_capi_run(int argc, const char *const *argv,
                  const char *stdin_text, unsigned flags, char **out_json);
```

`line_capi_run` takes `line-cli`'s own argument vector and runs it in process. The whole vocabulary is therefore reachable – every solver, every `-a` analysis, every input format, `--api`, `--find-solver` – because *the vocabulary is the argument vector* and this carries it verbatim.

The alternative was a structure mirroring the flags. `parse_args` accepts about a hundred and twenty of them and gains more with every solver, so that structure would have been a second parser to keep in step, and every host binding would have had to learn it. Passing argv means a host that can build a command line reaches everything the command line reaches, on the day a flag lands rather than on the day someone widens a structure.

The other eight functions are `abi_version`, `free`, `api_invoke`, `api_list`, `version_json`, `model_index`, `validate` and `set_interrupt`.

2.2.2 One document shape

Every call returns the same envelope:

```
{ "abi": 1, "status": 0,
  "documents": [ "{\"avg\":{...}}" ],
  "text": "SolverMVA arith=double method=mm1 ...",
  "diagnostics": "",
  "error": null }
```

`documents` holds the emitted JSON **as strings, not as parsed values**. The eleven emission sites in the CLI use three different indents and print their numbers at full precision so that a cross-codebase diff is not capped; parsing here and re-dumping in the host would reformat both, and the reformatting would stay invisible until someone diffed a golden. Storing the bytes the CLI printed is also what makes the acceptance test possible: each document must appear *verbatim* in the binary's own output.

`documents` is partial on failure by design. An `-a avg,sens` whose `sens` arm throws has already produced and printed the `avg` envelope, and discarding it would lose an answer the equivalent command line gives. `status` is the authority on whether the run as a whole succeeded.

2.2.3 Errors carry identity, not prose

A failure arrives with the stable identifier of `line/io/marshal.h` – `line:input`, `line:numeric`, `line:unsupported`, `line:error`, plus `line:internal` for a `std::exception` that `error_id` cannot classify and `line:interrupt` for a host-requested abort. The R side raises a condition whose class carries that identifier, so a caller can select on the *kind* of refusal rather than on the wording of a message. Those identifiers existed before this binding and had no consumer: `line_cli.cpp` did not include `marshal.h` at all.

2.3 The seam inside `cpp/`

Serving the whole vocabulary in process needed a reusable entry point, because the CLI's dispatch existed only behind `main()`.

- `cpp/include/line/cli/cli_run.h` declares `line::cli::run(Request) → Response` and `main_body`.
- `cpp/src/cli/line_cli_main.cpp` is the binary's ten-line `main`. **Its own file and not an `#ifdef`**, because an object file is indivisible: a static library carrying `main` would drag that symbol into the shared object R loads beside the R interpreter's own `main`. A guard macro would instead have meant compiling that translation unit twice, and it is the most expensive one in the project.
- `cpp/CMakeLists.txt` builds `line_cli_core`, a static library of `line_cli.cpp`, which both the binary and the ABI link. The translation unit is still compiled exactly once.

The acceptance test for the whole change is that `line-cli` is byte-identical afterwards, and `line-test.git/cpp/tests/test_cli_run.cpp` asserts it: for a dozen argument vectors spanning the vocabulary, every collected document must appear verbatim in the binary's standard output and the captured text must equal that output exactly.

2.3.1 Capture is at the file descriptor, and is not optional

Sixty-two sites print a solver banner unconditionally, even under `-o json -v silent`, so requiring `-o json` would not have been enough. More importantly, a library that writes to the process's standard output does not reach R's console on Windows or inside RStudio, and doing so is an outright CRAN policy violation. So `run` redirects file descriptors 1 and 2 for the duration of the call and returns the text as data.

The mechanism already existed: `run_invocation_captured` did the `mkstemp`/`dup`/`dup2` dance for server mode. The change factored it into an RAII `FdCapture` rather than inventing one. A temporary file and not a pipe, because a pipe holds 64 KB and nobody is draining it, so a solve that printed more would deadlock against its own output.

2.3.2 Four reentrancy defects, fixed rather than worked around

`g_jsim_input`	set once, never reset: a later `.json` model took the JSIM reader
`g_pnml_input`	the same
`g_verbosity`	sticky, so one `-v silent` call quietened the next
stdin buffer	a static local drained once and cached forever, so a second piped model solved the first

One of these was live in shipped code rather than hypothetical: the solve server serves one request per connection on the same process, so a request carrying `-i jsimg` poisoned every later request on that server. `reset_invocation_state` covers all four and `run_invocation_captured` now calls it too. Each has a test that fails on the tree as it stood before the seam existed.

2.3.3 Two gaps closed on the way

`--find-solver` ignored `-o json` entirely, so the structured candidate rows – with the *reason* each solver and method was refused – could not cross any boundary and every caller had to parse a fixed-width table. And `is_layered_json` took a *path*, so a `LayeredNetwork` piped on standard input was handed to the `Network` reader and refused with a message about the solver. Both are fixed for all five front ends, not only for R.

2.3.4 Reentrancy is bounded and stated

`line_capi_run` and its three siblings serialize on one process-wide mutex. Three separate things force it and none is cheap to remove: the file-scope state in `line_cli.cpp`, the `LineConsole` session singleton, and the fact that capturing output means redirecting process-global file descriptors. A finer lock would protect one of the three and leave the other two. R calls into `.Call` from one thread, so an R host pays nothing for it.

2.3.5 The interrupt is partial, and says so

`line_capi_set_interrupt` installs a callback polled between the analyses of a comma-separated `-a` list and at each document emission. It does **not** reach inside a single long solve: `-s ssa --samples 1e9` and a stiff fluid integration are in engine loops that take no callback, and threading one through them is an engine-wide change this seam has no business making alone.

This is stated in the header, in the manual and in the package documentation because a half-working interrupt that is advertised as working is worse than none. R's own `R_CheckUserInterrupt` unwinds by `longjmp`, which cannot cross C++ frames without skipping their destructors and leaking the capture's descriptors and temporary files, so the callback is the only safe shape.

2.4 The document layer

The R half that needs no engine is the `line-model` reader and writer. Two things about it are load-bearing.

Numbers are emitted by hand. The wire convention is the shortest round-tripping decimal, the same one Python's `repr` and the C++ writer's Grisu produce. `jsonlite::toJSON(digits = NA)` is documented as maximum precision and is in fact fifteen significant digits, which turns $1/3$ into a different double and $0.1 + 0.2$ into 0.3. It is exact for round numbers, which is why it passes any test written with them; it was caught only by round-tripping a document whose values were not round, and the regression test is written with such a value for that reason.

Enumerations cross as names. The numeric values differ across the codebases, so the R value *is* the wire string and there is no lookup table that can drift. A mistyped member is refused rather than returning `NULL`, because a bare list lookup would build a station with no scheduling key at all and the model would solve as something else with nothing in the output naming the cause.

2.5 Assembling results

The payloads are column-oriented, which is what an R data frame already is, but three things cannot be done the obvious way.

Column types must not depend on the data. `jsonlite`'s simplification is data-dependent: an all-integer column becomes an integer vector, the same column with one `null` becomes a list, a one-element column becomes a scalar. The reader therefore parses with `simplifyVector = FALSE` and assembles every column explicitly.

The table is not a complete grid. `emit_avg_table_named` drops any (station, class) row whose six metrics are all exactly zero, so the table's dimensions do not imply the model's and a reshape would silently produce the wrong shape. Matrices are allocated from the model index and scattered by name.

`NA` and zero are different answers. A `null` in a result column means the solver did not compute that metric, which some solvers legitimately do; it stays `NA`. The matrix path fills a missing row with zero instead, because there the missing row *was* zero. The asymmetry looks like a defect and is not, so it is commented at both sites.

2.6 What the suite measures

`R-JSON`	round-trips every `line-model` document in the checkout through the R reader and writer and compares *documents*. Needs no engine, and is the row that would actually catch an assembler defect.
`R2C`	builds a model in R, solves it through the engine, and compares against the shared golden with the project's own tolerance policy.

A green `R2C` is not corroboration of the `CPP` row. R implements no solver, so the row measures the R assembler and then re-asserts the engine's own numbers. Saying so beside the row is part of the row: without it, two green rows would look like two independent confirmations of one set of numbers.

The comparator is a port of the shared one, including `golden_slack` – half a unit in the last digit a golden actually carries. That is not a tolerance the R suite chose; goldens are written from printed tables at five significant digits, so a golden recording `2.222` for a value of `2.2220273` cannot be matched at a 10^{-6} relative gate by *any* engine, and every other consumer applies the same allowance.

An earlier version of `R2C` carried an escape hatch that excused a cell when the engine missed the golden too. It was removed: a wrong port reproduces itself perfectly, so the hatch turned a mis-ported example into a silently accepted one. A cell that misses the golden is a finding, and the ported examples are checked against their reference documents at *zero* tolerance separately.

Part I

The solver interface

Chapter 3

SolverAUTO

3.1 Overview

`SolverAUTO` is the automatic solver-selection meta-solver: given a model, it chooses which concrete solver and method to run. Solver selection is a nontrivial decision problem because the accuracy and cost of each paradigm depend on structural model features (product-form compliance, population size, distribution classes, scheduling disciplines) in ways that are only partially characterized analytically; `AUTO` encodes this decision as an explicit, inspectable policy. The class resides in `jline/solvers/auto/`, together with the `LINE` convenience facade and a `ModelAnalyzer` component that extracts decision features from `sn`.

3.2 Selection policy

The default policy is rule-based. `ModelAnalyzer` computes a feature vector including: openness of the workload (open, closed, mixed), number of stations, classes, and jobs, presence of non-exponential distributions and their SCVs, scheduling disciplines, load dependence, and topology properties (tandem, cyclic, fork-join, caches, Petri net elements). The heuristic then proceeds by elimination and preference:

1. candidate solvers are instantiated (`SolverMVA`, `SolverNC`, `SolverMAM`, `SolverFLD`, and the simulators) and filtered by `supports(model)`, i.e., feature-set admission;

2. among the admissible candidates, analytical solvers are ranked by expected accuracy on the detected structure: exact paradigms first when their cost bound is acceptable (exact MVA or NC for small product-form models), then the approximation whose known error profile best matches the model (AMVA for large closed product-form networks, MAM for autocorrelated open traffic, FLD for very large populations);

3. simulation (`LDES`, `SSA`) is the fallback when no analytical candidate admits the model or when the request targets metrics unavailable analytically. `JMT` is deliberately not a candidate: `LDES` subsumes its feature set, so automatic selection has nothing to gain from dispatching to the external simulator; `JMT` stays reachable through the explicit per-solver delegation `options.method='jmt'`, documented in the user manual's `AUTO` section.

Feature-set admission is necessary but not sufficient for the `CTMC` candidate: a model can declare only supported features yet still generate a state space too large to enumerate. `SolverCTMC.isStateSpaceTractable(model, options)` runs, on the phase-type-converted `sn` and without generating the state space, the same size estimator and memory gate the analyzer itself runs (§5); a refused `CTMC` candidate is dropped from ranking and selection continues with the next-best analytical or simulation candidate, e.g. `getTranProb` on a large multi-station Erlang model falls through to `FLD` rather than proposing a chain that cannot be built. This tractability screen applies only to ranked selection: the forced method name `SolverAUTO(model,'ctmc')`, and the three generator accessors below, are not screened and must reach `CTMC` and fail with its own diagnostic if the state space is intractable.

The choice is also sensitive to the invoked handle: `getAvg` requests may be answered by a different solver than `getTranAvg` or `getCdfRespT` on the same model, since transient and distributional support differs across solvers. The selected solver's name is recorded in the result for reproducibility.

`getStateSpace`, `getGenerator` and `getSymbolicGenerator` are `CTMC`-only concepts with no ranked-selection equivalent, so `AUTO` exposes them under the same names and resolves them directly to the `CTMC` candidate (`ctmcSolver()` in MATLAB and the JAR, `_ctmc_solver()` in Python), building one on demand if it is not already in the candidate pool. They bypass `delegate` and the ranked tables entirely, so they are exempt from the tractability screen described above.

3.3 Architectural flow

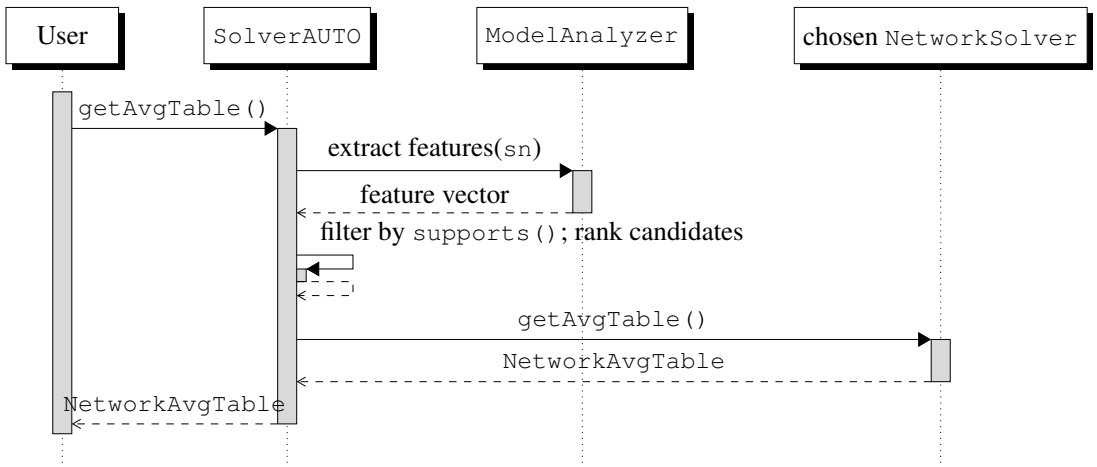


Figure 3.1: Sequence diagram of `SolverAUTO`: selection followed by delegation.

Figure 3.1 shows the delegation structure: `AUTO` performs no numerical work itself and inherits the full result object, including solver-specific diagnostics, from the delegate.

3.4 Feature and method support

`SolverAUTO` performs no numerical work of its own: the admissible features are the union of those of the candidate solvers (Chapters 9–7), and support for any given model reduces to that of the delegate it selects. What is specific to `AUTO` is the mapping from detected model structure to the selected solver, summarized in Table 3.1; the exact per-method support then follows from the corresponding chapter table.

Table 3.1: Structure-to-solver selection policy of `SolverAUTO` (default handle `getAvg`).

Detected model structure	Selected solver (method)
Small product-form closed/mixed	`NC`/`MVA` (exact)
Large closed product-form	`MVA` (amva)
Very large populations	`FLD` (closing)
Open with autocorrelated / MAP traffic	`MAM` (dec.mmap)
Open G/G/k decomposable	`MVA` (qna)
Cache network	`MVA`/`NC` (cache analyzer)
Small non-product-form state space	`CTMC`
Stochastic Petri net	`CTMC` / `LDES`
Layered network (LQN)	`LN`
No admissible analytical solver	`LDES` / `SSA` (simulation; `JMT` not a candidate)
Transient / response-time distribution	`FLD` / `CTMC` / `MAM`

3.5 Method algorithms

Table 3.2: High-level pseudocode of `SolverAUTO`.

Method	Pseudocode
heuristic (default)	1. extract the feature vector from `sn` (`ModelAnalyzer`): openness, sizes, distributions/SCVs, disciplines, load dependence, topology 2. instantiate candidate solvers and filter by `supports(model)` 3. rank the admissible candidates by expected accuracy vs. cost for the detected structure and the invoked handle (`getAvg`/`getTranAvg`/`getCdfRespT`) 4. delegate to the top solver; record its name in the result

3.6 Bibliographic notes

The layered solver taxonomy that makes automated selection well-posed in LINE is presented in [30]. Method-selection trade-offs among exact, asymptotic, and iterative product-form algorithms are analyzed in [29, 33]; the respective domains of decomposition and simulation methods are discussed in the chapters of Part II.

Part II

Analysis algorithms

Chapter 4

SolverBA

4.1 Overview

`SolverBA` encodes the bounding paradigm for closed queueing networks. Where `SolverMVA` and `SolverNC` return a point estimate, a bounding method returns one guaranteed side of an interval that contains the exact solution, so the `.lower` and `.upper` members of a family bracket the truth. The guarantee, rather than the accuracy, is what the solver sells: an interval of known sidedness composes safely inside an optimization loop or an admission test, whereas an approximation of unknown sign does not [29].

Two properties shape the design. First, every bound in the solver is a function of the service demands $D_{ir} = V_{ir}/\mu_{ir}$, the think times Z_r and the populations $\boldsymbol{N}$ alone; none reads a service distribution beyond its mean. The feature set is therefore deliberately narrow, admitting closed product-form-parameterized models – with one exception, §4.7, which trades the closed-model restriction for a Markovian-service one – and the solver is cheap enough to be called thousands of times. Second, the families form a partial order of computational effort against tightness, from closed-form asymptotic bounds evaluated in $O(MR)$ to linear programs over a polytope whose dimension grows with the population.

The solver resides in `jline/solvers/ba/`, mirrored by `matlab/src/solvers/BA/`, `python/line_solver/solvers/solver_ba/` and `cpp/include/line/solvers/ba/`. The numerical algorithms live in the `PFQN` API (functions prefixed `pfqn_`) and, for the reduction bounds, in the `MAPQN` API. The bound methods were previously hosted by `SolverMVA`; they now dispatch here, and the corresponding `SolverMVA` method names raise a redirect.

4.2 Software architecture

Table 4.1 lists the main components. `SolverBA.runAnalyzer()` normalizes the method name, resolves the aliases `default`, `lr` and `qr`, and routes to one of two analyzers: the native bound analyzer for the classical families and the hierarchies, and the QRF analyzer for the reduction bounds.

Each analyzer branch computes a throughput bound $X^{\pm}$ and then reconstructs the remaining metrics from it. This is the role of `ba_fill`: given a bounded chain throughput it applies the forced-flow law $T_i = XV_i$ and Little's law to obtain the companion queue-length and residence-time entries, so that the solver returns a fully populated result on the requested side rather than a single scalar. The reconstruction

Component	Class	Role
Solver	`SolverBA`	feature admission, alias resolution, analyzer dispatch
Options	`SolverOptions`	method, hierarchy depth `level` (default 2)
Analyzer	`solver_ba_analyzer`	asymptotic families, hierarchies, LP linear reduction
Analyzer	`solver_ba_qrf_analyzer`	QRF reduction bounds
Helper	`ba_sc_demands`	single-chain demand, visit and think-time extraction
Helper	`ba_fill`	reconstruction of Q, U, R, T, C from a bounded throughput

Table 4.1: Main components of `SolverBA`.

is consistent by construction but is not itself a bound on every metric, a point to keep in mind when reading the queue-length columns of a one-sided result.

4.3 Asymptotic families

The classical families derive from operational laws and require no iteration. Let $D_{\max} = \max_i D_i$, $D_{\text{sum}} = \sum_i D_i$ and Z be the think time of a single-class model with population N.

`aba` Asymptotic bound analysis [57, 118, 137]. The throughput cannot exceed the bottleneck capacity nor the light-load asymptote, giving

$$X^+ = \min\left(\frac{1}{D_{\max}}, \frac{N}{D_{\text{sum}} + Z}\right), \qquad X^- = \frac{N}{N D_{\text{sum}} + Z}, \tag{4.1}$$

the lower bound following from the worst case in which every job queues behind all the others. These are the crudest bounds in the solver and the cheapest.

`bjb` Balanced job bounds [118, 190]. Replacing the demand vector by a balanced one with the same total sharpens (4.1), because a balanced network maximizes throughput among all networks of equal total demand. The bounds are exact when the demands are already balanced.

`pb` Proportional bounds [95], which interpolate between the asymptotic and balanced cases using the demand distribution rather than only its extremes.

`gb` Geometric bounds [34], obtained from a geometric approximation of the queue-length growth with population, with error guarantees on throughput. The throughput bounds are `pfqn_xzgsbup` and `pfqn_xzgsblow`, and the companion queue-length bounds are `pfqn_qzgbup` and `pfqn_qzgblow`.

`sb` Simple bounds, evaluated in closed form from the power sums $A_k = \sum_i D_i^k$ of the demand vector following Harel, Namn and Sturm [86]. Unlike the other asymptotic families this one is defined only for delay-free models, and the analyzer rejects a model containing an infinite-server station.

`harel` The sharp bounds of the same paper [86], distinct from `sb`: rather than reading only the first three power sums, they extrapolate from the *exact* normalizing constant evaluated at the populations $n \le 7$, so the pair is tight where the population is small and degrades gracefully above the extrapolation range. Like `sb` they are defined for delay-free single-class models with single-server stations, and the analyzer rejects think times and multiserver stations by name.

mw The robust box bounds of Majumdar and Woodside [126] for multiclass networks, valid under the weaker NBUE assumption on service times rather than product form, obtained by optimizing over a box of admissible demand vectors.

No family dominates the others on both sides simultaneously, so which is sharpest is model dependent; the solver therefore exposes them all rather than selecting one.

4.4 Bound hierarchies

A bound hierarchy is a sequence of bounds indexed by a depth parameter, monotonically tightening toward the exact solution and reaching it at a finite depth. The depth is the `level` option, default 2. Raising it trades computation for tightness, which lets the caller buy exactly as much accuracy as an application needs.

pbh The Eager-Sevcik performance bound hierarchy initialized from the PB bound [63, 64] (`pfqn_pbh`). The MVA recursion (9.1) is unrolled for `level` steps from the population of interest downward, and the unknown queue lengths at the truncation point are replaced by their extreme admissible values, which bounds the recursion from either side. The hierarchy is exact once `level` reaches N, since the recursion is then unrolled to the empty network, whose solution is known.

bjbh The Eager-Sevcik hierarchy initialized from the BJB bound [63, 64] (implemented through `pfqn_bjbk`).

cbh The convolutional bound hierarchy of Dowdy, Eager, Gordon and Saxton [60] (`pfqn_cbh`), which bounds the normalizing constant (10.1) rather than the MVA recursion: stations are convolved exactly up to `level` and the remaining factors are bounded above and below by geometric envelopes. It is exact when `level` reaches the number of stations.

ssd The multiserver disaggregation bounds of Suri and Dallery [174] (`pfqn_ssd`), which bound a multiserver station by the two single-server networks obtained by disaggregating its capacity.

sib The successively improving bounds of Srinivasan [172] (`pfqn_sib`), refined as `level` grows. The construction is stated for $Z = 0$, so the analyzer rejects models with a non-zero think time, which the delay extension of Section 3.2 of the original paper would be needed to cover.

cub, mbjb The composite bound method of Kerola [103] for multiclass models (`pfqn_mcub`). The composite construction yields an upper bound only, exposed as `cub.upper`; the companion lower bound is the multiclass balanced job bound of the same paper, exposed separately as `mbjb.lower` because it is a distinct construction rather than the mirror image of `cub`. Naming them as one family would wrongly suggest that `cub.lower` exists.

looping Eager's Looping algorithm [61] (`pfqn_looping`), the multiclass bracket that seeds the multiple-class performance bound hierarchy. Built on the convolution identity $Q_{jk}(\boldsymbol{N} - \mathbf{1}_c) = [X_j^{+k}(\boldsymbol{N} - \mathbf{1}_c)/X_j(\boldsymbol{N})]\, Q_{jk}(\boldsymbol{N})$, it iterates queue-length lower bounds together with a per-class *heap* of congestion the current bounds have not yet accounted for, charged back at the pessimistic inflation factor $\max_k D_{ck}$ or the optimistic one $\min_k D_{ck}$, the largest and smallest delays one customer can inflict. It

is a bounding algorithm rather than a point estimator, returning the pessimistic bracket and its optimistic counterpart as `looping.lower`/`looping.upper`.

`ldac` The load-dependent BCMP closed-open equivalence bound of Anselmi and Cremonesi [4] (`pfqn_ldbcmp`), a lower bound only, and applicable only in the asymptotic regime $N \geq \hat{Q}$; outside it the algorithm returns `NaN` and the analyzer raises an error rather than reporting an unfounded bound.

4.5 Reduction bounds

The reduction bounds come from the Quadratic Reduction Framework [31], which bounds performance measures of networks with phase-type or Markovian arrival process service by optimizing over a relaxation of the stationary distribution. The exact chain is described by joint probabilities $p_2[j, n_j, k, i, n_i, h]$ of the population and service phase at pairs of stations. Writing the balance and marginal consistency conditions that the true stationary distribution must satisfy yields a polytope $\mathcal{P}$ containing it; optimizing a performance functional over $\mathcal{P}$ therefore bounds that functional. The relaxation is what makes the method tractable, since $\mathcal{P}$ is described by pairwise rather than full joint distributions.

Two distinct kinds of program are built over $\mathcal{P}$, and the distinction matters because the method names do not make it obvious.

`lr` The linear reduction bound. The objective is the utilization functional $U_i = \sum_{n_i \geq 1, k} p_2[i, n_i, k, i, n_i, k]$, which is *linear* in the decision variables, so minimizing and maximizing it over $\mathcal{P}$ is a pure linear program, solved by `linprog`, HiGHS or a simplex implementation according to the codebase. Both senses are valid bounds because the relaxation contains the exact solution: `lr.lower` minimizes and `lr.upper` maximizes, and the bare name `lr` means `lr.upper`. The backing algorithm is `mapqn_bnd_lr_pf`. Being a linear program, this is also the numerically best-behaved member of the family: the polytope is degenerate, with an empty interior, and simplex methods handle that natively.

`qr`, `qrf.*` The quadratic reduction bounds, in which the objective is a nonlinear entropy or mutual-information functional over the same constraints, so the program is a nonlinear one solved by `fmincon`. The members are `qrf.mmi` (mutual information, aliased `qr`), `qrf.mem` (maximum entropy), `qrf.mmi.ld` (load dependent) and `qrf.mmi.linear`. The blocking variants `qrf.bas.*` and `qrf.rsrd`, which cover blocking-after-service and RS-RD networks, are linear programs again.

The name `qrf.mmi.linear` is a trap worth stating plainly: it is *not* a linear program and it is *not* an alias of `lr`. The word *linear* there refers only to the way the constraints are represented, as explicit A_{eq}, b_{eq} matrices rather than as callback functions, while the objective remains the nonlinear maximum-entropy functional. A reader who conflates the two will attribute the LP method's numerical robustness to a method that does not have it.

4.6 Single-class bounds

`scb` answers a different question from every family above: given the demand vector L and population N of a *single-class* model, it does not bound that model's own solution, which single-class MVA already gives

exactly, but instead bounds the unknown multiclass system that the single class aggregates [59]. The lower side, `scb.lower`$= X_1$, is simply the exact single-class throughput from the ordinary MVA recursion with $Z = 0$. The upper side inflates it by a ratio capped in two independent ways: the population-sharing cap $(N + m - 1)/N$ with $m = \min(N, K)$ stations of maximal demand (Theorem 3), and the one-server-per-station capacity cap $1/(X_1 \max_i L_i)$ (Corollary 1), so $X^+ = X_1 \min\big((N + m - 1)/N,\ 1/(X_1 \max_i L_i)\big)$. The gap between the two sides is at most 50% and only ever closes as $m \to N$.

Three model-free companion functions share the same paper and require no solved model:

`pfqn_scbgap`(N, K, r) the maximum relative throughput error from merging r of N classes into one, $e = (\min(r, K) - 1)/(r + \min(r, K) - 1)$ in general (Expression 4/3) or, restricted to $r \le K$, the tighter undominated bound $e = r(r - 1)/(\min(N, K)(2r - 1))$ (Theorem 5).

`pfqn_usumbound`(R, K, N) an upper bound on the sum of station utilizations, $H = \min(R, K)$, $\sum_k U_k \le (H - 1) + (K - H + 1)(N - H + 1)/(K + N - 2H + 1)$ (Theorem 6).

`pfqn_minclasses`(U_{sum}, K, N) the inverse of the above: the minimum R consistent with a measured utilization sum, or `NaN` if no R attains it.

Because `scb.lower` already equals the exact throughput, folding `scb` into the `auto.*` candidate set of §4.9 would make `auto.lower` collapse onto it whenever a single-class model qualifies; it is excluded from that set for this reason, not because it is inadmissible.

4.7 Achievable-region bounds

`bgt` and `bpt` are the only families in the solver that admit an *open* model, and the only ones whose service restriction is Markovian rather than product-form-parameterized: both require every station to be exponential, single-server, fed from a Source, with deterministic and non-merging per-class routing (no delay, no multiserver station, no probabilistic split of a class across destinations). Both certify a property of *every* work-conserving (`bgt`) or non-idling (`bpt`) policy, not of the particular scheduling disciplines configured on the model, which is why the restriction is on the arrival/routing structure rather than on the scheduling discipline.

`bgt.upper` [15] bounds the steady-state queue lengths of a multitype open Markovian network from above. It solves the Down-Meyn global-stability linear program in a piecewise-linear Lyapunov function $\Phi(x) = \max_j L^{j\top} x$: a feasible drift margin $\gamma > 0$ certifies stability, and Theorem 4 of the source then yields the constant

$$\mathbb{E}[L^{j\top} Q] \le U = \frac{16 N J^2 (J - 1)(L_{\max} + \gamma)^3}{\gamma^2} + \frac{8(L_{\max} + \gamma/2)^2}{\gamma},$$

N the number of stations and J the number of Lyapunov facets, from which the per-class bound $\mathbb{E}[Q_{i,k}] \le U / \max_j L^j_{i,k}$ follows. The bound is loose by construction: the certificate and the tail-decay rate are the sharp content, the constant is not, and it can sit one to several orders of magnitude above the exact value even on an M/M/1.

`bpt.lower` [16] bounds the mean sojourn times of a multiclass open Markovian network from below by a first-order linear relaxation of the achievable region: from the steady-state balance of $\mathbb{E}[R(t)^2]$ for the aggregate $R(t) = \sum_r f(r)n_r(t)$, it derives linear variables for the per-class mean sojourn times and the pairwise idling/queueing indicators, and minimizes a linear objective over the resulting polytope. It is exact on M/M/1. Because only externally-fed classes couple tightly to the second-moment balance equation, the bound is markedly weaker on classes that are only reached by internal routing – a property of the relaxation, not a defect to work around.

4.8 Stochastic network calculus bounds

`snc.upper` (`solver_ba_snc_analyzer`) bounds the mean response times and queue lengths of a feed-forward open network by the moment-generating-function envelope calculus of Fidler and Rizk [65], and the bound holds for *every* work-conserving policy at every station rather than for the disciplines the model happens to declare. The envelope algebra is the `snc` API family: arrival envelopes (`snc_env_poisson`, `snc_env_cpoisson`, `snc_env_map`, `snc_env_tokenbucket`), service envelopes (`snc_srv_rate`, `snc_srv_exp`), the min-plus operations (`snc_conv` for tandem concatenation, `snc_leftover` for blind multiplexing, `snc_output` for the departure envelope), the tail bounds (`snc_bound_delay`, `snc_bound_backlog`) with their quantile and mean readings (`snc_perc_delay`, `snc_perc_backlog`, `snc_mean_delay`, `snc_mean_backlog`), and the Chernoff optimizer `snc_thetaopt`. The analyzer maps the model onto that algebra, propagates envelopes hop by hop and reads the bound back per station and class.

Units are jobs, not work, which is what lets the departure envelope of one station be the arrival envelope of the next: a work unit differs from station to station, a job does not. R_{ir} is the integral of the delay tail bound at that station, so each entry is a valid upper bound on its own; Q follows by Little's law from the bounded R and the exact throughput, since an open network's per-class rates are fixed by the traffic equations rather than by the policy, and U is exact for the same reason. The method requires a fully open network with a Source, one server per station and no delay station, exponential service (the source may be any Markovian (D_0, D_1) process), an acyclic station graph so that cross traffic is always determined upstream, deterministic routing downstream of the Source, and equal service rates among the classes sharing a station, since the blind-multiplexing leftover subtracts job counts from a job-count capacity; a probabilistic split at a Poisson Source is exact and allowed, one of an already-queued flow is refused rather than bounded by the whole flow, which would report a false instability.

Being policy-robust, the bound is loose on the mean: $2.4\times$ the exact M/M/1 mean response time at $\rho = 0.1$ and $10.4\times$ at $\rho = 0.95$. Its sharp object is the tail, whose decay rate it reproduces exactly, $(\lambda/\mu)^n$ for the backlog and $e^{-(\mu-\lambda)d}$ for the delay on an M/M/1, so the quantile getters `getDelayPerc`, `getBacklogPerc` and `getPercTable` are the intended reading when the tail is what matters.

4.9 Architectural flow

Figure 4.1 shows the path taken by `getBounds`, which resolves the family prefix of the current method and invokes the analyzer once per side, forwarding the caller's option set so that `level` applies to both sides

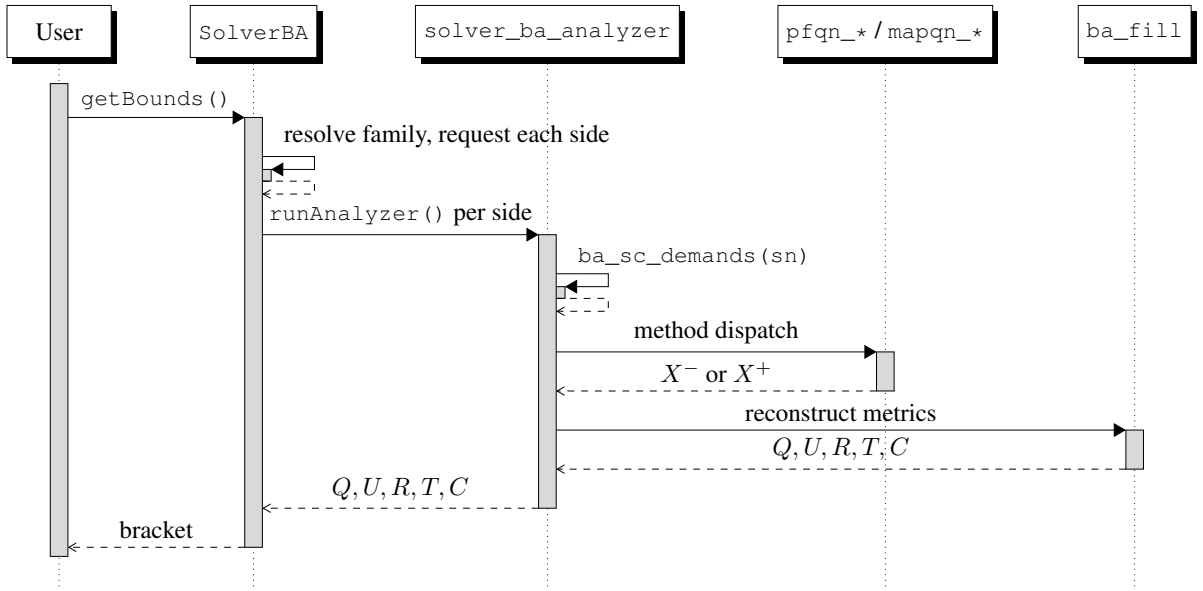


Figure 4.1: Sequence diagram of `SolverBA` for a two-sided bound request.

of the bracket. One-sided families are detected by checking the family name against `listValidMethods`, and the absent side is reported as `NaN` rather than being silently replaced by a finite value; this keeps the absence of a bound distinguishable from a bound equal to zero.

Two accessors sit on top of this path. `getBounds` returns the raw bracket as a structure of `Tlower`, `Tupper`, `Qlower` and `Qupper`, and is the programmatic form. `getBoundsTable` formats the same information per station and class in the layout of `getAvgTable`, reusing its labelling and its omission of disabled station-class pairs, and returns the table type native to each codebase: a `table` in MATLAB, a pandas `DataFrame` in Python, and a `NetworkBoundsTable` extending `AvgTable` in the JAR. The pair stands to each other exactly as `getAvg` does to `getAvgTable`.

4.10 Implementation notes

Bounds are computed on the single-chain projection of the model. `ba_sc_demands` aggregates the per-class demands and visits into the chain quantities that the algorithms expect, and raises an error when the model is genuinely multiclass and the requested family has no multiclass form. This is why `mw` and `cub`/`mbjb`, the two genuinely multiclass families, follow a separate extraction path.

Structural admissibility is checked per method rather than only through the feature set, because several families carry restrictions that the feature set cannot express. The geometric families and `sib` reject infinite-server stations, as do the alpha-free `qrf.*` reduction bounds, whose population-free transition rates give every station a single server so that a delay would silently be a different model rather than an approximation of the intended one. The two load-dependent arms are the exception: `qrf.mmi.ld` and `qrf.mmi.linear` carry a scaling $\alpha(i, n)$ that multiplies every rate out of station i at population n, which is exactly the rate

law of an infinite server ($\alpha = n$), of a c-server station ($\alpha = \min(n, c)$) and of limited load dependence, so those models are answered on their own chain rather than refused. The scaling is derived from the model by `sn_to_qrf_alpha`, with `options.config.qrf_alpha` kept as an override, and the one restriction that survives is that a station serving several jobs at once must have exponential service, since the local state carries a single phase per station. `lr` and `scb` additionally require a single-class closed model with single-server stations; `ldac` requires the asymptotic regime; `bgt`/`bpt` require an open model with a Source, single-server exponential stations only, and deterministic non-merging routing. In each case the analyzer raises an error rather than returning a value that would not be a bound. This is a deliberate choice: a bound that is silently invalid is worse than no bound, since the caller cannot detect the failure from the output.

The cross-language status is not uniform for the reduction bounds. The linear program methods are available in all four codebases. In MATLAB and native Python they agree with the reference model to 10^{-11}, and the C++ port solves the same programs with its exact-arithmetic-capable simplex; the JAR solves them with a sparse ADMM backend and attains roughly 10^{-5}, so JAR bound values are not interchangeable with the others at tight tolerances.

The nonlinear `qrf.*` methods are now available natively as well. They were withheld in native Python until the adapter path from a `Network` into the reduction library had been exercised end to end, on the grounds that the available nonlinear programming routines could not be used on a polytope whose empty interior makes the entropy objective singular at the optimum. Each ground was removed at its root rather than worked around: the redundant equality block is reduced to a maximal independent row set before it reaches the solver, and the program starts from a phase-one feasible point instead of the origin, which previously caused the solver to exit early and return the origin as though it were a solution. The tokens `qr`, `qrf.mmi`, `qrf.mem`, `qrf.mmi.ld` and `qrf.mmi.linear` are advertised in all four codebases, and the blocking variants have since followed: native Python serves `qrf.bas`, `qrf.rsrd`, `qrf.bas.mmi` and `qrf.bas.mem` through its HiGHS-backed `qr_bounds` path, and the C++ port carries the same set except `qrf.bas.mmi`.

Exercising that path is worth recording as a methodological point, because it found defects that no unit test on the reduction library itself could have found: the adapter transposed the background transition matrix relative to the completion matrix, which is invisible for a reversible process and for single-phase service but reverses the phase order of an Erlang; and it recovered throughput by inverting the queue length at the reference station, which presumes that station is an infinite server. Both produced plausible output rather than an error. The utilization is now read directly off the reduction variables, which is legitimate because that quantity is the probability that the station is non-empty, marginalized over phase, and is therefore bounded by one through the normalization constraint; the throughput follows from any finite-server station, and the resulting value agrees across stations to machine precision.

A caution on the oracle used to check such work: a product-form solver returns the exponential answer whatever the phase-type distribution, so it cannot adjudicate a phase-type model even though it runs without complaint. The reference must be a Markov chain solution or the linear program itself.

4.11 Feature and method support

Table 4.2 maps modelling features to the `SolverBA` method groups. All groups require closed classes: bounds are defined on the population recursion and have no meaning for a purely open model. The `asym`

column covers `aba`, `bjb`, `pb`, `gb` and `sb`; `hier` covers `pbh`, `bjbh`, `cbh`, `ssd` and `sib`; `mcl` covers the multiclass families `mw`, `cub` and `mbjb`; `lr` is the linear program; `qrf` covers the nonlinear reduction bounds. Service distributions beyond the mean are marked ○ because the classical bounds read only the mean demand and remain valid, though not tight, for any distribution with that mean, whereas the reduction bounds exploit the phase-type structure explicitly.

Table 4.2: Feature support of `SolverBA` method groups (● supported, ○ approximate/-conditional, blank unsupported).

Feature	asym	hier	mcl	lr	qrf
Job classes					
Closed class	●	●	●	●	●
Open class					
Multiclass	○	○	●		
Class switching	○	○	○		
Node types					
Queueing station	●	●	●	●	●
Delay (infinite server)	○	○	●		
Service distributions					
Exponential	●	●	●	●	●
Phase-type	○	○	○	○	●
General (mean only)	○	○	○		
Scheduling disciplines					
INF (delay)	○	○	●		
PS, FCFS, LCFS-PR	●	●	●	●	●
Advanced features					
Multiserver stations		○			
Load dependence					○
Blocking (BAS, RS-RD)					○
Analysis and output					
Throughput bound	●	●	●	●	●
Utilization bound	●	●	●	●	●
Two-sided bracket	●	●	○	●	○
Depth control (`level`)		●			

4.12 Method algorithms

Table 4.3 gives high-level pseudocode. All methods return a throughput bound from which `ba_fill` reconstructs the remaining metrics.

Table 4.3: High-level pseudocode of `SolverBA` methods.

Method	Pseudocode
`aba`	compute $D_{\max}$, D_{sum}, Z; return $X^{+} = \min(1/D_{\max}, N/(D_{\text{sum}} + Z))$ and $X^{-} = N/(ND_{\text{sum}} + Z)$; no iteration
`bjb`	replace the demand vector by the balanced one of equal total; evaluate the asymptotic formulas on it; exact when demands are already balanced

continued on next page

Method	Pseudocode
`pb`	interpolate between the asymptotic and balanced cases using the demand distribution
`gb`	fit a geometric envelope to the queue-length growth in N; evaluate the resulting closed-form throughput and queue-length bounds with error guarantees
`sb`	form the power sums $A_k = \sum_i D_i^k$ of the demand vector; evaluate the closed-form simple bounds; delay-free models only
`mw`	optimize the multiclass throughput over a box of demand vectors admissible under the NBUE assumption
`pbh`	1. unroll the MVA recursion `level` steps downward from $\boldsymbol{N}$ 2. replace the queue lengths at the truncation point by their extreme admissible values 3. propagate back up, giving X^- and X^+; exact at `level` $= N$
`bjbh`	run the Eager-Sevcik hierarchy from the balanced-job-bound seed, using `level` as its depth
`cbh`	convolve the first `level` station factors exactly; bound the remaining factors by geometric envelopes; form $G^{\pm}$ and hence $X^{\mp}$
`ssd`	disaggregate each multiserver station into single-server networks bounding it from either side; solve both
`sib`	refine power sums of the demand vector for `level` steps; delay-free models only
`harel`	evaluate the exact normalizing constant at populations $n \le \min(N, 7)$; extrapolate the sharp upper and lower throughput bounds from it; delay-free models only
`looping`	iterate queue-length lower bounds and the unaccounted-congestion heaps, charging each heap at the extreme per-customer inflation factors, to the pessimistic/optimistic throughput bracket
`cub`, `mbjb`	composite multiclass upper bound over per-class bottleneck constraints; `mbjb` is the separate multiclass balanced-job lower bound
`ldac`	map the closed load-dependent model onto an equivalent open one; return the lower bound, or `NaN` when $N < \hat{Q}$
`lr`	1. build the linear-reduction polytope $\mathcal{P}$ (balance, marginal and consistency constraints) 2. minimize (`lr.lower`) or maximize (`lr.upper`) the linear utilization functional over $\mathcal{P}$ by LP 3. derive the chain throughput from the bounded utilizations
`qrf.*`	build the same constraint set; optimize the nonlinear entropy or mutual-information objective by `fmincon`; blocking variants `qrf.bas.*`, `qrf.rsrd` are LPs
`scb`	run single-class MVA with $Z = 0$ to get the exact X_1 =`scb.lower`; scale by $\min((N+m-1)/N,\ 1/(X_1 \max_i L_i))$ for `scb.upper`
`bgt`	solve the Down-Meyn stability LP for the piecewise-linear Lyapunov drift margin γ; evaluate the closed-form constant of Theorem 4; upper bound only
`bpt`	build the first-order linear relaxation of the achievable region from the second-moment balance of the aggregate $R(t)$; minimize the linear sojourn-time objective by LP; lower bound only

4.13 Bibliographic notes

Asymptotic and balanced job bounds are classical and are presented systematically by Lazowska and co-authors [118]. Bound hierarchies originate with Eager and Sevcik [63], extended to multiple classes in [64], with the seeding Looping bracket in Eager's thesis [61] and related two-class constructions by Hsieh and Lam [95]. Successively improving bounds are due to Srinivasan [172], composite multiclass bounds to Kerola [103], and multiserver disaggregation to Suri and Dallery [174]. The convolutional hierarchy is due to Dowdy, Eager, Gordon and Saxton [60]. Geometric bounds and their error guarantees are from Casale, Muntz and Serazzi [34]; the convexity-based bounds of Harel and co-authors are in [86]. Robust multiclass bounds under NBUE service are due to Majumdar and Woodside [126]. The load-dependent closed-open equivalence bound is from Anselmi and Cremonesi [4]. The reduction bounds and the optimization framework behind `lr` and `qrf.*` are due to Casale, De Nitto Personé and Smirni [31]. The

single-class bounds and their model-free companions are due to Dowdy, Carlson, Krantz and Tripathi [59]. The piecewise-linear-Lyapunov queue-length bound is due to Bertsimas, Gamarnik and Tsitsiklis [15]; the achievable-region sojourn-time bound to Bertsimas, Paschalidis and Tsitsiklis [16]. The use of bounds inside optimization and inference loops, which motivates the guaranteed-sidedness requirement, is discussed in [29].

Chapter 5

SolverCTMC

5.1 Overview

`SolverCTMC` evaluates a model by explicit construction of its underlying continuous-time Markov chain: it enumerates the reachable state space, assembles the infinitesimal generator $\boldsymbol{Q}$, and solves the global balance equations $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$ for steady-state metrics, or the Kolmogorov forward equations for transient metrics [114, 173]. It is the semantic ground truth of LINE: every extended feature of the modeling language (phase-type services, class switching, impatience, retrials, finite capacity regions, Petri net primitives, heterogeneous servers, signals) is ultimately defined by the CTMC that this solver builds, and other solvers are validated against it. Its practical limitation is state-space explosion, so it is intended for models whose reachable space fits in memory, with a configurable cutoff (`options.cutoff`) truncating open models. On closed single-class networks, Section 5.5 describes the `mdd` method, which never materializes the generator but instead aggregates the exact reachable set stored as a decision diagram; on the narrower class of closed single-class product-form networks, the perfect samplers of `SolverNC` (Section 10.8) return the same steady-state metrics up to Monte Carlo error without building the chain.

The solver resides in `jline/solvers/ctmc/`, with state-space and Markov-chain algorithms in `jline/api/mc/` and the state marshalling machinery in `jline/lang/state/`.

5.2 State-space generation

The state of a stateful node is a vector encoding, per class, the queue contents (with buffer positions for order-sensitive disciplines), the service phases of in-service jobs, and local variables (routing pointers, cache lists, Petri net markings), following the state descriptors documented in the `NetworkStruct` reference. Two generation engines exist:

synchronization-based (default) Events are described by the `sn` synchronization structures (`sn.sync`): each synchronization couples an active action (e.g., a departure at node i in class r) with a passive action (the matching arrival). The generator `Ctmc_ssg` performs a reachability exploration from the initial state, applying `afterEvent` state-transition functions node by node; rates are composed from the active action rate and the passive routing probability. Fork-join models use the tag-augmented

generator `Ctmc_ssg_fj`, which introduces transient tag classes tracking sibling tasks so that join synchronization is Markovian.

flat A legacy engine that enumerates the Cartesian product of node state spaces and filters unreachable states; it is retained as a fallback for features not yet covered by the synchronized generator, e.g., certain MAP compositions and round-robin routers.

Immediate transitions (zero service times, Petri net immediate firings, pass-through routers) are eliminated by stochastic complementation [131] (`Ctmc_stochcomp`, `Ctmc_pseudostochcomp`): the generator is partitioned into tangible and vanishing states and the vanishing block is absorbed, as in generalized stochastic Petri net reductions [1]. A `MemoryGuard` component estimates the state-space cardinality before allocation and aborts with a diagnostic when the projected memory exceeds the configured budget.

5.3 Steady-state and transient solution

The generator is assembled by `Ctmc_makeinfgen` in sparse form. The steady-state algorithm `Ctmc_solve` orders methods by robustness: direct sparse LU on the augmented balance equations, with diagonal scaling; when the chain is reducible or nearly so, `Ctmc_solve_reducible` performs a block decomposition into communicating classes and solves each recurrent block (`Ctmc_solve_reducible_blkdecomp`). Iterative aggregation-disaggregation is available through the Takahashi method [108, 177] (`Ctmc_takahashi`) and Courtois decomposition for nearly completely decomposable chains [48, 49] (`Ctmc_courtois`); `Ctmc_kms` implements the Koury-McAllister-Stewart scheme. Kolmogorov reversibility testing (`Ctmc_testpf_kolmogorov`) and time reversal (`Ctmc_timereverse`) support product-form verification experiments [101].

Every steady-state solve, not only the ones explicitly requesting the reducible path, is now routed through a single entry point that treats the irreducible chain as the degenerate case of the block decomposition (one bottom strongly connected component, or BSCC, no transient states) rather than special-casing it: `CtmcStationary` in the JAR, `ctmc_stationary.m` in MATLAB, and the equivalent logic inlined unconditionally in the Python handler. All of a generator's stationary mass lives in its BSCCs, each weighted by the probability of being absorbed into it from the declared initial state; every other state is transient and carries exactly zero probability. `ctmc_stationary` seeds the absorption with a point mass at the model's initial state when that state is present in the enumerated space (it can be absent, e.g. when stochastic complementation eliminated it), and otherwise falls back to a uniform seed over the source SCCs, i.e. those with no incoming transition. Detecting an arc for the underlying strongly-connected-component decomposition is done by magnitude, $|Q_{ij}| >$ `ArcTol` (`ArcTol` $= 10^{-12}$), never by sign: a matrix-exponential (ME/CME) generator embeds off-diagonal entries that are genuinely negative, which a sign test misreads as absent arcs and which would then merge or split SCCs incorrectly. The joint, marginal and reward analyzers (`Solver_ctmc_joint`, `Solver_ctmc_marg`, `Solver_ctmc_reward`) share this entry point with the default analyzer, so BSCC handling is uniform across every output the solver produces rather than being present only for the metrics computed directly from `getAvg`.

When the fill-in of the sparse LU exceeds the memory the pre-gate allows, the same balance equations are solved iteratively by two Krylov kernels. `Ctmc_gmres` applies restarted GMRES [158] under an ILUT right preconditioner [157], falling back to a Jacobi preconditioner when the incomplete factorization breaks

down, and `Ctmc_bicgstab` the stabilized biconjugate gradient method [182], whose work and storage per iteration are constant rather than growing with the Krylov dimension, so it neither restarts nor pays the optimality that restarting costs GMRES. The two are complementary and are tried in that order: GMRES(m) stagnates when the useful subspace is wider than the restart window, where BiCGSTAB converges, and is the more robust of the two when it is not. Two preparation steps are shared and are not optional on a generator: rows are equilibrated to unit maximum norm, so that the $O(1)$ normalization row does not mix with rows carrying rates of a different magnitude, and the states are reordered by reverse Cuthill-McKee, since in the natural ordering of a birth-death chain the unpivoted elimination has growth factor $(\mu/\lambda)^n$ and overflows within a few thousand states. A fill-reducing ordering rather than pivoting is what removes that: a pivoting incomplete factorization was measured to produce two orders of magnitude more fill on a banded generator, which is what breaks the iteration rather than the recurrence itself. The block variants `Ctmc_gmres_multi` and `Ctmc_bicgstab_multi` solve a whole block of right-hand sides, reusing one factorization and warm-starting each column from the previous solution, which is the shape of the stochastic complement; `Ctmc_stochcomp`, `Ctmc_kms`, `Ctmc_takahashi` and the block-decomposition solver all check the convergence flag and fall back from GMRES to BiCGSTAB and then to the direct solve.

Two model reductions are also reachable from the solver as opt-in configuration entries rather than as methods. Under `options.config.chain_aggregation` the model is collapsed onto one class per chain by `ModelAdapter.aggregateChains`, solved, and split back per class by `sn_deaggregate_chain_results` (`ctmcChainAggregation`); the reduction is applied only when the model has fewer chains than classes, and is exact on a product-form model. Under `options.config.fes_stations` the named station subset is replaced by a single load-dependent station (`ctmcFesAggregation`); the surviving stations are answered by the reduced chain directly and a collapsed station i by the Chandy-Herzog-Woo conditional sum $E[Q_i] = \sum_{\boldsymbol{n}} P(N_{fes} = \boldsymbol{n})\, Q_i(\boldsymbol{n})$ [39], whose $Q_i(\boldsymbol{n})$ come from `fes_compute_metrics`. Throughput needs no conditioning, since flow through a station is fixed by the routing and an exact reduction leaves X unchanged. Both adapters predate any solver consumer: they were exercised by examples and tests alone until these two entries were added.

Transient analysis (`getTranAvg`, `getTranProb`) integrates $\frac{d}{dt}\boldsymbol{\pi}(t) = \boldsymbol{\pi}(t)\boldsymbol{Q}$ by uniformization [99, 173] (`Ctmc_randomization`, `Ctmc_transient`): with $\Lambda \geq \max_x |q_{xx}|$ and $\boldsymbol{P} = \boldsymbol{I} + \boldsymbol{Q}/\Lambda$,

$$\boldsymbol{\pi}(t) = \sum_{n \geq 0} e^{-\Lambda t} \frac{(\Lambda t)^n}{n!}\, \boldsymbol{\pi}(0) \boldsymbol{P}^n, \tag{5.1}$$

truncated adaptively to the requested tolerance. Time-averaged distributions used by reward and availability analysis are computed by `Ctmc_timeaverage`.

5.4 Analyzers and methods

The default analyzer `Solver_ctmc_analyzer` runs the pipeline above and derives station-class metrics from $\boldsymbol{\pi}$ through `Ctmc_avg_from_pi`, which projects state probabilities onto per-station, per-class counting functions; utilizations account for multiserver and heterogeneous-server stations by counting busy servers per state. Marginal and joint probability requests route to the handlers `Solver_ctmc_marg`, `Solver_ctmc_margaggr`, `Solver_ctmc_joint`, and `Solver_ctmc_jointaggr`, which differ in

whether phases are kept or aggregated. Reward analysis (`getAvgReward`, `Solver_ctmc_reward`) evaluates $\mathbb{E}[\rho] = \sum_x \pi_x \rho(x)$ for user-supplied state functions ρ registered in `sn.reward`, supporting nonlinear functionals such as $\mathbb{E}[n^2]$, in the tradition of Markov reward models [131, 159].

The `qrf` method invokes `Solver_ctmc_qrf_analyzer`, an optimization-based evaluation of complex stochastic networks (Quadratic Reduction Framework) that replaces full enumeration with a constrained optimization over aggregated state variables [31]. The `mdd` method replaces generation and solution together with a decision-diagram aggregation, described in Section 5.5. Stochastic Petri net models (Place/Transition nodes) are evaluated on the same pipeline, with markings as states and timed transitions as synchronizations, following GSPN semantics [1, 135, 138]; non-Markovian firing distributions (deterministic, Gamma, Weibull, lognormal, Pareto, uniform) are expanded into phase-type approximations before generation [19].

5.5 State-space aggregation via decision diagrams

The `mdd` method (`Solver_ctmc_mdd_analyzer`) targets closed, single-class networks whose exact reachable set is too large to enumerate as a flat generator but whose per-station local state is highly repetitive across the diagram. Rather than build $\boldsymbol{Q}$, it stores the reachable set as a multi-valued decision diagram (MDD) over K levels, one per station [132]: node k at level ℓ represents a set of reachable local-state combinations for stations $\ell, \ldots, K$, and identical suffixes are shared as a single node, giving a storage cost governed by the compression of the diagram rather than by $|S|$ directly. The local-state encoding is chosen by discipline: a count for exponential service under any work-conserving discipline; a count plus in-service phase for phase-type service under FCFS/LCFS/SIRO/HOL with one server; and a per-phase count vector for phase-type service under PS/DPS/GPS/INF. Phase-type service under LCFSPR/FCFSPR has no encoding and is rejected.

Given the diagram, `mdd_mcd` (Miner, Ciardo and Donatelli [133]) computes the stationary distribution by aggregation rather than by solving the full chain: it forms K coupled level-CTMCs, one per station, and iterates them to a fixed point, each level's balance equations solved by QR least squares over the row sums of its local generator restricted to the diagram nodes reachable at that level. Memory is $O(\sum_k |M_k|)$, the sum of per-level node counts, rather than $O(|S|)$; the two coincide only when the diagram compresses nothing (measured to never pay off at $K = 3$), and diverge sharply as K grows, since $\sum_k |M_k|$ is polynomial in K while $|S|$ is exponential in it.

The aggregation is exact whenever a single approximation, $\Pr\{i_k \mid \alpha\} = \Pr\{i_k \mid p\}$ (conditioning on the local state at level k is the same as conditioning on the diagram node p that encodes it), holds everywhere it is invoked; this holds trivially at any node reached by exactly one root-to-node path, and holds exactly for every product-form model regardless of diagram sharing. `mdd_mcd` therefore returns, alongside the stationary law, an *exactness certificate*: the maximum path count per level and a boolean `noAggregation` flag that is `true` whenever the sufficient single-path condition holds at every node, in which case the fixed point is certified exact with no reference solve required. The certificate is one-directional: `false` means "not certified exact by this sufficient condition", never "known approximate", since a product-form model with heavily shared diagram nodes is still exact. Where the certificate does not hold the method is an approximation, measured at approximately 0.2% error on the mean queue lengths of phase-type-at-FCFS models once the diagram compresses.

Because `mdd` never assembles $\boldsymbol{Q}$, the state space, event filtration and symbolic generator stay empty by

design, so the transient path, reward functionals, and the symbolic-generator/state-space export are unavailable under this method; its envelope is therefore an exception to Table 5.1, narrower than the shared envelope on two further axes beyond those unavailable outputs: closed networks only (an open arrival stream makes the marking unbounded, so no finite diagram exists) and a single class only (a multiclass network would need one diagram level per station-class pair rather than per station).

5.6 Symbolic solution and parametric sensitivity

Beyond the numeric solve, `SolverCTMC` exposes the chain as an algebraic object. The symbolic generator (`getSymbolicGenerator`) normalizes each event filtration by its smallest positive rate and scales it by a symbol x_e, so that $\boldsymbol{Q}$ is *linear* in $x_1, \ldots, x_E$; assembling it therefore needs no computer algebra, and substituting each symbol with its event's nominal rate recovers the numeric generator exactly. Solving $\boldsymbol{\pi Q} = \mathbf{0}$ over the field of rational functions in those symbols is a different matter, and `getSymbolicSolution` delegates it to the external backend named by `options.config.symbolic`, the SageMath service documented in the user manual under "Symbolic analysis backend"; the vector is returned both entry by entry and over one common denominator. Because printed normal forms depend on the engine and its version, and symbol numbering follows event enumeration order, a symbolic result is compared across codebases by substituting rates and comparing numbers, never by matching text.

Parametric sensitivity follows Trivedi and Bobbio [181], Sec. 9.7, and is reached through `getSensitivity` in two ways that differ in where the differencing happens. The `fd` method central-differences $\boldsymbol{Q}$ with respect to θ with the state space held fixed, which is legitimate because the space depends on the topology and the cutoff and not on rate values, then hands $\mathrm{d}\boldsymbol{Q}/\mathrm{d}\theta$ to `ctmc_sens`: one solve of the differentiated balance equations, whose coefficient matrix is the one the stationary solve already factorized. It needs no backend and is exact to $O(\text{step}^2)$. The `symbolic` method instead differentiates the rational function $\boldsymbol{\pi}(x_1, \ldots, x_E)$ exactly and closes with the chain rule over the rate map $x_e(\theta)$, which is affine in θ whenever θ sets or scales a rate, so the one remaining difference is reproduced exactly and the $O(\text{step}^2)$ term of `fd`, which comes entirely from differencing through the solve, disappears. A parameter whose perturbation reshapes a filtration rather than scaling it breaks the chain rule and is refused by name. Both forms report the unscaled derivative of Eq. (9.79) and the dimensionless scaled form of Eq. (9.80), and both take $\mathrm{d}r/\mathrm{d}\theta = 0$: a reward whose own rates depend on θ needs the second term of Eq. (9.83) and is outside what either computes. Agreement between the two is evidence about both, since they share nothing but the model, and that is how the cross-codebase tests assert them.

5.7 Architectural flow

Figure 5.1 shows the standard path. Transient requests replace `ctmc_solve` with uniformization over the same generator; probability requests replace the projection step with the marginal or joint handlers. The sampling methods branch out of `runAnalyzer` before the memory guard and skip the diagram from generation onwards.

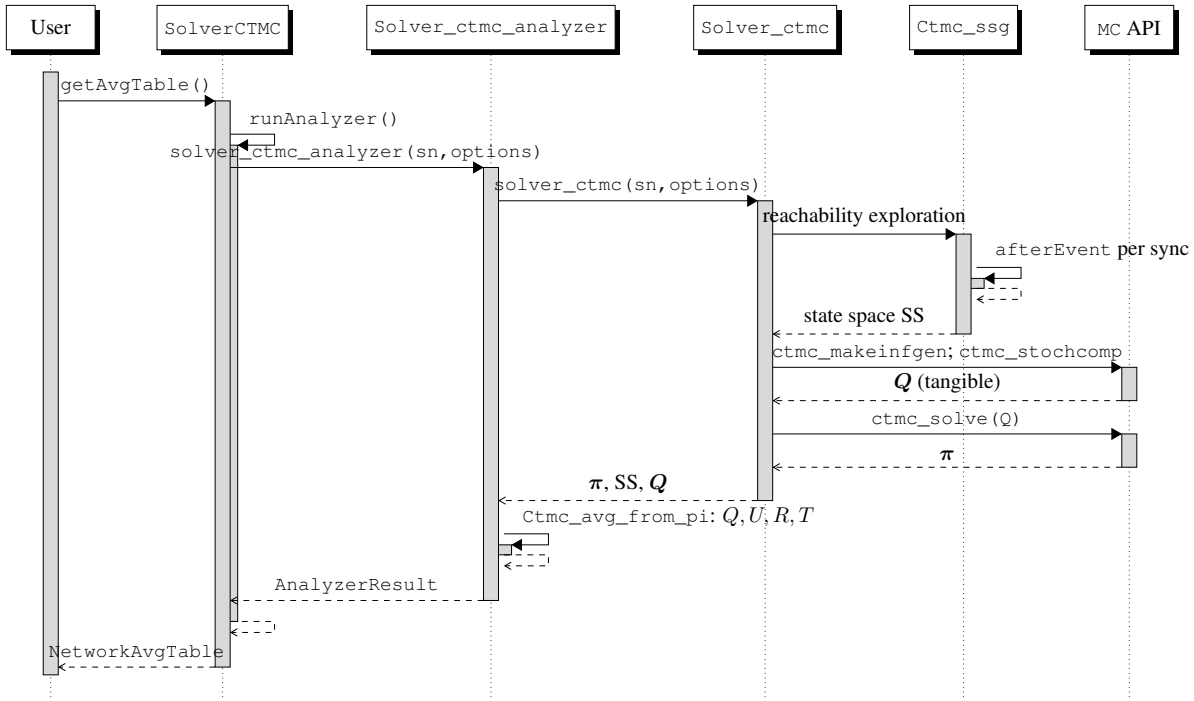


Figure 5.1: Sequence diagram of `SolverCTMC` for a steady-state average request with the synchronization-based generator.

5.8 Implementation notes

State vectors are canonicalized and hashed so that reachability exploration is a breadth-first search with $O(1)$ duplicate detection; `sn` index-conversion methods map node states to station metrics. The solver can export the symbolic generator (`getSymbolicGenerator`) with transition labels, supporting derivation of parametric chains, and can save the state space for inspection (`getStateSpace`, `ctmc_simulate` for sample-path cross-checks). Open classes require a finite cutoff; the truncation error is not bounded by the solver and should be assessed by cutoff refinement. The analyzer caps reported utilizations at 1 for unstable open models and emits a warning, consistently with the tool-wide convention. For models with cache nodes, hit and miss probabilities are exact functionals of $\boldsymbol{\pi}$, making the solver the reference for cache-analysis validation.

5.9 Feature and method support

Because `SolverCTMC` builds the exact chain, feature admission does not vary by solution algorithm: the steady-state methods (direct sparse solve, `Ctmc_solve_reducible`, Takahashi, Courtois, KMS) and the transient uniformization path all operate on the same generator and therefore share the feature envelope of Table 5.1. The methods differ in numerics and in output: transient averages and probabilities (`getTranAvg`,

`getTranProb`) use uniformization, marginal and joint probabilities use the `marg`/`joint` handlers, and reward functionals use `Solver_ctmc_reward`; all are available for every feature in the envelope. The `qrf` optimization method covers a structural subset. Non-Markovian firing and service distributions are admitted through phase-type expansion, hence ○. The `mdd` method of Section 5.5 is the exception, because it never builds the generator: closed and single-class only, and none of the transient, reward or symbolic-generator/state-space outputs, since `Q`, the state space and the event filtration are never assembled under it.

Table 5.1: Feature envelope of `SolverCTMC` (● supported, ○ supported via phase-type expansion).

Feature	CTMC	**Feature**	CTMC
Open / closed / self-looping class	●	Class switching	●
Priority classes	●	Signal / catastrophe	●
Source, Sink, Queue, Delay	●	Router	●
Fork / Join	●	Cache node	●
Place, Transition (timed/immediate)	●	Enabling / inhibitor arc	●
Exponential	●	Phase-type (Erlang, HyperExp, Coxian, APH, PH)	●
MAP, MMPP2	●	General (Gamma, Weibull, Lognormal, LogLogistic, MassAtOrigin, Pareto, Uniform, Det)	○
INF, FCFS, PS	●	DPS, GPS	●
LCFS, LCFS-PR	●	SEPT, LEPT	●
SIRO, HOL	●	LPS	●
PAS (pass-and-swap)	●	Priority variants (`*PRIO`)	●
Routing PROB, RAND	●	RROBIN, WRROBIN	●
JSQ, KCHOICES	●	Cache repl. (RR/FIFO/SFIFO/LRU)	●
Load dependence	●	Retrial (orbit)	●
Balking	●	Reneging (impatience)	●
Steady means (`getAvg`)	●	Marginal/joint prob.	●
Transient (`getTranAvg/Prob`)	●	Rewards (`getAvgReward`)	●
Symbolic generator	●	State-space export	●
Symbolic solution	●	Parametric sensitivity	●

The symbolic solution (`getSymbolicSolution`) and the `symbolic` branch of the parametric sensitivity (`getSensitivity`) are the two entries that need an external computer algebra backend; the `fd` branch of the sensitivity needs none. Both are unavailable under `mdd`, which assembles no generator. See §5.6.

5.10 Method algorithms

All methods share the generation stage of Table 5.2 and differ in how the generator is solved.

Table 5.2: High-level pseudocode of `SolverCTMC` stages and methods.

Stage / method	**Pseudocode**
generation	1. BFS from the initial state; for each synchronization apply `afterEvent` node-by-node to reach successors, hashing states for $O(1)$ deduplication 2. assemble sparse $\boldsymbol{Q}$ (`makeinfgen`); eliminate immediate/vanishing states by stochastic complementation

continued on next page

Stage / method	Pseudocode
`default` (direct)	solve $\boldsymbol{\pi Q} = \mathbf{0}$, $\boldsymbol{\pi 1} = 1$ by sparse LU on the augmented, diagonally scaled system
reducible	decompose into communicating classes; solve each recurrent block and weight by reachability
Takahashi / Courtois / KMS	iterative aggregation–disaggregation: partition states, solve the aggregated chain, disaggregate, iterate to convergence (NCD chains)
transient	uniformize with $\Lambda \geq \max_x \lvert q_{xx} \rvert$, $\boldsymbol{P} = \boldsymbol{I} + \boldsymbol{Q}/\Lambda$; $\boldsymbol{\pi}(t) = \sum_{n \geq 0} e^{-\Lambda t} \frac{(\Lambda t)^n}{n!} \boldsymbol{\pi}(0) \boldsymbol{P}^n$, truncated to tolerance
reward	$\mathbb{E}[\rho] = \sum_x \pi_x \, \rho(x)$ for each registered state functional ρ (steady or time-averaged)
`qrf`	replace enumeration by constrained optimization over aggregated state variables (quadratic reduction)
`mdd`	1. build the exact reachable set as a K-level multi-valued decision diagram, one level per station, sharing identical suffixes 2. form K coupled level-CTMCs from the diagram and iterate to a fixed point (QR least squares per level) 3. certify exactness when every diagram node is reached by a single root-to-node path (sufficient, not necessary)
metrics	project $\boldsymbol{\pi}$ onto per-station, per-class counting functions (`avg_from_pi`); marginal/joint handlers keep or aggregate phases

5.11 Bibliographic notes

Numerical solution of Markov chains is treated comprehensively by Stewart [173]; uniformization goes back to Jensen [99]. Aggregation-disaggregation methods are due to Takahashi [177] and Koury, McAllister and Stewart [108], and decomposition of nearly completely decomposable chains to Courtois [48, 49]. Stochastic complementation theory is from Meyer [131]. GSPN semantics and vanishing-state elimination follow [1, 56, 135, 138]. Markov reward models are surveyed in [159]. The QRF optimization framework is introduced in [31]. Perfect sampling by coupling from the past is due to Propp and Wilson [147]; its monotone specialization to closed Jackson networks, together with the rapidly-mixing chain and its mixing-time bound, is due to Kijima and Matsui [104]. Product-form testing connects to the reversibility theory of Kelly [101] and to algorithmic product-form detection [8, 127, 128]. The multi-valued decision diagram representation of a reachable set is due to Miner and Ciardo [132]; the level-CTMC aggregation implemented by `mdd` is due to Miner, Ciardo and Donatelli [133].

Chapter 6

SolverFLD

6.1 Overview

`SolverFLD` analyzes queueing networks by fluid and mean-field approximation: the stochastic population process is replaced by the solution of a system of ordinary differential equations that becomes exact in the many-jobs scaling limit of Kurtz [115, 116]. For a network with phase-type services, the state is the vector $\boldsymbol{x}(t)$ of expected jobs per station, class, and phase, and the ODE system takes the form

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{x}(t)\,\boldsymbol{W}(\boldsymbol{x}(t)), \tag{6.1}$$

where the rate matrix $\boldsymbol{W}$ collects phase-type dynamics, routing, and a scheduling-dependent capacity-sharing term, e.g., $\min(x_i, c_i)/x_i$ for PS multiserver stations. Steady-state metrics are read from the equilibrium $\boldsymbol{x}(\infty)$, and transient metrics from the trajectory; response-time distributions are obtained by augmenting the system with passage-time tracer classes. Fluid limits of this kind for closed multiclass networks with PS and delay stations were derived in [22, 36, 180], and the solver descends from that line of work, with FCFS handled by smoothed approximations [155].

The solver resides in `jline/solvers/fluid/`, with analyzers in `analyzers/` and ODE right-hand sides in `handlers/`. Numerical integration uses stiff and non-stiff solvers from Apache Commons Math, plus an LSODA extension (`LSODAExt`) with automatic stiffness switching.

6.2 Analyzers and methods

`closing`, `softmin`, `tbi` `ClosingAndStateDepMethodsAnalyzer` with the `ClosingAndStateDepMethodsODE` right-hand side. The ODE system (6.1) is built per phase with closing of the service processes: each station's phase-type representation $(\boldsymbol{D}_0, \boldsymbol{D}_1)$ contributes linear intra-phase dynamics and completion flows routed by `sn` routing matrices. Scheduling enters through state-dependent saturation functions; the `softmin` option replaces $\min(x_i, c_i)$ with a smoothed softmin to improve integrability near the corner point, following the smoothing analysis of [155]; `softmin` requests that smoothing by name. `tbi` runs the same field through the trajectory-based iteration of Sheldon, Tuncer and Casale [169]: the station set is partitioned into cells and, on each time segment, every cell's initial-value problem is solved with the other cells frozen at the trajectory of the previous sweep (waveform

relaxation, Gauss-Seidel or Jacobi), iterating until the sup-norm gap between successive trajectories falls below `options.config.tbi_tol`; cross-cell inflows are read off frozen trajectories and interior flows off the live cell state, which is what lets a very large transient decompose.

`statedep` The same analyzer with fully state-dependent rate scalings, covering load-dependent stations by interpolating `sn.lldscaling` inside $\boldsymbol{W}(\boldsymbol{x})$.

`matrix`, `pnorm` (default) `MatrixMethodAnalyzer` with `MatrixMethodODE`: for networks whose fluid equations are linear except for the capacity term, the right-hand side is assembled once in matrix form, enabling faster evaluation and, in the linear regime, matrix-exponential solutions. `pnorm` is the same assembly with the p-norm (p^*) smoothing of the capacity term, and the pair is what `default` resolves to.

`mfq` `MFQAnalyzer` evaluates Markov-modulated fluid queues: infinite-buffer fluid models driven by a background CTMC, solved by the spectral and matrix-analytic techniques for stochastic fluid flows [3, 12, 83, 162], with compositional approximations in the style of [66]. This analyzer underpins the `mfq_*` algorithms ported from established fluid-flow toolboxes and connects to fluid stochastic Petri nets [91]. On an eligible topology it also serves Age-of-Information analysis (`solver_mfq_aoi`) and fluid priority queues (`solver_mfq_prio`); a model outside the single-queue shapes it recognizes falls back to `matrix` with a warning.

`refined`, `rmf` Model-aware aliases for refined mean field. Without cache nodes, the analyzer applies the min-normal closure of Guenther, Stefanek and Bradley [82] and the Gast–van Houdt $O(1/N)$ correction about the mean-field fixed point; with cache nodes, it splits the model into caches solved by the refined mean-field limits of Gast and Van Houdt [74] and a queueing network in which each cache is relabeled as a class switch, alternating until request rates stabilize.

`minnormal` The min-normal closure without the $O(1/N)$ refined-mean correction, alternating the mean ODE at a held covariance with a Lyapunov solve for the covariance at the resulting fixed point. It supplies the second-order output of `getMoments`.

`dae` On a stochastic Petri net with fluid places [91], each fluid level is one more coordinate of the drift and its two bounds are active-set constraints; a level pinned at a bound is reported as the bound (`options.config.fluid_boundary='reflect'`, default) or as the mean of a reflected Brownian motion with the net inflow's drift and variance rate (`'diffusion'`) [87], which is approximate. On a queueing network it is the min-normal closure stated and solved as one differential-algebraic system (`solver_fluid_dae`): in steady state the drift residual, the population conservation and the closure consistency are solved simultaneously by damped Newton, and over a finite horizon the index-1 DAE is integrated by the RODAS Rosenbrock method of Hairer and Wanner [85], so the transient carries a time-varying covariance. Finite capacity regions and per-class station buffers enter as algebraic constraint rows, with one staging coordinate per (region, class) holding the mass a throttled admission leaves waiting outside the region; a station buffer gets no staging room, matching the reference semantics in which the blocked job remains at its upstream station.

diffusion A diffusion approximation that integrates the stochastic differential equation of the population process by the Euler-Maruyama scheme (`solver_fluid_diffusion`), for closed networks only; the reported means average the simulated diffusion paths.

kp `KoPenderAnalyzer` implements the fluid and diffusion limits of Ko and Pender [106] for the open $(\text{MAP}_t/\text{Ph}_t/\infty)^N$ network: time-inhomogeneous `MAPt`/`PHt` arrival and service processes, restricted to infinite-server (delay) and finite-server stations. Unlike every other method in this chapter, `kp` integrates the mean *and* the covariance jointly,

$$\dot{\boldsymbol{q}} = F(t,\boldsymbol{q}) = \boldsymbol{A} f(t,\boldsymbol{q}), \qquad \dot{\boldsymbol{\Sigma}} = \boldsymbol{J}\boldsymbol{\Sigma} + \boldsymbol{\Sigma}\boldsymbol{J}^{\top} + \boldsymbol{G}, \qquad \boldsymbol{J} = \boldsymbol{A}\,\partial f/\partial \boldsymbol{q}, \quad \boldsymbol{G} = \boldsymbol{A}\,\mathrm{diag}(f)\,\boldsymbol{A}^{\top},$$

with $\boldsymbol{A}$ the jump matrix (one column per event, the event's jump vector) and f the event-rate vector; $\boldsymbol{G}$ is $d\boldsymbol{H}\,d\boldsymbol{H}^{\top}$ of Theorem 3.3 of [106], each independent Poisson term contributing $\boldsymbol{l}_e\boldsymbol{l}_e^{\top} f_e$. Where f is affine in $\boldsymbol{q}$ – every infinite-server station and the arrival phase process – $\boldsymbol{J}$ does not depend on $\boldsymbol{q}$ and both equations close exactly, so the mean and covariance are exact rather than asymptotic for the pure $(\text{MAP}_t/\text{Ph}_t/\infty)^N$ case; a finite-server station reintroduces the $\min(x,c)$ nonlinearity and the covariance degrades to a linear-noise approximation. `kp` does not reuse the `closing` ODE: that formulation routes a departure from source to destination through the *stationary* arrival-instant vector $\boldsymbol{\pi}$, replacing the $\boldsymbol{D}_1^{\top}$ operator with the rank-one map $\boldsymbol{\pi}(\boldsymbol{D}_1\boldsymbol{e})^{\top}$ – i.e. it treats the arrival stream as a PH renewal process with representation $(\boldsymbol{\pi}, \boldsymbol{D}_0)$. That renewal's stationary rate is exact but its autocorrelation is gone, and preserving a non-renewal arrival stream's autocorrelation is the entire point of a MAP, so `kp` tracks the arrival phase occupancy directly instead. It is available only for open models (no closed class has an arrival process to modulate) and is the sole `SolverFLD` method returning a second moment, exposed via `getTranAvgVar`.

Immediate transitions are removed before integration by `ImmediateElimination`, mirroring the vanishing-state reduction of the CTMC solver. The `PStarSearcher` and `PStarOptimisationFunction` components calibrate the blending parameter p^* used to interpolate FCFS stations between PS-like and delay-like fluid regimes, solving a scalar optimization per station [155].

6.3 Transient analysis and passage times

Transient averages (`getTranAvg`) return the integrated trajectory $\boldsymbol{x}(t)$ on the requested time grid, with the initial condition taken from the model state (`sn.state`) or a marginal prior. Response-time distributions (`getCdfRespT`) use `PassageTimeODE`: a tagged infinitesimal tracer population is injected at the reference station and the CDF is read from its absorption dynamics, following the fluid passage-time methodology developed for SLA assessment in [143, 145]. The ODE right-hand side can also be exported symbolically (`FluidODEsExporter`, `exportODEs`) as LaTeX or code, enabling external verification of the generated vector field.

6.4 Architectural flow

Figure 6.1 shows the default path. Stationarity is detected by a step handler (`MethodStepHandler`) monitoring the norm of the derivative; if the integrator stalls, the analyzer switches solver class (non-stiff to stiff)

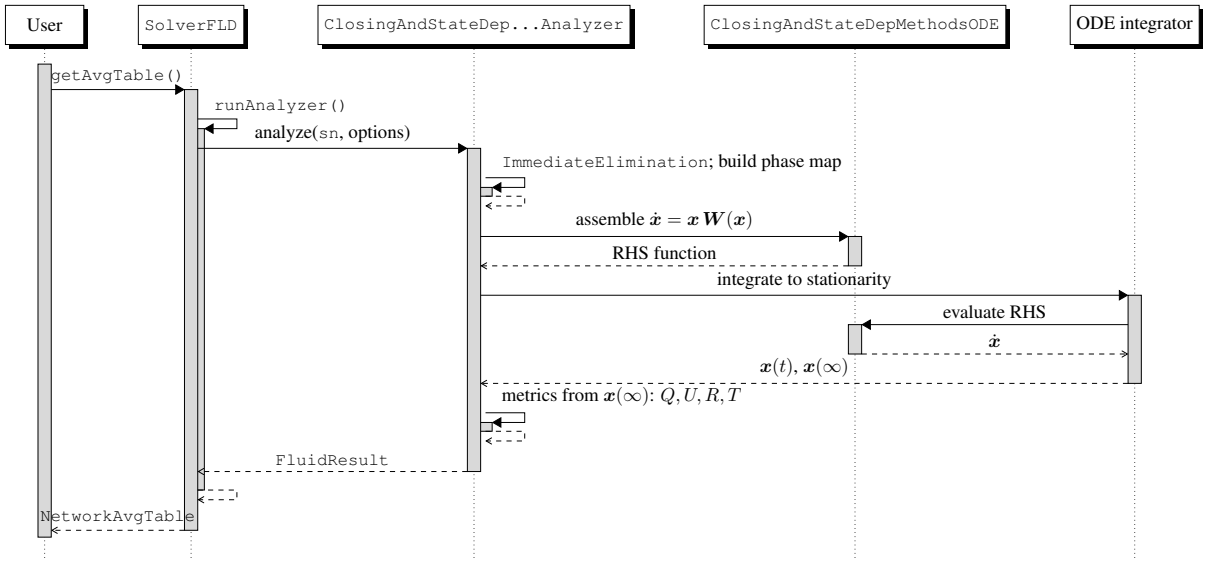


Figure 6.1: Sequence diagram of `SolverFLD` with the default closing method.

and retries.

6.5 Implementation notes

The codebases use different ODE integrators (MATLAB `ode15s`/`ode23s`, Apache Commons Math and LSODA in Java, SciPy in Python, and the C++ port's own embedded stiff and non-stiff steppers, with RODAS behind `dae`), so small numerical deviations across implementations are expected and tolerated by the parity suite. The phase-expanded state dimension is $\sum_{i,r}\texttt{phases}(i,r)$, and Jacobians for the stiff solvers are assembled analytically in the matrix method and numerically otherwise. Utilizations are derived from the equilibrium inflow rates rather than from $\boldsymbol{x}(\infty)$ directly, which is more robust when the fluid fixed point lies on the saturation boundary. Known limitations are documented in the feature set: multiclass FCFS with highly variable service is approximated through the p^* blending and should be cross-checked against simulation.

6.6 Feature and method support

Table 6.1 maps modelling features to the `SolverFLD` methods. `closing` is the general-purpose analyzer; `statedep` specializes it to state-dependent (load-dependent) scalings; `matrix` assembles the same field once for the linear-plus-capacity regime; `mfq` is the Markov-modulated fluid-queue analyzer, a distinct model class (fluid buffer driven by a background CTMC) rather than a job-population network; `rmf`/`refined` select the generic correction or cache decomposition from the model; and `kp` is the time-inhomogeneous open-network mean/covariance limit of Ko and Pender [106], restricted to `MAPt`/`PHt` arrivals and services at infinite- and finite-server stations. The methods absent from the table share the

envelope of the column they derive from: `pnorm` and `softmin` are the smoothing variants of `matrix` and `closing`, `tbi` is the transient waveform relaxation over the `closing` field, `minnormal` and `refined` are the moment closures over the closing envelope (with GPS admitted by `minnormal` only), `dae` is the DAE statement of `minnormal`, narrower on the closing-family disciplines (SIRO, LCFS, LCFS-PR, DPS, GPS are refused) but alone in admitting finite capacity regions, and admitting load dependence as `statedep` does, and `diffusion` is closed-networks-only. FCFS is captured only through the p^* smoothing and is therefore ○; general distributions enter through phase-type/moment closure (○).

Table 6.1: Feature support of `SolverFLD` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	closing	statedep	matrix	mfq	rmf	kp
Job classes						
Open class	●	●	●	○		●
Closed class	●	●	●		●	
Self-looping class	●	●	●		○	
Class switching	●	●	●		○	
Node types						
Source, Sink	●	●	●	○		●
Queueing station	●	●	●	●	●	○
Delay (infinite server)	●	●	●		●	●
Cache node	●	●	●			
Service distributions						
Exponential	●	●	●	●	●	●
Phase-type (Erlang, HyperExp, Coxian, Cox2, APH)	●	●	●	○	●	●
General (Gamma, Weibull, Lognormal, Pareto, Uniform, Det)	○	○	○		○	
Time-inhomogeneous (`MAPt`, `PHt`)						●
Scheduling disciplines						
INF (delay)	●	●	●		●	●
PS	●	●	●		●	
DPS	●	●	●		○	
FCFS	○	○	○		○	
LCFS, LCFS-PR	●	●	●		○	
SIRO	●	●	●			
Advanced features						
Load dependence	○	●	●			
Markov-modulated fluid (background CTMC)				●		
Refined mean-field $1/N$ correction					●	
Joint mean/covariance limit (2nd moment)						●
Analysis and output						
Steady-state means (`getAvg`)	●	●	●	●	●	●
Transient averages (`getTranAvg`)	●	●	●	○		●
Response-time CDF (`getCdfRespT`)	●	●	○			
Symbolic ODE export (`exportODEs`)	●	●	●			
Transient covariance (`getTranAvgVar`)						●

6.7 Method algorithms

Table 6.2 gives high-level pseudocode for each `SolverFLD` method.

Table 6.2: High-level pseudocode of `SolverFLD` methods.

Method	Pseudocode
`closing`	1. build the phase-expanded state $\boldsymbol{x}$ and the ODE $\dot{\boldsymbol{x}} = \boldsymbol{x}\,\boldsymbol{W}(\boldsymbol{x})$ from $(\boldsymbol{D}_0, \boldsymbol{D}_1)$, routing, and the capacity term $\min(x_i, c_i)/x_i$ (`softmin` smoothing near the corner) 2. eliminate immediate transitions; integrate to stationarity (step handler on $\|\dot{\boldsymbol{x}}\|$, stiff/non-stiff switch) 3. read Q, U, R, T from $\boldsymbol{x}(\infty)$ (utilizations from inflow rates)
`statedep`	as `closing` with fully state-dependent rate scalings ($\boldsymbol{W}(\boldsymbol{x})$ interpolates `lldscaling`)
`matrix`	assemble $\boldsymbol{W}$ once in matrix form (linear except the capacity term); integrate, using matrix-exponential solution in the linear regime
`mfq`	Markov-modulated fluid queue: solve the fluid buffer driven by the background CTMC by spectral / matrix-analytic techniques for the stationary buffer distribution
`rmf/refined`	without a cache: solve the min-normal closure and add the Gast–van Houdt $O(1/N)$ correction about the mean-field fixed point; with a cache: alternate the isolated cache limit and the relabeled queueing layer until request rates stabilize
`minnormal`	1. integrate the mean ODE to its fixed point at a held covariance 2. solve the Lyapunov equation for the covariance there; extract the min-normal closure terms 3. repeat until mean and covariance are mutually consistent; report Q, U, R, T and the second-order output
`dae`	steady state: solve drift residual + population conservation + closure consistency simultaneously by damped Newton, one Lyapunov solve per residual; transient: integrate the index-1 DAE by RODAS; capacity caps enter as algebraic rows with staging coordinates for regions
`tbi`	partition the stations into cells; per sweep, integrate each cell's IVP with the other cells frozen at the previous sweep's trajectory; repeat until the trajectory sup-norm gap falls below `tbi_tol`
`diffusion`	integrate the population SDE by Euler-Maruyama; average the simulated paths (closed networks only)
`kp`	1. lay out the state as arrival-MAP-phase blocks (one per Source/class) followed by service-phase blocks (one per station/class) 2. enumerate the five Ko-Pender event families – arrival-phase change without arrival, arrival-phase change with arrival, service-phase change, completion leaving the network, completion routed onward – as jump vectors $\boldsymbol{A}$ and rate functions $f(t, \boldsymbol{q})$ evaluated on the (possibly piecewise-constant) `MAPt/PHt` schedule 3. integrate $\dot{\boldsymbol{q}} = \boldsymbol{A} f(t, \boldsymbol{q})$ and $\dot{\boldsymbol{\Sigma}} = \boldsymbol{J}\boldsymbol{\Sigma} + \boldsymbol{\Sigma}\boldsymbol{J}^{\top} + \boldsymbol{G}$ jointly 4. read Q, U, R, T from $\boldsymbol{q}(\infty)$ (or the requested time grid) and the second moment from $\boldsymbol{\Sigma}$

6.8 Bibliographic notes

Mean-field limits for Markov population processes are due to Kurtz [115, 116]. Fluid analysis of closed multiclass queueing networks and of PS stations was developed in [22, 36], with reward-based extensions in [180] and model-transformation applications in [136]. Passage-time and percentile analysis via fluid tracers is from [143,145]. The refined mean-field approximation is due to Gast and Van Houdt [75], and its cache decomposition to [74], and the smoothed FCFS fluid model to Ruuskanen et al. [155]. Markov-modulated fluid queues originate with Anick, Mitra and Sondhi [3], with matrix-analytic treatments in [12, 83, 162], compositional approximations in [66], and fluid stochastic Petri nets in [91]. The joint fluid/diffusion limit of the time-varying $(\mathrm{MAP}_t/\mathrm{Ph}_t/\infty)^N$ network implemented by `kp` is due to Ko and Pender [106], extending single-station time-varying limits to the network setting. The min-normal moment closure behind

`minnormal` and `dae` is due to Guenther, Stefanek and Bradley [82], the $O(1/N)$ refinement of `refined` to Gast and Van Houdt [75], building on the $1/N$-accuracy result of Gast [73], the RODAS integrator of the DAE transient to Hairer and Wanner [85], and the trajectory-based iteration of `tbi` to Sheldon, Tuncer and Casale [169].

Chapter 7

SolverLDES

7.1 Overview

`SolverLDES` implements `LDES`, an event-driven simulation engine built on the SSJ stochastic simulation library [119]. In contrast with `SolverSSA`, which samples the Markov jump process of the phase-type expanded state vector, `LDES` simulates job entities flowing through the network: arrivals, service completions, and routing decisions are scheduled as timestamped events on a future-event list, and service times are drawn directly from the configured distributions. This makes the solver applicable to non-Markovian timing (deterministic, uniform, Pareto, Weibull, lognormal, empirical traces via `Replayer`) without phase-type expansion, and to fine-grained scheduling semantics (SRPT, FB, EDF, pass-and-swap, polling, fair queueing variants) that are cumbersome to encode as Markovian synchronizations.

The solver resides in `jline/solvers/ldes/`, with the simulation algorithm in `handlers/Solver_ssj.java` and its layered-network counterpart in `handlers/Solver_ssj_ln.java`. LDES is now *two* engines behind one interface: the native C++ engine (`cpp/include/line/solvers/ldes/`, distributed as the `common/ldes` binary) and the Java one (`common/ldes.jar`), which remains the fallback for what the C++ engine does not cover. Both answer to the same command-line contract and emit the same result document key for key; MATLAB and Python invoke whichever is available as a JSON-mediated subprocess. The C++ engine reproduces SSJ's random streams bit-exactly (MRG32k3a and `java.util.Random`, through SSJ's own inverse-CDF quantiles, one stream per node-class pair at the reference's seed offsets), so on a model whose routing consumes no extra draws the two engines walk the *same sample path*: queue lengths, utilizations and throughputs agree to machine zero and the time-based metrics to 10^{-14} on the seeded acceptance models. A model with feedback or re-entrance consumes the routing stream in a different order, and there the two paths part company at the ordinary Monte-Carlo scale; a claim of cross-engine identity must therefore state which of the two cases the model is in, and a seeded golden regenerated against one engine must not be read against the other. The authoritative statement of supported model features is `SolverLDES.getFeatureSet()`, and `getLNFeatureSet()` for layered networks.

7.2 Simulation algorithm

`Solver_ssj` compiles `sn` into an array of station runtime objects, each holding its buffer structures, server tokens, and per-class distribution samplers backed by SSJ random streams. The event loop is driven by SSJ's `simevents` scheduler: `simulator.simulate(maxEvents)` advances simulated time until `options.samples` events have been processed (steady-state mode) or until the end of the requested horizon (transient mode, `solver_ssj_transient`). The main event types are arrivals (from `Source` nodes or upstream departures), service completions (per in-service job, with preemption bookkeeping for the preemptive-resume and preemptive-repeat disciplines), impatience firings (reneging, balking, retrial orbit re-attempts), cache lookups with replacement-list updates, Petri net transition firings with atomic token consumption and production, and state-change events for modulated arrival processes.

Scheduling is implemented per station by policy-specific insertion and selection rules over the buffer: insertion order realizes FCFS/LCFS/SIRO/priority/deadline orders; size-based policies (SJF, SEPT, SRPT, PSJF, FB/SETF) maintain attained-service or remaining-work keys [142, 188]; processor-sharing families (PS, DPS, GPS) are simulated by rescheduling the earliest completion whenever the sharing weights change; polling stations implement exhaustive, gated, and decrementing service with switchover times [176]; pass-and-swap (PAS) stations apply the swap-graph semantics at each completion. Fork nodes replicate jobs into sibling tasks and join nodes synchronize them, including quorum joins; finite-capacity stations and regions apply blocking-after-service or drop rules at admission. Batch arrivals release a whole batch per arrival epoch (`Source.setArrivalBatch`); server breakdowns alternate each server between up and down on exponential failure and repair clocks, the job in service holding its residual work across the outage, with an optional degraded-rate mode in place of a stop; and time-inhomogeneous `MAPt`/`PHt` processes carry their modulating phase across successive activations. A slotted mode (`LDESOptions.slotted`) runs the whole engine on a discrete time lattice, refusing any sampled time off the lattice rather than rounding it and ordering intra-slot events by a fixed phase, which is what makes Geo/Geo/1-type discrete queues exact. Busy-period measurement (`busyPeriodOrders`) accumulates the mean busy period of orders $1..K$ per station, station-class pair and declared station subset, and cache item sizes with per-list cost caps (`setItemSizes`, `setCostCaps`) refuse a promotion or insertion that would breach a cap without disturbing the cache state.

7.3 Analyzers, estimation, and rewards

`Solver_ldes_analyzer` validates the model against the feature set, normalizes the initial state (the model's declared state, `sn.state`, gives the per-class count at each queue and delay, injected station by station with classes descending, in both the Java and the native C++ engine; a station declaring nothing, or counts that do not conserve a closed population, fall back to the reference-station placement, and a prior over states with different counts is refused as `LINE:unsupported:StatePrior`), draws the master random stream from `options.seed`, and calls the algorithm in steady-state or transient mode. A warm start (`SolverLDES(model, initSolver)`/`initFromSolver`) seeds the model initial state from an auxiliary solver's steady state through `initFromMarginal`, as for every state-driven solver, and disables warmup removal, which shortens the transient the estimators must outlive. A *job-level* initial state (`initFromJobs`, held in `options.init_jobs` because only LDES can honour it) places individual jobs with their remaining work, attained service, ages, server order and an optional tag; it takes precedence over `init_sol`,

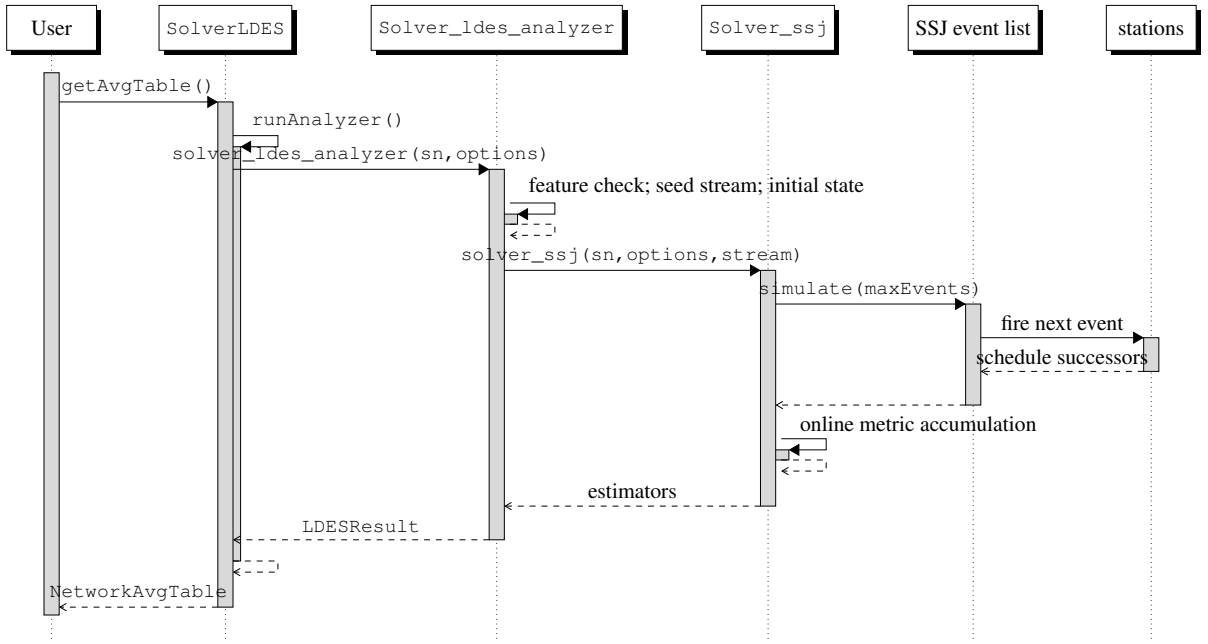


Figure 7.1: Sequence diagram of `SolverLDES` for a steady-state request.

which itself takes precedence over the model state, is honoured at FCFS, the PS family and Delay, and with `track_jobs` the run returns its final jobs in the same layout plus the departures of tagged jobs, so a later run continues exactly where the previous one stopped. `Solver_ldes_analyzer_parallel` runs independent replications on separate SSJ substreams and pools the estimators. Point estimates are computed online: utilizations and queue lengths by time-weighted integrals, throughputs by event counting, and response times by per-job timestamps; transient mode emits trajectories on the requested grid. Reward metrics (`getAvgReward`, `getTranReward`) are computed by event-level integration of user-defined state functionals along the sample path, permitting nonlinear rewards such as $\mathbb{E}[n^2]$, and are validated against the CTMC reward engine. Verification of the engine itself follows a two-sample testing methodology: the acceptance suite compares LDES against `SolverJMT` with two-sample t-tests at the 1% significance level.

Layered networks are simulated natively by `Solver_ssj_ln` through `Solver_ldes_ln_analyzer`: hosts, tasks, entries, activities, and synchronous or asynchronous calls are simulated directly on the LQN topology (Chapter 13 describes the LQN formalism), avoiding the layer-decomposition error of analytical LQN solvers [67].

7.4 Architectural flow

Figure 7.1 shows the standard path; in the MATLAB and Python codebases, the call into `Solver_ldes_analyzer` is preceded by a subprocess dispatch that serializes the model, invokes the JAR or the native binary, and parses the returned result (`LDESResultIO`).

7.5 Implementation notes

The algorithm is a single translation unit by design, keeping the event-handling code paths monomorphic for JIT friendliness; a lightweight standalone artifact (`ldes.jar`, plus a GraalVM native binary) repackages it for tool-independent distribution. Random streams follow SSJ's stream/substream discipline [119], guaranteeing reproducibility and independence across replications. Petri net firings are atomic: input tokens are consumed and output tokens produced in one event, which avoids the artifact states of token-reservation implementations. Cache simulation updates replacement lists (LRU/FIFO/RR/SFIFO/HLRU/CLIMB/QLRU) at lookup time and supports delayed-hit retrieval semantics in which concurrent misses on an item coalesce onto one back-end fetch. For LQN models, an external cross-check against the `lqsim` simulator of the LQNS distribution [69] is available through the `SolverLQNS` wrapper.

7.6 Feature and method support

`SolverLDES` has the widest feature envelope in LINE, since discrete-event simulation of job entities imposes no product-form or Markovian restriction. Feature admission does not vary across the simulation methods: the steady-state and transient engines and the parallel-replication engine accept the same models (Table 7.1) and differ only in output (transient trajectories, replication confidence intervals, reward functionals) and cost. The authoritative list is `SolverLDES.getFeatureSet()` (and `getLNFeatureSet()` for layered networks).

Table 7.1: Feature envelope of `SolverLDES` (• supported).

Feature	LDES	Feature	LDES
Job classes and nodes			
Open / closed / self-looping class	•	Class switching	•
Priority classes	•	Source, Sink, Queue, Delay	•
Router	•	Fork / Join (with quorum)	•
Logger / LogTunnel	•	Cache node	•
Place, Transition (timed/immediate)	•	Queueing place (QPN)	•
Finite-capacity region (FCR)	•	Finite-capacity blocking	•
Service and arrival distributions			
Exponential, Erlang, HyperExp	•	Coxian, Cox2, APH, PH	•
MAP, DMAP, MMAP, BMAP, MMPP2	•	ME, RAP	•
Deterministic, Immediate, Disabled	•	Uniform, Gamma, Pareto	•
Weibull, Lognormal	•	NHPP	•
LogLogistic, MassAtOrigin	•	Replayer / Trace (empirical)	•
Scheduling disciplines			
INF, FCFS (+PR/PI)	•	LCFS (+PR/PI)	•
PS, DPS, GPS	•	LPS (limited PS)	•
SIRO, HOL	•	SJF, LJF	•
SEPT, LEPT	•	SRPT (+PRIO)	•
PSJF, FB, SETF, LRPT, FSP	•	EDD, EDF (deadline)	•
PAS (pass-and-swap)	•	POLLING	•
EXT (external self-loop)	•	Priority variants (`*PRIO`)	•
Routing, cache, advanced			
Routing PROB, RAND	•	RROBIN, WRROBIN	•

continued on next page

Feature	LDES	Feature	LDES
JSQ, KCHOICES	•	Cache repl. (RR/FIFO/SFIFO/LRU/HLRU/CLIMB/QLRU)	•
Delayed-hit retrieval	•	Load dependence	•
Retrial (orbit)	•	Balking	•
Reneging (impatience)	•	Multiserver ($c > 1$)	•
Batch arrivals	•	Server breakdowns	•
Slotted (discrete-time) mode	•	Busy-period orders	•
Cache item sizes / cost caps	•	`MAPt`, `PHt` (time-inhomogeneous)	•
Warm start (`initFromSolver`)	•	Marking-dependent firing rates	•
Layered networks and output			
LQN (host/task/entry/activity)	•	Sync / async calls	•
Activity precedence (SEQ/AND/OR)	•	Steady means (`getAvg`)	•
Transient (`getTranAvg`)	•	Rewards (`getAvgReward/getTranReward`)	•

7.7 Method algorithms

Table 7.2 gives high-level pseudocode for the simulation engine and its variants.

Table 7.2: High-level pseudocode of `SolverLDES` engines.

Engine	Pseudocode
steady-state	1. compile `sn` to station runtime objects with per-class SSJ samplers; seed streams from `seed`; inject the initial state 2. pop the next event from the SSJ future-event list (arrival, completion, routing, impatience, cache lookup, transition firing, modulation) 3. apply it to the station and schedule successor events; update time-weighted queue/utilization integrals, event counts, and per-job timestamps 4. repeat until `samples` events; remove warm-up; report Q, U, R, T
transient	as above but advance to the requested horizon and emit trajectories on the time grid
`parallel`	run independent replications on separate substreams; pool the estimators and report confidence intervals
reward	integrate user-defined state functionals along the sample path (`getAvgReward/getTranReward`)
LQN (`ssj_ln`)	simulate hosts, tasks, entries, activities, and synchronous/asynchronous calls directly on the LQN topology

7.8 Bibliographic notes

The SSJ library and its stream architecture are described by L'Ecuyer et al. [119]. Scheduling-policy analysis relevant to the implemented disciplines is surveyed in [142, 188], polling systems in [176], and MAP/trace-driven input modeling in [25, 38]. The simulator's role within LINE, including its validation methodology against JMT, is discussed in [30].

Chapter 8

SolverMAM

8.1 Overview

`SolverMAM` encodes matrix-analytic methods for queueing models with Markovian arrival processes (MAPs) and phase-type or MAP services [117, 140, 141]. Its core object is the quasi-birth-death (QBD) process: for a MAP/MAP/1 queue, the CTMC on levels (queue lengths) and phases has block-tridiagonal generator with blocks $(\boldsymbol{A}_0, \boldsymbol{A}_1, \boldsymbol{A}_2)$, and the stationary distribution is matrix-geometric, $\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n \boldsymbol{R}$, where $\boldsymbol{R}$ solves $\boldsymbol{A}_0 + \boldsymbol{R}\boldsymbol{A}_1 + \boldsymbol{R}^2\boldsymbol{A}_2 = \boldsymbol{0}$ [140]. Networks are handled by decomposition: each station is reduced to a quasi-birth-death queue, and inter-station traffic is propagated as a marked MAP (MMAP), iterating until the flows stabilize. This makes the solver the tool of choice for models whose temporal correlations (autocorrelated arrivals, non-renewal departures) invalidate product-form and two-moment decompositions.

The solver resides in `jline/solvers/mam/`, with QBD, fitting, and process-algebra algorithms in `jline/api/mam/` and `jline/api/map/`, several of which are ports of the BuTools [94] and SMCSolver/Q-MAM [17] libraries.

8.2 Analyzers and methods

`Solver_mam_analyzer` first recognizes closed-form cases: an isolated MAP/MAP/1 station routes to `Solver_mam_mapmap1_exact`, and an M/M/c($/K$) station to `Solver_mam_mmck_exact`. Otherwise it dispatches by method:

`default`, `dec.mmap` The MMAP decomposition pipeline (`Solver_mam_basic`, `Solver_mam_basic_mmap`): each station is analyzed as a MAP/MAP/c QBD, its departure process is approximated as an MMAP preserving marginal rates and lag correlations, and `Solver_mam_traffic` implements the flow operations (superposition, Bernoulli splitting, class marking) in the MMAP algebra [25, 93]. The iteration continues until arrival-process parameters converge.

`dec.source`, `dec.poisson`, `dec.source.mmap` Simplified decompositions in which internal flows are replaced by renewal (source-equivalent) or Poisson processes, providing cheaper and more robust,

if less accurate, baselines; `Qna_superpos` supplies the two-moment superposition used in hybrid schemes [111, 187]. `dec.source.mmap` keeps the source-equivalent flows but synchronizes fork-join sections through the `mmap_max` maximum operator on the sibling MMAPs; it is what the analyzer selects on a general fork-join topology.

`mna` The Matrix Network Analyzer (`Solver_mna`, `Solver_mna_open`, `Solver_mna_closed`), a decomposition algorithm for phase-type queueing networks that propagates full Markovian representations of the flows and supports both open and closed topologies [121, 122].

`inap`, `inaprc`, `inapinf`, `exact` The RCAT family (`Solver_mam_ag`, `Solver_mam_build_ag`), which composes the model as cooperating structured Markov processes in the sense of RCAT product-form theory [127, 128]. `inap` is the iterative numerical algorithm of Marin, Rota Bulò and Balsamo, alternating per-component solutions with reversed-rate updates until a fixed point; `inaprc` estimates each action's reversed rate by rate conservation without normalization, after the AutoCAT construction of Casale and Harrison [32]; `inapinf` solves the isolated open components matrix-geometrically on the infinite state space, reporting per-process geometric tail decays; and `exact` demands that the RCAT conditions hold and returns the product form itself. These four are also the only methods that read a G-network signal marking (`sn.issignal`): every other `MAM` decomposition would treat a signal as an ordinary customer, so the solver refuses a signal-bearing model by name unless one of them is requested.

`bgchain` A mixed-network solver (`Solver_mam_bgchain`) that treats the closed classes as a background modulating chain and the open classes as matrix-analytic queues driven by it. The closed population vector is the only bounded part of a mixed model, so it is a finite CTMC in its own right: `Mam_bgchain_ctmc` builds and solves it given the mean open occupancy, `Mam_bgchain_env` lumps it onto the closed occupancy of one station, and `Mam_bgchain_station` solves that station as a level-dependent QBD whose phase carries the number of closed jobs competing for its server; the mean open occupancy closes the fixed point. With $R > 1$ closed chains the state space is exponential in R, so one chain is tagged and carried exactly while the others are replaced by flow-equivalent aggregates of Chandy-Herzog-Woo type [39], grouped by similarity of per-station demand in `Mam_bgchain_groups`; each chain takes its turn as the tagged one. At a PS station the open service law is exponentialized first, which is the insensitivity property rather than an approximation, since the QBD tracks a single service phase per station. Purely open or purely closed models, class priorities and fork-join sections are refused by name.

`sync.faster`, `sync.slower` Synchronous approximations of a single M/G/c/K station (one open class, Poisson `Source`, one FCFS `Queue`, $c \leq K$, drop, deterministic or phase-type service) [77]. The c concurrent service clocks are synchronized into one batch, and the resulting Markov regenerative process is solved through its embedded chain. In the *faster* model an arrival finding a free server joins the running batch, which bounds Q, R and the loss from below; in the *slower* model it restarts the batch, which bounds them from above. `options.config.sync_phases` $= N \geq 1$ selects the synchronous phase expansion (deterministic service only): one server keeps its exact service time and the others count phases of length d/N on its lattice, so the rounding affects one phase and the

two bounds close in as N grows; $N = 0$ (default) is the approximation without expansion. A model outside the shape gate is refused by name, and `default` never resolves to either method.

Batch arrivals and services are covered by the BMAP/MAP/1 and MAP/BMAP/1 handlers (`Solver_mam_bmap_map_1`, `Solver_mam_map_bmap_1`) [25, 141]. Retrial queues with orbit are solved by `Solver_mam_retrial` and the algorithm `Qsys_bmapphnn_retrial`, using the level-dependent constructions of retrial theory [146]. Fork-join stations use `Solver_mam_fj`. Priority queues are treated through `PriorityAnalysis` with the ETAQA-style aggregate solutions of Riska and Smirni [150, 151].

8.3 Level-dependent and transient QBDs

Load-dependent stations yield level-dependent QBDs (LDQBDs), solved by `Solver_mam_ldqbd` with the matrix-continued-fraction construction of Bright and Taylor [23]; the `statevec` variant returns the full level-phase distribution for consumption by `SolverENV` (Chapter 12). The method serves single-class models in two regimes, a closed one with an infinite-server delay and a finite population and an open one with a Source feeding the queue and a level index truncated at `options.cutoff`, and it is also invoked automatically on retrial topologies, where the orbit retrial rate is what makes the generator level-dependent. Multiserver phase-type service is exact rather than aggregated: the block builder `ldqbd_mphc` carries, inside each level, the *multiset* of the phases occupied by the $\min(n, c)$ busy servers, of order $\binom{m_s+k-1}{k}$ rather than m_s^k [7] (the companion `ph_multisets` enumerates and indexes the multisets), and an optional per-level scaling vector multiplies the station's total service rate, so limited load dependence composes with the exact multiserver chain. Transient analysis is provided by `Solver_mam_transient_qbd` and `Solver_mam_ldqbd_transient`, which compute time-dependent level distributions by Laplace-transform methods with numerical inversion, automatically selected by `getTranAvg` for MAP/MAP/1($/N$) stations. First-passage metrics (response-time distributions, `getCdfRespT`) are computed by `Solver_mam_passage_time` from the fundamental-period matrices $\boldsymbol{G}$ and $\boldsymbol{U}$ of the QBD [117].

The QBD algorithms in `jline/api/mam/` implement cyclic reduction and logarithmic reduction for the $\boldsymbol{R}$ and $\boldsymbol{G}$ matrices with the numerically stable variants of [17]; spectral shortcuts are used when the phase spaces are small. MAP fitting utilities in `jline/api/map/` include moment and autocorrelation fitting, notably the KPC-Toolbox recipes [38], two-phase MMPP fitting [92, 93], and the count-process moment matching used by trace-driven workflows [25].

8.4 Architectural flow

Figure 8.1 shows the decomposition path; exact single-station cases bypass the traffic iteration entirely.

8.5 Implementation notes

Phase-space growth is the main cost driver: MMAP superposition multiplies phase dimensions, so the traffic handlers compress departure processes back to small-order MMAPs by moment and correlation matching

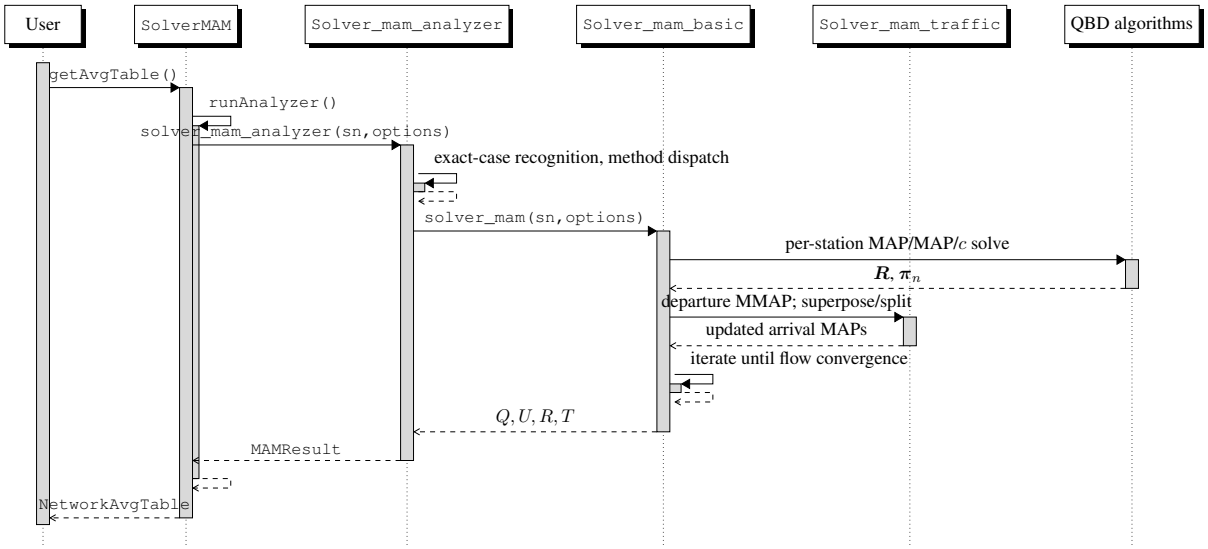


Figure 8.1: Sequence diagram of `SolverMAM` with the MMAP decomposition method.

before propagation, an accuracy-cost trade-off that dominates the method's behavior on large topologies. Default orders and compression tolerances live in `MAMOptions`. Closed networks impose population constraints handled by `Solver_mna_closed` through a population-coupling fixed point. Known accuracy caveats are recorded in the test suite; notably, service-process autocorrelation propagation through multi-hop routes is validated against JMT simulation.

8.6 Feature and method support

Table 8.1 maps modelling features to the `SolverMAM` methods. Per the method-aware gate (`supportsModelMethod`), `mna` rejects mixed open/closed models and `ldqbd` requires a single-class model; the `retrial` and `fj` columns are the retrial-orbit and fork-join analyzers, selected by topology (a fork-join model is routed to `fj`, not rejected), and `st.exact` covers the closed-form single-station recognizers (`Solver_mam_mapmap1_exact`, `Solver_mam_mmck_exact`), reached regardless of the requested method; the RCAT `exact` shares the `inap` column. `dec.mmap` (default) propagates full marked-MAP flows; `dec.source` replaces internal flows by renewal approximations, so it captures Markovian arrivals only approximately. General distributions enter through MAP fitting (`o`).

Table 8.1: Feature support of `SolverMAM` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	dec.mmap	dec.source	mna	inap	ldqbd	st.exact	retrial	fj
Job classes								
Open class	●	●	●	●	○	●	●	●
Closed class	●	●	●	●	●			●
Mixed open/closed	●	●		○				○
Single-class only					●			
Class switching	●	●	●	●				
Node types								
Source, Sink	●	●	●	●	●	●	●	●
Queueing station	●	●	●	●	●	●	●	●
Delay (infinite server)	●	●	●	●	●	●		●
Multiserver (MAP/MAP/c)	●	●	●	●	●	●		
Fork / Join								●
Service and arrival distributions								
Exponential	●	●	●	●	●	●	●	●
Phase-type (Erlang, HyperExp, Coxian, APH, PH)	●	●	●	●	●	●	●	●
MAP, MMPP2, DMAP	●	○	●	●	●	●	●	○
Batch (BMAP, MMAP)	●		○	○		●		
Matrix-exponential (ME), RAP	●	○	○	○		●		
General (Gamma, Weibull, Lognormal, LogLogistic, MassAtOrigin, Pareto, Uniform, Det)	○	○	○	○	○	○		
Scheduling disciplines								
INF (delay)	●	●	●	●		●		●
FCFS	●	●	●	●	●	●	●	●
PS	○	○	○	○		○		
HOL (priority)	○					○		
Advanced features								
Retrial (orbit)							●	
Load dependence (LDQBD)					●			
Analysis and output								
Steady-state means (`getAvg`)	●	●	●	●	●	●	●	●
Response-time CDF (`getCdfRespT`)	○				○	●		
Transient averages (`getTranAvg`)	○				○	●		

8.7 Method algorithms

Table 8.2 gives high-level pseudocode for each `SolverMAM` method.

Table 8.2: High-level pseudocode of `SolverMAM` methods.

Method	Pseudocode
`dec.mmap`	1. for each station build the QBD blocks $(\boldsymbol{A}_0, \boldsymbol{A}_1, \boldsymbol{A}_2)$ from arrival and service MAPs 2. solve $\boldsymbol{A}_0 + \boldsymbol{R}\boldsymbol{A}_1 + \boldsymbol{R}^2\boldsymbol{A}_2 = \boldsymbol{0}$ (cyclic/logarithmic reduction); $\boldsymbol{\pi}_{n+1} = \boldsymbol{\pi}_n\boldsymbol{R}$ 3. approximate the departure process as an MMAP; superpose/split/mark into downstream arrivals (compress order) 4. iterate steps 1–3 until the arrival descriptors converge
`dec.source`	as `dec.mmap` but replace internal flows by renewal (source-equivalent) or Poisson processes; cheaper, more robust, less accurate
`mna`	propagate full PH/MAP flow descriptors through the network (open or closed topology); closed models add a population-coupling fixed point
`ldqbd`	level-dependent QBD via the matrix-continued-fraction construction (Bright–Taylor); single class; multiserver PH service carried exactly on the phase-multiset coordinate (`ldqbd_mphc`); returns the level-phase distribution
`inap`, `exact`	compose the model as cooperating processes; iterate per-component solutions with reversed-rate updates to a fixed point (`inap`/`inaprc`/`inapinf` variants), or return the certified product form (`exact`)
`retrial`	build the level-dependent QBD with an orbit level; solve the matrix-geometric orbit/queue distribution
`fj`	model the fork-join synchronization as a QBD over sibling-task states and solve the join completion
`sync.faster`, `sync.slower`	1. check the M/G/c/K shape gate; with `sync_phases` $= N \geq 1$ expand the state to (owner busy, phase multiset of the other servers, waiting jobs) on a d/N lattice 2. build the Markov regenerative process with one synchronized clock, an arrival finding a free server joining (faster) or restarting (slower) the running batch 3. solve the embedded chain, with Ω and Ψ evaluated by uniformization at rate λ, and convert to time averages through the conversion matrix 4. report Q, U, R, T and the loss probability; faster gives the lower and slower the upper value

8.8 Bibliographic notes

Matrix-geometric methods originate with Neuts [140, 141]; algorithmic treatments and software are covered by Latouche and Ramaswami [117] and by the SMCSolver/Q-MAM tools [17], and the MAMSolver line [150, 151]. LDQBD algorithms are due to Bright and Taylor [23]. MAP/MMAP modeling and fitting are surveyed in [25], with fitting recipes in [38, 92, 93]. Decomposition of non-renewal networks descends from QNA [111, 187]; the MNA algorithm is introduced in [121, 122]. Retrial queue theory is surveyed in [146]; entropy-based alternatives for general networks appear in [109].

Chapter 9

SolverMVA

9.1 Overview

`SolverMVA` encodes the mean-value analysis paradigm for product-form queueing networks and its approximate extensions to non-product-form models. Exact mean-value analysis (MVA) computes mean performance metrics of closed product-form networks by a recursion over populations without ever evaluating the normalizing constant [148, 149]. Approximate mean-value analysis (AMVA) replaces the population recursion with a fixed-point iteration over interpolated queue lengths, trading exactness for a cost that is independent of the population size [10,40,165]. On top of these two pillars, the solver hosts a family of specialized analyzers for bounds, caches, polling systems, isolated queueing systems, load-dependent stations, and open-network decomposition.

The solver resides in `jline/solvers/mva/` with the analyzer classes in `analyzers/` and the method handlers in `handlers/`. Its numerical core is the `PFQN` API (`jline/api/pfqn/`), complemented by `NPFQN`, `QSYS`, `CACHE`, and `POLLING` algorithms.

9.2 Software architecture

Table 9.1 lists the main classes. The entry point `SolverMVA.runAnalyzer()` inspects `sn` and routes the request to one of eight analyzers; each analyzer selects a handler according to `options.method`.

9.3 Default method selection

When `options.method='default'`, `Solver_mva_analyzer` applies the following policy, in order:

1. if the model is a single-chain closed network with blocking-after-service (BAS) drop rules, run the `sqd` handler;

2. if the model mixes open and closed classes, has only single-server stations, satisfies the BCMP product-form conditions [11], and the closed populations are integral, run exact mixed MVA

Component	Class	Role
Solver	`SolverMVA`	feature admission, analyzer dispatch, result handles
Options	`MVAOptions`	method, tolerance `iter_tol`, iteration cap `iter_max`
Result	`MVAResult`, `ProbabilityResult`	metric matrices, iteration counts, logNormConst
Analyzer	`Solver_mva_analyzer`	default dispatch for queueing networks
Analyzer	`Solver_mvald_analyzer`	load-dependent networks
Analyzer	`Solver_mva_bound_analyzer`	non-iterative bound methods
Analyzer	`Solver_mva_qsys_analyzer`	isolated open queueing systems
Analyzer	`Solver_mva_cache_analyzer`	standalone caches
Analyzer	`Solver_mva_cacheqn_analyzer`	cache-queueing networks
Analyzer	`Solver_mva_retrieval_analyzer`	delayed-hit retrieval caches
Analyzer	`Solver_mva_polling_analyzer`	polling stations
Handler	`MVARunner`	orchestration, fork-join transformations
Handler	`Solver_mva`, `Solver_mvald`	exact MVA, exact load-dependent MVA
Handler	`Solver_amva`, `Solver_amvald`	AMVA fixed-point schemes
Handler	`Solver_qna`	open-network decomposition (QNA)
Handler	`Solver_sqd`	blocking-after-service single-chain models
Handler	`Solver_mva_lcfsqn`	LCFS-PR closed networks

Table 9.1: Main classes of `SolverMVA`.

(`pfqn_mvamx`) [24], since the AMVA linearizer is known to inflate closed-class response times by double-counting open-class interference;

3. if the model is product-form with at most 4 chains and at most 20 jobs, run exact MVA;
4. otherwise, run AMVA, whose internal default is the linearizer family described below.

This policy reflects the general design rule of LINE that exact algorithms are preferred whenever their combinatorial cost is provably small, and asymptotically scalable approximations are used otherwise.

9.4 Exact MVA methods

The `exact` (alias `mva`) method evaluates closed product-form networks using the arrival theorem [149,167]: an arriving class-r job observes the network in equilibrium with population $\boldsymbol{N} - \mathbf{1}_r$. For single-server FCFS/PS/LCFS-PR stations and infinite-server delays, the recursion is

$$R_{ir}(\boldsymbol{N}) = D_{ir}\,[1 + Q_i(\boldsymbol{N} - \mathbf{1}_r)] \quad \text{(queueing stations)}, \qquad R_{ir}(\boldsymbol{N}) = D_{ir} \quad \text{(delay stations)},$$
$$T_r(\boldsymbol{N}) = \frac{N_r}{Z_r + \sum_i R_{ir}(\boldsymbol{N})}, \qquad Q_{ir}(\boldsymbol{N}) = T_r(\boldsymbol{N})\, R_{ir}(\boldsymbol{N}), \tag{9.1}$$

iterated over all population vectors $\mathbf{0} \le \boldsymbol{n} \le \boldsymbol{N}$ in lexicographic order. The implementation is `pfqn_mva`; its time complexity is $O(MR\prod_r(N_r+1))$ and its space complexity is reduced by retaining only the layers of the population lattice needed by the recursion. Multiserver stations are handled by `pfqn_mvams`, which augments (9.1) with the exact marginal-probability recursion of queue-dependent MVA [148]; mixed open-closed models are solved by `pfqn_mvamx`, which first computes the open-class utilizations and then inflates closed-class demands by $(1 - U_i^{\text{open}})^{-1}$ following the exact BCMP decomposition [11,24].

Exact MVA for load-dependent stations is provided by the `Solver_mvald` handler through `pfqn_mvald`, which propagates the full marginal queue-length distributions $\pi_i(n \mid \boldsymbol{N})$ across populations.

Since the load-dependent recursion is numerically unstable in the presence of near-zero marginal probabilities [33, 148], the handler monitors the probability mass and rescales it, flagging results where stabilization was applied.

9.5 AMVA methods

The `Solver_amva` and `Solver_amvald` handlers implement the approximate MVA family. All methods share the fixed-point skeleton: guess queue lengths Q_{ir}, estimate the arrival-instant queue lengths A_{ir} seen by a class-r arrival at station i, evaluate response times, apply Little's law, and repeat until the queue lengths change by less than `iter_tol`. The methods differ in the interpolation used for A_{ir}:

`bs` Bard-Schweitzer proportional estimator $A_{ir} = Q_i(\boldsymbol{N}) - \frac{Q_{ir}(\boldsymbol{N})}{N_r}$ [10, 165]. Cost $O(MR)$ per iteration; existence and uniqueness of the fixed point are known for single-class models.

`lin` The Linearizer of Chandy and Neuse [40], a three-level scheme that estimates the fractional deviations δ_{irs} between populations $\boldsymbol{N}$ and $\boldsymbol{N} - \mathbf{1}_s$ and is typically accurate to a fraction of a percent. Multiserver stations use the extensions of Conway [46] and the multiserver AMVA corrections of Suri et al. [175] (`pfqn_linearizerms`, `pfqn_conwayms`); mixed models use `pfqn_linearizermx`. The general-form variants `gflin` and `egflin` (`pfqn_gflinearizer`, `pfqn_egflinearizer`) parameterize the deviation update in the spirit of the unified framework of Cremonesi, Schweitzer and Serazzi [51], which subsumes Linearizer variants under a common template; `fli` is the fraction-line one-step variant from the same framework.

`aql` The aggregated-queue-length method of Zahorjan, Eager and Sweillam [189] (`pfqn_aql`), which carries $R + 1$ population points and a correction $\gamma_{ir} = Q_i(\boldsymbol{N})/\sum_r N_r - Q_i(\boldsymbol{N} - \mathbf{1}_r)/(\sum_r N_r - 1)$ removing the Schweitzer proportionality error. It lives only in the product-form, load-independent branch of `Solver_amva` and rejects multiserver stations, so `listValidMethods` advertises it only when the model qualifies; `qdaql` has no handler arm and is refused by name rather than silently computing `qd`.

`qd`, `qdlin`, `qli` Queue-dependent AMVA (QD-AMVA) [35]: the arrival estimator is made a function of the local queue length to interpolate load-dependent rates, yielding methods that handle multiserver and limited load-dependent stations within the fixed-point scheme (all three are assembled in `Solver_amvald`; `qdlin` combines this with Linearizer deviations, `qli` is the queue-line variant).

`sqni` A single-queue network interpolation built on the QRF optimization view of interpolated networks [31] (`pfqn_sqni`).

`schmidt`, `schmidt-ext` The exact-arrival-theorem AMVA of Schmidt for networks with FCFS multiserver stations (`pfqn_schmidt_amva`), including its extended variant.

`ab` The Akyildiz-Bolch load-dependent approximation [2] (`pfqn_ab_amva`).

`tay` The Tay-Suri arrival-instant approximation [178], which derives the auxiliary queue lengths A_{ir} from throughput elasticities with respect to the population, solved as R linear equations per station-class

pair rather than by the Bard-Schweitzer or Linearizer interpolations (`pfqn_tay`). Restricted to single-server stations.

`qsa` The queue-shift approximation of Schweitzer, Serazzi and Broglia [166] (`pfqn_qsa`). Where Linearizer extrapolates the fractional deviations, QSA extrapolates the *absolute* shift of the aggregate queue length, $Y_{ri}(\boldsymbol{N}) = 1 + Q_i(\boldsymbol{N} - \mathbf{1}_r) - Q_i(\boldsymbol{N})$, one unknown per station rather than per station-class, imposing the core equation at $\boldsymbol{N}$, at every $\boldsymbol{N} - \mathbf{1}_s$ and, through an affine extrapolation, at every $\boldsymbol{N} - \mathbf{1}_s - \mathbf{1}_t$, and solving the whole system by damped Newton. Its advantage over Linearizer grows with the population and holds when the per-station service time is class-independent; restricted, like `aql`, to strict product-form, load-independent, single-server models.

`lcp`, `chow`, `pamb/pami/pamt`, `clust`, `dmlin` The classical AMVA variants, each behind its own `pfqn_*` kernel: Bard's large-customer-population interpolation [10] (`pfqn_lcp`), JMT-compatible Chow using the same full-population aggregate-queue fixed point [43] (`pfqn_chow`), the Hsieh-Lam proportional approximation methods [96] in their bottleneck, intermediate and throughput variants (`pfqn_pam`), the de Souza e Silva-Lavenberg-Muntz clustering approximation [54] (`pfqn_clust`), which partitions the stations into subnetworks solved exactly and coupled through flow-equivalent servers, and the de Souza e Silva-Muntz improved Linearizer [56] (`pfqn_dmlin`), the variant JMT exposes as `jmva.dmlin`.

`scat` The Neuse-Chandy SCAT approximation [139], sharing the Bard-Schweitzer proportional correction $\Delta_{irs} = Q_{ir}(\boldsymbol{N} - \mathbf{1}_s)/(N - \mathbf{1}_s)_r - Q_{ir}(\boldsymbol{N})/N_r$ with Linearizer but refreshing it once per outer fixed-point pass rather than in Linearizer's three nested levels. Accuracy sits between `bs` ($\Delta \equiv 0$) and `lin`, at roughly a third of the per-iteration cost of the latter. Restricted, like `aql`, to strict product-form, load-independent, single-server models.

Priority classes are handled inside `Solver_amvald` by shadow-server reductions in the tradition of Sevcik [167] and the AMVA priority approximation of Eager and Lipscomb [62]. First-come first-served stations with non-exponential service are corrected inside `Solver_amvald_forward` itself, under `options.config.highvar='hvmva'`, which blends the product-form residence time with the c_{ir}^2-weighted term $\sum_s S_{is}U_{is}(1 + c_{is}^2)/2$; comparative background on such heuristics is given in [21, 184].

Load-dependent AMVA (`Solver_amvald`) supports limited load dependence (LLD) and class dependence (CD), following the factorization approach for load-dependent normalizing constants of [33]. The handler reads the scaling data in `sn` (`lldscaling`, `cdscaling`) and embeds them in the arrival estimator. Class dependence is per class: `cdscaling`$\{i\}$ returns either a scalar shared by all classes or the vector $[\beta_{i,1}(n), \ldots, \beta_{i,R}(n)]$, which subsumes the joint-dependence tables of earlier releases.

9.6 Bound methods

`Solver_mva_bound_analyzer` serves the non-iterative bound methods, each in `.upper`/`.lower` pairs: `aba` (asymptotic bound analysis) and `bjb` (balanced job bounds) [118], `pb` (proportional bounds), `gb` (geometric bounds, a noniterative technique with error guarantees on throughput [34], algorithms `pfqn_qzgblow`/`pfqn_qzgbup`), `sb` (scaled bounds), and `harel` (the convexity-based bounds of Harel,

Namn and Sturm [86], algorithm `Pfqn_harel_bounds`). These methods require no iteration and are mainly intended for optimization loops, where monotonicity and guaranteed sidedness matter more than accuracy [29].

9.7 Specialized analyzers

Queueing systems. When the model consists of a single open queue (Source, Queue, Sink), `Solver_mva_qsys_analyzer` bypasses network iteration and evaluates closed-form or quadrature results from the `QSYS` API: M/M/1, M/M/k, M/G/1 via the Pollaczek-Khinchine formula, G/M/1, and G/G/1 through a portfolio of approximations selectable by method name, including Allen-Cunneen, Kraemer and Langenbach-Belz [111], Kobayashi's diffusion approximation [107], Kingman bounds, Marchal, Heyman, Gelenbe, Kimura, and Myskja variants, plus Cosmetatos and Whitt corrections for G/G/k. Scheduling-sensitive metrics for M/G/1 (SRPT, FB, priorities) follow the exact transform analyses surveyed in [142, 188]; the SRPT mean response time is computed by the Schrage-Miller quadrature [164].

Polling. `Solver_mva_polling_analyzer` evaluates exhaustive, gated, and decrementing (limited) polling stations with switchover times using the classical mean-delay results collected by Takagi [176], implemented in `jline/api/polling/`.

Caches. `Solver_mva_cache_analyzer` computes cache hit and miss probabilities via fixed-point iterations over the cache MVA equations, using the `CACHE` API algorithms (Che-style characteristic-time approximations [41] and list-based replacement models [52]); `Solver_mva_cacheqn_analyzer` embeds these probabilities into the surrounding queueing network by class switching, iterating between the cache solution and the network solution until the miss rates converge. The `Solver_mva_retrieval_analyzer` extends this scheme to delayed-hit retrieval caches, in which misses trigger a back-end fetch and concurrent requests for the same item coalesce.

Open network decomposition. The `qna` method (`Solver_qna`) implements two-moment parametric decomposition for open networks of G/G/1 and G/G/k queues in the style of the Queueing Network Analyzer of Whitt [187], with the Kraemer and Langenbach-Belz waiting-time approximation [111] at each station, and flow superposition, splitting, and departure SCV propagation following [129, 187].

Robust queueing. `Solver_rqt` (`pfqn_rqt`/`solver_rqt`) implements Robust Queueing Theory [9] for single-class open networks fed from a Source, dispatched directly by `mva_dispatch` rather than through the AMVA fixed point of §9.5. It replaces the stochastic primitives by polyhedral uncertainty sets and reports a worst-case system time rather than an expectation, in four steps that follow Section 7.2 of the reference: (1) the external stream is assigned $\Gamma_a = \sigma_a$; (2) `npfqn_traffic_rqt` composes the robust-Burke queue-passage, superposition and thinning theorems (Theorems 4-6 and 10) along the visit-ratio path to get the effective $(\lambda, \Gamma_a, \alpha)$ at every node, skipping the reference's own path-enumeration step since LINE already has the visit ratios; (3) `qsys_gigk_rqt_gamma` regresses the service variability parameter Γ_s against simulation (Table 1 of the reference, regimes `independent`/`normal`/`pareto`), restoring a factor of 2 that is

missing from the published formula – without it the M/M/1 error would be roughly 40% at $\rho = 0.9$ against the reference's own stated $\leq 9.5\%$; (4) `qsys_gigk_rqt` evaluates the closed-form worst-case bound of Theorem 3/8, or, when `options.config.rqt_exact` is set, the exact worst case of eq. 45 by a one-dimensional integer optimization. Being regressed in heavy traffic, the method is markedly less accurate at low utilization (about 8% error at $\rho = 0.9$ on M/M/1, over 50% at $\rho = 0.5$), so `mva_carries_interlock` (§9.10) explicitly excludes `rqt` from carrying an interlock correction matrix, since it reaches neither the exact nor the AMVA-forward kernel that correction assumes.

Shortest job next. `Solver_mva_sjn_analyzer` (`pfqn_mvasjn`/`pfqn_amvasjn`) is dispatched automatically, ahead of any other AMVA routing, whenever a closed model contains a single-server station with non-preemptive shortest-job-next scheduling (`SchedStrategy.SJF`). Every other station in scope must be single-server FCFS, PS, LCFS-PR, SIRO, or a delay station. Unlike an ordinary AMVA station, an SJF station is not carried by a scalar queue length: it is carried by the conditional waiting time $W(x, n)$ of a tagged job of size x at population n [100],

$$W(x,n) = \frac{(1+C^2)\,s\,U(n-1)/2 + X(n-1)\,\phi(x,n-1)}{1 - X(n-1)\,\theta(x)}, \qquad \theta(x) = \int_0^x t f(t)\,dt, \quad \phi(x,n) = \int_0^x W(t,n)\,t f(t)\,dt,$$

with the station response time $R(n) = s + \int_0^\infty W(x,n) f(x)\,dx$. The service density f is not an input: it is rebuilt from the station's mean and SCV by a two-moment fit (branching Erlang below $C^2 = 1$, balanced-means hyperexponential above), which keeps θ and the tail integrals closed form; the x-integrals themselves run on a fixed composite-Simpson grid (`sjn_ns` panels over $[0, \texttt{sjn_lfactor} \cdot \max_r s_r]$) with an exponential-tail closure $W = a - b\exp(-c(x - L_x))$ past the grid boundary. `sjn.mva` steps the full population lattice ($\prod_r (N_r + 1)$ states) and is exact for the equation; `sjn.amva` replaces it with a Schweitzer fixed point applied to the size-resolved queue-length density $\lambda_k W_k(x) f_k(x)$, deflating the tagged class by $(N_r - 1)/N_r$ and integrating over x to recover the ordinary rule at the other stations – it must be initialized from `pfqn_bs`, since a light-load guess can put the deflated utilization above one, where the fixed point has no solution. Multiclass models are read two ways, selected by whether the classes carry distinct priority levels: with none, every job is compared by size directly (Kant eq. 6, generalized); with distinct levels, SJN applies within a class under non-preemptive priority across classes (Kant method A, eq. 21). Because the equation is derived for an open system, its denominator $1 - U(x)$ can drive a congested station's residence time below its true value and lose the recursion's solution entirely (Kant's "no results" regime); `sjn_cap` pre-empts this by bounding the station's utilization at `sjn_umax` through a bisected inflation of its waiting time, never by capping throughput directly, so the population balance $X(Z + \sum_m C) = N$ still holds exactly and a binding cap only warns rather than silently discarding load.

9.8 Fork-join transformations

Fork-join subnetworks are handled by a model transformation applied before the MVA algorithms are invoked: the `ht` (`heidelberger-trivedi`) method implements the classical asynchronous-task transformation of Heidelberger and Trivedi [89], while `mmt` (the default) and `fjt` implement mixed-model transformations that replace fork-join sections with auxiliary open classes calibrated iteratively [58]. Under either

transformation every fork becomes a router, every join a zero-service delay, and the parallelism is carried by auxiliary open classes of arrival rate $(\phi_r - 1)\lambda_r$, where ϕ_r is the fanout of the fork feeding auxiliary class r. The transformation loop re-invokes the analyzer on each transformed model, recomputes the synchronization delays from the resulting queue lengths and throughputs, updates λ_r, and terminates when the queue lengths stop moving under a mixed absolute/relative test.

A solver-agnostic driver. The loop reads no solver internal: it consumes only the metric matrices Q, U, R, T, C, X of an inner solve on the transformed (fork-free) model. It is therefore implemented once per codebase, outside any single solver, as `@NetworkSolver/fjFixedPoint.m` in MATLAB, `jline.solvers.fj.FJFixedPoint` in the Java library, and the `ForkJoinDriverMixin` of `line_solver.solvers.fork_join_driver` in Python. `SolverMVA` and `SolverNC` are the two consumers: each supplies an inner solve and inherits fork-join support, so a fork-join model may be analyzed either by mean-value analysis or by the normalizing-constant analyzers of Chapter 10 without duplicating the fixed point. On a model without forks the driver runs the inner solve exactly once and returns it unchanged.

The callback contract is the analyzer dispatch itself. In MATLAB and Java it is a function handle respectively an `InnerSolve` interface returning the metric matrices; in Python, where the loop constructs its inner solver rather than calling an analyzer, the same split is expressed by four overridable seams (`_fj_inner_solver`, `_fj_publish`, `_fj_result_dict`, `_fj_network_type`), the last two adapting between the result containers the two solvers use. The retained state, the cached transformation and the last iterate of λ_r, lives on the solver base class, so an outer fixed point such as `SolverLN` warm-starts the fork-join loop instead of restarting it from scratch at every outer iteration.

On an open transformed model the two consumers agree closely, since both apply exact open-network formulas inside the same loop; on closed models both are approximate, the MVA route through its AMVA iteration and the NC route through the auxiliary open classes, whose arrival rates start at the numerical floor `GlobalConstants.FineTol`.

Quorum joins and variable forking levels. A *partial* join fires on the first k of its n siblings and discards the rest. Both transformations charge the k-th order statistic $E[X_{(k)}]$ of the branch completion times at the join instead of $E[\max]$ (`fj_ordstat_exp`), and the request tail is read from the k-of-n approximation `fj_tail_ordstat` behind the `'forktail'` percentile method, which otherwise returned the AND-join percentiles whatever k was. The feature is declared as `JoinPartial` by `SolverMVA`, `SolverNC`, `SolverJMT` and `SolverLDES`, and refused by name by `SolverCTMC`, `SolverSSA` and `SolverMAM`, whose exact state construction fixes the sibling set when the state space is built. The loss table reports a join row as $\max(0, \texttt{ArvR} - k\,\texttt{Tput})$, the discarded siblings being a genuine loss, where the plain difference reported a ratio of $(n-1)/n$ at every join including standard ones.

A fork may also emit a number of tasks that varies by link, by class or by draw. The scalar fanout ϕ_r that the transformations read is then the *expected* degree, with the branch activation probability folded into it, and the MMT correction is linear in it, so the transformation sees a well-defined mean model; a random degree is exact only under the simulators, which draw it at the fork epoch. The synchronization is taken over the $w{\cdot}B$ sibling multiset rather than by scaling $E[X_{(B)}]$ by w, which is what makes an unequal per-link count consistent with the order statistic charged at the join.

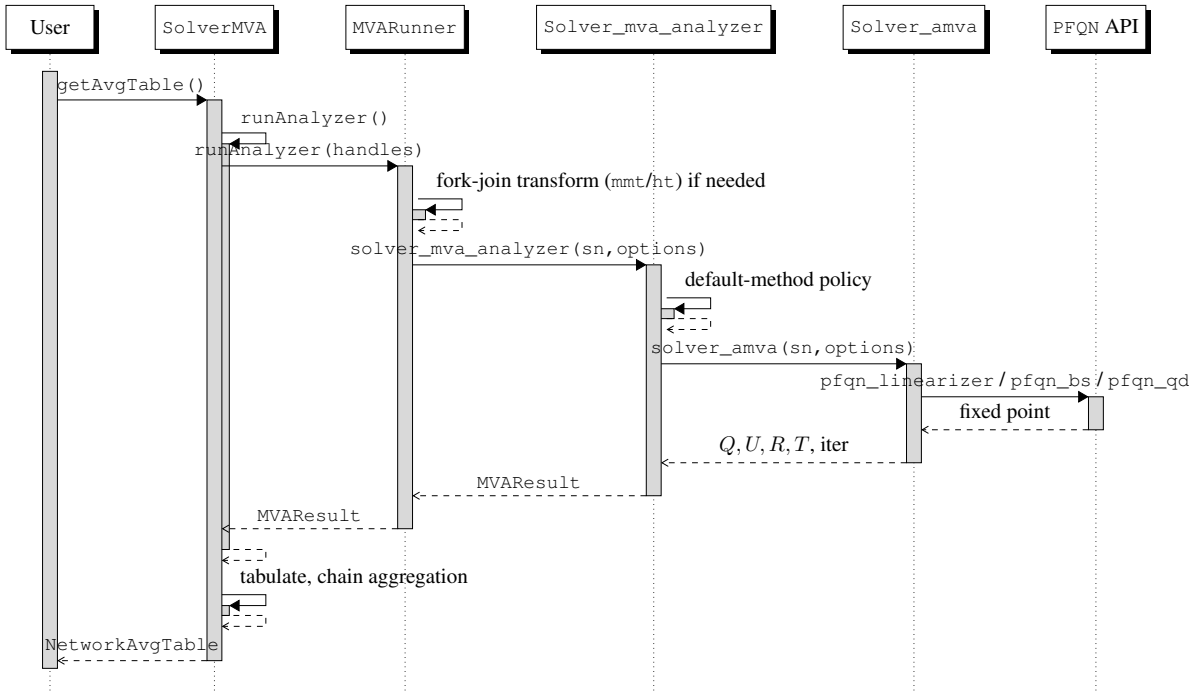


Figure 9.1: Sequence diagram of `SolverMVA` for a default steady-state request.

9.9 Architectural flow

Figure 9.1 shows the sequence of calls for a steady-state average request with the default method on a closed non-product-form network, the most common execution path.

9.10 Implementation notes

The exact and approximate algorithms operate on demands D_{ir} obtained from `sn` as V_{ir}/μ_{ir} after chain-level visit computation (`sn.visits`); class-to-chain aggregation and disaggregation are performed by the `SN` API so that all PFQN algorithms see one class per chain. Numerical safeguards include demand perturbation to break exact ties among stations (which can stall Linearizer convergence), utilization capping at 1 for unstable open models, and the DTMC visit solver ordering (`dtmc_solve` primary, reducible fallback) that guarantees consistent visit ratios in fork-join models. Iteration counts and the achieved residual are reported in `MVAResult` for reproducibility.

`mva_carries_interlock` guards a distinct hazard that only arises when `SolverMVA` is called as the layer solver of `SolverLN` (§13.6): whether the MVA path a given model is *already* dispatched to is one of the two kernels able to carry a class-level interlock correction matrix (exact `pfqn_mva` and the `Solver_amvald` forward step) without silently being redirected to a different algorithm to accommodate it. It returns false for `mvac`, `sqd`, `sum`, `qna`, `rqna` and `rqt`, which reach neither kernel, so a model routed to any of them carries no interlock term even when the layer declares one; `mva_is_bas_model` makes the

same check for closed single-chain BAS-blocking models routed to `solver_sqd`, which has no interlock term either. The reason this is a gate rather than a forced substitution is that swapping the underlying MVA algorithm to accommodate the correction – rather than declining to apply it – can turn a converging `SolverLN` Picard iteration into a limit cycle, because the perturbation is the algorithm swap itself, not the correction's numeric magnitude (observed as a period-17 oscillation on a non-converging layered benchmark even when the interlock matrix was scaled to near zero).

9.11 Feature and method support

Table 9.2 maps modelling features to the `SolverMVA` methods. A cell is marked • when the method admits and handles the feature and ○ when it does so approximately or only conditionally; a blank cell means the feature is not admitted by that method or is routed to a different analyzer. Columns `cache` and `cacheqn` denote the standalone cache analyzer (`Solver_mva_cache_analyzer`) and the cache-queueing-network analyzer (`Solver_mva_cacheqn_analyzer`, which also drives delayed-hit retrieval), and `poll` the polling analyzer, and `sjn` the shortest-job-next analyzer; all four are selected automatically by topology. Per the method-aware gate, `qna` and `rqna` are open-only (they drop closed and self-looping classes), and `rqna` additionally admits Markovian arrival processes; `sqd` and the bound methods are single-chain closed, and `sjn` is closed-only for the opposite reason, the conditional waiting time being a population recursion. Exact MVA is exact for phase-type service under insensitive disciplines (PS, LCFS-PR, INF) but approximate for FCFS with non-exponential service; AMVA is approximate in general.

Table 9.2: Feature support of `SolverMVA` methods (• supported, ○ approximate/conditional, blank unsupported).

Feature	exact	amva	qna	rqna	sqd	bounds	cache	cacheqn	poll	sjn
Job classes										
Open class	•	•	•	•			•	•	•	
Closed class	•	•			•	•	•	•	•	•
Self-looping class	•	•			•	•	•	•	•	
Mixed open/closed	•	•						•		
Class switching	•	•	•	•			•	•		
Priority classes (HOL)		○								○
Node types										
Source, Sink	•	•	•	•			•	•	•	
Queueing station	•	•	•	•	•	•		•	•	•
Delay (infinite server)	•	•	•	•		•		•		•
Multiserver ($c > 1$)	•	•	•	○		○				
Fork / Join	•	•								
Cache node							•	•		
Service and arrival distributions										
Exponential	•	•	•	•	•	•	•	•	•	•
Phase-type (Erlang, HyperExp, Coxian, APH)	•	•	•	•	○	•	•	•	○	○
General two-moment (Det, Uniform, Pareto, Weibull, Lognormal, LogLogistic, MassAtOrigin)	○	○	•	•	○	○	○	○	○	○
Batch (BMAP)	○	○	○	○						
Markovian (MAP, MMPP2, MMAP, RAP)				•						

continued on next page

Feature	exact	amva	qna	rqna	sqd	bounds	cache	cacheqn	poll	sjn
Scheduling disciplines										
INF (delay)	●	●	●	●		●		●		●
FCFS	○	●	●	●	●	●		●		●
PS	●	●				●		●		●
DPS		○								
LCFS, LCFS-PR	●	●				●		●		●
SIRO	●	●				●		●		●
POLLING									●	
SJF (shortest job next, single server)										●
Routing strategies										
Probabilistic (PROB)	●	●	●	●	●	●	●	●	●	●
Random (RAND)	●	●	●	●	●	●	●	●	●	●
Cache and advanced station features										
Cache replacement (RR, FIFO, LRU)							●	●		
Delayed-hit retrieval							○	●		
Load dependence	●	●								
Finite-capacity blocking (BAS)					●					
Analysis and output										
Steady-state means (`getAvg`)	●	●	●	●	●	●	●	●	●	●
Bounds (upper/lower)						●				

9.12 Method algorithms

Table 9.3 gives high-level pseudocode for each `SolverMVA` method.

Table 9.3: High-level pseudocode of `SolverMVA` methods.

Method	Pseudocode
`exact`	1. for each population $\boldsymbol{n}$ from $\mathbf{0}$ to $\boldsymbol{N}$ (lexicographic): 2. $R_{ir} = D_{ir}\,[1 + Q_i(\boldsymbol{n} - \mathbf{1}_r)]$ at queues, $R_{ir} = D_{ir}$ at delays (arrival theorem) 3. $T_r = n_r/(Z_r + \sum_i R_{ir})$; $Q_{ir} = T_r R_{ir}$ 4. return Q, U, R, T at $\boldsymbol{n} = \boldsymbol{N}$ (multiserver: add marginal-probability recursion; mixed: inflate closed demands by $(1 - U_i^{\text{open}})^{-1}$)
`amva`	1. initialize $Q_{ir} = N_r/M$ 2. repeat: estimate arrival-instant lengths A_{ir} (Bard-Schweitzer $A_{ir} = Q_i - Q_{ir}/N_r$, or Linearizer deviations δ_{irs}) 3. $R_{ir} = D_{ir}(1 + A_{ir})$; $T_r = N_r/(Z_r + \sum_i R_{ir})$; $Q_{ir} = T_r R_{ir}$ 4. until $\max_{ir} \lvert\Delta Q_{ir}\rvert <$ `iter_tol`
`qna`	1. solve traffic equations for per-station arrival rates λ_i 2. propagate SCVs through merge, split, and departure operators to get $c_{a,i}^2$ 3. per station evaluate G/G/k waiting time (Kraemer–Langenbach-Belz) 4. assemble Q, U, R, T (open only)
`rqna`	as `qna`, but propagate MAP/MMPP2 flow descriptors (rate, SCV, lag-1 correlation) instead of two moments; per-station queue solved from the fitted MAP

continued on next page

Method	Pseudocode
`rqt`	1. set $\Gamma_a = \sigma_a$ at the external stream 2. compose robust-Burke queue-passage, superposition and thinning along the visit-ratio path for effective $(\lambda, \Gamma_a, \alpha)$ 3. regress the service variability Γ_s against simulation, adaptation regime `rqt_regime` 4. evaluate the closed-form worst-case system time (or the exact worst case under `rqt_exact`); single-class open only
`scat`	as `amva`, but refresh the Bard-Schweitzer deviation Δ_{irs} once per outer pass instead of Linearizer's three inner levels; single-server product-form only
`sqd`	1. mark blocking-after-service stations; 2. iterate MVA with blocked-job holding at the downstream station until the blocking probabilities and throughputs converge (single-chain closed)
`bounds`	compute $D_{\max} = \max_i D_i$, $D_{\text{sum}} = \sum_i D_i$, Z; return $X_{\text{up}} = \min(1/D_{\max}, N/(D_{\text{sum}} + Z))$ and the companion lower bound (asymptotic, balanced-job, geometric, or Harel), no iteration
`cache`	1. initialize per-item miss probabilities 2. solve the characteristic time t_C from $\sum_k h_k(t_C) =$ capacity (Che approximation) 3. update item hit probabilities h_k; iterate to fixed point; return hit/miss rates
`cacheqn`	1. solve the network (`exact`/`amva`) with current hit/miss class-switch probabilities 2. recompute cache hit/miss from the resulting per-item request rates (`cache`) 3. iterate steps 1–2 until miss rates converge (delayed-hit retrieval adds back-end fetch classes)
`poll`	evaluate exhaustive/gated/limited mean waiting time in closed form from utilizations and switchover moments (Takagi)
`sjn` (`sjn.mva`/`sjn.amva`)	1. fit each SJF station's service density f from (mean, SCV) by a two-moment Erlang/hyperexponential mixture 2. `sjn.mva`: step $W(x,n)$ and $R(n)$ over the full population lattice $\mathbf{0}$ to $\boldsymbol{N}$ (exact) `sjn.amva`: from a `pfqn_bs` start, iterate a Schweitzer fixed point on the size-resolved density $\lambda_k W_k(x) f_k(x)$ 3. bound a near-saturated station's utilization at `sjn_umax` by bisecting an inflation of its waiting time (warns if binding) 4. combine with ordinary MVA at the non-SJF stations; return Q, U, R, T
fork-join (`mmt`/`ht`)	1. replace each fork-join subnetwork by auxiliary open classes (`mmt`) or asynchronous tasks (`ht`) 2. solve the transformed model with the selected method 3. update synchronization delays from sibling response times; repeat until stable

9.13 Bibliographic notes

Exact MVA is due to Reiser and Lavenberg [149], with the load-dependent form in [148] and the mixed-network form in [24]. The AMVA lineage starts with Bard [10] and Schweitzer [165], continues with Linearizer [40] and its cost reductions, the AQL method [189], multiserver extensions [46, 175], and unified templates [51]; queue-dependent AMVA is introduced in [35] and load-dependent factorization in [33]. Balanced job bounds and asymptotic bounds are classical [118]; geometric bounds are from [34] and the Harel bounds from [86]. Open-network decomposition follows Whitt [187] and Kraemer and Langenbach-Belz [111]. Fork-join approximations trace to Heidelberger and Trivedi [89] and, for the mixed-model transformations, to [58]. Polling results are surveyed by Takagi [176], and M/G/1 scheduling analysis by [142, 188]. The shortest-job-next conditional waiting time equation is due to Kant [100], its fixed-point closure following Schweitzer [165]. The arrival-instant approximation via throughput elasticities is due to Tay and Suri [178]. Robust Queueing Theory is due to Bandi, Bertsimas and Youssef [9]. SCAT is due to Neuse and Chandy [139].

Chapter 10

SolverNC

10.1 Overview

`SolverNC` encodes the normalizing-constant paradigm for product-form queueing networks. For a closed BCMP network [11] with population $\boldsymbol{N}$, the equilibrium distribution factorizes as

$$\pi(\boldsymbol{n}) = \frac{1}{G(\boldsymbol{N})} \prod_{i=1}^{M} F_i(\boldsymbol{n}_i), \qquad G(\boldsymbol{N}) = \sum_{\boldsymbol{n}\,:\,\sum_i \boldsymbol{n}_i = \boldsymbol{N}} \prod_{i=1}^{M} F_i(\boldsymbol{n}_i), \tag{10.1}$$

where F_i are station factors determined by demands and scheduling. All mean and marginal metrics are ratios of normalizing constants evaluated at shifted populations; for instance $T_r(\boldsymbol{N}) = G(\boldsymbol{N} - \mathbf{1}_r)/G(\boldsymbol{N})$. The solver therefore reduces performance evaluation to the computation, exact or asymptotic, of $G(\boldsymbol{N})$, in contrast with `SolverMVA`, which eliminates G altogether. Working with G pays off for models with many classes but few stations, for marginal and joint probabilities, for load-dependent stations, and for models where asymptotic expansions are accurate.

The solver resides in `jline/solvers/nc/`, with the numerical algorithms in `jline/api/pfqn/nc/` (functions prefixed `pfqn_`). All computations are carried out in logarithmic space (`logNormConstAggr`) to avoid overflow, since $G(\boldsymbol{N})$ grows combinatorially in $\boldsymbol{N}$.

10.2 Software architecture

`SolverNC.runAnalyzer()` dispatches to six analyzers: the default `Solver_nc_analyzer`, the load-dependent `Solver_ncld_analyzer`, cache analyzers (`Solver_nc_cache_analyzer`, `Solver_nc_cache_qn_analyzer`, `Solver_nc_retrieval_analyzer`), and the loss-network analyzer `Solver_nc_lossn_analyzer`. Handlers implement metric extraction from normalizing constants: `Solver_nc` (means), `Solver_nc_conv` (convolution), `Solver_nc_marg`/`Solver_nc_margaggr` (marginal probabilities), `Solver_nc_joint`/`Solver_nc_jointaggr` (joint probabilities), `Solver_ncld` (load-dependent means), and `Solver_nc_lcfsqn` (LCFS networks). The method string selects the algorithm used to evaluate G inside `pfqn_nc`, which acts as a facade that also sanitizes demands (`pfqn_nc_sanitize`: removal of replicated stations, zero-demand classes, and scaling for conditioning).

`Solver_nc_conv` is the route taken when a station declares a class- or joint-dependent scaling (`sn.cdscaling`, `sn.jdscaling`): it evaluates the multichain convolution of Sauer [161] over the population lattice, folding a joint dependence into the same per-station handle as a class dependence, and recovers the marginal queue lengths from $P_m(\boldsymbol{n}) = X_m(\boldsymbol{n})\, G_{-m}(\boldsymbol{N}-\boldsymbol{n})/G(\boldsymbol{N})$. The recursion runs on *chains*, not classes: the handler aggregates demands with `sn_get_demands_chain`, enumerates the lattice of $\boldsymbol{N}_{\text{chain}}$, and deaggregates with `sn_deaggregate_chain_results`. Running it per class would charge nothing to a station visited only after a class switch, because `sn.njobs` is zero in every class that holds no reference jobs; each layer built by `SolverLN` has exactly that shape. Two consequences follow for the handler: the response time it passes to the deaggregation is the per-visit $Q_{\text{chain}}/T_{\text{chain}}$, since the deaggregation reinstates the visit ratio, and the per-class peak rates collapse to one normalizer per chain, taken as the largest peak among the classes of that chain. The scaling handles are therefore called with a per-chain population vector here and in `Solver_amvald`, but with a per-class one under CTMC and the exact recursions, so a handle must answer in the index space of its argument.

A model containing a fork is intercepted before any of these routes, since none of them can represent a `Fork` node: the solver drives the shared fork-join fixed point described in Section 9.8 with an inner solve of its own (`ncDispatch`), which calls `Solver_nc_analyzer`, or `Solver_ncld_analyzer` when load- or class-dependent scaling is present, on the transformed fork-free network. No method override is applied: the transformed model carries auxiliary open classes at a vanishing arrival rate, which the default normalizing-constant route resolves exactly. The auxiliary-class columns are merged back and dropped by the driver, with the caveat that C is per-class in mean-value analysis but per-chain here, so it is left untouched rather than reshaped.

10.3 Exact methods

`ca` The convolution algorithm of Buzen [26], generalized to multiclass networks [118,148], computes G by dynamic programming over stations and populations in $O(MR\prod_r(N_r+1))$ time (`pfqn_ca`). It is the default exact fallback and also serves load-dependent stations through its queue-dependent form.

`comom` The class-oriented method of moments [27,28] evaluates G by propagating a basis of normalizing constants at neighboring populations through exact linear systems, with complexity polynomial in the total population for a fixed number of classes; it is the method of choice for models with many jobs and few classes (`pfqn_comom`, with repairmen specializations `pfqn_comomrm` and the multiserver variant `pfqn_comomrm_ms`). The `comomld` variant extends the basis propagation to load-dependent stations.

`kt` The Koenigsberg-Tier style recursion implemented as the Knessl-Tier asymptotic-exact hybrid [105] (`pfqn_kt`).

`bkt` BKT (not part of the reference): `pfqn_kt` minus the exact Stirling remainder $s(N_r) = \log N_r! - (N_r \log N_r - N_r + \frac{1}{2}\log 2\pi N_r)$ of every class direction the saddle point Laplaces, one per class with jobs, self-looping classes excepted since their coefficient is extracted exactly (`pfqn_bkt`). Steepest descent on $[\xi^N]e^{Z\xi} = Z^N/N!$ returns Stirling's approximation of $\log N!$ in place of $\log N!$, which is where the remainder comes from; $s(1)$ is the constant of `ble`, so the two corrections are one reading of

Laplace's method applied to the class and to the station directions. With a think time the two corrected expansions are ONE estimator, the R-dimensional and $(M-1)$-dimensional saddle points being one point in dual coordinates (ξ_r^* the class throughputs of the `le` fixed point, $v^*u_k^* = 1/(1-U_k)$) and Sylvester's identity exchanging the two Hessian determinants; without one they differ by a constant that depends only on $N+M$, the remainder of the radial direction that `le` integrates exactly. On the 1562 models of the SIGMETRICS'17 suite with $Z_r = 100$ the median $\log G$ error falls from 0.083 (`kt`) to $1.4 \cdot 10^{-5}$ nats; truncating s at $1/(12N_r)$ would leave $1.9 \cdot 10^{-4}$, which is why the remainder is evaluated exactly.

`lekt` The estimator of the previous item as a method in its own right (`pfqn_lekt`): since `ble` and `bkt` are one function of $(\boldsymbol{L}, \boldsymbol{N}, \boldsymbol{Z})$, the choice between them is cost alone, an R-dimensional convex solve and an $R \times R$ determinant against an M-dimensional fixed point and an $(M-1) \times (M-1)$ one, so `lekt` takes the Knessl–Tier side when $R \leq M$, or when a class self-loops (a single demand and no think time, which that side extracts exactly), and the logistic side otherwise. Without a think time the logistic side is given the constant $M(1-\log(2\pi)/2) - r(N+M)$ in place of `ble`'s $(M-1)(1-\log(2\pi)/2)$, r the Stirling remainder of the radial $\mathrm{Gamma}(N+M)$ direction that `le` integrates exactly and `kt` does not, which is what makes the two sides agree there as well; on interior modes that constant is the more accurate of the two, at a vertex `ble`'s is.

`rd`, `nrp`, `nrl` Exact recursions on generating functions and partial fractions in the tradition of Harrison and Lee [88] and residue methods [14, 42].

`rgf` Recursion by generating functions [50]: for a single-class closed network, each station contributes the coefficient sequence of its own generating function, and a group of m stations sharing the same demand p collapses into the single negative-binomial sequence $r(k) = \binom{k+m-1}{k} p^k$ (the delay contributes the Poisson sequence $r(k) = Z^k/k!$), so convolving the G distinct sequences gives $g(0..N)$ exactly in $O(GN^2)$ time (`pfqn_rgf`), against $O(MN)$ for `ca`; the whole recursion runs in the log domain to avoid overflow. This makes it the cheaper exact route precisely when the model is heavily replicated ($GN < M$) and the population is moderate. Restricted to single-class closed networks.

10.4 Asymptotic and integral methods

For large populations, (10.1) admits integral representations

$$G(\boldsymbol{N}) \;=\; \frac{1}{\prod_r N_r!} \int_{\mathbb{R}_+^M} \prod_r \Big(\textstyle\sum_i D_{ir} u_i + Z_r \Big)^{N_r} e^{-\sum_i u_i} \, d\boldsymbol{u}, \tag{10.2}$$

in the style of McKenna and Mitra [130], which the following methods evaluate:

`le` Logistic expansion [33]: a change of variables maps (10.2) onto the simplex, where a Laplace approximation around the fixed point of the logistic transform gives an asymptotic expansion with computable correction terms (`pfqn_le`, with fixed-point algorithms `pfqn_le_fpi`, `pfqn_le_fpiZ` and Hessian terms `pfqn_le_hessian`, `pfqn_le_hessianZ`). `pfqn_lap` provides the plain Laplace approximation. The reference's Theorem 4.1 requires the expansion parameter $\epsilon \geq \epsilon_N > 0$; `pfqn_le` instead

evaluates it at $\epsilon \to 0$, which is simpler but carries an $O(1)$ bias of $(e/\sqrt{2\pi})^{M-1}$ that grows with the number of stations.

ble BLE (not part of the reference): `pfqn_le` with the empirical correction $(M-1)(1-\log(2\pi)/2)$ added before the $\epsilon \to 0$ limit, cancelling the bias above to leading order. Measured across the full example suite, it lowers the median $\log G$ error from 0.561 to 0.041 nats relative to `le`; because the correction is additive in $\log G$ it cancels exactly in throughput ratios $G(\boldsymbol{N}-\boldsymbol{e}_r)/G(\boldsymbol{N})$, so mean-value metrics derived from `le` and `ble` agree even though the reported normalizing constants differ. `default` routes normally-loaded closed models through `ble` rather than `le`; `le` is kept unmodified as the published form for anyone who needs to reproduce it exactly.

aghq Adaptive Gauss–Hermite quadrature ([124]). At $\boldsymbol{Z} = \boldsymbol{0}$ the integrand of (10.2) is homogeneous, so the radius separates and integrates exactly to $\Gamma(N+M)$, leaving $\int_\Delta \prod_r (\boldsymbol{L}_r^\top \boldsymbol{x})^{N_r}\, d\boldsymbol{x}$: all of the error of `le` is the closure of that simplex integral. Reading $(\boldsymbol{u}^*, \boldsymbol{A})$ as a quadrature rule over it rather than as a closure, $\boldsymbol{w} = \boldsymbol{w}^* + \boldsymbol{A}^{-1/2}\boldsymbol{z}$ with q nodes per simplex direction. At $q = 1$ the single node is the mode and the weight $\sqrt{2\pi}$, so the rule IS `le`: the logistic expansion is the first term of a convergent quadrature rather than an approximation of unknown accuracy. Cost q^{M-1}, set by `options.config.aghq_nodes`. Convergence is fast when the integrand is close to Gaussian in $\boldsymbol{w}$ and slow in the corner regime, where the non-bottleneck directions are locally $\mathrm{Gamma}(1)$ and the constant of `ble` is the better instrument.

nre Saddle-tilted Edgeworth evaluation of the Norlund-Rice integral form of the limited load-dependent constant [33] (`pfqn_nre`). It differs from `nrl` and `nrp` in two respects. The integrand is invariant under $\boldsymbol{t} \to \boldsymbol{t} + c\boldsymbol{1}$, since its homogeneity degree cancels against $e^{-\imath \boldsymbol{N}\boldsymbol{t}}$, so the redundant direction is quotiented out and the integral is $(R-1)$-dimensional; and the contour radii are tilted per class to the saddle point, the $\boldsymbol{X}$ solving $X_r\, \partial \log h / \partial X_r = N_r$, at which the origin is a stationary point of the phase, whereas on the untilted contour $\boldsymbol{X} = \boldsymbol{1}$ it is not, which is the leading source of bias in the other two. A second-order Edgeworth term built from the third and fourth cumulants of the tilted distribution then leaves a relative error of $O(1/|\boldsymbol{N}|^2)$ in place of $O(1)$. Every integrand evaluation is at real positive demands, so the method needs no complex arithmetic and runs entirely in the log domain through `pfqn_gldsingle`, at a cost of $O(IR^2 + R^4)$ single-class evaluations.

ls Logistic sampling [33]: Monte Carlo integration under the logistic change of variables, asymptotically unbiased with variance vanishing in the large-population limit (`pfqn_ls`).

mci, imci Plain and importance-sampling Monte Carlo integration of (10.2) (`pfqn_mci`), in the spirit of the sampling estimators for closed networks [29].

pana, panald The asymptotic expansion of PANACEA [130] for networks with delays, valid in the normal-usage regime; `panald` is its load-dependent extension through the pseudonetwork coefficients of Mitra and McKenna [134].

mmint2 Two-dimensional repairmen-type integral evaluated by Gauss-Legendre or Gauss-Laguerre quadrature (`pfqn_mmint2_gausslegendre`, `pfqn_mmint2_gausslaguerre`); `gleint` is the general Gauss-Legendre integration path.

ger The closed-form normalizing constant of Gerasimov [76] (`pfqn_gerasimov`), which evaluates G exactly by iterated residues of its rational generating function, eliminating one class per residue.

propfair The proportionally fair allocation estimate (`pfqn_propfair`), which bounds G through the utility-maximizing throughput allocation; its asymptotics connect to the bottleneck-convergence theory of closed networks under congestion [5].

gm, grm Grundmann-Moeller simplicial cubature applied to the simplex form of (10.2) (`pfqn_grnmol`); `cub` selects general cubature (`pfqn_cub`).

clw The Choudhury-Leung-Whitt inversion of the generating function of G [42], including its load-dependent form (`pfqn_clw`, `pfqn_clw_lld`).

is Importance sampling over job *orderings* rather than over the integral (10.2), for closed models only. The balance function of a product-form station is the sum, over the orderings q of a per-class population $\boldsymbol{n}$, of an ordered product of per-position factors,

$$F_i(\boldsymbol{n}) = \frac{|\boldsymbol{n}|!}{\prod_r n_r!} \frac{\prod_r L_{ir}^{n_r}}{\prod_{k=1}^{|\boldsymbol{n}|} \mu_i(k)} = \sum_{q:\,|q|=\boldsymbol{n}} \prod_{p=1}^{|\boldsymbol{n}|} \frac{L_{i,q_p}}{\mu_i(p)},$$

so $G(\boldsymbol{N})$ is recovered by drawing an ordering c of all $\ell = \sum_r N_r$ jobs with probability $p(c)$ and averaging $S(c)/p(c)$, where $S(c)$ sums the ordered products over every way of cutting c into contiguous per-station segments. The inner sum is evaluated exactly by dynamic programming in $O(M\ell^2)$, so the ordering is the only Monte-Carlo dimension and the estimator is unbiased. The per-position factor specializes the method to four cases, dispatched automatically: a single-server queue uses L_{i,q_p} (`pfqn_is`); a load-dependent or multiserver station uses $L_{i,q_p}/\mu_i(p)$ with $\mu_i(k) = \min(k, c_i)$ (`pfqn_ld_is`); a delay uses Z_{q_p}/p; and an order-independent or pass-and-swap station uses the reciprocal rank rate $1/\mu_i(\mathrm{supp}(q_1 \dots q_p))$, which depends on the support reached at position p rather than on the class at p (`pfqn_oi_is`, `pfqn_pas_is`). For a pass-and-swap station with a non-empty swap graph the ordered-state chain is reducible [45]: only the linear extensions of the placement partial order are sampled, and the estimate is the normalizing constant $G_{\mathcal{C}}$ of the recurrent communicating class.

The default method chooses adaptively: exact recursions for small models, `comom` when the class structure permits, and logistic expansion or sampling for large populations, with the decision logic centralized in `pfqn_nc`.

10.5 Birman-Kogan methods

`bk`, `bkue`, `lc` and `lc.ue` [18] are a second asymptotic family, evaluating the same generating function as §10.4 by contour/saddle-point methods rather than the logistic or Laplace routes above.

bk (`pfqn_bk`) Saddle-point evaluation of $G(\boldsymbol{N})$ by Cauchy inversion of the generating function. Algorithm 1 of the reference detects chains whose *dedicated station* – a station serving one chain alone,

itself part of a group of identical stations – is a bottleneck, and pins that chain's coordinate on the pole of the generating function rather than at the saddle point, which the plain saddle-point approximation gets wrong. The routine reports separately which chains were treated as non-saturated dedicated chains and which as bottlenecked.

bkue (pfqn_bkue) Single-chain uniform expansion: when the saddle point sits close to a dominant pole, bk's separation of the two breaks down, and this variant keeps them together through a van der Waerden-style erfc term. Defined for a single chain only; on a multiclass model it falls back to bk.

lc, lc.ue (pfqn_bklc) Algorithm 2 of the reference, the load-concealment reduction of a multichain network to a sequence of single-chain problems: a Gauss-Seidel sweep solves each chain l in isolation against the residual capacity $A_i = 1 - \sum_{k \neq l} L(i,k) X_k$ left by the current throughput estimate of every other chain, iterating to a fixed point (tol, maxiter). The inner single-chain solve is mva by default or ue (pfqn_bkue) under lc.ue.

10.6 Probability and specialized analyzers

Because SolverNC manipulates G directly, it exposes exact state probabilities that mean-value methods cannot produce: getProbAggr evaluates marginal probabilities $\Pr[\boldsymbol{n}_i]$ as ratios of normalizing constants with station i removed (Solver_nc_marg), and getProbSysAggr evaluates joint states (Solver_nc_joint), including load-dependent variants; getProbSysMarg evaluates the joint law of the per-station totals with the classes summed out (Solver_nc_jointmarg, §10.6.2). The loss-network analyzer (Solver_nc_lossn_analyzer) solves multi-rate loss networks exactly from the normalizing constant (exact, rec), by the Erlang fixed-point approximation [102] (erlangfp), by the large-capacity limit of Kelly (asy, lossn_asy), which minimizes the convex dual $\sum_k \nu_k e^{-\sum_j A_{jk} y_j} + \sum_j C_j y_j$ by coordinate descent and sets the link blocking to $a_j = 1 - e^{-y_j}$, or by Monte Carlo summation (mci); any other method name is refused there. The analyzer also returns the loss-network parameters $(\boldsymbol{\nu}, \mathbf{A}, \mathbf{C})$, the link names and the method it ran, which SolverNC keeps with the result so that getAvgLinkTable can evaluate the per-link blocking $1 - G(\mathbf{C} - A_{jr}\mathbf{e}_j)/G(\mathbf{C})$ through lossn_links with the same method [154]. Cache analyzers compute exact or asymptotic hit probabilities for capacity-constrained caches by normalizing-constant methods over the cache state space, embedding them in queueing networks by the same class-switching iteration used in SolverMVA; the theory underlying the cache normalizing constants follows time-to-live and characteristic-time equivalences [41, 52]. The LCFS-network analyzer (Solver_nc_lcfsqn, pfqn_lcfsqn_nc) evaluates G for multiclass LCFS networks as a sum, over the $K+1$ possible positions of a distinguished cut ($K = \sum_r N_r$ the total population), of the matrix permanent of a per-position matrix built from the per-class geometric sequences α_r^j and $\alpha_r^j \beta_r$; each term is expanded to a $K \times K$ matrix by replicating class r's row N_r times before the permanent is taken.

10.6.1 Matrix permanent library

The permanent, $\mathrm{perm}(A) = \sum_{\sigma \in S_n} \prod_i A_{i,\sigma(i)}$, is the determinant with every sign fixed to $+1$; computing it is #P-complete, so jline.lib.perm (jline_solver.api.perm in Python, matlab/util/perm*.m

in MATLAB, `line::perm` in C++) offers three families of algorithm, all operating on nonnegative matrices, so a caller can trade exactness for scale:

Exact `Permanent` evaluates the Ryser inclusion-exclusion formula [156] exploiting repeated rows or columns, the form used by `pfqn_lcfsqn_nc` above since its expanded matrix repeats each class row N_r times; `RyzerPermanent` offers the Gray-code recurrence and a naive variant of the same formula, and `NaivePermanent` enumerates all $n!$ permutations directly. All three are exponential in n and are intended for the small, structured matrices that arise from a demand matrix, not for general dense inputs.

Deterministic approximation `BethePermanent` evaluates the sum-product (belief-propagation) Bethe approximation of the permanent [183]. `HeuristicPermanent` first rescales the matrix to doubly stochastic form by Sinkhorn iteration [170] and then reports both a mean-field estimate and the van der Waerden/Gurvits capacity bound [84] on the rescaled matrix.

Randomized approximation `AdaPartSampler` recursively partitions the permutation space, bounds each part by the Soules bound, draws a part with probability proportional to its bound, and turns the resulting acceptance ratio into an unbiased estimate of the permanent [113] (citation method names `adapart`/`perm.adapart`). `HuberLawSampler` rescales the matrix to doubly stochastic form and draws a permutation column by column under the Huber-Law bound on the remaining permanent, again turning the acceptance ratio (scaled by the rescaling constant) into an estimate [97] (citation method names `huberlaw`/`perm.huberlaw`). Both samplers support a `'classic'` (fixed acceptance budget), `'time'` (wall-clock budget) or `'sample'` (fixed draw budget) stopping mode; the Java sampler draws from an unseeded `java.util.Random`, while the Python port takes an explicit seed into a `numpy` generator and the C++ port a seeded `std::mt19937_64`, so the ports agree in distribution but not sample by sample. Two defects of the published AdaPart procedure are corrected in every port: the column search must consider only the columns still unassigned, since scoring an already-assigned column re-derives its own constraint and always looks cheapest, so the sampler would re-split one column forever; and the refinement step must discount the bound of the element actually removed rather than the bound at the root. Even so the Soules bound is tight on a matrix with equal entries, where no refinement can improve it, so the loop stops at the first non-improving expansion instead of iterating until improvement.

A companion submodule (`NetworkNoThink`, `NetworkThink`, `preprocessing_ds`) computes permanent-based marginal probabilities of closed product-form networks, since the marginal probability of a queue-length vector is itself a permanent of the (state-replicated) service-demand matrix and any of the solvers above can be plugged into it, exact or approximate.

All three families are reachable from a solver through `getProbSysMarg` (§10.6.2), whose optional second argument names the estimator; the exact family is the default, and the two approximate families are offered for models whose class count puts the exact expansion out of reach. They are refused, rather than applied to a floored matrix, whenever the replicated demand matrix has a structural zero: the Sinkhorn scaling behind `HeuristicPermanent` and `HuberLawSampler` does not converge without full support, and the Bethe estimate is a state-dependent lower bound whose gap does not cancel when several states are normalized against one another. Measured over the ten total states of a three-station, two-class closed network

and renormalized on that lattice, the mean relative error against the exact value is 0.5% for `huberlaw`, 1.6% for `bethe`, 2.6% for `heur` and 4.1% for `adapart`; the raw masses before renormalization are 0.998, 0.398, 0.971 and 1.028 respectively, the Bethe figure being the visible signature of its lower-bound property.

10.6.2 The joint law of the per-station totals

`getProbSysMarg` answers the joint law of the per-station TOTAL queue lengths, $\Pr[n_1 = k_1, \ldots, n_M = k_M]$, with the classes summed out. It is a metric rather than a method: nothing is added to `listValidMethods` and nothing is read from `options.method`, since the quantity has one algorithm and its optional second argument selects only the estimator of the permanent.

The distinction from the other state-probability getters is the index set, not the normalization. `getProbSysAggr` fixes the per-class population of every station and is a product form; each value returned here is the SUM of that one over the whole fibre of per-class tables with the prescribed row sums. On a model with $M = 3$, $R = 3$ and $N = (2, 2, 1)$ that collapses 108 per-class states onto 21 total states, and the fibre grows combinatorially, so summation is not a route to it. The closed form is the permanent identity

$$\Pr[n_1, \ldots, n_M] \;=\; \frac{\operatorname{perm}(A)}{\left(\prod_r N_r!\right)\left(\prod_{j \in \mathcal{I}} n_j!\right) G(N)},$$

with A the demand matrix whose column r is repeated N_r times and whose row i is repeated n_i times, hence square of order $\sum_r N_r$, and $\mathcal{I}$ the set of infinite-server stations. The implementation does not materialize that expansion: it passes the $(\sum_i n_i) \times R$ row-replicated matrix to `pfqn_perm` together with the column multiplicities N, which is the same quantity at $\prod_r (N_r + 1)$ evaluations instead of $(\sum_r N_r)!$.

EVERY infinite-server station keeps its own row and contributes its own $1/n_j!$, which is what separates `pfqn_jointmarg` from the totals branch of `pfqn_joint`, where the think time enters as a single aggregated row. A model with two delays is normalized by that difference alone: dividing once leaves the law summing to less than one. The normalizing constant itself is exempt, because the delay stations aggregate by the multinomial theorem, so G may still be evaluated with the infinite-server rows summed into Z, and all four ports do so; the cached `result.Prob.logNormConstAggr` is reused, so a sweep of the whole lattice pays for G once rather than once per state.

Zero entries are safe under the exact estimator and only under it. A station holding no jobs contributes no row of A, a class with no jobs contributes no column, a zero demand is an ordinary zero entry, and the permanent of the empty matrix is one, so the empty network and the zero-population class need no special case. An occupancy vector whose total does not match the closed population is not an error but has probability zero, which is what lets a caller sweep the lattice without a feasibility test.

The identity relies on exactly one factor $n_i!$ per queueing station, which neither a multiserver nor a load-dependent station supplies. Those, together with open and mixed models and class-dependent scaling, are refused by name inside the getter. The refusal is imperative rather than expressed as a feature name, which is how `SolverNC` expresses every other method-specific restriction, and it is the reason `getProbSysMarg` is inert on every existing path: no `getAvg` result can move because of it. Attribution follows the estimator actually used, through the `lastPermEngine` field that the getter sets and `citations` reads, so an approximate run reports its own estimator beside Ryser's formula.

10.7 State-dependent routing

Adaptive routing rules such as JSQ and SQ(d) destroy the product form, but the rule of Krzesinski [112], the multiclass generalization of Towsley's, does not: the routing probabilities out of an entry center depend on the populations of the branches and subnetworks it feeds, and the stationary distribution keeps a product form with the weights of the reference's eq. (16). `SolverNC` recognises a model carrying such a declaration and routes it to `Solver_nc_sdr_analyzer`, under the method names `sdr` and `sdr.mva`.

The routing structure declared on the entry node is reduced to its derived coefficients by `pfqn_sdrcoeff`, the routing probabilities of eq. (10) are evaluated by `pfqn_sdrprob`, and the visit coefficients ξ of the reference's Section 3.2 by `pfqn_sdrvisits`. The two solution routes are then:

`sdr` (`pfqn_sdr`) evaluates eq. (16) exactly by enumerating the state space, which places no restriction on the branch topology but costs the size of that space.

`sdr.mva` (`pfqn_sdrmva`) runs the reference's Section 4 mean value analysis and convolution instead, level by level and outermost subnetwork last: a modified MVA of the centers of one level in isolation, whose arrival theorem carries the admission coefficient $\delta_{ti}(n) = d_{ti} - n$ and the center's own queue-length distribution (Sec. 4.2.1), a convolution of that level with the already solved inner subnetwork weighted by $\Omega_{t-1,t}/\Omega_{tt}$ (Sec. 4.2.2), which also renormalizes the inner queue lengths (Sec. 4.2.3), and a final convolution against the complement $M - V$ (Sec. 4.3). The cost is $O(JTM(V_1 \cdots V_J)^2)$. Two restrictions come from the reference itself and are refused by name rather than approximated: every branch must be a single center, the general case being deferred to an unpublished technical report, and every C_t must be negative. A structure with $C_t < 0$ but not -1 is rescaled internally, which leaves both eq. (10) and eq. (16) unchanged because the factors telescope away.

One deviation from the printed rule is deliberate. Section 3.2 sets $\xi = \xi_e$ at the entry and departure centers of a branch, which is exact only for single-center branches; with feedback onto a branch departure the traffic equations as implemented reproduce the CTMC to $5.6 \cdot 10^{-16}$ where the literal rule is off by $3 \cdot 10^{-1}$ in the queue lengths. Equation (16) itself extends unchanged.

10.8 Perfect sampling

The methods `cftp` and `cftp.approx` answer a steady-state request by sampling the product form rather than evaluating it: they neither generate the state space nor compute the normalizing constant, so they return the stationary distribution of models whose reachable space does not fit in memory and report no $\log G$. They were `SolverCTMC` methods until 2026-09-27 and moved here because they read the balance function and never a generator. They are admitted only on closed single-class product-form networks, where the stationary distribution over the aggregate state $\boldsymbol{n} = (n_1, \ldots, n_M)$, $\sum_i n_i = N$, is

$$\pi(\boldsymbol{n}) = \frac{1}{G(N)} \prod_{i=1}^{M} \alpha_i(n_i), \qquad \alpha_i(s) = \frac{L_i^s}{\prod_{m=1}^{s} \min(m, c_i)}, \tag{10.3}$$

with L_i the service demand of station i, c_i its number of servers and $c_i = \infty$ at a delay. The analyzer `Solver_nc_cftp_analyzer` obtains $\boldsymbol{L}$, the visit ratios and the reference station from `sn_get_demands_chain`, then delegates the draws to `Pfqn_cftp`.

The sampler is the monotone coupling-from-the-past algorithm of Propp and Wilson [147] in the form given for closed Jackson networks by Kijima and Matsui [104]. A move of the underlying chain picks a pair of stations (i, j) and redraws the split of the $k = n_i + n_j$ jobs they hold from the conditional law $\Pr[n_i = s \mid k] \propto \alpha_i(s)\alpha_j(k-s)$, $s = 0, \ldots, k$. The normalizing constant $G(N)$ cancels in this conditional, so no normalizing constant is ever evaluated, and the balance functions are accumulated in logarithmic form, $\log \alpha_i(s) = s \log L_i - \sum_{m \le s} \log \min(m, c_i)$, with a max-shift before exponentiation. The state space is ordered componentwise and the move is coupled across states by reusing the same uniform variate, which makes the chain monotone: the exact variant runs the coupling backwards from the past from the two extremal states, doubling the horizon until they coalesce, and returns the common state, which is distributed exactly as (10.3). The approximate variant runs instead the rapidly-mixing chain forward for the number of steps given by its mixing bound, $O(M^2 \log(N/\epsilon))$ at accuracy ϵ, trading a residual bias for a deterministic cost. Both report per sample the coalescence horizon or the number of mixing steps.

The reduction from samples to metrics follows the conventions of the other LINE solvers. Queue lengths are the sample means of the drawn occupancies, and the busy-server counts $b_i = \mathbb{E}[\min(n_i, c_i)]$ are estimated on the same draws. Throughput is taken at the reference station, $X = b_{\text{ref}}/(S_{\text{ref}} V_{\text{ref}})$, and propagated through the visit ratios as $T_i = V_i X$; deriving each T_i from its own station would give every station an independent Monte Carlo error and break flow balance in the reported table. Utilization keeps its own estimator $U_i = b_i/c_i$ ($U_i = Q_i$ at a delay), which is unbiased and confined to $[0, 1]$ by construction, whereas propagating it from X can push a saturated station above one. Response times follow from Little's law, $R_i = Q_i/T_i$, and the system metrics from $C = N/X$; only Q and X therefore carry Monte Carlo error, of order $O(\text{samples}^{-1/2})$. The distinct drawn states and their empirical frequencies are returned in place of the enumerated state space and of $\boldsymbol{\pi}$, and the raw draws and horizons are exposed as well, since they are the only representation of the stationary distribution this method produces.

An admission gate runs before any sampling and refuses, rather than approximates, every model outside the class: transient requests, multiclass or open models, fewer than two stations, node types other than Queue, Delay and Router, scheduling other than `INF`, `PS`, `FCFS`, `SIRO` and `LCFSPR`, non-exponential service (more than one phase), finite buffers, load-, class- or joint-dependent scaling, finite capacity regions, and state-dependent routing.

10.9 Architectural flow

Figure 10.1 shows the standard path. Note the two-level dispatch: the analyzer selects the handler by request type (means, marginals, joint), while the `pfqn_nc` facade selects the numerical algorithm by method and model size.

10.10 Implementation notes

All algorithms return $\log G$ and the handlers form metric ratios by exponentiating differences, which keeps the pipeline stable up to populations of several thousands. Demand sanitization eliminates numerically troublesome structure before algorithm selection: zero-demand classes are absorbed into think times, identical stations are merged with multiplicities, and demands are rescaled with the scaling factor restored in

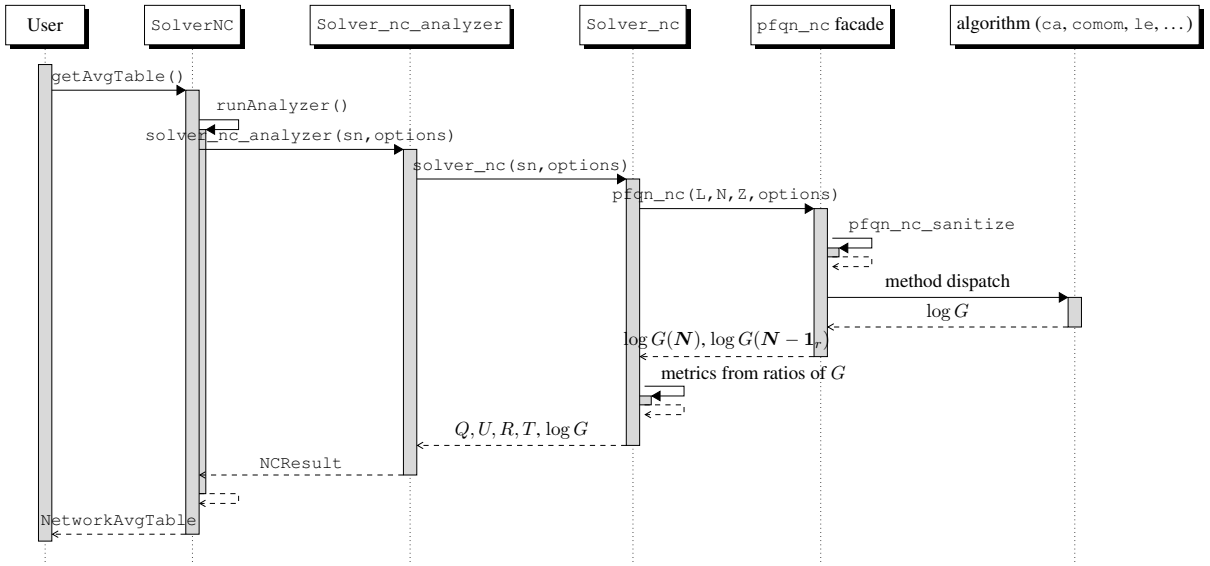


Figure 10.1: Sequence diagram of `SolverNC` for a steady-state average request.

$\log G$ afterwards. The load-dependent path stores station factors as cumulative log-products of rate scalings, consistent with the factorization form of [33]. Results include `logNormConstAggr` so that downstream consumers, notably `SolverLN` and the sensitivity API (`pfqn/sens`), can reuse the constant without recomputation.

10.11 Feature and method support

Table 10.1 maps modelling features to the `SolverNC` methods. Exact normalizing-constant evaluation (`ca`, `comom`) requires product-form structure; the asymptotic and integral methods (`le`/`ls`, `mci`, `kt`) target large closed populations; `mem` is the maximum-entropy path for non-product-form open and closed networks, gated per model class composition (`supportsModelMethod`). Column `spec.` denotes the specialized analyzers (cache, retrieval, loss network), selected by topology. Phase-type service is exact under insensitive disciplines (PS, LCFS-PR, INF) and FCFS is exact only with class-independent exponential service (BCMP), hence ○. The perfect samplers `cftp` and `cftp.approx` have no column: their envelope is the closed single-class exponential product form of Section 10.8, and their gate refuses everything else.

Table 10.1: Feature support of `SolverNC` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	ca	comom	le/ls	mci	kt	mem	spec.
Job classes							
Open class	●	●		●		●	●

continued on next page

Feature	ca	comom	le/ls	mci	kt	mem	spec.
Closed class	●	●	●	●	●	●	●
Self-looping class	●	●	●	●	●		●
Mixed open/closed	●	●		●		●	●
Class switching	●	●	●	●	●		●
Node types							
Source, Sink	●	●		●		●	●
Queueing station	●	●	●	●	●	●	●
Delay (infinite server)	●	●	●	●	●	●	●
Cache node							●
Fork / Join	○	○	○	○	○		○
Service distributions							
Exponential	●	●	●	●	●	●	●
Phase-type (Erlang, HyperExp, Coxian, APH)	●	●	●	●	●	○	●
General two-moment (Det)	○	○	○	○		○	○
Scheduling disciplines							
INF (delay)	●	●	●	●	●	●	●
PS	●	●	●	●	●	●	●
LCFS-PR	●	●	●	●	●	●	●
SIRO	●	●	●	●	●		●
FCFS	○	○	○	○	○	●	○
Advanced and specialized features							
Load dependence	●	●	●				
Cache replacement (RR, FIFO)							●
Delayed-hit retrieval							●
Loss network (exact, Erlang fixed point, Kelly limit, Monte Carlo)							●
Analysis and output							
Steady-state means (`getAvg`)	●	●	●	●	●	●	●
Marginal probabilities (`getProbAggr`)	●	●	○	○			●
Joint probabilities (`getProbSysAggr`)	●	●	○	○			
Normalizing constant (`logNormConst`)	●	●	●	●	●	●	●

10.12 Method algorithms

Table 10.2 gives high-level pseudocode for each `SolverNC` method; all operate in logarithmic space and return $\log G(\boldsymbol{N})$ and the shifted $\log G(\boldsymbol{N}-\mathbf{1}_r)$, except the perfect samplers `cftp` and `cftp.approx`, which draw states in place of evaluating G (Section 10.8).

Table 10.2: High-level pseudocode of `SolverNC` methods.

Method	Pseudocode
`ca`	1. $g \leftarrow \delta_{\boldsymbol{n},\mathbf{0}}$ 2. for each station i: convolve g with the station factor sequence $F_i(\boldsymbol{n}_i)$ over the population lattice 3. $G(\boldsymbol{N}) = g(\boldsymbol{N})$; form metrics as ratios $G(\boldsymbol{N}-\mathbf{1}_r)/G(\boldsymbol{N})$ (load-dependent: queue-dependent factors)

continued on next page

Method	**Pseudocode**
`comom`	1. assemble a basis of normalizing constants at neighboring populations 2. propagate the basis from small to full $\boldsymbol{N}$ through exact linear systems (class-oriented moment equations) 3. read $\log G$ and shifts from the final basis (`comomld`: load-dependent factors)
`le`	1. map the integral to the simplex via the logistic change of variables 2. solve the fixed point $\boldsymbol{u}^*$ of the logistic transform 3. Laplace expansion around $\boldsymbol{u}^*$ with Hessian correction terms $\Rightarrow \log G$
`ls`	Monte Carlo integration under the logistic change of variables; average the integrand over samples; variance vanishes in the large-population limit
`mci`	draw $\boldsymbol{u} \sim \exp$; average $\prod_r (\sum_i D_{ir} u_i + Z_r)^{N_r}$; `imci` adds importance sampling
`kt`	Knessl–Tier asymptotic-exact hybrid recursion on G for large populations
`bkt`	as `kt`, but subtract the exact Stirling remainder of each Laplaced class direction; coincides with `ble` when there is a think time
`lekt`	the common value of `ble` and `bkt`, computed on the cheaper side ($R \leq M$: the Knessl–Tier saddle; otherwise the logistic mode)
`ble`	as `le`, but add $(M-1)(1-\log(2\pi)/2)$ to the Laplace expansion before the $\epsilon \to 0$ limit; default route for normally-loaded closed models
`bk`	saddle-point Cauchy inversion of the generating function; pin the coordinate of a bottlenecked dedicated chain on the pole instead of the saddle
`bkue`	as `bk`, single chain only; combine pole and saddle via an erfc term when they are close; falls back to `bk` on multiple chains
`lc`	Gauss-Seidel: solve each chain's single-chain problem against the residual capacity left by the current throughput estimate of the others; iterate to a fixed point; inner solver `mva` (`lc`) or `ue` (`lc.ue`)
`mem`	1. form the maximum-entropy product-form marginal per station with Lagrange multipliers 2. match multipliers to mean queue length / utilization constraints 3. iterate global flow balance until the marginals are consistent (non-product-form, open and closed)
`cftp`	skip generation; per sample, run the monotone chain backwards from horizon t, doubling t until the two extremal states coalesce under common uniforms; each move redraws n_i from $\Pr[n_i = s \mid n_i + n_j = k] \propto \alpha_i(s)\alpha_j(k-s)$ in log space, in which $G(N)$ cancels
`cftp.approx`	as above, but run the rapidly-mixing chain forward for $O(M^2 \log(N/\epsilon))$ steps instead of coupling, giving a deterministic cost and a residual mixing bias
sampled metrics	$Q_i = \overline{n_i}$, $b_i = \overline{\min(n_i, c_i)}$, $X = b_{\mathrm{ref}}/(S_{\mathrm{ref}} V_{\mathrm{ref}})$, $T_i = V_i X$, $U_i = b_i/c_i$, $R_i = Q_i/T_i$, $C = N/X$; empirical frequencies of the distinct draws replace $\boldsymbol{\pi}$
`spec.`	cache: normalizing-constant / characteristic-time fixed point over cache states; loss network: normalizing constant over $\mathbf{An} \leq \mathbf{C}$, Erlang fixed-point (Kaufman–Roberts) recursion over occupancy, or the Kelly limit from the convex dual

10.13 Bibliographic notes

Product-form theory originates with Jackson [98], Gordon and Newell [80], and BCMP [11]; the convolution algorithm is due to Buzen [26] with multiclass and load-dependent treatments in [118, 148]. RECAL and related recursions by chains are surveyed in [44, 47]. Integral representations and asymptotics were developed by McKenna and Mitra [130], Knessl and Tier [105], and via residues and transform inversion in [14, 42, 88]. The method of moments and its class-oriented specialization are from [27, 28]; logistic expansion and sampling from [33]; asymptotic performance inference applications in [29]. Loss-network fixed points and the large-capacity limit follow Kelly [102]; the per-link blocking, the critical-loading asymptotic of a single link and the revenue sensitivities with implied costs follow Ross [154]. Recursion by generating functions, which groups replicated stations into a single negative-binomial sequence, is due to Coury

and Harrison [50]. A unified treatment of normalizing-constant methods appears in [20]. The saddle-point, uniform-expansion and load-concealment methods of §10.5 are due to Birman and Kogan [18].

Chapter 11

SolverSSA

11.1 Overview

`SolverSSA` evaluates models by stochastic simulation of the same continuous-time Markov chain that `SolverCTMC` builds explicitly, without ever materializing the state space. It implements exact-sampling algorithms in the lineage of Gillespie's stochastic simulation algorithm (SSA) [79]: from the current state, the set of enabled events and their rates is computed, the sojourn time is sampled from the exponential distribution with the total exit rate, and the next event is drawn proportionally to its rate. Metrics are time averages over the generated trajectory. The solver therefore shares the full feature coverage of the CTMC semantics while scaling to state spaces far beyond enumeration, at the cost of statistical rather than numerical error.

The solver resides in `jline/solvers/ssa/`. Unlike `SolverLDES` (Chapter 7), which is an event-driven discrete-event simulator operating on job entities, `SolverSSA` samples Markov jump processes at the state-vector level, which makes it exact for the model semantics but restricted to Markovian (phase-type expanded) timing.

11.2 Analyzers and simulation engines

`Solver_ssa_analyzer` dispatches on `options.method` to four engines:

`nrm` (default when eligible) The next-reaction-method engine `Solver_ssa_nrm`, after Gibson and Bruck [78]: each synchronization maintains a tentative firing time in an indexed priority queue, and only the rates affected by the executed event are resampled, using a dependency graph between synchronizations. This reduces the per-event cost from $O(E)$ to $O(\log E)$ for models with E synchronizations and sparse interactions. Since the NRM engine reached full-capability parity with the serial engine (denoted milestone M5 in the development notes), it is preferred by the `default` method, and by the `parallel` replication driver, whenever the model passes the eligibility gate described below.

`serial` The direct-method engine `Solver_ssa`. Each step calls `Solver_ssa_findenabled` to enumerate enabled synchronizations from `sn` (`sn.sync`), evaluates their rates via the same `afterEvent`

transition functions used by the CTMC generator, and samples the next event by inverse transform. Hashed states accumulate sojourn times so that stationary metrics are computed from the empirical occupancy measure; an optional event log preserves the sample path for trace-driven post-processing. It is the reference engine against which the NRM engines are validated, and the fallback for any model outside the NRM eligibility gate. Requesting a sample-path event log (`record_events`) also forces the serial engine, which is the one that materializes the trajectory.

`parallel` (alias `para`) The replication driver `Solver_ssa_analyzer_parallel` runs independent trajectories with distinct substreams and averages their stationary estimators, reducing wall-clock time on multicore hosts; the estimator is the across-replication mean, following standard independent-replication methodology [89, 119]. Each replication runs the NRM or serial engine according to the same eligibility gate.

`nrm.space` A state-space-aware NRM variant (`Solver_ssa_nrm_space`) that additionally tracks the visited-state table for reachability and probability output (`Solver_ssa_reachability`), effectively providing simulation-based state-space exploration for models too large for `Ctmc_ssg`.

Tau-leaping acceleration of Markovian queueing simulation, which inspired parts of this architecture, is studied in [168]; the production engines in LINE sample exactly and do not leap.

NRM eligibility gate. The `default` method selects the NRM engine unless a feature the NRM path does not yet cover forces a fallback to `serial`. A model is routed to serial when it contains Fork/Join nodes, state-dependent routing other than JSQ (which NRM handles natively), the delayed-hit cache retrieval system, or a non-exponential service process outside the exactly expandable families (see below). Stochastic Petri net models (Place/Transition nodes) take a dedicated NRM path (`nrm_spn`) under the default method, while an explicit `serial`/`ssa` request is honoured by the serial `afterEvent` engine. All other extended features – finite-capacity regions under both DROP and WAITQ rules, POLLING, KCHOICES with Anselmi's SQ(d,N) memory, retrials, pass-and-swap, state-based (queue-length) balking and memoryless reneging, and G-network signals – are simulated by the NRM engine directly. Phase-type service is expanded exactly at the INF/PS family and, through the in-service phase multiset (`svcph`), at the non-preemptive buffered (FCFS/HOL/SIRO) family; RROBIN routing draws the entry phase from the routed node. Outside these cases a non-exponential process falls back to serial. A net with *fluid places* (`Place.setFluid`) is simulated as a piecewise-deterministic Markov process [53]: `nrm_spn` advances every level linearly between firings at the net flow rate of the enabled modes, and schedules each guard crossing and each bound hit as an event that changes the drift but fires no transition. `default` resolves to `nrm` on such a net and `serial` refuses it.

11.3 Statistical output analysis

The trajectory length is controlled by `options.samples` and the initial transient is removed by discarding a configurable warm-up fraction. Steady-state estimators are ratio estimators over the retained horizon: utilizations and mean queue lengths integrate state indicator functions against sojourn times, throughputs count

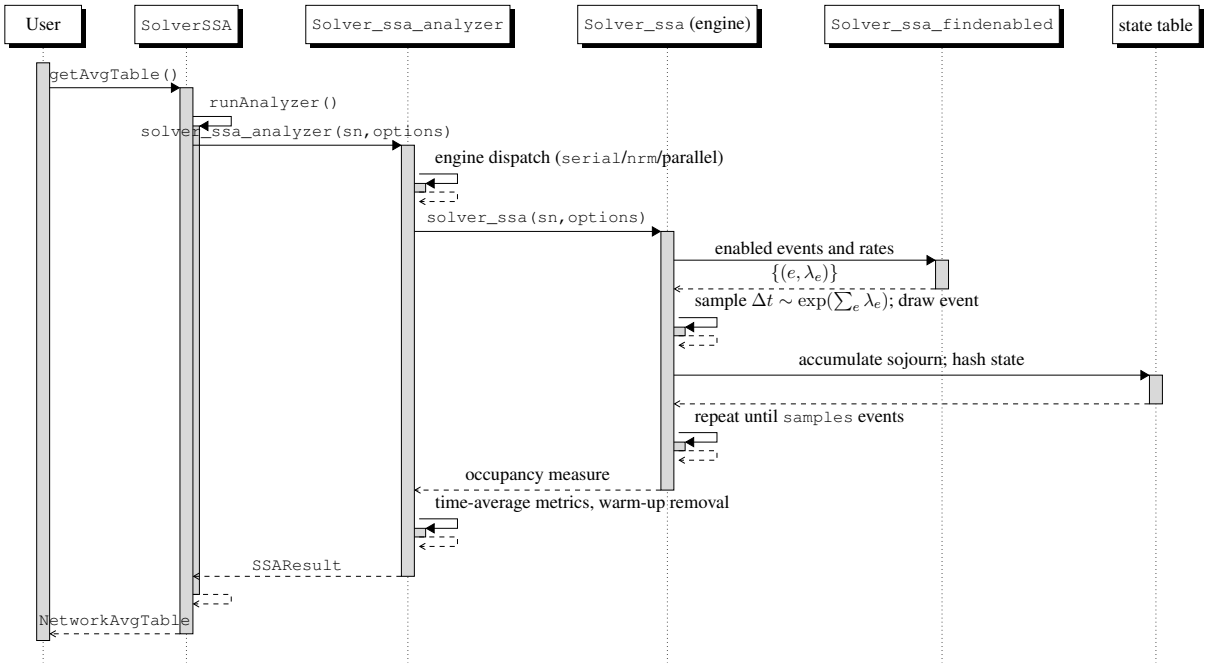


Figure 11.1: Sequence diagram of `SolverSSA` with the serial direct-method engine.

event firings per unit time, and response times follow from Little's law applied at station level [123]. Replicated runs (parallel engine) report confidence intervals from the sample variance of replication means. The random number generator is seeded from `options.seed`, and all engines derive per-replication substreams deterministically from it, making runs bit-reproducible across the codebases for a fixed seed.

11.4 Architectural flow

Figure 11.1 shows the serial path; the NRM engines replace the per-step rate enumeration with priority-queue updates driven by the synchronization dependency graph.

11.5 Implementation notes

The engines snapshot and restore any `sn` fields they must specialize (e.g., effective server counts for infinite-server stations), so that the shared structure is never mutated across calls. Immediate-feedback suppression (`sn.immfeed`) short-circuits departures that re-enter the same station in the same class, matching the CTMC reduction of vanishing states. Fractional routing stoichiometry is disallowed: events fire with integral job movements, and the NRM dependency graph is rebuilt whenever class switching changes the enabling structure. For models with cache nodes, the engines update the cache list variables inside `afterEvent`, so hit/miss ratios are simulated exactly rather than approximated; the NRM path handles the base cache (read modelled as an immediate state-dependent hit/miss class switch) and, on the serial engine, the delayed-hit

retrieval system. The serial engine is the reference implementation; the NRM engine is now the default whenever eligible and is validated against serial (and, on enumerable models, against CTMC) in the parity suite. The serial and NRM engines are distinct samplers of the same chain, so at finite sample counts their estimators agree only within simulation error, not bit-for-bit.

11.6 Feature and method support

`SolverSSA` samples the same Markov chain as `SolverCTMC`, so its feature envelope (Table 11.1) is that of the model semantics, not of a particular engine: `serial`, `parallel`, `nrm`, and `nrm.space` accept the same models and differ only in cost and in output. What differs between the NRM and serial engines is not the envelope but which of the two the `default` method selects; the NRM eligibility gate (Section 11.2) determines that choice, and any model it excludes is served identically by the serial engine. The `parallel` engine adds replication confidence intervals, and `nrm.space` additionally exposes reachability and state probabilities. Non-Markovian distributions enter through phase-type expansion (○). Unlike CTMC, SSA also admits external (EXT) scheduling.

Table 11.1: Feature envelope of `SolverSSA` (● supported, ○ via phase-type expansion).

Feature	SSA	**Feature**	SSA
Open / closed / self-looping class	●	Class switching	●
Priority classes	●	Source, Sink, Queue, Delay	●
Router	●	Fork / Join	●
Cache node	●	Cache repl. (RR/FIFO/SFIFO/LRU)	●
Place, Transition (timed/immediate)	●	Enabling / inhibitor arc	●
Signal / catastrophe (G-network)	●	Finite capacity region (DROP/WAITQ)	●
Exponential	●	Phase-type (Erlang, HyperExp, Coxian, APH, PH)	●
MAP, MMPP2	●	General (Gamma, Weibull, Lognormal, LogLogistic, MassAtOrigin, Pareto, Uniform, Det)	○
INF, FCFS, PS	●	DPS, GPS, LPS	●
LCFS, LCFS-PR	●	SEPT, LEPT	●
SIRO, HOL	●	POLLING	●
PAS (pass-and-swap)	●	Priority variants (`*PRIO`)	●
EXT (external self-loop)	●	Routing PROB, RAND	●
RROBIN, WRROBIN	●	JSQ, KCHOICES	●
Load dependence	●		
Retrial (orbit)	●	Balking	●
Reneging (impatience)	●	Steady means (`getAvg`)	●
Confidence intervals (parallel)	●	Probabilities (`nrm.space`)	●
State-space reachability (`nrm.space`)	●		

11.7 Method algorithms

Table 11.2 gives high-level pseudocode for each simulation engine.

Table 11.2: High-level pseudocode of `SolverSSA` engines.

Engine	Pseudocode
`serial`	1. from the current state enumerate enabled synchronizations and rates $\{(e, \lambda_e)\}$ (`findenabled`) 2. sample sojourn $\Delta t \sim \exp(\sum_e \lambda_e)$; draw event e with probability $\lambda_e / \sum \lambda_e$ 3. apply `afterEvent`; accumulate sojourn against the hashed state 4. repeat until `samples` events; time-average the occupancy measure after warm-up removal
`parallel`	run R independent trajectories on distinct substreams; average the per-replication stationary estimators; report confidence intervals from their variance
`nrm`	maintain a tentative firing time per synchronization in an indexed priority queue; fire the minimum; resample only the rates affected by the executed event (dependency graph), $O(\log E)$ per step. Default engine when the model passes the eligibility gate (no Fork/Join, no non-JSQ state-dependent routing, no delayed-hit retrieval, service phase-type only in the exactly expandable INF/PS and buffered families); a dedicated `nrm_spn` path handles Place/Transition models
`nrm.space`	as `nrm`, additionally recording the visited-state table for reachability and state-probability output

11.8 Bibliographic notes

The direct method is due to Gillespie [79]; the next reaction method to Gibson and Bruck [78]. Tau-leaping applied to queueing networks is explored in TauSSA [168]. General discrete-event and replication methodology can be found in [119]. The synchronization formalism that both `SolverSSA` and `SolverCTMC` interpret descends from the event-based semantics of stochastic process algebras and stochastic automata networks [90, 173], adapted to the extended queueing network features of LINE [30].

Part III

Compositional solvers

Chapter 12

SolverENV

12.1 Overview

`SolverENV` analyzes queueing networks operating in a random environment: a finite set of environment stages $e = 1, \ldots, E$, each associated with a `Network` submodel describing the system dynamics while the environment sojourns in that stage, and a stochastic process governing stage transitions. Typical applications are systems with breakdowns and repairs, bursty reconfigurations, or phased workloads [36, 144]. The environment process is specified by inter-stage transition distributions; Markovian environments yield an exact modulated-chain semantics, while phase-type stage holding times are supported by expansion. The solver is an ensemble meta-solver (`jline/solvers/env/`): the user supplies one `NetworkSolver` per stage, and `SolverENV` coordinates them.

12.2 The blending method

The default analysis implements the blending scheme of Casale, Tribastone and Harrison [37]. Let $\boldsymbol{\pi}^{(e)}$ denote the metric vector (e.g., mean queue lengths) of the stage-e submodel, and let α_e be the stationary probability that the environment resides in stage e, computed from the environment generator or its semi-Markov embedding. Blending accounts for the fact that a stage does not start from its own steady state but from the state inherited at the preceding transition: the method iterates

1. solve each stage submodel for its equilibrium metrics, given its current initialization;
2. compute stage-entry distributions by applying the user-specified reset functions (`resetQLFun`, mapping queue lengths at a transition $e \to h$ to the initial condition of stage h);
3. update the per-stage transient-averaged metrics over the stage holding time, using the stage solver's transient handle;
4. mix the stage metrics with weights α_e and repeat until the blended metrics converge.

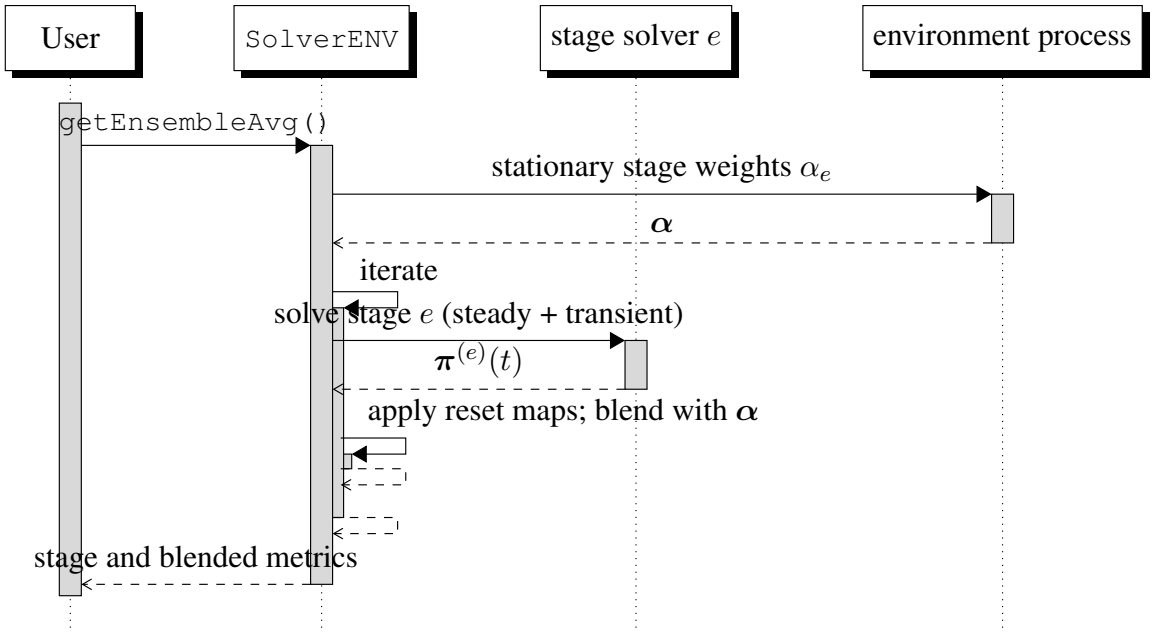


Figure 12.1: Sequence diagram of `SolverENV` with the blending method.

This scheme is exact in the limits of fast and slow environments and interpolates between them; its accuracy for two-stage fluid environments is studied in [36]. Because step 3 requires transient solutions, the natural stage solvers are `SolverFLD`, `SolverCTMC`, and `SolverMAM`.

12.3 The state-vector method

The `statevec` method avoids the blending approximation for stages whose submodels admit an explicit state-space representation. Each stage is compiled to a generator (CTMC backend) or to a level-structured process (MAM/LDQBD backend, Section 8.3); the solver then assembles the modulated joint process over (environment stage, system state), with per-stage entry distributions `piEnter`, per-destination exit distributions `piExitDest`, and sojourn-end distributions `piTimeAvg` computed from the stage holding-time distributions. Metrics follow by direct solution of the joint process. This is exact for Markovian environments and phase-type holding times, at the cost of the product state space; it corresponds to the classical Markov-modulated construction of queues in random environments [140, 162]. Semi-Markov environments with general holding times are supported through the `smp` option by embedding at transition epochs.

12.4 Architectural flow

Figure 12.1 shows the blending loop; the `statevec` method replaces the loop body with the assembly and solution of the joint modulated process.

12.5 Implementation notes

Stage submodels must be structurally compatible (same stations and classes) so that reset maps are well-defined; `SolverENV` verifies stage-solver admission (`supports`) upfront and rejects non-Markovian transition distributions unless the semi-Markov path is selected. The blending iteration reuses each stage solver instance, so expensive structural compilations are performed once per stage. The `statevec` backends are validated to produce bit-identical results across the MATLAB, Java, and Python implementations.

12.6 Feature and method support

`SolverENV` is a meta-solver: the modelling features admissible *within* a stage are those of the per-stage solver, so the columns of Table 12.1 concern the treatment of the environment process and holding times, which is where the methods differ. `blending` requires transient stage solutions (natural backends: FLD, CTMC, MAM) and is exact in the fast- and slow-environment limits; `statevec` assembles the exact joint (stage, state) process from CTMC or MAM/LDQBD stage backends; `smp` embeds a semi-Markov environment at transition epochs. Its environment can be fitted from observed stage histories by `infer_smp` (intensity or sojourn approach, parametric holding-time families, optional covariates), whose fit `infer_smp_env` turns into an `Environment` over given stage models [6].

Table 12.1: Feature support of `SolverENV` methods (● supported, ○ approximate/conditional, blank unsupported).

Feature	blending	statevec	smp
Environment process			
Markovian environment	●	●	○
Semi-Markov environment	○		●
Phase-type stage holding times	●	●	●
General stage holding times	○		●
Exact for Markovian environment		●	
Stage submodel and coupling			
Open / closed stage classes	●	●	●
Class switching	●	●	●
Reset maps (`resetQLFun`)	●	○	○
FLD stage backend	●		
CTMC stage backend	●	●	●
MAM / LDQBD stage backend	●	●	○
Output			
Ensemble averages (`getEnsembleAvg`)	●	●	●
Per-stage metrics	●	●	●

12.7 Method algorithms

Table 12.2: High-level pseudocode of `SolverENV` methods.

Method	Pseudocode
`blending`	1. compute stationary stage weights α_e from the environment generator 2. repeat: solve each stage submodel (steady + transient over the stage holding time) from its current initialization 3. apply the reset maps (`resetQLFun`) to obtain stage-entry distributions 4. mix the per-stage metrics with weights α_e 5. until the blended metrics converge
`statevec`	1. compile each stage to a generator (CTMC) or level-structured process (MAM/LDQBD) 2. assemble the joint (stage, system state) modulated process with `piEnter`/`piExitDest`/`piTimeAvg` 3. solve the joint process directly (exact for Markovian environments)
`smp`	embed a semi-Markov environment at transition epochs; weight stage solutions by the embedded stationary distribution and mean holding times

12.8 Bibliographic notes

Blending of randomness in closed networks is introduced in [37]; fluid analysis of two-stage random environments in [36]. Markov-modulated queues are classical in the matrix-analytic literature [117, 140, 162]. The application of LINE to reliability-aware software performance, which motivated this solver, is described in [144], with related model-driven reliability formalisms in [159, 179].

Chapter 13

SolverLN

13.1 Overview

`SolverLN` is the native layered-network meta-solver of LINE. It evaluates layered queueing networks (LQNs), the software-performance formalism in which processors host tasks, tasks expose entries, entries execute activity graphs, and activities issue synchronous or asynchronous calls to other entries [67, 71, 153]. Contention arises simultaneously at hardware (processor) and software (task multiplicity) layers, and the standard solution strategy is layer decomposition: the LQN is split into an ensemble of ordinary queueing networks, one per layer, whose parameters depend on each other's solutions; a fixed-point iteration alternates layer solutions until the metrics converge. This is the architecture of the classical LQNS solver's method of layers [153] and of SRVN-style task decomposition [67, 70]; `SolverLN` generalizes it by allowing an arbitrary LINE solver per layer, so each submodel may be solved by MVA, NC, MAM, FLD, or simulation.

The solver resides in `jline/solvers/ln/` as a subclass of `EnsembleSolver`, operating on the `LayeredNetworkStruct` representation (`lqn`) documented in the user manuals.

13.2 Layer construction

`buildLayers()` traverses the LQN graph and materializes one `Network` submodel per served element: a host layer for each processor (its tasks act as customers competing for the processor) and a task layer for each non-reference task (its callers act as customers competing for the task's entries). `buildLayersRecursive` walks the call graph so that each layer receives: a delay station representing the callers' think and non-local service times, a queueing station representing the served resource with its multiplicity and scheduling discipline, and one closed class per caller chain, with populations equal to caller multiplicities and demands compiled from activity host demands and call means. Phase-two activities, activity precedences (sequences, AND/OR forks and joins, loops), and replies are compiled into the class routing and the service processes of the layer submodels. Caches appear as `CacheTask` elements and produce cache nodes in the affected layers.

13.2.1 Two layerings and two encodings

A method name of `SolverLN` carries two independent decisions, and its two-part form states both: the part before the dot is the *layering*, which fixes what a submodel is, and the part after it is the *encoding*, which fixes how an activity graph is written into that submodel. `lqn_ln_method` normalizes a requested name onto the set the solver dispatches on, and the outcome is stored once in the `lnmethod` property; every later dispatch — `updateLayers`, `updateMetrics`, `updateThinkTimes`, `getEnsembleAvg` — reads that property and never `options.method`, so a reconstruction cannot disagree with the layers it is reading.

`srvn.cs`: the graph as routing. One submodel per served element, with the activity graph encoded as class switching. The walk creates a class for every task, entry, activity and call it reaches and lays down the routing that moves a job between them, minting a `Fork`/`Join` pair for an AND precedence, a `Router` per fork branch, and the class switch that an off-diagonal $P_{r,s}$ implies. A call whose target is served in this layer routes the job to the server station; a call that leaves the layer is charged as a delay at the client, and the mean number of calls is folded either into the demand (below one call per visit, a Bernoulli pass through a `.Aux` class) or into a geometric re-entry loop (above one). This is the encoding of every earlier release and serves the full LQN vocabulary.

`srvn.ph`: the graph as one phase-type law. The same layering, with each entry's activity graph composed instead into a single phase-type service law by the exact series-parallel reduction of Sahner and Trivedi [160]: a sequence by convolution, an OR branch by mixture, a loop by the geometric compound, and an AND fork by the serialized host law, since a processor sees the work of the parallel branches and not their elapsed time. A layer collapses to a two-station cycle, a client delay and the server, with one closed class per *calling task* rather than one per graph element. The per-element quantities the rest of the solver reads — `servt`, `residt`, `callservt`, `callresidt` — are recovered analytically from the series-parallel weights, splitting each visit's queueing inflation R/S across the leaves in proportion to their means, so the pieces sum back to R exactly. A `SetupTask` is prefixed to the composed law as the mixture $p\,(\text{setup} \to \text{entry}) + (1-p)$ entry with p the probability that the delay-off countdown expired first [72]. The encoding cannot represent a second phase (the composed law has no reply point), a forwarding call (whose target is not in the caller's graph), a cache task, a quorum AND join, a routed call group or a per-entry admission constraint, and it refuses such a model by name.

`srvn`: the alias. The default. It probes whether `srvn.ph` can serve the model — first the feature gate, then the composition of every entry workflow, non-destructively, keeping the composed laws for reuse — and builds `srvn.ph` where it can and `srvn.cs` where it cannot. Asking for `srvn.ph` by name is therefore not the same as taking the default: the alias falls back, the explicit name raises.

`flat.cs`: the squashed layering. Every processor and every called task becomes a station of one submodel. The builder is the same walk, generalized from a single served element to a set: a station is created per element, and each routing decision resolves the station of the element it concerns — an activity's host demand goes to the station of *its own* processor, a call to the station of the *callee*, an entry arrival to the station of the entry's task — falling back to the client delay when that element is not served here. Under

a single served element every one of those lookups returns the layer's own server, which is why the same code path serves both layerings. What changes is that two servers now contend inside one network instead of seeing each other through surrogate delays, and that a group of call targets can share a submodel, which is what makes a routed call group (`synchCallRoundRobin`, `synchCallJSQ`) expressible at all. Squashing is refused rather than approximated for a replicated element, a cache task or a setup task, each of which carries state that only a submodel of its own can hold. `flat` is the unconditional alias of `flat.cs`: it does not probe `flat.ph`, because a model is squashed in order to express the routed call groups that only the routing encoding dispatches.

`flat.ph`: the squashed layering, composed. The same single submodel and the same server set, but the activity graph is reduced into one phase-type law per (server, caller) rather than written as a class chain, so a caller task is ONE closed class visiting each server it uses once per invocation. Squashing does not conflict with the composition, because a task station's service law is already the *inflated* entry law — host demand plus call counts times callee service and waiting — which the outer fixed point updates, exactly as `lqns` does under `--squashed-layering`; a task appears twice, as a server station its own clients visit and as a client chain visiting its callees, so inflating and visiting sit at different levels and do not double count. Everything downstream is shared verbatim with `srvn.ph`, and the one genuine difference is the surrogate delay, now a single closure: a client delay stands for whatever of a thread's cycle the model does *not* hold as a station of its own, which under `srvn.ph` reduces term for term to the previous per-layer formulas and under `flat.ph` leaves only the think times. Two traps are worth naming: a class that never visits a station must be marked `Disabled` there rather than given a small placeholder law, since an FCFS station carries one service law across its classes and a placeholder is mixed into the multiserver correction; and the inflation cap is bounded by the population of the *model*, not of the station. What `flat.ph` costs is the routed call group, whose dispatch the composed law folds into one visit.

Because a squashed layer serves many elements at once, "the station of element i" is no longer the layer's single server: the layer carries a map from LQN element to station index (`model.attribute.serverIdxOf`, `Layer::server_idx_of`), populated under *both* layerings so that a consumer reads it without knowing which one built the layer, and a companion lookup resolves the station a *class* is served at (the processor of an activity, the called task of a call). The reconstruction in `getEnsembleAvg` then walks the layer's host stations one at a time, charging each processor only with the activities that run on it.

An activity's synchronous calls to a *group* of target entries (`synchCallRoundRobin`, `synchCallJSQ`; `lqn.callgroups{}`, holding the caller, the routing strategy and the target list) are readable only by the squashed (`'flat'`) layering, since the grouped targets must land in one shared submodel for a dispatch decision to be meaningful, and by a layer solver capable of state-dependent routing (`CTMC` or `SSA`); `assertCallGroups` rejects the model before it reaches any other layer solver, rather than letting it silently resolve to a probabilistic split that would misreport a deterministic policy as one. `sn_rt_stations` performs the companion projection for open-network solvers outside the layered path, absorbing a round-robin router's non-station row into the surrounding stations' routing matrix by $P_{\mathrm{st}} = P_{AA}+P_{AB}(I-P_{BB})^{-1}P_{BA}$.

A host or task may declare a service-rate dependence on the station that represents it in its layer (`setLoadDependence`, `setClassDependence`, `setJointDependence`, stored in `lqn.lldscaling`, `lqn.cdscaling` and `lqn.jdscaling`). `buildLayersRecursive` applies these right after the admis-

sion constraints, once per replica of the served element. A load dependence transfers verbatim, since it reads the total population of the station. The class- and joint-dependence handles instead read a per-*operand* vector, where the operands are the tasks of a host or the entries of a task, in `tasksof`/`entriesof` order; the layer station reads job classes, so each operand is resolved to the set of layer classes that carry its work (the classes of its activities in a host layer, of its incoming calls in a task layer) and the declared handle is lifted into one over classes. A scalar return applies to every class of every operand and classes belonging to no operand keep a unit scaling. Where each operand resolves to a single class the lift preserves the product-form structure of a class dependence; otherwise the lifted handle is installed as a joint dependence, numerically identical but without the exactness claim. The lift must be index-space agnostic: layer solvers call the handle with a per-class vector under CTMC and the exact recursions, and with a per-chain vector under `Solver_amvald` and the convolution of Section 10.2, so it branches on the length of its argument, resolves the chain columns lazily from `sn.chains`, and returns a vector of the length it was given.

13.3 The fixed-point iteration

`runAnalyzer()` executes the ensemble loop, per iteration `it`:

1. `updateLayers(it)`: refresh the parameters of every layer submodel from the metrics of the other layers computed at the previous iteration;
2. solve each layer with its assigned solver (in parallel where the backend permits);
3. `updateMetrics(it)`: map layer results back to LQN elements (entry throughputs and utilizations, activity response times), with a default moment-matching update (`updateMetricsDefault`) and a moment-based variant (`updateMetricsMomentBased`) that propagates higher service moments to make the decomposition sensitive to service variability;
4. `updateThinkTimes(it)`: recompute the callers' effective think times in every layer as the response time they experience elsewhere, which is the coupling device of the method of layers [153];
5. `updateRoutingProbabilities(it)`: refresh call routing for OR-forks and multiplicity-dependent branching;
6. test convergence of entry-level metrics against `options.iter_tol`; under-relaxation and, for models with coexisting fixed points, warm-started initialization stabilize the iteration.

Upon convergence, `getEnsembleAvg` returns per-layer tables and the LQN-level summary table (task and entry throughputs, utilizations, response times). Transient ensemble analysis (`getTranAvg`) integrates the layer submodels' transient solutions over the iteration fixed point.

Because each layer is an ordinary `Network`, all extended features of Part II solvers are available inside layers; e.g., using `SolverFLD` per layer yields a scalable LQN solver in the spirit of fluid LQN analysis [143, 179], while `SolverMAM` layers propagate service variability. Correctness is routinely cross-checked against the external LQNS solver and its `lqsim` simulator [67,69], accessed through the `SolverLQNS` wrapper, and against native LQN simulation in LDES (Chapter 7).

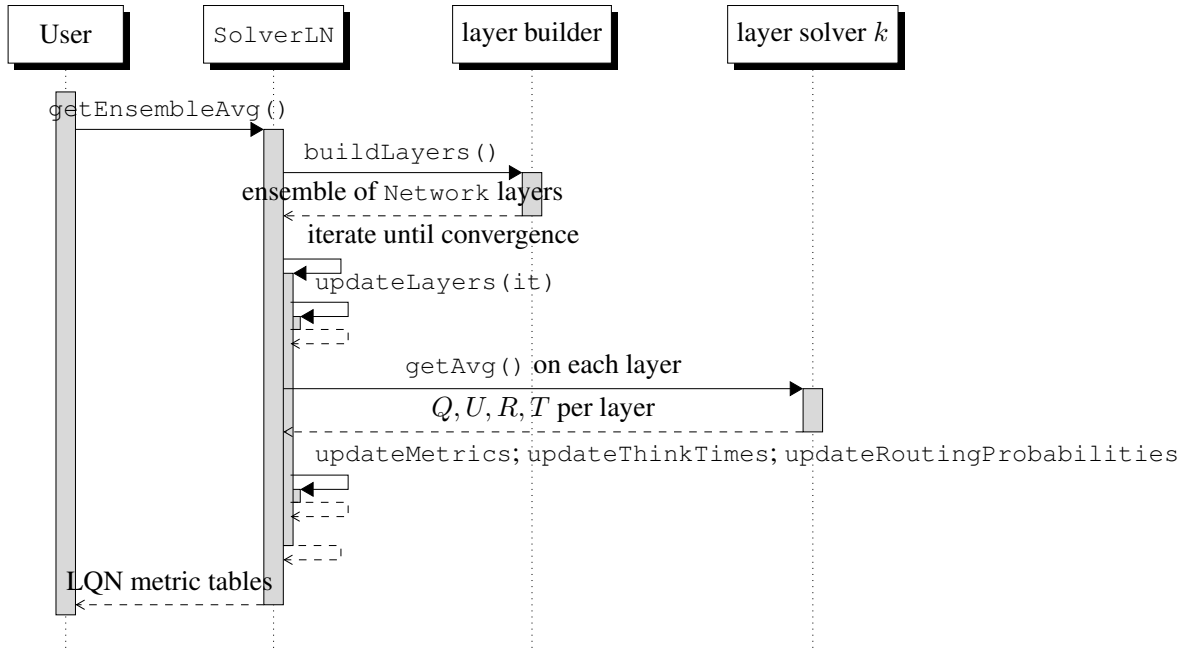


Figure 13.1: Sequence diagram of `SolverLN`: layer decomposition with fixed-point coupling.

13.4 Architectural flow

Figure 13.1 shows the ensemble loop. The per-layer solver is selected by a `SolverFactory`, defaulting to `SolverMVA`.

13.5 Implementation notes

The solver caches layer structures across iterations and re-parameterizes them in place; the AMVA layers additionally reuse the previous fixed point as a warm start, which both accelerates convergence and pins the iteration to a consistent solution when the AMVA equations admit multiple fixed points. Convergence is declared on the maximum relative change of entry metrics; a small number of extra smoothing iterations guard against oscillation. Second-phase service and asynchronous calls introduce non-renewal effects that layer decomposition captures only approximately; discrepancies of a few percent against simulation on adversarial models are expected and documented in the test suite, consistent with the accuracy envelope reported for analytic LQN solvers [67].

13.6 Interlock correction

When two tasks on a shared downstream call path impute traffic to one another, treating each layer's arriving demand as independent overstates the load: `initInterlock` performs the LQNS V5 static-path

analysis [67] once at solver initialization, locating interlocked task pairs and the phase-2 and external sources feeding them, and `updatePopulations`/`updateThinkTimes` apply the resulting interlock matrix `util_ilock` (element (i, j): the fraction of task i's utilization to impute to task j) as a correction to the call residence times each layer solves against, gated by `options.config.interlocking` (default `true`).

The correction is not simply forced onto every layer, because applying it can require the layer's MVA dispatch to be routed to a different underlying algorithm than the one it would otherwise use, and `mva_carries_interlock` (§9.10) exists to prevent exactly that substitution. Only two kernels carry the correction matrix: exact `pfqn_mva` and the `Solver_amvald` forward step. A layer whose method resolves to `mvac`, `sqd`, `sum`, `qna`, `rqna` or `rqt` reaches neither, so it is solved without the correction even when `interlocking` is on and the model has interlocked paths; `mva_is_bas_model` makes the equivalent check for closed single-chain BAS-blocking models routed to `solver_sqd`. The alternative – swapping the layer to a kernel that can carry the correction – was tried and rejected: it can turn a converging outer Picard iteration into a limit cycle, since the perturbation destabilizing the fixed point is the algorithm swap itself and not the numeric magnitude of the correction, observed as a period-17 oscillation on a non-converging layered benchmark (`model_C2_L2_T4_P2_1_doesnotconverge_v3`) even after scaling the interlock matrix toward zero.

13.7 Feature and method support

`SolverLN` admits the full LQN element vocabulary (`getLNFeatureSet`: hosts/processors, tasks, entries, activities, synchronous and asynchronous calls, and the sequence, AND, and OR activity precedences); every layer is materialized as an ordinary `Network`, so the "method" axis in Table 13.1 is the per-layer solver chosen through the `SolverFactory` (default `SolverMVA`). All per-layer solvers admit the LQN elements; they differ in the within-layer effects they capture, e.g., service-variability sensitivity (moment-based MVA update, MAM, LDES), large-population scaling (FLD), exact product-form per layer (MVA, NC), and transient ensemble analysis.

Table 13.1: Per-layer solver support in `SolverLN` (● supported, ○ approximate/conditional, blank unsupported).

Feature	MVA (def.)	NC	MAM	FLD	LDES
LQN elements (admitted by all layer solvers)					
Host / Processor	●	●	●	●	●
Task (multiplicity)	●	●	●	●	●
Reference task	●	●	●	●	●
Entry, Activity graph	●	●	●	●	●
Synchronous call	●	●	●	●	●
Asynchronous call	●	○	○	○	●
Precedence: sequence	●	●	●	●	●
Precedence: AND fork/join	●	○	○	○	●
Precedence: OR fork/join	●	●	●	●	●
Second phase	●	○	○	○	●

Feature	MVA (def.)	NC	MAM	FLD	LDES
Cache task	●	●	○	○	●
Within-layer capability					
Exact product-form per layer	○	●			
Server load dependence	○	●		○	●
Server class/joint dependence	○	●			○
Service-variability sensitivity	○		●	○	●
Large-population scaling	○		○	●	○
Output					
Ensemble averages (`getEnsembleAvg`)	●	●	●	●	●
Transient ensemble (`getTranAvg`)			○	●	●

13.8 Method algorithms

Table 13.2: High-level pseudocode of `SolverLN` (layer decomposition).

Stage / method	Pseudocode
`buildLayers`	resolve the method name into a layering and an encoding, once, into `lnmethod`; then traverse the LQN graph and materialize the submodels
`srvn.cs`	one `Network` per served element (host layer per processor, task layer per non-reference task) with a delay station for callers, a queueing station for the resource, and one closed class per caller chain; the activity graph becomes class routing
`srvn.ph`	same layering, but each entry's graph is composed into one phase-type law by series-parallel reduction, so a layer is a two-station cycle with one class per calling task; per-element metrics are split back out of the composition weights
`srvn`	probe `srvn.ph` against the model (feature gate, then composition of every entry workflow); build it if it serves, `srvn.cs` otherwise
`flat.cs`	one `Network` for the whole model: a station per processor and per called task, with every hop resolving the station of the element it concerns rather than a single server
`flat.ph`	the same single submodel, but each (server, caller) activity graph is composed into one phase-type law, so a caller task is one closed class visiting each server it uses once per invocation; refused on routed call groups
iteration	1. `updateLayers(it)` : refresh each layer's parameters from the other layers' previous metrics 2. solve each layer with its assigned per-layer solver (default `SolverMVA`) 3. `updateMetrics`: map layer results back to entries/activities (default or moment-based update) 4. `updateThinkTimes`: set caller think times to the response times experienced elsewhere 5. `updateRoutingProbabilities`: refresh OR-fork and multiplicity-dependent branching 6. test entry-metric convergence vs. `iter_tol` (under-relaxation, warm start); repeat
moment-based	as above but propagate higher service moments in step 3 to make the decomposition sensitive to service variability

Stage / method	Pseudocode
`nlp`	1. fold every activity graph into visit counts (sequence, loop, OR-branch); refuse an AND-fork unless `nlp_fork='exp'`, which composes it by the expected maximum of exponential branches 2. write flows, entry composition and thread closures as linear constraints, and the QD-AMVA congestion law and Little's law as residuals, with each station served at rate $r_i(n)$ (multiservers, `config.lld`) and $\min(n, m)$ replaced by a smooth minimum of sharpness `nlp_alpha` 3. root the system by least-squares Newton, falling back to a descent bounded by `nlp_maxiter`; no layers, no outer fixed point [35, 153]

13.9 Bibliographic notes

The LQN formalism consolidates the stochastic rendezvous network model of Woodside et al. and the method of layers of Rolia and Sevcik [153]; the canonical solver architecture and semantics are given by Franks et al. [67, 69, 71]. Enhancements relevant to this implementation include activity and phase modeling [70], multiserver approximations for large task pools [192], fluid and mean-field LQN analysis [143, 179], and LQN generation from software design models [81, 110, 120]. Applications of `SolverLN` within model-driven engineering pipelines are described in [30, 144].

Chapter 14

SolverUQ

14.1 Overview

`SolverUQ` is the uncertainty-quantification meta-solver. It addresses the setting in which model parameters, typically service or arrival distributions, are not known exactly but only up to a prior: the user attaches a `Prior` object to a service or arrival process, listing alternative distributions with prior weights. Such epistemic uncertainty is the norm when demands are estimated from noisy measurements [171, 185, 186], and propagating it to the performance metrics is preferable to reporting point estimates alone.

The solver resides in `jline/solvers/uq/` as a subclass of `EnsembleSolver`. It is solver-agnostic: a `SolverFactory` supplied at construction decides which concrete solver evaluates each realization.

14.2 Method

`SolverUQ` scans the model for `Prior` occurrences and records, for each, the node index, class index, and whether it parameterizes a service or an arrival process (`PriorInfo`). It then expands the model into the Cartesian family of deterministic networks $\{m_1, \ldots, m_K\}$, one per combination of prior alternatives, with joint weights w_k obtained as products of the marginal prior probabilities. Each realization is evaluated by a solver created from the factory, and the results are aggregated as prior-weighted expectations,

$$\mathbb{E}[\theta] \;=\; \sum_{k=1}^{K} w_k \, \theta(m_k), \tag{14.1}$$

for every metric θ in the average tables; higher posterior functionals (metric variances across realizations, ranges) follow from the same ensemble. This is a discrete-prior Bayesian model-averaging scheme: it is exact with respect to the stated prior, embarrassingly parallel across realizations, and inherits the accuracy profile of the underlying solver. When the priors arise from parameter inference, the weights can encode a posterior computed externally, e.g., by MCMC demand estimation [185] or by maximum-likelihood estimation from queue-length data [186]; surveys of demand-estimation approaches are given in [171].

14.3 Architectural flow

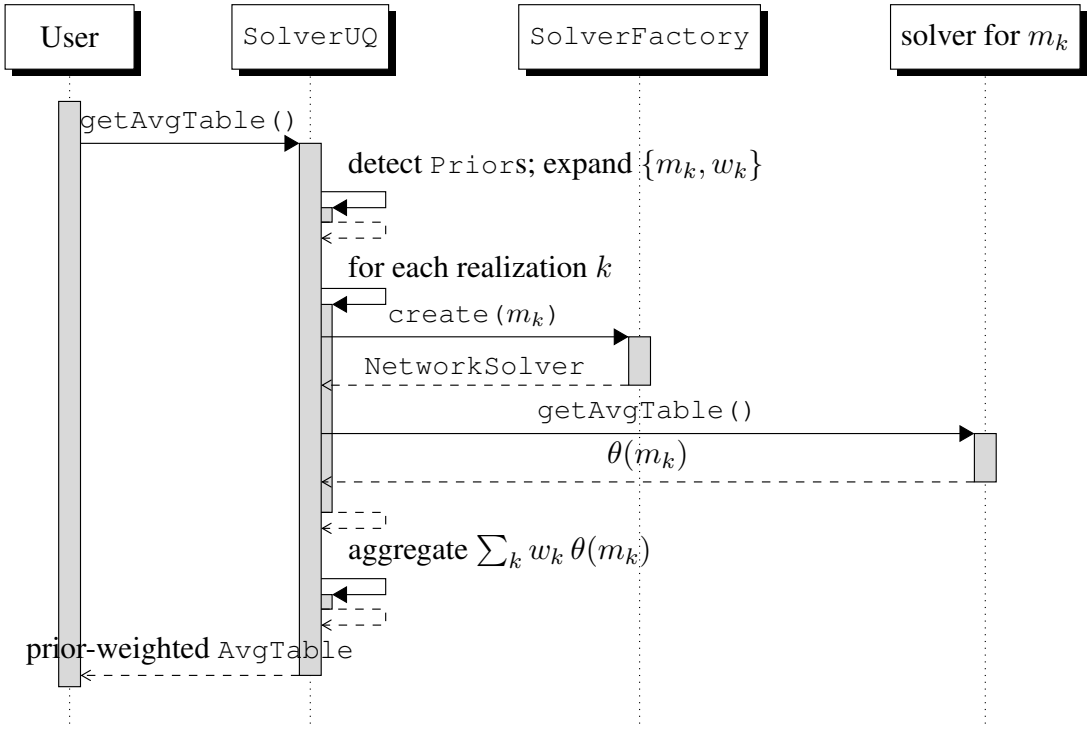


Figure 14.1: Sequence diagram of `SolverUQ`.

Figure 14.1 shows the expansion-evaluation-aggregation pipeline inherited from `EnsembleSolver`.

14.4 Feature and method support

`SolverUQ` is solver-agnostic: each realization m_k is evaluated by a solver built from the supplied `SolverFactory`, so the admissible modelling features are exactly those of that delegate (Chapters 9–7), and every metric the delegate reports is propagated as a prior-weighted expectation (14.1). The only feature specific to `UQ` is the `Prior` attached to a service or arrival process; it is supported wherever the delegate admits the corresponding deterministic distributions, since a prior is expanded into the Cartesian family of deterministic realizations before solution. Consequently `UQ` adds no per-method feature table of its own: consult the table of the factory solver for the concrete envelope, and read every supported metric as its posterior expectation (with variance and range available from the same ensemble).

14.5 Support-only uncertainty: interval-valued parameters

A prior asserts more than the user often knows. When only a range is defensible, `getInterval` (and its tabular form `getIntervalTable`) discards the weights and reports each metric as a pair of endpoints. The flag `exact` on the returned object separates two regimes, and they carry different guarantees.

If the model is a closed single-class network of load-independent single-server queues and delay stations, and every `Prior` parameterizes a service process, the gate `qualifiesForIntervalMVA` passes and the interval is the exact hull of MVA over the demand box, computed by `pfqn_mva_interval` [125]. Single-class MVA is monotone in each input: writing d_i for the demands, z for the think time and n for the population,

$$\frac{\partial x}{\partial d_i} \le 0, \quad \frac{\partial x}{\partial z} \le 0, \quad \frac{\partial x}{\partial n} \ge 0, \qquad \frac{\partial q_k}{\partial d_k} \ge 0, \quad \frac{\partial q_k}{\partial d_i} \le 0 \ (i \ne k), \quad \frac{\partial q_k}{\partial z} \le 0, \tag{14.2}$$

with the totals r and q increasing in every d_i and in n and decreasing in z. A function monotone in each argument attains its range on the input box at a corner of that box, so every endpoint is one ordinary MVA solution at the corner the sign pattern in (14.2) selects: $2(m+2)$ solutions for m demand intervals of nonzero width, the extra pair per thick station being needed because q_k and r_k are maximized with d_k high and the other demands low. The ensemble is not built at all on this path. Evaluating the MVA recursion in interval arithmetic instead is sound but far weaker: each input recurs at every step of the recursion, and the resulting enclosure is 14 times too wide on the example of [125].

Otherwise the solver falls back on the range of the metrics over the realizations it did evaluate, reporting `exact` as false together with the violated condition. That range covers the whole support for a discrete prior, but for a continuous one it is an *inner* approximation, since a quadrature node is the median of an equal-mass stratum and never an endpoint of the support.

The utilization is the one metric whose interval is not tight: $u_k = x d_k$ is absent from (14.2), so it is enclosed as a product of intervals intersected with $[0, 1]$, and is exact only where d_k has zero width.

The semantics deserve emphasis, because the output shape coincides with that of `SolverBA` (Chapter 4) while the guarantee does not. A bound family in `SolverBA` brackets the exact solution of a model whose demands are known; an interval here brackets the predictions of MVA over a set of admissible demands, conditional on the truth lying in that set, and is silent on the error of MVA itself. Intersecting or composing the two would claim a guarantee that neither provides.

14.6 Method algorithms

Table 14.1: High-level pseudocode of `SolverUQ`.

Method	Pseudocode
model averaging	1. scan the model for `Prior` occurrences (`PriorInfo`: node, class, service/arrival) 2. expand the Cartesian family $\{m_k\}$ of deterministic realizations with joint weights $w_k = \prod$ marginal prior probabilities 3. evaluate each m_k with a solver from the `SolverFactory` (embarrassingly parallel) 4. aggregate $\mathbb{E}[\theta] = \sum_k w_k\, \theta(m_k)$ per metric; variances and ranges from the same ensemble

Method	Pseudocode
interval (support only)	1. gate the model on `qualifiesForIntervalMVA`: single closed class, load-independent single-server queues and delays, priors on service processes only 2. build the demand box, widening each prior-carrying station to the range of its mean service time; fold the delay stations into the think-time interval 3. call `pfqn_mva_interval`: $2(m+2)$ ordinary MVA solutions at the corners selected by (14.2), no ensemble 4. on a refusal, return the range over the evaluated realizations instead, flagged inexact and carrying the violated condition

14.7 Bibliographic notes

Bayesian inference of queueing model parameters is developed in [185]; likelihood-based demand estimation in [186]; a comparative survey of resource-demand estimation appears in [171] and estimation in multi-threaded applications in [145]. Blending analyses that similarly average network solutions over an exogenous randomization are treated in [37]. The interval-valued alternative, in which parameters are bounded rather than distributed, and the monotonicity properties of single-class MVA that make its output intervals exact, are due to [125].

Part IV

External engines

Chapter 15

SolverJMT

15.1 Overview

`SolverJMT` is a wrapper around Java Modelling Tools (JMT) [13], an established open-source suite for the analysis of queueing networks by discrete-event simulation and by analytical product-form algorithms. LINE does not reimplement the JMT algorithms; the wrapper marshals the `sn` structure into a JMT input document, launches the appropriate JMT engine as a subprocess, and parses the results back into the standard LINE result object. The wrapper is the default simulation reference against which the native simulators (`SolverSSA`, `SolverLDES`) are validated, since JMT is an independently developed and widely used tool.

The wrapper resides in `jline/solvers/jmt/`. It requires `JMT.jar` on the classpath; the external GUI need not be installed.

15.2 Wrapper architecture

Two JMT engines are targeted, selected by `options.method`:

JSIMG (simulation) The default path exports `sn` to the JMT simulation format (`jsimg`/`jsimw` XML), including nodes, classes, routing, phase-type service, finite capacity, fork-join, blocking regions, and Petri net primitives. The wrapper writes the seed and the number of samples, runs the JMT simulator, and parses the per-measure output. Because seed and sample budget are marshalled explicitly, JSIMG results are reproducible and match LINE's native simulators for a fixed assignment.

JMVA (analytical) For product-form models the wrapper writes a JMVA document (`writeJMVA`) and sets the `algType` element from the method name: exact MVA (default), RECAL (`jmva.recal`), CoMoM (`jmva.comom`), Chow (`jmva.chow`), Bard-Schweitzer (`jmva.bs`/`jmva.amva`), AQL (`jmva.aql`), Linearizer (`jmva.lin`), and De Souza-Muntz Linearizer (`jmva.dmlin`), with tolerance and `maxSamples` attributes.

The `JMTResult`/`JMTOptions` classes carry the parsed metrics and the wrapper configuration; a temporary working directory holds the exported model and the JMT output, and is cleaned up after parsing.

15.3 Feature and method support

JMT has the broadest feature envelope among the analytical/simulation backends (Table 15.1); admission does not vary between the JSIMG engine and the network structure, whereas the JMVA analytical methods are restricted to product-form models. Non-Markovian distributions are simulated directly by JSIMG.

Table 15.1: Feature envelope of `SolverJMT` (● supported by JSIMG; ○ JMVA product-form only).

Feature	JMT	Feature	JMT
Open / closed / self-looping class	●	Class switching	●
Priority classes	●	Source, Sink, Queue, Delay	●
Router, Logger	●	Fork / Join	●
Cache node	●	Place, Transition, Queueing place	●
Finite-capacity blocking / region	●	Product-form (JMVA)	○
Exponential, Erlang, HyperExp	●	Coxian, Cox2, APH, PH	●
MAP, MMPP2	●	Normal, Gamma, Pareto, Weibull, Lognormal, Uniform, Det	●
Replayer / Trace	●	Cache repl. (RR/FIFO/SFIFO/LRU)	●
INF, FCFS, PS, DPS, GPS	●	LCFS, LCFS-PR	●
SIRO, HOL	●	SEPT, LEPT, SJF, LJF	●
SRPT (+PRIO)	●	LPS, POLLING	●
Priority variants (`*PRIO`)	●	EXT (external self-loop)	●
Routing PROB, RAND	●	RROBIN, WRROBIN	●
JSQ, KCHOICES	●	Steady means (`getAvg`)	●
Response-time distribution	●	Transient / confidence intervals	●

Table 15.2 gives high-level pseudocode of the wrapper dispatch.

Table 15.2: High-level pseudocode of `SolverJMT` methods.

Method	Pseudocode
JSIMG (default)	1. export `sn` to `jsimg` XML (nodes, classes, routing, distributions, capacity, measures) 2. write `seed` and `samples`; run the JMT simulator as a subprocess 3. parse the per-measure results into `JMTResult`
`jmva.*`	1. export `sn` to JMVA XML (`writeJMVA`); set `algType` (MVA/RECAL/CoMoM/Chow/Bard-Schweitzer/AQL/Linearizer/DM-Linearizer) with tolerance and `maxSamples` 2. run the JMT analytical engine; parse Q, U, R, T back

15.4 Architectural flow

15.5 Bibliographic notes

JMT is described by Bertoli, Casale and Serazzi [13]; its role within LINE and the validation methodology (two-sample t-tests of the native simulators against JMT) are discussed in [30]. The JMVA analytical algorithms it exposes are the exact and approximate product-form methods of Part II: MVA [149], RECAL [47], CoMoM [28], and the AMVA family [10, 40, 165, 189].

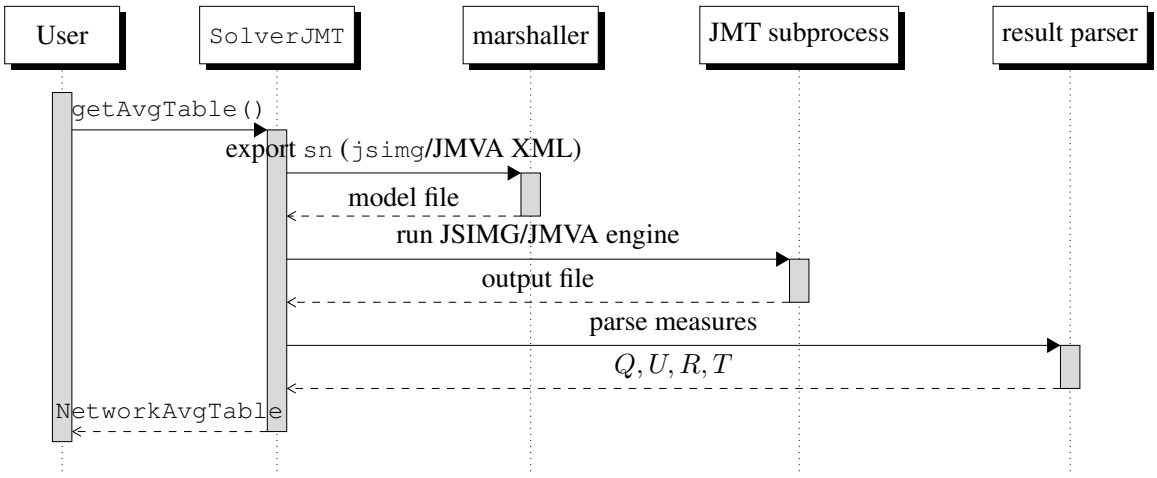


Figure 15.1: Sequence diagram of `SolverJMT`: marshal, run subprocess, parse.

Chapter 16

SolverLQNS

16.1 Overview

`SolverLQNS` is a wrapper around the Layered Queueing Network Solver (LQNS) suite of Franks et al. [67, 69], the reference toolset for the analysis of layered queueing networks by the method of layers and its analytical refinements, together with its discrete-event companion `lqsim` and the product-form network tool `qnsolver`. LINE does not reimplement these algorithms; the wrapper marshals the model to the external tool's input format, invokes the command-line executable as a subprocess, and parses the returned results. On layered models it serves as the external cross-check for the native `SolverLN` (Chapter 13) and `SolverLDES`, while on flat networks it cross-checks the multiserver AMVA methods of `SolverMVA`.

The wrapper resides in `jline/solvers/wrappers/lqns/` (with `qnsolver` marshalling in `Solver_qns_analyzer`). It extends `NetworkSolver`, accepting both a `LayeredNetwork` and a flat `Network` in the way `SolverJMT` hosts both JSIM and JMVA; this flat-network path replaces the former standalone `SolverQNS`. On a `LayeredNetwork` the station table `getAvgTable()` of `NetworkSolver` does not apply, and the layered results are returned by `getLNAvgTable()`. It requires the `lqns`, `lqsim`, and `qnsolver` executables on the system `PATH` (`SolverLQNS.hasQnsolver()` and `Solver_qns_analyzer.isQNSolverAvailable()` guard `qnsolver` availability); optionally `lqns` and `lqsim` may be dispatched through a container (`options.config.container`).

16.2 Wrapper architecture

Depending on the model type and `options.method`, the wrapper targets three external paths:

`lqns` (layered default) Writes the `LayeredNetwork` to the `.lqnx`/`.lqxo` XML schema, runs the analytical solver, and parses task, entry, and activity metrics (U, R, T). The method of layers is controlled through LQNS pragmas exposed by `options.config`, notably the multiserver approximation (`conway`, `rolia`, `zhou`, `suri`, `schmidt`, etc.) and the layering and convergence pragmas.

`lqsim` (layered simulation) When `options.method='lqsim'`, invokes the LQNS discrete-event simulator on the `.lqnx` file, passing the simulation length and seed to provide a simulation ground truth for the same LQN input.

`qns.*` (flat `Network` via `qnsolver`) On a flat `Network`, `options.method` selects the multiserver approximation mapped to `options.config.multiserver`: `qns.conway` [46], `qns.rolia` [71, 152, 153], `qns.zhou` [191, 192], `qns.suri` [175], `qns.reiser` [148, 149], and `qns.schmidt` [163] (`default` and `qns` select `rolia`, reported as `default/qns.rolia`; in `SolverAUTO` these belong to `lqns.qns.*`). The `qns*` methods are refused on a `LayeredNetwork`, and `qns.suri`/`qns.schmidt` are refused on the `qnsolver` path when a multiserver station is present since `qnsolver -m` does not accept them. Product-form or open models are marshalled directly to `qnsolver` (`Solver_qns_analyzer`) to obtain Q, U, R, T; non-product-form closed models are converted to an equivalent `LayeredNetwork` via `QN2LQN` and solved through `lqns`.

16.3 Feature and method support

On a `LayeredNetwork`, the wrapper admits the LQN element vocabulary (hosts/processors, tasks, entries, activities with sequence/AND/OR precedences, synchronous and asynchronous calls) as exported to the LQNS schema; the underlying station model is restricted to the disciplines LQNS itself supports (Table 16.1). The `lqns` and `lqsim` methods share the same input; `lqns` is analytical (subject to layer decomposition error) while `lqsim` is a simulation with statistical error. On a flat `Network`, all `qns.*` methods share the same feature envelope: open and closed classes, standard nodes (Source, Sink, Queue, Delay, Router, Fork/Join, Logger, Place/Transition), multiserver stations ($c > 1$), common Markovian and non-Markovian distributions (Exp, Erlang, HyperExp, Coxian, PH/APH, MAP/MMPP2, Det, Uniform, Normal, Lognormal, Gamma, Pareto, Weibull, Replayer), scheduling disciplines (INF, FCFS, PS/DPS/GPS, LCFS/LCFS-PR, SIRO, HOL, SJF/LJF, SEPT/LEPT, EXT), and routing strategies (PROB, RAND, RROBIN, WRROBIN, KCHOICES), with self-looping classes and non-product-form closed models handled via the `QN2LQN` fallback to `lqns`.

Table 16.1: Feature support of `SolverLQNS` on a `LayeredNetwork` (• supported, ◦ approximate).

Feature	lqns	Feature	lqsim
Host / Processor	•	Host / Processor	•
Task (multiplicity)	•	Task (multiplicity)	•
Reference task	•	Reference task	•
Entry, Activity graph	•	Entry, Activity graph	•
Synchronous call	•	Synchronous call	•
Asynchronous call	•	Asynchronous call	•
Precedence: sequence	•	Precedence: sequence	•
Precedence: AND fork/join	◦	Precedence: AND fork/join	•
Precedence: OR fork/join	•	Precedence: OR fork/join	•
Second phase	•	Second phase	•
Multiserver (conway/rolia/...)	◦	Multiserver	•
Exp / Erlang / HyperExp / Coxian	•	Non-exponential service	•
FCFS, PS host scheduling	•	FCFS, PS host scheduling	•
Steady means (`getAvg`)	•	Confidence intervals	•

Table 16.2 gives high-level pseudocode of the wrapper dispatch across layered and flat network inputs.

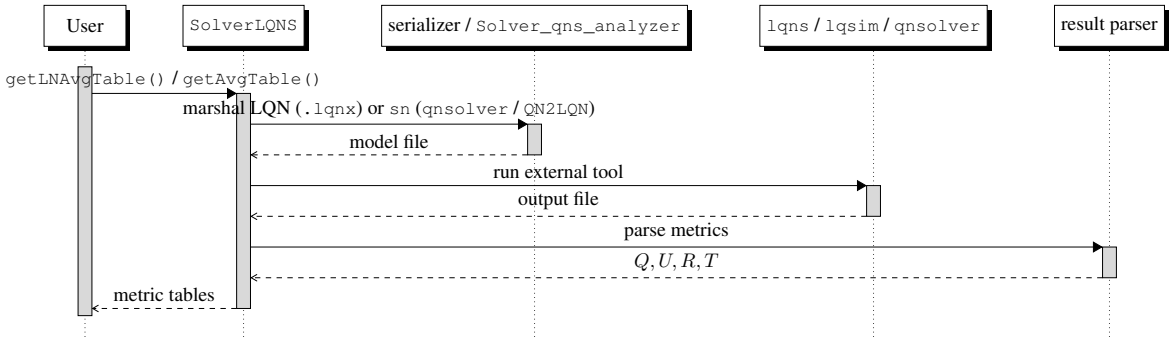


Figure 16.1: Sequence diagram of `SolverLQNS`: serialize LQN or flat `Network`, run `lqns`/`lqsim`/`qnsolver`, parse.

Table 16.2: High-level pseudocode of `SolverLQNS` methods.

Method	Pseudocode
`lqns`	1. serialize the LQN to `.lqnx` XML 2. build the `lqns` invocation with multiserver and layering pragmas from `options.config` 3. run `lqns` (locally or in a container); parse the `.lqxo` results into task/entry/activity metrics
`lqsim`	as `lqns` but invoke the `lqsim` simulator with the requested run length and seed; parse the simulated metrics and confidence intervals
`qns.*` (Network)	1. map the `qns.*` method to the `qnsolver` multiserver configuration (`conway`, `rolia`, `zhou`, `suri`, `reiser`, `schmidt`) 2. if product-form or open: marshal `sn` to `qnsolver` via `Solver_qns_analyzer`, run the subprocess, and parse Q, U, R, T 3. else (non-product-form closed): convert to an equivalent LQN with `QN2LQN` and solve it via `lqns`

16.4 Architectural flow

16.5 Bibliographic notes

The LQNS solver, its method of layers, activity and phase modeling, the `lqsim` simulator, and the companion `qnsolver` tool are described by Franks et al. [67, 69–71]. The multiserver approximations exposed by both `lqns` pragmas and `qnsolver` are those of Conway [46, 55], Reiser and Lavenberg [148, 149], Rolia [152, 153] in the per-class *Rolia-Franks* form of Franks [71], Zhou's arrival-theorem binomial approximation [191, 192], Suri et al. [175], and Schmidt [163], surveyed in [191, Ch. 3]; the underlying product-form theory is BCMP [11]. The comparison of `SolverLN` against LQNS and the accuracy envelope of layer decomposition are discussed in [30].

Part V

Extending LINE

Chapter 17

Registering a new solver or method

17.1 Scope

The preceding chapters describe the solvers as they are. This chapter describes how to add one. It is written for a researcher who has an algorithm and wants it to become a first-class citizen of LINE, that is, selectable by name, admissible only on the models it is valid for, classified as exact or approximate, attributed to its paper, and covered by tests.

Two granularities of extension are possible, and they differ in cost by roughly an order of magnitude:

A new *method* inside an existing solver. The algorithm shares the solution paradigm of a solver already present, e.g., it is another fixed-point approximation of the mean-value analysis equations. It is selected by `options.method` and reuses the solver class, its options, its result tabulation, and most of its feature envelope. Section 17.3 walks through this case.

A new *solver* when the algorithm encodes a solution paradigm of its own, i.e., it consumes `sn` and produces the metric matrices through machinery that no existing solver hosts. This requires a solver class, an options object, a feature envelope, a registration in the solver-type enumeration, and an entry in the automatic-selection logic. Section 17.4 walks through this case.

A useful test for which of the two applies: if the new algorithm would sit in the same `switch` as an existing one and return the same tuple (Q, U, R, T, C, X) from the same `sn`, it is a method. If it needs its own model representation, its own result type, or its own dispatch pipeline, it is a solver.

Remark 1 LINE *is maintained at functional parity across four codebases (MATLAB, the Java JAR, native Python, and the C++ port). An extension is complete only when it is present in all four, or when the omission is deliberate and documented. The checklists in this chapter therefore address every codebase in every edition of this manual; the layout table covers the three class-based codebases, and the C++ shape, which is a convention rather than a class hierarchy, is stated in the C++ blocks alongside each step.*

17.2 The solver contract

Every network solver inherits from `NetworkSolver` and must supply the members of Table 17.1. Three of them carry a semantic obligation that is easy to violate silently and is worth stating explicitly:

- `getFeatureSet` is a *claim*. A feature name declared here asserts that the solver handles that feature correctly, not that it tolerates its presence. Declaring a name the analyzer ignores produces a plausible wrong number instead of a rejection, which is the worst possible failure mode for a performance tool.

- `listValidMethods` is also a claim: every name it returns must run to completion on the model it was called with. A name that is listed and then raises is a false advertisement, and the automatic-selection logic will try it. Names admissible only under structural conditions must therefore be appended conditionally, as `SolverMVA` does for `qna` (open models), `sqd` (blocking-after-service), and `mvac` (closed models).

- `runAnalyzer` must reject before it computes. Feature admission, structural gates, and unsupported-option checks belong at the top of the method; once the numerical algorithm has started, a silent fallback to a different model is indistinguishable, in the output, from a correct solve.

Table 17.1: The `NetworkSolver` contract. Static members are marked S.

Member	Kind	Obligation
constructor	instance	call the base constructor with the solver name, then install the parsed options
`defaultOptions`	S	return the options object seeded by the solver type; never mutate a shared instance
`getFeatureSet`	S	the feature envelope of the solver, as a union over its methods
`methodFeatureSet`	S	per-method deltas on that envelope, when a method is narrower or wider than the solver
`supports`	S	envelope test plus any structural gate that has no feature name (finite capacity, regions)
`listValidMethods`	instance	the method names admissible *for this model*
`resolveMethod`	instance	expansion of `default` into a concrete method, when the choice is feature-driven
`runAnalyzer`	instance	reject, dispatch to an analyzer, store the result and the runtime
`getLibrariesUsed`	S	third-party libraries the solve went through, for attribution

Above the solver sit two registries that are not part of the class and are the ones most often forgotten:

- the *option and method whitelist*, consulted by the varargin parser to tell a method token from an option key. A token absent from it is not recognized in the multi-argument constructor form;

- the *method-type registry*, which classifies each method as exact, approximate, or bound, and as deterministic or randomized. It is what the solver banner prints, and it exists in four twins that must agree.

17.3 Walkthrough: adding a method to an existing solver

The running example of this chapter is a deliberately trivial algorithm, called here the *toy* method, registered under the name `toy`. On a closed single-class model it allocates the population in proportion to the service demands,

$$Q_i = N \frac{D_i}{\sum_j D_j}, \qquad X = \frac{N}{\sum_j D_j}, \qquad U_i = X D_i, \qquad R_i = Q_i / X, \tag{17.1}$$

and returns. It is crude, non-iterative, and makes no claim to accuracy; that is the point. Every step below concerns registration rather than numerics, so the algorithm is kept small enough to read in full, and a real extension differs from this one only in the body of steps 1 and 2. We first register the toy method inside `SolverMVA`, whose paradigm it shares in the weak sense that it consumes the same `sn` and returns the same tuple; Section 17.4 then re-registers the same algorithm as a solver of its own, so that the two costs can be compared directly.

Step 1: implement the handler. The handler is a pure function of `sn` and `options` that returns the metric matrices and, for an iterative method, the iteration count. It must not touch the object model, and it must not read solver state.

Step 2: wire the analyzer dispatch. The analyzer normalizes the method name, then selects the handler. Add one branch; do not fold the new name into a neighbouring case, because that re-baselines every golden result of the case you joined.

Step 3: advertise the method. Append the name to `listValidMethods`, conditionally when the method is admissible only on part of the model space. Equation (17.1) is written for a closed model, so the toy method is appended under the same guard that governs `mvac`. Advertising it unconditionally would be the archetypal violation of the second obligation of Section 17.2: on an open model the handler divides by an infinite population and the listed name raises.

Step 4: register the token in the option whitelist. The varargin parser separates option keys from method tokens by consulting a static list. A method absent from it is not recognized when passed among other arguments.

Step 5: declare the per-method feature envelope. If the new method supports strictly less, or strictly more, than the solver envelope, express the delta in the per-method feature set rather than by an ad-hoc test inside the handler. `SolverMVA` does this for `qna` (drops closed classes) and for `rqna` (adds the non-renewal point processes). The toy method reads only demands and populations, so it inherits the solver envelope unchanged and needs no entry. A method that ignores a declared feature rather than handling it, for instance one that reads `sn` as if every station were single-server, must narrow its envelope here; leaving it wide is how a solver comes to return a plausible number for a model it does not solve.

Step 6: classify the method. Add the method name to the method-type registry so that the solver banner reports it correctly. The registry exists in four twins that must agree: `matlab/src/io/line_method_type.m`, `jar/src/main/java/jline/solvers/MethodType.java`, `python/line_solver/solvers/base.py`, and the C++ `method_type.h`. The toy method is approximate and deterministic, so it joins the `APPROX_DET` group next to `amva`, `bs`, `aql`, and `lin`. An asymptotic expansion is approximate even when it is computed to machine precision; conversely, numerical truncation alone does not make an otherwise exact method approximate. A method absent from the registry prints as unclassified, and in one codebase only if the four twins were not updated together, which surfaces later as a spurious parity difference.

Step 7: record the attribution. Any method implementing a published algorithm requires an entry in the citation registry, `matlab/src/io/line_citations.m` and its twins, carrying the bibliography key as it appears in `doc/latex/biblio.bib`, a short reference, and a one-line statement of which part of the solution process the paper accounts for:

```
add('<method>', '<BibKey>', '<author, title, venue, year>', '<what it covers>');
add('mva.<method>', '<BibKey>', '...', '<what it covers>');
```

Attribution in LINE is pull-based: nothing is printed during a solve, and the user retrieves the references with `solver.citations()`, so a method that reaches a user without an entry is a silent loss of attribution. The toy method has no paper behind it and therefore takes no entry; an algorithm with no literature is the only admissible omission. When the paper is not yet cited by the manuals, add the BibTeX entry to `doc/latex/biblio.bib` and to `doc/latex-internals/biblio.bib`.

Step 8: decide whether default should ever select it. Adding a method does not change the default dispatch, and it should not unless there is evidence that the new method dominates the incumbent on an identifiable class of models. The toy method plainly does not qualify. The default branch of the analyzer is a policy, not a fallback; every change to it re-baselines the regression goldens of every model that reaches that branch.

17.4 Walkthrough: adding a new solver

A new solver is warranted when the algorithm does not fit the dispatch of any existing one. To keep the comparison with Section 17.3 exact, we now register the same toy algorithm as a solver of its own, named `SolverCustom` and addressed by the short name `CUSTOM`, hosting a single method `toy`. In a real extension the paradigm would be one that no existing solver hosts; the mechanics below are unchanged. The steps are additional to the eight above, which still apply to each method the new solver hosts.

Step 1: the solver type. Add the symbol to the solver-type enumeration, which the options object keys on: `jar/src/main/java/jline/lang/constant/SolverType.java`, `python/line_solver/constants.py` (class `SolverType`), and, in MATLAB, the corresponding method name accepted by `SolverOptions`.

Step 2: the solver class and its options. Follow the per-codebase layout of Table 17.2. The constructor calls the base constructor with the solver name and installs `Solver.parseOptions(varargin, SolverCustom.defaultOptions)`: Beware that `parseOptions` *replaces* the defaults with what it is given, so fields must be merged, never assumed to survive.

Step 3: the feature envelope. Write `getFeatureSet` by starting from the empty set and adding only what an analyzer demonstrably handles. For `SolverCustom` that is a short list, since Equation (17.1) reads demands and populations and nothing else: `Queue`, `Delay`, `Source`, `Sink`, `ClosedClass`, `Exp`, and the scheduling strategies whose demands are meaningful under it. Feature names come from a canonical registry shared by the codebases: `SolverFeatureSet.fields` in MATLAB, `FeatureSet.java` in the JAR, and `SolverFeatureSet.FIELDS` in `python/line_solver/solvers/base.py`. A name outside the registry is invisible to the support test, so a model using it would pass the gate as if the feature were absent; if the new solver introduces a genuinely new model feature, the name must be added to every registry first.

Step 4: analyzer and result. Implement `runAnalyzer`: reject unsupported models with the standard error function, seed the random generator when the method is randomized, dispatch to the analyzer, and store Q, U, R, T together with the runtime and the actual method used. Reuse `NetworkAvgTable` and the existing result types unless the solver returns a genuinely new object. `SolverCustom` needs no new result type: the toy handler of step 1 of Section 17.3 is carried over unchanged, moved from the MVA handler directory to its own.

Step 5: the short alias. Users address solvers by short name. Add `CUSTOM` as an alias of `SolverCustom`: a one-line subclass in `matlab/src/solvers/CUSTOM.m`, a thin class in the JAR package, and an entry in the alias block and in `__all__` of `python/line_solver/solvers/__init__.py`.

Step 6: automatic selection. `SolverAUTO` keeps an ordered list of candidate solvers and a feature-driven pre-filter built from the candidates' feature sets. A new solver that is not registered there is never chosen automatically, which is the right default, and the right one for `SolverCustom`: register it only once its accuracy envelope is characterized, and place it in the order according to cost, analytical solvers preceding simulators.

Step 7: the external interfaces. The command-line interface maps a solver name to a constructor (`jar/src/main/java/jline/cli/LineCLI.java` and `python/line_solver/cli.py`); a solver absent from that mapping is unreachable from `line-cli`, from the REST and WebSocket servers, and from the MCP server.

Table 17.2: Where the artefacts of a solver named `CUSTOM` live, by codebase.

Artefact	MATLAB	JAR	Python
solver class	`src/solvers/CUSTOM/@SolverCustom/SolverCustom.m`	`jline/solvers/custom/SolverCustom.java`	`solvers/solver_custom/solver_custom.py`

Artefact	MATLAB	JAR	Python
short alias	src/solvers/CUSTOM.m	jline/solvers/custom/CUSTOM.java	alias block of solvers/__init__.py
options	SolverOptions('CUSTOM')	CustomOptions.java	SolverOptions(SolverType.CUSTOM)
analyzer	solver_custom_analyzer.m	custom/analyzers/Solver_custom_analyzer.java	solver_custom_analyzer.py
handlers	solver_custom_toy.m	custom/handlers/Solver_custom_toy.java	branches in solver_custom.py
solver type	method name in SolverOptions.m	lang/constant/SolverType.java	constants.py
feature names	SolverFeatureSet.m	lang/FeatureSet.java	solvers/base.py
method type	io/line_method_type.m	solvers/MethodType.java	solvers/base.py
citations	io/line_citations.m	io/LineCitations.java	api/io/citations.py
option whitelist	Solver.listValidOptions	Solver.listValidOptions	option parser in solvers/_solver_base.py
auto selection	AUTO/@SolverAUTO/SolverAUTO.m	solvers/auto/SolverAUTO.java	solvers/solver_auto/
CLI mapping	line-cli wrapper	cli/LineCLI.java	cli.py

MATLAB requires no path registration: `lineStart` adds the whole source tree with `genpath`. In the JAR, the package is picked up by the Maven build with no descriptor entry. In Python, the class must be imported and exported explicitly in `solvers/__init__.py`.

17.5 Verification obligations

A method or solver is not finished when it runs. The following are the acceptance conditions used in the project.

1. *Agreement with an exact reference.* On models small enough for `SolverCTMC`, or product-form enough for exact MVA, the new method must agree with the exact answer to a stated tolerance, or the deviation must be explained by the approximation it makes. Simulators are compared against `SolverJMT` by a two-sample t-test rather than by point equality.

2. *Cross-codebase parity.* The implementations must return the same numbers on the shared example suite. The parity harness lives in the companion test repository and compares MATLAB, native Python, and the MATLAB `lang='java'` dispatch row by row; the C++ port asserts against the shared golden baselines in `goldens/`, as the in-repo Python and JAR suites also do.

3. *Envelope honesty.* For every feature declared in `getFeatureSet`, a test must exercise it; for every feature not declared, the solver must reject rather than approximate. The corresponding negative tests are as important as the positive ones.

4. *Method-list honesty.* A test that iterates `listValidMethods` on a representative model and runs each name to completion catches the most common regression, namely a method advertised after its handler stopped accepting the model.

5. *No expectation edits.* When a new method disagrees with an existing golden value, the implementation is wrong until proven otherwise. Test assertions and tolerances are not adjusted to accommodate new code.

17.6 Documentation obligations

The artefacts below are part of the definition of done, in the same change that adds the code:

- this manual: a chapter for a new solver in the part its paradigm belongs to, or a row in the method tables of the existing solver's chapter, including its feature envelope table and its pseudocode entry;
- the user manuals (`doc/latex/manual.tex`): the solver list, the method summary table, the per-solver method description, and the supported feature tables;
- the knowledge base (`_kb/`): the solver catalogue page and, when dispatch or a structure field changes, the corresponding reference pages;
- the bibliography, when the attribution added in step 7 cites a paper not already referenced by the manuals.

17.7 Pitfalls

The following failure modes have all occurred in practice and are silent, in the sense that they produce a number rather than an error.

- *Declaring a feature the analyzer ignores.* The model is admitted and solved as if the feature were absent. This is the single most damaging mistake available when extending a solver.
- *Listing a method that raises.* Automatic selection and the method-sweep tests will both call it.
- *Folding a new method into an existing case.* Spelling out a fall-through must preserve the path taken; joining a neighbouring case changes the answer for the models that already used it.
- *Assuming the defaults survive option parsing.* The option parser replaces the default structure with the caller's, so a field the new method reads may be absent unless it was merged.
- *Defensive guards instead of a fix.* Bounds checks, null checks, and try/catch blocks that mask an underlying indexing or structural error convert a crash into a wrong number, which is strictly worse.

- *Comparing enumeration values numerically across codebases.* The process-type enumerations start at different offsets in MATLAB and Python; compare names.
- *Forgetting one of the four method-type twins.* The banner then reports the method as unclassified in one codebase only, which surfaces as a spurious parity difference.

Part VI

Reference

Appendix A

The NetworkStruct reference

This appendix documents field by field the `sn` structure produced by `Network.getStruct()`, the flat model representation on which every analyzer of this manual operates (Section 1.3). The tables are given once per codebase, since the three implementations agree on the field names and semantics but differ in the container types they use: MATLAB stores cell arrays and matrices, the JAR stores `Matrix` objects and typed `Map`s keyed by model elements, and native Python stores NumPy arrays, lists and dictionaries. Fields absent from a table are absent from that codebase; the cross-language caveats are collected in the user manuals and in the repository knowledge base.

Two groups of fields are distinguished. *Static* properties carry the parameterization supplied at model definition time. *Computed* properties are derived from the static ones by `refreshStruct` and its component refresh routines (visits, routing tables, chains, capacities), and are the ones a solver may legitimately recompute after a parameter change.

A.1 Nodes, stations, and stateful nodes

Internally to LINE there is an explicit differentiation between properties of nodes, stations, and stateful nodes. This distinction has impact in particular over routing and class-switching mechanisms, and also allows solvers to better differentiate between different kinds of nodes.

The three categories form a nested hierarchy. Every element of a `Network` is a *node*, the most general category, uniquely identified by a node index in $1, \ldots,$ `nnodes`. A *stateful node* is a node that carries local state variables needed to determine its behaviour; stateful nodes are separately indexed in $1, \ldots,$ `nstateful` and flagged by the `isstateful` field. A *station* is a stateful node in which jobs may additionally spend time (sojourn), so that it contributes to queueing and response-time metrics; stations are separately indexed in $1, \ldots,$ `nstations` and flagged by the `isstation` field. Hence every station is a stateful node, and every stateful node is a node, but not conversely. For example, a `Sink` is a plain stateless node, since jobs cannot sojourn in it and no state variable is required; a `Cache` is a stateful node but not a station, since it must track the cached items yet passage through it is instantaneous; and a `Queue` is a station, since jobs sojourn in its buffer and servers. A `Source` is likewise treated as a station, because the inter-arrival time of jobs from the external world is modelled as time spent in the source, although this time is not counted as part of the system response time. Table A.1 classifies the nodes available in `Network` models according to these three

categories.

Table A.1: Classification of `Network` nodes as stateful nodes and stations

Node	Stateful	Station	Category
`Source`	yes	yes	station
`Queue`	yes	yes	station
`Delay`	yes	yes	station
`Join`	yes	yes	station
`Place`	yes	yes	station
`Router`	yes	no	stateful node
`Cache`	yes	no	stateful node
`Transition`	yes	no	stateful node
`ClassSwitch`	no	no	stateless node
`Fork`	no	no	stateless node
`Logger`	no	no	stateless node
`Sink`	no	no	stateless node

In some cases, one may want to access some properties of nodes that are contained in `NetworkStruct` fields that are however referenced by station or stateful node index. To help this and similar situations, the `NetworkStruct` class also provides static methods to quickly convert the indexing of nodes, stations, and stateful nodes, which is used in referencing its data structures:

- `nodeToStateful` (`node_to_stateful` in Python)
- `nodeToStation` (`node_to_station` in Python)
- `stationToNode` (`station_to_node` in Python)
- `stationToStateful` (`station_to_stateful` in Python)
- `statefulToNode` (`stateful_to_node` in Python)

As an example, we can determine the portion of the `nodevisits` field that refers to stateful nodes in chain $c = 1$ as follows

A.2 Static properties

A.3 Computed properties

A.4 Node parameters

For advanced nodes, such as `Cache` and `Transition`, additional parameters are carried by the `nodeparam` entry of the corresponding node. The tables below give these parameters.

Appendix B

The LayeredNetworkStruct reference

B.1 Representation of the model structure

It is possible to access the internal representation of a `LayeredNetwork` model in a similar way as for MATLAB `Network` objects, i.e.: The return `lqn` structure, of class `LayeredNetworkStruct`, contains all the information about the specified model. It relies on relative and absolute indexing for the elements of the `LayeredNetwork`.

- A *relative* index is a number between 1 and the number of similar elements in the model, e.g., for a model with 3 tasks, the relative index t of a task would be a number in $[1, 3]$.
- An *absolute* index is a number between 1 and the total number of elements (of any kind, except calls) in the model, e.g., for a model with 2 hosts, 3 tasks, 5 entries, and 8 activities, the total number of elements is `nidx`$= 18$ and last activity a may have an absolute index `aidx`$= 18$ and a relative index `a`$= 8$.
- The difference between the relative and the absolute index of an element is referred to as *shift*, e.g., in the previous example `ashift`$= 18 - 8 = 10$.
- Absolute and relative indexing for calls and hosts are identical, call index `cidx` ranges in $[1, \texttt{ncalls}]$ and host index `hidx` ranges in $[1, \texttt{nhosts}]$.

Using the above convention, the internal representation of the model is described in the table below. As in the examples above, relative and absolute indexes are differentiated by using the suffix `idx` in the latter (e.g., `a` vs. `aidx`). This indexing style is used throughout the codebase as well.

B.2 Decomposition into layers

Layers are a form of decomposition where we model the performance of one or more servers. The activity of clients not detailed in that layer is taken into account through an artificial delay station, placed in a closed loop to the servers [153]. This artificial delay is used to model the inter-arrival time between calls issued by that client.

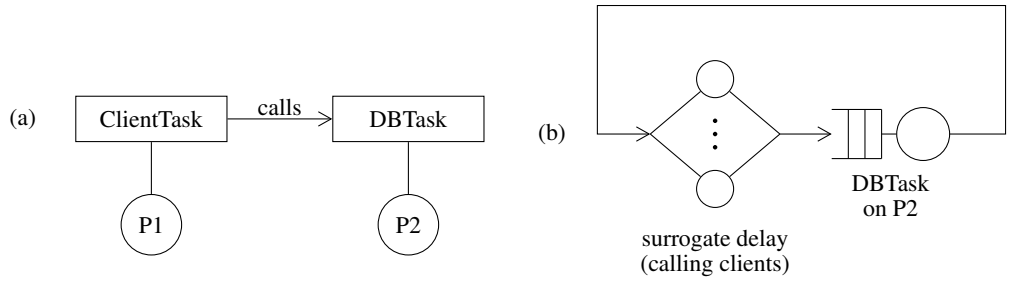


Figure B.1: SRVN-type layering. (a) A layered model in which the client task calls the database task. (b) The layer of `DBTask`: the clients not detailed in this layer are replaced by a surrogate delay placed in a closed loop with the server, its delay being the inter-call time.

Figure B.1 illustrates the construction.

The current version of LINE adopts SRVN-type layering [68], whereby a layer corresponds to one and only one resource, either a processor or a task. The `getLayers` method returns a cell array consisting of the MATLAB `Network` objects corresponding to each layer The decomposition is performed through the `LN` solver described later.

Within each layer, classes are used to model the time a job spends in a given activity or call, with synchronous calls being modeled by classed with label including an arrow, e.g., `'AS1=>E3'` is a closed class used represent synchronous calls from activity `AS1` to entry `E3`, whereas `'AS1->E3'` denotes an asynchronous call. Artificial delays and reference nodes are modelled as a delay station named `'Clients'`, whereas the task or processor assigned to the layer is modelled as the other node in the layer.

Appendix C

API Function Reference

This appendix catalogs the API functions of the `jline.api` package that are called from outside their own package: the algorithm entry points a solver, an analyzer or a user reaches for. The table lists each function name, its organizational package, and a brief description of its purpose. Result and option carrier classes, and helpers private to one algorithm, are documented in the source rather than here.

Table C.1: Complete API Function Reference

Function Name	Package	Description
`amap2_fit_gamma`	mam	Fits AMAP(2) distributions to match moments and correlation characteristics
`amap2_fit_gamma_map`	mam	Fits AMAP(2) by approximating arbitrary-order MAP with preserved correlation structure
`amap2_fit_gamma_trace`	mam	Fits AMAP(2) from empirical traces while preserving autocorrelation characteristics
`aph2_adjust`	mam	Adjusts moments to ensure feasibility bounds for APH(2) fitting procedures
`aph2_assemble`	mam	Constructs APH(2) transition matrices from specified rates and transition probabilities
`aph2_fit`	mam	Fits APH(2) distributions to match given moments with automatic feasibility adjustment
`aph2_fit_map`	mam	Fits APH(2) distributions by approximating arbitrary-order MAP processes
`aph2_fit_trace`	mam	Fits APH(2) distributions from empirical inter-arrival time traces
`aph2_fitall`	mam	Fits multiple APH(2) distributions to match given moments with exhaustive parameter search
`aph_bernstein`	mam	Constructs APH distributions using Bernstein exponential approximation methods
`aph_convseq`	mam	Convolves a sequence of APH distributions by repeated sequential simplification
`aph_fit`	mam	Fits APH distributions to specified moments using optimization and approximation techniques
`aph_rand`	mam	Generates a random Acyclic Phase-type (APH) distribution with the specified number of phases
`aph_simplify`	mam	Simplifies and combines APH distributions using structural pattern operations
`cache_erec`	cache	Implements exact recursive (EREC) algorithms for cache system analysis
`cache_gamma`	cache	Computes cache access factors from request arrival rates and routing matrices

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
cache_gamma_lp	cache	Computes cache access factors using linear programming optimization methods
cache_is	cache	Estimates the cache normalizing constant using Monte Carlo importance sampling
cache_miss	cache	Provides general-purpose algorithms for computing cache miss rates across
cache_miss_asy	cache	Provides asymptotic approximation methods for cache miss rate analysis
cache_miss_fpi	cache	Computes cache miss probabilities using fixed-point iteration methods
cache_miss_is	cache	Computes global, per-user, and per-item cache miss rates using importance sampling
cache_miss_rayint	cache	Estimates cache miss rates using ray method for partial differential equations
cache_miss_spm	cache	Estimates cache miss rates and per-user/per-item miss metrics using the SPM PDE method
cache_mva	cache	Implements Mean Value Analysis algorithms for cache system performance
cache_mva_miss	cache	Implements Mean Value Analysis (MVA) algorithms for computing cache miss
cache_prob_erec	cache	Computes exact cache state probabilities using recursive methods based on
cache_prob_fpi	cache	Computes cache state probabilities using fixed-point iteration algorithms
cache_prob_is	cache	Computes per-item cache hit and miss probabilities using Monte Carlo importance sampling
cache_prob_rayint	cache	Computes cache state probabilities using ray integration methods for
cache_prob_spm	cache	Computes cache state probabilities using the SPM ray-method approximation
cache_rayint	cache	Implements ray integration techniques for cache system analysis using
cache_rrm_meanfield_ode	cache	Implements the mean field ordinary differential equation system for Random
cache_spm	cache	Approximates the cache normalizing constant using the SPM (saddle-point) method
cache_t_hlru	cache	Computes response time metrics for Hierarchical Least Recently Used (H-LRU)
cache_t_lrum	cache	Computes response time metrics for Least Recently Used with Multiple servers
cache_t_lrum_map	cache	Analyzes response times for LRUM cache systems with Markovian Arrival
cache_ttl_hlru	cache	Implements TTL approximation for Hierarchical LRU (H-LRU) cache systems
cache_ttl_lrua	cache	Implementation of Time-To-Live (TTL) approximation for LRU(A) cache systems
cache_ttl_lrum	cache	Implementation of Time-To-Live approximation for Least Recently Used with
cache_ttl_lrum_map	cache	Combines TTL approximation with LRUM cache policies and Markovian Arrival
cache_ttl_tree	cache	Tree-based TTL cache analysis implementation for the LINE solver framework
cache_xi_bvh	cache	Computes cache xi terms using the iterative method from Gast-van Houdt
cache_xi_fp	cache	Estimates cache xi terms using fixed-point algorithms. Xi terms represent
cache_xi_iter	cache	Iteratively computes cache xi terms assuming monotone access factors with list index
ctmc_bicgstab	mc	Solves a sparse nonsymmetric system by the stabilized biconjugate gradient method, with row equilibration and reverse Cuthill-McKee reordering
ctmc_bicgstab_multi	mc	Solves a whole block of right-hand sides by stabilized biconjugate gradients, reusing one factorization across the columns
ctmc_courtois	mc	Courtois decomposition for nearly completely decomposable CTMCs
ctmc_fau	mc	Transient distribution of a CTMC by fast adaptive uniformization
ctmc_hitting_time	mc	Mean time to reach a target state set from each state of a CTMC

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ctmc_kms	mc	Koury-McAllister-Stewart aggregation-disaggregation method for CTMCs
ctmc_makeinfgen	mc	Constructs and validates infinitesimal generator matrices for continuous-time
ctmc_multi	mc	Multi-level aggregation method for CTMCs
ctmc_passage_lst	mc	Laplace-Stieltjes transform of the first passage time of a CTMC into a target state set
ctmc_passage_moments	mc	Moments of the first passage time of a CTMC into a target state set
ctmc_passage_ph	mc	Phase-type representation of the first passage time of a CTMC into a target state set
ctmc_passage_time	mc	Distribution and density of the first passage time of a CTMC into a target state set
ctmc_pseudostochcomp	mc	Computes a pseudo-stochastic complement that approximates a CTMC by reweighting eliminated states by stationary probability
ctmc_rand	mc	Generates random infinitesimal generator matrices for continuous-time Markov chains
ctmc_saddlepoint	mc	Saddlepoint approximation of $\Pr\{N(t) = k\}$ for a MAP counting process, from the Perron root of $D_0 + e^{\theta} D_1$; cost independent of t and k, log-probability returned
dqsys_bernoulli1	dqsys	Exact analysis of the discrete-time Bernoulli/Bernoulli/1 queue on the slot lattice
dqsys_geogeo1	dqsys	Exact analysis of the discrete-time Geo/Geo/1 queue, the analytical reference of the slotted LDES mode
dqsys_geoxgeo1	dqsys	Exact analysis of the discrete-time Geo^X/Geo/1 queue with geometric batch arrivals
ctmc_randomization	mc	Converts a continuous-time Markov chain into an equivalent discrete-time chain
ctmc_relsolve	mc	Equilibrium distribution of a continuous-time Markov chain re-normalized with respect to
ctmc_sens	mc	Sensitivity of the stationary distribution to a scalar parameter, from the differentiated balance equations $\boldsymbol{\pi}'\boldsymbol{Q} = -\boldsymbol{\pi}\,\mathrm{d}\boldsymbol{Q}/\mathrm{d}\theta$, which share the coefficient matrix of the stationary solve
ctmc_simulate	mc	Generates sample paths for CTMCs using the standard simulation algorithm with
ctmc_solve	mc	Computes the steady-state probability distribution for CTMCs by solving the linear
ctmc_solve_reducible	mc	Solve reducible CTMCs by converting to DTMC via randomization
ctmc_solve_reducible_blkdecomp	mc	Solves reducible CTMCs via SCC-based block decomposition of the generator matrix
ctmc_ssg	mc	Generates the CTMC state space and aggregated representation from a NetworkStruct using the configured cutoffs
ctmc_ssg_reachability	mc	CTMC State Space Generator for Reachability Analysis
ctmc_stmonotone	mc	Computes the stochastically monotone upper bound for a CTMC
ctmc_stochcomp	mc	Implements stochastic complementarity analysis for CTMCs to identify strongly
ctmc_takahashi	mc	Takahashi's aggregation-disaggregation method for CTMCs
ctmc_testpf_kolmogorov	mc	Test if a CTMC has product form using Kolmogorov's criteria
ctmc_timereverse	mc	Computes the infinitesimal generator of the time-reversed continuous-time
ctmc_transient	mc	Computes transient probabilities for CTMCs using numerical integration of the
ctmc_uniformization	mc	CTMC Transient Analysis via Uniformization
da_cacheqn_itemprob	da	Per-item occupancy of a cache from the converged access probabilities of a decomposition-aggregation solve
dmap_dist	mam	Computes squared L2 distances between joint PMFs and ACFs of two discrete-time MAPs
dmap_moment	mam	Computes raw moments of inter-arrival times of a discrete-time MAP
dmap_pie	mam	Computes the stationary vector at arrival epochs of a discrete-time MAP

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
dmap_sample	mam	Generates inter-arrival time samples from a discrete-time MAP
dtmc_isfeasible	mc	Check if a matrix represents a feasible DTMC transition matrix
dtmc_makestochastic	mc	Converts non-negative matrices into valid discrete-time Markov chain transition
dtmc_rand	mc	Generates random stochastic transition matrices for discrete-time Markov chains
dtmc_simulate	mc	Generates sample trajectories for DTMCs by sampling from the transition probability
dtmc_solve	mc	Computes the steady-state probability distribution for DTMCs by converting the
dtmc_solve_reducible	mc	Estimates the limiting distribution of a possibly reducible DTMC using strongly connected component decomposition
dtmc_stochcomp	mc	Returns the stochastic complement of a DTMC
dtmc_timereverse	mc	Compute the infinitesimal generator of the time-reversed DTMC
dtmc_uniformization	mc	Computes transient DTMC probabilities by converting to a CTMC generator and applying CTMC uniformization
euler_get_alpha	lti	Nodes of the Euler (Abate-Whitt) quadrature used by the Laplace inversion
euler_get_eta	lti	Euler (Abate-Whitt) weights eta of the Laplace inversion quadrature
euler_get_omega	lti	Weights of the Euler (Abate-Whitt) quadrature used by the Laplace inversion
fes_compute_metrics	fes	Per-station metrics of an isolated subnetwork at every population state, for flow-equivalent server aggregation
fj_amva	fj	Mean value analysis of a closed network of fork-join subnetworks
fj_char_max_blom	fj	Blom-corrected plotting position for the characteristic maximum
fj_char_max_discrete	fj	Characteristic maximum of a discrete random variable
fj_cox_fit	fj	Two-stage Coxian fit of a mean and a squared coefficient of variation
fj_dag_makespan	fj	Makespan of a task system with precedence constraints
fj_delay_opt	fj	Deterministic subtask delays that minimise mean dispersion
fj_dispersion	fj	Mean subtask dispersion of a split-merge system with Erlang branches
fj_ism_green	fj	Green's independent server model of simultaneous server requests
fj_lst_max_het	fj	Laplace-Stieltjes transform of the maximum of heterogeneous exponentials
fj_ordstat_exp	fj	Mean of the k-th smallest of n independent exponential branch completion times, as charged at a quorum join
fj_qgb	fj	Geometric bound on the queue length of a fork-join subnetwork
fj_respt_bulk	fj	Centralized splitting analysed as an M[K]/M/c bulk arrival system
fj_respt_closed	fj	Varki bound on the residence time of a closed fork-join subnetwork
fj_respt_nosplit	fj	Mean response time of the distributed no-splitting parallel system
fj_serialization	fj	Blocking probability and pseudoserver delay of serialization phases
fj_simplex	fj	Two-phase dense simplex for small equality-form linear programs
fj_tail_ordstat	fj	Tail latency of a k-of-n (quorum) fork-join request
fj_tsm_capacity	fj	Saturation throughput of the team service model
fj_xmax_coxian	fj	Expected maximum of K i.i.d. two-stage Coxian variables
fj_xmax_het	fj	Exact moments of the maximum of heterogeneous exponential branch times
fj_xmax_hz	fj	Harrison-Zertal approximation of the maximum of i.i.d. variables
fj_xmax_hz_het	fj	Harrison-Zertal approximation of the maximum of general variables
fj_xmax_moments_het	fj	Moments of the maximum of heterogeneous exponentials by recurrence
gaver_stehfest_get_alpha	lti	Nodes of the Gaver-Stehfest Laplace inversion, on an even number of terms
gaver_stehfest_get_omega	lti	Gaver-Stehfest weights. N is rounded DOWN to even. Twin of the native Python
ge_fit	fj	Two-moment fit of a generalized exponential law
infer_variational	infer	Variational inference of service rates from noisy queue-length readings, returning a Gamma posterior per rate
laplace_invert	lti	Invert a Laplace transform at a single time point T. METHOD is one of

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
laplace_invert_cdf	lti	Inverts the transform of a density into a distribution function on a time grid
laplace_invert_cme	lti	Invert a Laplace transform at T by the Concentrated Matrix Exponential
laplace_invert_euler	lti	Invert a Laplace transform at T by the Euler (Abate-Whitt) method. F is a
laplace_invert_gaver_stehfest	lti	Invert a Laplace transform at T by Gaver-Stehfest. This method samples the
laplace_invert_pdf	lti	Inverts the transform of a density on a time grid
laplace_invert_talbot	lti	Invert a Laplace transform at T by Talbot's deformed contour. F is a handle
laplace_invert_weeks	lti	Invert a Laplace transform by the Laguerre (Weeks) series
laplace_weeks_coeffs	lti	Laguerre coefficients of the damped and scaled function inverted by the Weeks method
laplace_weeks_scaling	lti	Automatic selection of the damping and scaling parameters of the Weeks inversion
ldqbd	mam	Computes rate matrices for level-dependent QBD processes via matrix continued fractions
ldqbd_mphc	mam	Block-tridiagonal LDQBD generator of the c-server FCFS queue with PH service, exact on the multiset-of-phases coordinate, with optional per-level rate scaling
ljd_delinearize	pfqn	Recovers a per-class population vector from its linearized index, the inverse of ljd_linearize
lossn_asy	lossn	Large-capacity (Kelly) limit of a loss network: link blocking from the convex dual of the fluid optimization
lossn_erlangfp	lossn	Implements fixed-point algorithms for analyzing loss networks using Erlang
lossn_kr	lossn	Kaufman-Roberts recursion: occupancy distribution and per-class blocking of one multirate link
lossn_kr_asy	lossn	Critical-loading normal approximation of the blocking of one multirate link
lossn_kr_dim	lossn	Smallest link capacity meeting per-class blocking targets
lossn_links	lossn	Per-link blocking and mean occupancy of a loss network, by any loss-network method
lossn_revsens	lossn	Revenue sensitivities and route and link implied costs of a loss network
lqn_boxbounds	lqn	Computes Majumdar-Woodside robust box bounds on per-task throughput for a layered queueing network (processor-contention model)
lqn_entry_workflow	lqn	Activity graph of a layered network entry, expressed as a Workflow
lqn_ph_moments	lqn	First two moments of a phase-type law without building its representation
lqn_ph_serial_law	lqn	Composed law of a workflow whose AND-fork branches are executed serially
lqn_ref_thinktime	lqn	Declared think time of a task as it enters the thread cycle
lqn_setup_charge	lqn	Mean cold start charged to one request of a task with a setup time
lsn_max_multiplicity	lsn	Computes maximum multiplicity constraints for load sharing network (LSN)
m3pp22_fitc_approx_cov	mam.m3pp	Implements parameter fitting for second-order Marked Markov Modulated Poisson Process
m3pp22_fitc_approx_cov_multiclass	mam.m3pp	Implements constrained optimization for fitting M3PP(2,2) parameters given an underlying
m3pp22_interleave_fitc	mam.m3pp	Implements lumped superposition of multiple M3PP(2,2) processes using interleaved
m3pp2m_fitc	mam.m3pp	Implements exact fitting of second-order Marked Markov Modulated Poisson Process
m3pp2m_fitc_approx	mam.m3pp	Implements approximation-based fitting for M3PP(2,m) using optimization methods
m3pp2m_fitc_approx_ag	mam.m3pp	Implements auto-gamma approximation method for M3PP(2,m) parameter fitting
m3pp2m_fitc_approx_ag_multiclass	mam.m3pp	Implements multiclass auto-gamma fitting for M3PP(2,m) with variance and covariance
m3pp2m_fitc_theoretical	mam.m3pp	Fits the theoretical characteristics of an MMAP(n,m) with an M3PP(2,m) under several methods

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
m3pp2m_fitc_trace	mam.m3pp	Fits an M3PP(2,m) from trace data using counting-process characteristics
m3pp2m_interleave	mam.m3pp	Implements interleaved superposition of multiple M3PP(2,m) processes to construct
m3pp_interleave_fitc	mam.m3pp	Implements fitting and interleaving of k second-order M3PP processes with varying
m3pp_interleave_fitc_theoretical	mam.m3pp	Implements theoretical MMAP fitting through M3PP interleaving using analytical
m3pp_interleave_fitc_trace	mam.m3pp	Implements M3PP interleaving and fitting directly from empirical trace data
m3pp_rand	mam.m3pp	Implements random generation of Markovian Multi-class Point Processes (M3PP)
m3pp_superpos_fitc	mam.m3pp	Implements superposition-based fitting of k second-order M3PP processes into
m3pp_superpos_fitc_theoretical	mam.m3pp	Implements superposition fitting of k second-order M3PP processes using theoretical
m3pp_superpos_fitc_trace	mam.m3pp	Superposes k M3PP processes to fit a multi-class trace with m classes
mamap22_fit_gamma_fs_trace	mam	Fits MAMAP(2,2) from trace data using gamma autocorrelation and forward-sigma characteristics
mamap22_fit_multiclass	mam	Fits MAMAP(2,2) processes for two-class systems with forward moments and sigma characteristics
mamap2m_coefficients	mam	Computes coefficients for MAMAP(2,m) fitting formulas in canonical forms
mamap2m_fit	mam	Fits MAMAP(2,m) processes matching moments, autocorrelation, and class characteristics
mamap2m_fit_fb_multiclass	mam	Fits MAMAP using combined forward and backward moment characteristics for multiclass systems
mamap2m_fit_gamma_fb_mmap	mam	Fits MAMAP with autocorrelation control using forward-backward moments from MMAP input
mamap2m_fit_gamma_fb_trace	mam	Fits a second-order acyclic MMAP from a marked trace matching class probabilities and forward-backward moments
mamap2m_fit_mmap	mam	Fits MAPH/MAMAP(2,m) by approximating characteristics of input MMAP processes
mamap2m_fit_trace	mam	Fits MAMAP(2,m) processes from empirical trace data with inter-arrival times and class labels
map2_fit	mam	Fits MAP(2) processes to match specified moments and autocorrelation decay rates
map2mmpp	mam	Converts MAP representations to MMPP format for compatibility with MMPP-specific algorithms
map2ph	mam	Converts a MAP into a phase-type distribution and the associated PH-renewal process
map_acf	mam	Computes autocorrelation function (ACF) values for MAP inter-arrival times at specified lags
map_acfc	mam	Computes ACFC values for MAP counting processes over time intervals, measuring temporal correlation
map_anfit	mam	Fits a MAP using the Andersen-Nielsen IPP-superposition method calibrated to the Hurst parameter
map_bernstein	mam	Constructs a MAP from a continuous distribution PDF via Bernstein polynomial approximation
map_block	mam	Constructs MAP(2) representations from moment and autocorrelation parameters using fallback
map_ccdf_derivative	mam	Computes derivatives of MAP complementary cumulative distribution functions at zero
map_cdf	mam	Computes CDF values for MAP inter-arrival times using CTMC uniformization techniques
map_checkfeasible	mam	Comprehensive validation of MAP matrices including stochastic properties, numerical stability,
map_count_mean	mam	Computes mean number of arrivals in MAP counting processes over specified time intervals

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
map_count_moment	mam	Computes power moments of MAP counting processes using moment generating functions and
map_count_var	mam	Computes variance of MAP counting processes over specified time intervals using matrix
map_dist	mam	Computes the squared L2 distance between lag-L joint densities of two MAPs
map_dist_acf	mam	Computes the squared L2 distance between autocorrelation functions of two MAPs
map_embedded	mam	Computes embedded DTMC matrices from MAP representations by extracting transition
map_erlang	mam	Constructs MAP representations of Erlang-k processes with specified means and phases
map_exp_mul_int	mam	Computes the inner product of lag-L joint densities of two MAPs via recursive Sylvester equations
map_exponential	mam	Creates MAP representations of exponential inter-arrival time distributions with specified
map_feasblock	mam	Constructs feasible MAP representations when exact moment matching fails by adjusting
map_feastol	mam	Provides standard tolerance values for numerical feasibility checks in MAP algorithms
map_gamma	mam	Computes gamma parameter measuring autocorrelation decay rates in MAP processes
map_gamma2	mam	Computes largest non-unit eigenvalue of embedded DTMC for MAP correlation characterization
map_hyperexp	mam	Constructs MAP representations of two-phase hyperexponential renewal processes
map_idc	mam	Computes asymptotic index of dispersion for MAP counting processes, measuring long-term
map_infgen	mam	Computes infinitesimal generator matrix of underlying CTMC by combining MAP transition
map_isfeasible	mam	Provides convenient interface for MAP feasibility validation with configurable tolerance
map_joint	mam	Computes joint moments of MAP inter-arrival times for advanced statistical characterization
map_jointpdf_derivative	mam	Computes partial derivatives of MAP joint probability density functions at origin
map_kpc	mam	Computes Kronecker product composition of multiple MAPs for building complex arrival processes
map_kurt	mam	Computes kurtosis of MAP inter-arrival times measuring tail heaviness and distribution shape
map_lambda	mam	Computes the long-run arrival rate of a MAP from its hidden and visible transition matrices
map_largemap	mam	Provides size thresholds for determining when MAP algorithms should switch to
map_mark	mam	Creates Marked MAP (MMAP) representations by adding class labels to MAP arrivals
map_max	mam	Computes MAP representation of maximum inter-arrival times from independent MAP processes
map_mean	mam	Computes the mean inter-arrival time of a MAP as the reciprocal of its arrival rate
map_mixture	mam	Creates probabilistic mixtures of MAP processes with specified mixture probabilities
map_mmpp2	mam	Fits a 2-state MMPP as a MAP from mean, SCV, skewness, and lag-1 autocorrelation
map_moment	mam	Computes raw moments of MAP inter-arrival times using matrix inversion techniques
map_normalize	mam	Sanitizes MAP matrices by ensuring non-negativity constraints and proper diagonal adjustments

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
map_pdf	mam	Computes PDF values for MAP inter-arrival times using matrix exponential techniques
map_pie	mam	Computes steady-state probability vector of embedded discrete-time Markov chain
map_piq	mam	Computes steady-state probability vector of underlying continuous-time Markov chain
map_pntiter	mam	Computes exact arrival probabilities using iterative numerical methods based on Neuts and Li
map_pntquad	mam	Computes MAP point process probabilities using ODE quadrature methods with Runge-Kutta integration
map_prob	mam	Computes equilibrium probability distribution of underlying CTMC for MAP analysis
map_rand	mam	Generates random MAP representations for testing, simulation, and statistical analysis
map_randn	mam	Generates random MAP samples with added numerical noise for robustness testing
map_renewal	mam	Creates renewal MAP by removing correlations to obtain memoryless arrival processes
map_sample	mam	Generates random samples from MAP distributions for simulation and empirical analysis
map_scale	mam	Rescales MAP inter-arrival time distributions to achieve specified mean values
map_scv	mam	Computes SCV of MAP inter-arrival times as normalized dispersion measure
map_skew	mam	Computes skewness of MAP inter-arrival times measuring asymmetry in distributions
map_stochcomp	mam	Performs state elimination through stochastic complementation while preserving MAP properties
map_sum	mam	Computes MAP representations of sums of identical MAP processes for load scaling
map_sumind	mam	Computes MAP representations of sums of independent MAP processes for modeling
map_super	mam	Creates superposition of MAP processes using Kronecker product techniques
map_timereverse	mam	Computes time-reversed MAP by adjusting transition rates based on stationary distributions
map_var	mam	Computes the variance of MAP inter-arrival times from the second moment and squared mean
map_varcount	mam	Computes variance of event counts in MAP processes over specified time intervals
maph2m_fit	mam	Fits MAPH(2,m) processes to match ordinary moments, class probabilities, and backward moments
maph2m_fit_mmap	mam	Fits MAPH(2,m) by approximating characteristics of input MMAP processes
maph2m_fit_multiclass	mam	Fits MAPH(2,m) models to multiclass characteristics with class-specific parameters
maph2m_fit_trace	mam	Fits MAPH(2,m) from empirical trace data for multiclass service time modeling
mapqn_bnd_lr	mapqn	Implements general linear reduction methods for computing performance
mapqn_amva	mapqn	Horizontal-cut mean value analysis of a closed network of one exponential delay station and one FCFS MAP queue
mapqn_bnd_lr_mva	mapqn	Implements linear reduction bounds for MAP queueing networks using Mean
mapqn_bnd_lr_pf	mapqn	Implements linear reduction bounds specialized for product-form MAP
mapqn_bnd_qr	mapqn	Implements general quadratic reduction methods for computing performance
mapqn_bnd_qr_delay	mapqn	Implements quadratic reduction bounds for delay systems in MAP queueing

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mapqn_bnd_qr_ld	mapqn	Implements quadratic reduction bounds for load-dependent MAP queueing
mapqn_lpmodel	mapqn	Base class for representing MAP queueing network linear programming models
mapqn_parameters	mapqn	Defines the base parameter structure for MAP queueing network analysis
mapqn_parameters_factory	mapqn	Factory class for creating parameter objects for MAP queueing network
mapqn_qr_bounds_bas	mapqn	Implements Queue-Router bounds using the Balanced Asymptotic Scaling (BAS)
mapqn_qr_bounds_rsrd	mapqn	Implements Queue-Router (QR) bounds using the Randomized Simultaneous
me_mean	mam	Computes the mean of a Matrix Exponential (ME) distribution
me_pie	mam	Computes the stationary initial probability vector for an ME/RAP distribution
me_sample	mam	Generates random samples from an ME distribution using inverse CDF interpolation
me_scv	mam	Computes the squared coefficient of variation of a Matrix Exponential (ME) distribution
me_var	mam	Computes the variance of a Matrix Exponential (ME) distribution
mmap_backward_moment	mam	Computes backward moments of MMAP inter-arrival times for each marked class
mmap_compress	mam	Compresses MMAP using various approximation methods including mixture, matching,
mmap_count_idc	mam	Computes IDC values for each marked class in MMAP counting processes
mmap_count_lambda	mam	Computes arrival rate vectors for each marked class in MMAP processes
mmap_count_mcov	mam	Computes count covariance matrices between marked classes in MMAP processes
mmap_count_mean	mam	Computes mean count vectors for each marked class in MMAP counting processes
mmap_count_var	mam	Computes variance vectors for counting processes of each marked class in MMAP
mmap_cross_moment	mam	Computes cross-moment matrices between different marked classes in MMAP processes
mmap_embedded	mam	Computes embedded discrete-time Markov chain for MMAP processes
mmap_exponential	mam	Constructs MMAP with exponential inter-arrival distributions for each marked class
mmap_forward_moment	mam	Computes forward moments of MMAP inter-arrival times for each marked class
mmap_hide	mam	Hides specified arrival classes in MMAP processes by removing observable events
mmap_idc	mam	Computes asymptotic IDC for each marked class in MMAP as time approaches infinity
mmap_isfeasible	mam	Validates mathematical feasibility of MMAP representations including stochastic
mmap_issym	mam	Checks if an MMAP is symmetric
mmap_lambda	mam	Returns the per-class arrival rate vector of a Marked MAP (alias for mmap_count_lambda)
mmap_maps	mam	Extracts individual MAP processes for each marked class from MMAP representations
mmap_mark	mam	Converts a Markovian Arrival Process with marked arrivals (MMAP) into a new MMAP with redefined classes based on a given probability matrix
mmap_max	mam	Computes element-wise maximum of MMAP processes for synchronization analysis
mmap_mixture	mam	Creates probabilistic mixtures of MMAP processes with specified weights
mmap_mixture_fit	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given cross-moments
mmap_mixture_fit_mmap	mam	Fits a mixture of Markovian Arrival Processes (MMAPs) to match the given moments

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
mmap_mixture_fit_trace	mam	Fits an MMAP with m classes from trace data using a mixture of m squared PH-distributions
mmap_mixture_order2	mam	Creates a second-order MMAP mixture from a collection of MMAPs
mmap_modulate	mam	Modulates an MMAP by another MMAP, creating a compound arrival process
mmap_normalize	mam	Normalizes MMAP matrices to ensure feasibility and mathematical validity
mmap_pc	mam	Computes the proportion of counts (PC) for each type in a Markovian Arrival Process with marked arrivals (MMAP)
mmap_pie	mam	Computes steady-state probability vectors for each marked class in MMAP processes
mmap_rand	mam	Generates random MMAP representations for testing and simulation purposes
mmap_sample	mam	Generates random samples from MMAP distributions for each marked class
mmap_scale	mam	Rescales MMAP inter-arrival distributions to achieve specified mean values
mmap_shorten	mam	Converts an MMAP representation from M3A format to BUTools format
mmap_sigma	mam	Computes one-step class transition probabilities for a Marked Markovian Arrival Process (MMAP)
mmap_sigma2	mam	Computes two-step class transition probabilities for a Markovian Arrival Process (MMAP)
mmap_sum	mam	Computes superposition of MMAP processes creating independent multiclass arrival streams
mmap_super	mam	Combines multiple MMAP processes into superposed multiclass arrival streams
mmap_super_safe	mam	Combines multiple MMAPs into a single superposition while keeping the resulting order below a given maximum
mmap_timereverse	mam	Computes the time-reversed version of a Markovian Arrival Process with marked arrivals (MMAP)
mmpp2_fit	mam	Fits MMPP(2) models to match specified moments and correlation characteristics
mmpp2_fit1	mam	Fits MMPP(2) models using simplified single-parameter approach for specific scenarios
mmpp2_fit_count	mam	Fits an MMPP(2) using the Heffes-Lucantoni method based on counting process IDC values
mmpp2_fit_count_approx	mam	Fits an MMPP(2) by optimization-based approximation matching counting-process IDCs
mmpp2_fitc	mam	Fits MMPP(2) models using count statistics and index of dispersion criteria
mmpp2_fitc_approx	mam	Fits MMPP(2) using optimization-based approximation methods for count statistics
mmpp_rand	mam	Generates random MMPP models with diagonal D1 matrices for testing and simulation
ms_additivesymmetricchisquared	measures	Additive symmetric chi-squared distance between two probability distributions
ms_anderson_darling	measures	Implements the Anderson-Darling test for assessing whether a sample comes from
ms_avgl1linfty	measures	Average L1 L-infinity distance between two probability distributions
ms_bhattacharyya	measures	Computes the Bhattacharyya distance measuring the similarity between probability
ms_canberra	measures	Canberra distance between two probability distributions
ms_chebyshev	measures	Chebyshev Distance for Probability Distributions
ms_cityblock	measures	Implements the City block distance between probability distributions
ms_clark	measures	Clark distance between two probability distributions
ms_condentropy	measures	Computes conditional entropy measuring the remaining uncertainty
ms_cosine	measures	Computes cosine distance (1 - cosine similarity) measuring the angle between two

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_cramer_von_mises	measures	Implements the Cramer-von Mises test statistic for comparing two empirical distributions
ms_czekanowski	measures	Czekanowski distance between two probability distributions
ms_dice	measures	Dice distance between two probability distributions
ms_divergence	measures	Divergence distance between two probability distributions
ms_entropy	measures	Implements Shannon entropy for discrete random variables
ms_euclidean	measures	Implements the standard Euclidean distance between probability distributions
ms_fidelity	measures	Fidelity distance between two probability distributions
ms_gower	measures	Gower distance between two probability distributions
ms_harmonicmean	measures	Harmonic mean distance between two probability distributions
ms_hellinger	measures	Computes the Hellinger distance measuring dissimilarity between probability distributions
ms_intersection	measures	Intersection distance between two probability distributions
ms_jaccard	measures	Jaccard distance between two probability distributions
ms_jeffreys	measures	Jeffreys divergence between two probability distributions
ms_jensendifference	measures	Jensen difference divergence between two probability distributions
ms_jensenshannon	measures	Implements Jensen-Shannon divergence, a symmetric and bounded version of
ms_jointentropy	measures	Computes joint entropy measuring the uncertainty of joint distributions
ms_kdivergence	measures	K-divergence between two probability distributions
ms_kolmogorov_smirnov	measures	Implements the Kolmogorov-Smirnov test for determining if a sample follows
ms_kuiper	measures	Implements the Kuiper test statistic, a rotation-invariant variant of the Kolmogorov-Smirnov
ms_kulczynskid	measures	Kulczynski d distance between two probability distributions
ms_kulczynskis	measures	Kulczynski s distance between two probability distributions
ms_kullbackleibler	measures	Implements the Kullback-Leibler divergence measuring distribution differences
ms_kumarhassebrook	measures	Kumar-Hassebrook distance between two probability distributions
ms_kumarjohnson	measures	Kumar-Johnson distance between two probability distributions
ms_lorentzian	measures	Lorentzian distance between two probability distributions
ms_matusita	measures	Matusita distance between two probability distributions
ms_minkowski	measures	Minkowski Distance for Probability Distributions
ms_motyka	measures	Motyka distance between two probability distributions
ms_mutinfo	measures	Computes mutual information measuring the amount of shared information
ms_neymanchisquared	measures	Neyman chi-squared distance between two probability distributions
ms_nmi	measures	Computes normalized mutual information providing a scale-invariant measure
ms_nvi	measures	Computes normalized variation information measuring the normalized distance
ms_pearsonchisquared	measures	Pearson chi-squared distance between two probability distributions
ms_probsymmchisquared	measures	Probabilistic symmetry chi-squared distance between two probability distributions
ms_product	measures	Product distance between two probability distributions
ms_relatentropy	measures	Computes relative entropy measuring the information difference
ms_ruzicka	measures	Ruzicka distance between two probability distributions
ms_soergel	measures	Soergel distance between two probability distributions
ms_sorensen	measures	Sorensen distance between two probability distributions
ms_squaredchisquared	measures	Squared chi-squared distance between two probability distributions
ms_squaredchord	measures	Squared chord distance between two probability distributions
ms_squaredeuclidean	measures	Squared Euclidean distance between two probability distributions
ms_taneja	measures	Taneja distance between two probability distributions
ms_tanimoto	measures	Tanimoto distance between two probability distributions
ms_topsoe	measures	Topsoe distance between two probability distributions
ms_wasserstein	measures	Implements the Wasserstein distance measuring the minimum cost to transform

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
ms_wavehegdes	measures	Wave-Hedges distance between two probability distributions
mtrace_backward_moment	trace	Computes backward moments of a multi-class trace
mtrace_bootstrap	trace	Implements bootstrap resampling methods for multi-class empirical trace data
mtrace_count	trace	Computes count statistics from a multi-class trace over specified time windows
mtrace_cov	trace	Computes the covariance matrix for multi-type traces
mtrace_cross_moment	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_forward_moment	trace	Computes the forward moments of a marked trace
mtrace_iat2counts	trace	Computes the per-class counting processes of T, i.e., the counts after
mtrace_joint	trace	Given a multi-class trace, computes the empirical class-dependent joint
mtrace_mean	trace	Computes per-class means for multi-class empirical trace data. Enables separate
mtrace_merge	trace	Merges two traces in a single marked (multiclass) trace
mtrace_moment	trace	Computes empirical class-dependent statistical moments for multi-class trace data
mtrace_moment_simple	trace	Computes the k-th order moment of the inter-arrival time between an event
mtrace_pc	trace	Computes the probabilities of arrival for each class
mtrace_sigma	trace	Computes the empirical probability of observing a specific 2-element
mtrace_sigma2	trace	Computes the empirical probability of observing a specific 3-element
mtrace_split	trace	Given a multi-class trace with inter-arrivals T and labels L,
mtrace_summary	trace	Computes summary statistics for multiple trace analysis, providing
npfqn_bnd_bgt	npfqn	Piecewise-linear Lyapunov upper bound on the steady-state queue lengths of an open multiclass Markovian network
npfqn_bnd_bpt	npfqn	Achievable-region linear-programming lower bound on the sojourn times of an open multiclass Markovian network
npfqn_nonexp_approx	npfqn	Implements approximation methods for non-product-form queueing networks
npfqn_traffic_merge	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_merge_cs	npfqn	Implements traffic merging algorithms for non-product-form queueing networks
npfqn_traffic_rqt	npfqn	Effective arrival process perceived at each node of a single-class open network under robust queueing theory
npfqn_traffic_split_cs	npfqn	Implements traffic splitting algorithms for non-product-form queueing networks
pfqn_ab	pfqn.ld	Implements the Akyildiz-Bolch linearizer method for analyzing closed product-form queueing networks
pfqn_ab_amva	pfqn.mva	Akyildiz-Bolch AMVA approximation for closed BCMP networks with multi-server stations
pfqn_aghq	pfqn.nc	Adaptive Gauss-Hermite quadrature of the simplex factor of the McKenna-Mitra integral, q nodes per direction; $q = 1$ reproduces pfqn_le
pfqn_aql	pfqn.mva	Implements the Aggregate Queue Length (AQL) approximation method for analyzing closed
pfqn_bk	pfqn	Birman-Kogan saddle point normalizing constant with bottleneck detection
pfqn_bklc	pfqn	Birman-Kogan load concealment algorithm: multichain solved as single chain problems
pfqn_bkt	pfqn.nc	Knessl-Tier expansion with the exact Stirling remainder of each Laplaced class direction subtracted (BKT)
pfqn_ble	pfqn.nc	Logistic expansion with the empirical $(M-1)(1-\log(2\pi)/2)$ bias correction applied before the $\epsilon \to 0$ limit (BLE)
pfqn_bkue	pfqn	Birman-Kogan uniform (van der Waerden) expansion for a single chain
pfqn_bs	pfqn.mva	Implements the classic Bard-Schweitzer approximate MVA algorithm for closed queueing networks
pfqn_busyp	pfqn	Mean busy periods of orders $1..K$ of a closed product-form network, per station, station-class pair and station subset (Daduna)

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_ca	pfqn.nc	Convolution Algorithm for Product-Form Networks
pfqn_cdfun	pfqn.ld	Provides functionality to evaluate class-dependent scaling functions in load-dependent queueing networks
pfqn_cftp	pfqn.nc	Perfect sampling of the stationary queue-length vector of a closed single-class product-form network by monotone coupling from the past [104], with no normalizing constant
pfqn_chow	pfqn	JMT-compatible Chow approximate MVA
pfqn_clust	pfqn	de Souza e Silva-Lavenberg-Muntz Clustering Approximation (CA)
pfqn_clw	pfqn.nc	Choudhury-Leung-Whitt normalizing constant by numerical inversion of the generating function [42]
pfqn_clw_lld	pfqn.nc	Choudhury-Leung-Whitt inversion extended to limited load-dependent stations via Bertozzi-McKenna transforms [14]
pfqn_cntol	pfqn	Chandy-Neuse population-scaled termination cutoff for approximate MVA
pfqn_comom	pfqn.nc	Computes the log normalizing constant of a multi-station closed network using CoMoM matrix recursion
pfqn_comomrm	pfqn.nc	Implements the Class-Oriented Method of Moments specialized for repairman queueing models
pfqn_comomrm_ld	pfqn.ld	Implements the Class-Oriented Method of Moments (COMOM) for computing normalizing constants in
pfqn_comomrm_ms	pfqn.nc	CoMoM normalizing-constant method for multiserver repairman models with load-dependent rates
pfqn_comomrm_orig	pfqn.nc	Original CoMoM implementation for the single-station finite repairman model
pfqn_conv	pfqn.ld	Multichain convolution algorithm for closed networks with class-dependent service rates (Sauer eq. (40))
pfqn_conwayms	pfqn.mva	Implements the Conway-Maxwell approximation method for analyzing closed queueing networks
pfqn_cub	pfqn.nc	Implements the cubature (multi-dimensional integration) approach for computing normalizing
pfqn_cyclet_ofree	pfqn	Exact passage-time density, distribution and moments along an overtake-free path of a closed single-chain product-form network
pfqn_dmlin	pfqn	de Souza e Silva-Muntz Improved Linearizer (IL)
pfqn_egflinearizer	pfqn.mva	Implements the Extended General-Form linearizer approximation for closed queueing networks
pfqn_fnc	pfqn.ld	Computes scaling factors for load-dependent functional servers in product-form queueing networks
pfqn_gerasimov	pfqn	Gerasimov closed-form residue evaluation of the normalizing constant
pfqn_gflinearizer	pfqn.mva	Implements the general-form linearizer approximation for closed queueing networks with
pfqn_gld	pfqn.ld	Implements the generalized convolution algorithm for computing normalizing constants in
pfqn_gld_complex	pfqn.ld	Extends the generalized load-dependent convolution algorithm to handle complex-valued
pfqn_gldsingle	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_gldsingle_complex	pfqn.ld	Provides specialized auxiliary function for computing normalizing constants in single-class
pfqn_grnmol	pfqn.nc	Computes normalizing constants using the Grundmann-Moeller simplex quadrature rule
pfqn_harel_bounds	pfqn	Computes Harel-Namn-Sturm throughput bounds for closed queueing networks
pfqn_joint	pfqn	Computes joint queue-length probabilities for closed product-form networks via normalizing constants
pfqn_jointmarg	pfqn	Computes the joint law of the per-station total queue lengths, classes summed out, as a permanent of the demand matrix replicated once per job; every infinite-server station contributes its own $1/n_j!$

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_kt	pfqn.nc	Implements the Knessl-Tier asymptotic expansion using the ray method for computing
pfqn_lap	pfqn.nc	Laplace (saddle-point) approximation for the log normalizing constant of a product-form network
pfqn_lcfsqn_ca	pfqn.lcfs	Convolution algorithm for normalizing constants in 2-station LCFS queueing networks
pfqn_lcfsqn_mva	pfqn.lcfs	Exact log-space MVA for 2-station LCFS and LCFS-PR queueing networks
pfqn_lcfsqn_nc	pfqn.lcfs	Computes the normalizing constant of LCFS queueing networks via matrix permanent calculations
pfqn_lcp	pfqn	Bard Large Customer Population (LCP) approximate MVA
pfqn_le	pfqn.nc	Implements the Logistic expansion approach for computing normalizing constants in
pfqn_le_fpi	pfqn.nc	Implements the fixed-point iteration algorithm used in the Logistic expansion method
pfqn_le_fpiz	pfqn.nc	Implements the fixed-point iteration algorithm for the Logistic expansion method
pfqn_le_hessian	pfqn.nc	Computes the Hessian matrix used in the Logistic expansion method for second-order
pfqn_le_hessianz	pfqn.nc	Computes the Hessian matrix used in the Logistic expansion method for closed queueing
pfqn_lekt	pfqn.nc	The estimator common to pfqn_ble and pfqn_bkt, evaluated on the cheaper of the saddle-point and logistic sides
pfqn_linearizer	pfqn.mva	Linearizer Approximate MVA for Product-Form Networks
pfqn_linearizerms	pfqn.mva	Implements the multi-server version of Krzesinski's linearizer approximation for closed
pfqn_linearizermx	pfqn.mva	Implements linearizer-based approximation methods for mixed queueing networks with
pfqn_linearizerpp	pfqn.mva	Implements the Linearizer++ algorithm for closed queueing networks with enhanced accuracy
pfqn_lldfun	pfqn.ld	Evaluates limited load-dependent (LLD) scaling functions using spline interpolation for
pfqn_lld	pfqn.ld	Normalizing constant of a multiclass closed model with limited load-dependent stations
pfqn_lldsingle	pfqn.ld	Exact normalizing constant of a single-class limited load-dependent model, linear in the population
pfqn_looping	pfqn	Eager Looping approximate MVA
pfqn_ls	pfqn.nc	Implements the logistic sampling approach for computing normalizing constants in
pfqn_mci	pfqn.nc	Implements Monte Carlo integration approaches including Importance Monte Carlo Integration
pfqn_minclasses	pfqn	Lower bound on the number of customer classes needed to explain a set of measurements
pfqn_mmint2	pfqn.nc	Implements numerical integration for computing normalizing constants in multi-class
pfqn_mmint2_gausslaguerre	pfqn.nc	Computes the McKenna-Mitra normalizing constant for repairman models via Gauss-Laguerre quadrature
pfqn_mmint2_gausslegendre	pfqn.nc	Implements Gauss-Legendre quadrature integration for computing normalizing constants
pfqn_mmsample2	pfqn.nc	Implements importance sampling for computing normalizing constants in multi-class
pfqn_mom	pfqn.nc	Implements the Method of Moments using exact arithmetic with BigFraction for computing
pfqn_momlin	pfqn.mva	Moment linearizer: mean queue lengths from the Schweitzer-Bard fixed point and their variances and covariances from its analytic demand derivatives, polynomial rather than exponential in the number of classes
pfqn_mu_ms	pfqn.ld	Computes load-dependent scaling factors for multi-server queueing stations with finite

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_mushift	pfqn.ld	Provides utility function for shifting load-dependent scaling vectors by one position,
pfqn_mva	pfqn.mva	Mean Value Analysis for Product-Form Queueing Networks
pfqn_mva_ilock	pfqn	Exact MVA recursion carrying the interlocked-flow correction
pfqn_mva_interval	pfqn.mva	Exact output intervals of single-class MVA under interval-valued demands, think time and population
pfqn_mvald	pfqn.ld	Load-Dependent Mean Value Analysis
pfqn_mvaldms	pfqn.ld	Provides wrapper functionality for load-dependent Mean Value Analysis with automatic
pfqn_mvaldmx	pfqn.ld	Implements Mean Value Analysis for mixed queueing networks with both open and closed classes
pfqn_mvaldmx_ec	pfqn.ld	Provides auxiliary functionality for computing EC terms used in load-dependent Mean Value
pfqn_mvams	pfqn.mva	Provides comprehensive MVA solution for mixed queueing networks with multi-server stations
pfqn_mvams_ilock	pfqn	MVA entry point for models carrying the interlocked-flow correction
pfqn_mvamx	pfqn.mva	Implements MVA for mixed networks containing both open and closed classes without multi-server
pfqn_mwrbb	pfqn	Computes Majumdar-Woodside robust box bounds on per-class throughput for closed multiclass networks (NBUE service, FIFO/PS/priority disciplines)
pfqn_nc	pfqn.nc	Normalizing Constant Methods for Product-Form Networks
pfqn_nc_sanitize	pfqn.nc	Sanitizes and preprocesses parameters for product-form queueing network models to
pfqn_nca	pfqn.nc	Implements the Normalizing Constant Approximation method for single-class closed
pfqn_ncld	pfqn.ld	Provides the main entry point for computing normalizing constants in load-dependent
pfqn_nre	pfqn	Normalizing constant via saddle-tilted Edgeworth (NRE) approximation
pfqn_nrl	pfqn.nc	Implements the Norlund-Rice Logit approach for computing normalizing constants
pfqn_nrp	pfqn.nc	Implements the Normal Random Permutation approach for computing normalizing constants
pfqn_pam	pfqn	Hsieh-Lam Proportional Approximation Methods (PAMB/PAMI/PAMT)
pfqn_qsa	pfqn	Queue-shift approximation for closed product-form networks, solving the absolute aggregate queue-length shifts by damped Newton
pfqn_qdamva	pfqn.ld	Queue-dependent approximate mean value analysis for load-dependent closed networks
pfqn_qdlin	pfqn.ld	The Linearizer arm of AMVA-LD, evaluated on a plain demand matrix
pfqn_panacea	pfqn.nc	Implements the PANACEA approximation method for computing normalizing constants in
pfqn_perm	pfqn	Computes the permanent of a demand matrix with optional column multiplicities, by Ryser's inclusion-exclusion formula
pfqn_pff_delay	pfqn.nc	Computes the product-form factor for delay stations in closed queueing networks
pfqn_procomom	pfqn.nc	Computes marginal queue-length probabilities using the Probabilistic CoMoM matrix recursion
pfqn_procomom2	pfqn.ld	Implements the probabilistic class-oriented method of moments for analyzing
pfqn_propfair	pfqn.nc	Implements the proportionally fair allocation method using convex optimization
pfqn_qzgblow	pfqn.mva	Computes the lower Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_qzgbup	pfqn.mva	Computes the upper Geometric Bound (GB) for queue lengths in closed single-class queueing
pfqn_rd	pfqn.nc	Implements the Reduction Heuristic approach for computing normalizing constants in

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
pfqn_recal	pfqn.nc	Implements the RECAL (Recursive Calculation) algorithm for computing normalizing constants
pfqn_replicas	pfqn	Identifies replicated stations with identical demands and consolidates them into unique stations with multiplicity
pfqn_scat	pfqn	Neuse-Chandy SCAT (Self-Correcting Approximation Technique) AMVA
pfqn_scb	pfqn	Dowdy-Carlson-Krantz-Tripathi (1992) single-class bounds on the
pfqn_scbgap	pfqn	Maximum relative throughput error of aggregating several customer classes into one
pfqn_schmidt	pfqn.ld	Schmidt method for load-dependent MVA with multi-server stations
pfqn_schmidt_amva	pfqn.mva	Schmidt MVA algorithm for multi-class FCFS queueing networks with class-dependent service
pfqn_sdr	pfqn	Exact product form of a multiclass network with state-dependent routing
pfqn_sdrcoeff	pfqn	Derived coefficients of a Krzesinski state-dependent routing structure
pfqn_sdrmva	pfqn	MVA and convolution solution of a network with state-dependent routing
pfqn_sdrprob	pfqn	Krzesinski state-dependent routing probabilities
pfqn_sdrvisits	pfqn	Section 3.2 coefficients xi of a network with state-dependent routing
pfqn_sens	pfqn.sens	Exact Jacobian of the mean measures of a closed product-form network with respect to the service demands and think times, plus the queue-length covariances
pfqn_sens_ldmx_ec	pfqn.sens	Effective capacity terms of the mixed load-dependent MVA together with their exact derivatives with respect to the open-class load
pfqn_sens_linearizer	pfqn.sens	Approximates the queue-length moments of a closed network in polynomial time by differentiating the Linearizer fixed point
pfqn_sens_mom	pfqn.sens	Exact moments up to the third of the queue length of any grouping of the classes (per class, per chain, or per station) of a closed network
pfqn_sens_mva	pfqn.sens	Exact per-class queue-length variances and covariances of a closed network by an MVA-like recursion, without forming the sensitivity Jacobian
pfqn_sens_mvaldmx	pfqn.sens	Exact queue-length variances and covariances for mixed open/closed networks with limited load dependence
pfqn_sens_respt	pfqn.sens	Exact moments of the sojourn time at FCFS multiserver centers of a closed network, via the arrival theorem
pfqn_sqni	pfqn.mva	Implements the Square-root Non-iterative method for analyzing multi-class closed
pfqn_stdf	pfqn.nc	Implements McKenna's 1987 method for computing sojourn time distributions at
pfqn_stdf_heur	pfqn.nc	Implements a heuristic variant of McKenna's 1987 method for computing sojourn time
pfqn_usumbound	pfqn	Upper bound on the sum of the device utilizations of a closed multiclass network
pfqn_xia	pfqn.ld	Implements Xia's asymptotic approximation method for computing normalizing constants
pfqn_xzabalow	pfqn.mva	Computes the lower ABA bound for throughput in closed single-class queueing networks
pfqn_xzabaup	pfqn.mva	Computes the upper ABA bound for throughput in closed single-class queueing networks
pfqn_xzgsblow	pfqn.mva	Computes the lower GSB for throughput in closed single-class queueing networks using
pfqn_xzgsbup	pfqn.mva	Computes the upper GSB for throughput in closed single-class queueing networks using
ph_reindex	mam	Reindexes phase-type distribution maps for network models using integer station and class indices
ph_multisets	mam	Enumerates and indexes the multisets of busy-server phases used by the exact multiserver LDQBD and MAP/PH/c chains
polling_qsys_1limited	polling	Implements analysis algorithms for 1-limited polling systems where the
polling_qsys_exhaustive	polling	Implements analysis algorithms for exhaustive polling systems where the
polling_qsys_gated	polling	Implements analysis algorithms for gated polling systems where the server
qbd_bmapbmap1	mam	Analyzes batch arrival and service systems using QBD matrix methods

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
qbd_depproc_etaqa	mam	Constructs a MAP departure-process approximation for MAP/MAP/1-FCFS via ETAQA truncation
qbd_depproc_etaqa_ps	mam	Constructs a MAP departure-process approximation for MAP/MAP/1-PS via ETAQA truncation
qbd_depproc_jointmom	mam	Computes joint moments of consecutive inter-departure times for MAP/MAP/1 queues
qbd_mapmap1	mam	Analyzes MAP/MAP/1 queueing systems using QBD matrix analytic methods
qbd_r	mam	Computes the QBD R matrix using successive substitutions until convergence
qbd_r_logred	mam	Computes QBD R-matrix using logarithmic reduction method for numerical stability
qbd_raprap1	mam	Analyzes RAP/RAP/1 queueing systems using QBD methods with rational arrival processes
qbd_rg	mam	Computes fundamental R and G matrices for QBD analysis of MAP/MAP/1 queues
qbd_setupdelayoff	mam	Analyzes queueing systems with server setup delays and switch-off mechanisms
qsys_gg1	qsys	Provides comprehensive analysis of G/G/1 queues with general arrival and service processes
qsys_gig1_approx_allencunneen	qsys	Implements the widely-used Allen-Cunneen approximation for general G/G/1 queueing
qsys_gig1_approx_gelenbe	qsys	G/G/1 queue approximation using Gelenbe's method
qsys_gig1_approx_heyman	qsys	Analyzes a G/G/1 queueing system using Heyman's approximation
qsys_gig1_approx_kimura	qsys	G/G/1 queue approximation using Kimura's method
qsys_gig1_approx_klb	qsys	Analyzes a G/G/1 queueing system using the Kramer-Langenbach-Belz (KLB) approximation
qsys_gig1_approx_kobayashi	qsys	Analyzes a G/G/1 queueing system using Kobayashi's approximation
qsys_gig1_approx_marchal	qsys	Analyzes a G/G/1 queueing system using Marchal's approximation
qsys_gig1_approx_myskja	qsys	G/G/1 queue approximation using Myskja's method
qsys_gig1_approx_myskja2	qsys	G/G/1 queue approximation using enhanced Myskja's method
qsys_gig1_lbnd	qsys	G/G/1 queue lower bounds
qsys_gig1_rqt	qsys	Robust Queueing Theory (RQT) worst-case system time of a G/G/1 FCFS queue,
qsys_gig1_ubnd_kingman	qsys	Calculates an upper bound on the waiting time for a G/G/1 system using Kingman's formula
qsys_gigk_approx	qsys	Analyzes a G/G/k queueing system using an approximation method
qsys_gigk_approx_cosmetatos	qsys	G/G/k queue approximation using Cosmetatos method
qsys_gigk_approx_kingman	qsys	Analyzes a G/G/k queueing system using Kingman's approximation
qsys_gigk_approx_whitt	qsys	G/G/k queue approximation using Whitt's method
qsys_gigk_rqt	qsys	Robust Queueing Theory (RQT) worst-case system time of a G/G/k FCFS queue
qsys_gigk_rqt_gamma	qsys	Service variability parameter of the Robust Queueing Theory (RQT) framework,
qsys_gm1	qsys	G/M/1 Queueing System Analysis
qsys_mapg1	qsys	Analyzes MAP/G/1 queues with MAP arrivals and general service times using BUTools
qsys_mapm1	qsys	MAP/M/1 queueing system analysis with MAP arrivals and exponential service
qsys_mapmap1	qsys	Analyzes MAP/MAP/1 queues with MAP arrivals and MAP service times
qsys_mapmc	qsys	MAP/M/c multiserver queueing system analysis with MAP arrivals
qsys_mapphc	qsys	Exact MAP/PH/c queue on the multiset-of-phases chain: queue-length distribution, per-server utilization, waiting probability, waiting-time moments and CCDF
qsys_mmapgk1	qsys	Exact MMAP[K]/G[K]/1 FCFS queue by transforms: per-class waiting-time moments, LST and distribution function
qsys_mapph1	qsys	Analyzes MAP/PH/1 queues with MAP arrivals and phase-type service distributions

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
qsys_mg1	qsys	Implements the Pollaczek-Khinchine formula for M/G/1 queues with Poisson arrivals
qsys_mg1_prio	qsys	M/G/1 queueing system with non-preemptive class priorities
qsys_mg1_ps	qsys	Exact Ott-Yashkov sojourn time transform of the M/G/1-PS queue, in closed form for phase-type service
qsys_mg1k_loss	qsys	M/G/1/K loss probability calculation
qsys_mg1k_loss_mgs	qsys	M/G/1/K loss probability using MacGregor Smith approximation
qsys_mginf	qsys	M/G/inf queue analysis (infinite servers)
qsys_mm1	qsys	Implements exact analytical solutions for the M/M/1 queue (Poisson arrivals, exponential
qsys_mm1k_loss	qsys	M/M/1/K loss probability calculation
qsys_mmk	qsys	Implements exact analytical solutions for M/M/k queues with Poisson arrivals,
qsys_phph1	qsys	Analyzes PH/PH/1 queues with phase-type arrivals and service distributions
qsys_tandem_ub_ciucu	qsys	Polynomial-exponential upper bounds on the end-to-end waiting and sojourn time tails of a two-station tandem
randp	mam	Provides random value selection based on relative probability distributions
rap_sample	mam	Generates random samples from a Rational Arrival Process (RAP) marginal distribution
smp_passage_lst	mc	Laplace-Stieltjes transform of the first passage time of a semi-Markov chain into a target state set
smp_passage_moments	mc	Moments of the first passage time of a semi-Markov chain into a target state set
smp_passage_time	mc	Distribution and density of the first passage time of a semi-Markov chain into a target state set
sn_deaggregate_chain_results	sn	Calculate class-based performance metrics for a queueing network based on performance measures of its chains
sn_fj_visits_spn	sn	Computes fork-join visit ratios by building auxiliary closed SPN models solved with SolverCTMC
sn_get_arv_r_from_tput	sn	Calculates the average arrival rates at each station from the network throughputs
sn_get_demands_chain	sn	Calculate new queueing network parameters after aggregating classes into chains
sn_get_node_arv_r_from_tput	sn	Computes per-node arrival rates from station throughputs by applying nodevisit ratios for non-station nodes
sn_get_node_tput_from_tput	sn	Computes per-node throughputs from station throughputs by combining nodevisit ratios with the routing matrix
sn_get_product_form_chain_params	sn	Calculate the parameters at class and chain level for a queueing network model
sn_get_product_form_params	sn	Extracts essential parameters (service demands, populations, visit ratios) from
sn_get_residt_from_respt	sn	Calculates the residence times at each station from the response times
sn_get_state_aggr	sn	Aggregates the state of the network
sn_has_class_switching	sn	Checks if the network uses class-switching
sn_has_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_dps	sn	Check if the network includes a node with DPS scheduling
sn_has_dps_prio	sn	Check if the network includes a node with DPSPRIO scheduling
sn_has_fb	sn	Check if the network includes a node with FB (LAS) scheduling
sn_has_fcfs	sn	Identifies queueing networks using First-Come-First-Served scheduling disciplines
sn_has_fork_join	sn	Checks if the network uses fork and/or join nodes
sn_has_fractional_populations	sn	Checks if the network has closed classes with non-integer populations
sn_has_gps	sn	Check if the network includes a node with GPS scheduling
sn_has_gps_prio	sn	Check if the network includes a node with GPSPRIO scheduling
sn_has_hol	sn	Check if the network includes a node with HOL scheduling
sn_has_homogeneous_scheduling	sn	Checks if the network uses an identical scheduling strategy at every station

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_has_inf	sn	Check if the network includes a node with INF scheduling
sn_has_lcfs	sn	Check if the network includes a node with LCFS scheduling
sn_has_lcfs_pi	sn	Check if the network includes a node with LCFSPI scheduling
sn_has_lcfs_pr	sn	Check if the network includes a node with LCFSPR scheduling
sn_has_lept	sn	Check if the network includes a node with LEPT scheduling
sn_has_ljf	sn	Check if the network includes a node with LJF scheduling
sn_has_load_dependence	sn	Checks if the network has a station with load-dependent service process
sn_has_lrpt	sn	Check if the network includes a node with LRPT scheduling
sn_has_mixed_classes	sn	Checks if the network has both open and closed classes
sn_has_multi_chain	sn	Check if the network has multiple chains
sn_has_multi_class	sn	Identifies queueing networks with multiple job classes, which require specialized
sn_has_multi_class_fcfs	sn	Checks if the network has any FCFS station with non-zero service rates for more than one class
sn_has_multi_class_heter_exp_fcfs	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
sn_has_multi_class_heter_fcfs	sn	Checks if the network has one or more stations with multiclass heterogeneous FCFS
sn_has_multi_server	sn	Check if the network includes a multi-server node
sn_has_multiple_closed_classes	sn	Checks if the network has one or more closed classes
sn_has_open_classes	sn	Checks if the network has one or more open classes
sn_has_polling	sn	Check if the network includes a node with polling
sn_has_priorities	sn	Checks if the network uses class priorities
sn_has_product_form	sn	Determines if a queueing network has a known product-form solution by validating
sn_has_product_form_except_multi_class_heter_exp_fcfs	sn	Checks product-form assumptions except for multiclass heterogeneous exponential FCFS stations
sn_has_product_form_not_het_fcfs	sn	Checks if the network satisfies product-form assumptions (does not have heterogeneous FCFS)
sn_has_ps	sn	Check if the network includes a node with PS scheduling
sn_has_ps_prio	sn	Check if the network includes a node with PSPRIO scheduling
sn_has_psjf	sn	Check if the network includes a node with PSJF scheduling
sn_has_quorum_join	sn	Checks if the network has a quorum (k-of-n) join
sn_has_sd_routing	sn	Checks if the network has state-dependent routing strategies that violate product-form
sn_has_sept	sn	Check if the network includes a node with SEPT scheduling
sn_has_single_chain	sn	Check if the network has a single chain
sn_has_single_class	sn	Check if the network has a single class
sn_has_siro	sn	Check if the network includes a node with SIRO scheduling
sn_has_sjf	sn	Check if the network includes a node with SJF scheduling
sn_has_srpt	sn	Check if the network includes a node with SRPT scheduling
sn_interlock_chain	sn	Aggregates a class-indexed interlock matrix to the chain basis read by the MVA recursion
sn_is_closed_model	sn	Identifies closed queueing network models with finite job populations and no external
sn_is_mixed_model	sn	Checks if the network is a mixed model
sn_is_open_model	sn	Identifies open queueing network models with external arrivals and infinite
sn_is_population_model	sn	Checks if the model is a population model (only specific scheduling strategies without priorities or fork-join)
sn_is_state_valid	sn	Stochastic Network State Validation Utility
sn_join_droprate	fj	Rate at which sibling tasks are discarded at each Join of a fork-join network
sn_join_quorum	fj	Number of sibling tasks the Join node JOINIDX waits for in class R, given
sn_join_siblings	fj	Number of sibling tasks forked per parent job of class R on the fork-join
sn_modify_options	sn	Defines options (e.g., in-place vs. copy mode) controlling NetworkStruct modification methods

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
sn_nonmarkov_to_ph	sn	Converts non-Markovian distributions in a network to phase-type via Bernstein approximation
sn_print	sn	Prints comprehensive information about a NetworkStruct
sn_print_routing_matrix	sn	Prints the routing matrix of the network, optionally for a specific job class
sn_refresh_process_fields	sn	Refreshes service-process fields (mu, phi, proc, pie, phases) from current rates and SCVs
sn_refresh_visits	sn	Stochastic Network Visit Ratio Calculator
sn_rtnodes_to_rtorig	sn	Converts routing matrices from nodes to original format, specifically handling class switching nodes
sn_set_arrival	sn	Updates the arrival rate of an open class at a Source station; delegates to sn_set_service
sn_set_arrival_batch	sn	Batch update of arrival rates for multiple class indexes at a Source station
sn_set_fork_fanout	sn	Updates the fanout of a Fork node by writing to nodeparam.fanOut
sn_set_population	sn	Updates the population of a closed class and recalculates nclosedjobs
sn_set_priority	sn	Updates the scheduling priority of a class via the classprio vector
sn_set_routing	sn	Replaces the full routing matrix rt with optional auto-refresh of derived fields
sn_set_routing_prob	sn	Updates a single entry of rt for a (from,to) stateful node and class pair
sn_set_servers	sn	Updates the number of servers at a station via the nservers vector
sn_set_service	sn	Updates service rate and squared coefficient of variation for a (station,class) pair
sn_set_service_batch	sn	Batch update of service rates and SCV values for multiple (station,class) pairs; NaN entries are skipped
sn_set_service_coc	sn	Updates service rate in cell-of-cells representation, refreshing mu, phi, pie, and proc
sn_to_ag	sn	Converts a LINE network structure into the agent (RCAT) format used by SolverAG
sn_validate	sn	Validates NetworkStruct consistency and returns a list of validation errors
spn_lpbnd	spn	Linear-programming bounds on the mean marking and the throughputs of a stochastic timed Petri net, over the uniformized moment polytope
spn_properties	spn	Aggregate Petri-net property analysis: invariants, siphons and traps, structural LP checks and reachability-graph verdicts
spn_reachgraph	spn	Reachability graph of a stochastic Petri net under the GSPN firing rule (inhibitors, priorities, immediate events)
spn_reachprops	spn	Boundedness, deadlocks, L4 liveness and reversibility from the reachability graph
spn_sinvariants	spn	Minimal-support S-invariants (P-invariants) of a stochastic Petri net and the load vector they induce
spn_siphons	spn	All minimal siphons of a stochastic Petri net and Commoner's siphon-trap condition
spn_structprops	spn	Structural boundedness, conservativeness, consistency and repetitiveness by linear programming, and the net classes
spn_tinvariants	spn	Minimal-support T-invariants of a stochastic Petri net
spn_traps	spn	All minimal traps of a stochastic Petri net
snc_bound_backlog	snc	Violation probability of a backlog level (stochastic network calculus)
snc_bound_delay	snc	Violation probability of a delay target (stochastic network calculus)
snc_conv	snc	Min-plus convolution of two service envelopes (tandem concatenation)
snc_env_cpoisson	snc	MGF arrival envelope of a compound Poisson flow with Exp job sizes
snc_env_map	snc	MGF arrival envelope of a MAP/MMPP flow with unit-size jobs
snc_env_poisson	snc	MGF arrival envelope of a Poisson flow with unit-size jobs
snc_env_tokenbucket	snc	Deterministic token-bucket arrival envelope
snc_leftover	snc	Leftover service envelope under blind multiplexing
snc_mean_backlog	snc	Upper bound on the mean backlog from the tail bound
snc_mean_delay	snc	Upper bound on the mean delay from the tail bound
snc_output	snc	Output arrival envelope of a flow leaving a server
snc_perc_backlog	snc	Backlog quantile at a prescribed violation probability

Continued on next page

Table C.1 – API Function Reference. *Continued from previous page*

Function Name	Package	Description
snc_perc_delay	snc	Delay quantile at a prescribed violation probability
snc_srv_exp	snc	MGF service envelope of an exponential server, in job units
snc_srv_rate	snc	MGF service envelope of a constant-rate work-conserving server
snc_thetaopt	snc	Minimize a Chernoff bound over the free parameter theta
talbot_get_alpha	lti	Nodes of the Talbot deformed contour used by the Laplace inversion
talbot_get_omega	lti	Weights of the Talbot deformed contour used by the Laplace inversion
trace_mean	trace	Computes the arithmetic mean of empirical trace data. Fundamental statistical
trace_skew	trace	Computes the skewness of the trace data using Apache Commons Math
trace_var	trace	Computes sample variance and related statistics for empirical trace data
wf_analyzer	wf	Provides comprehensive workflow analysis capabilities including pattern
wf_auto_integration	wf	Provides automatic integration capabilities for workflow analysis with
wf_branch_detector	wf	Implements algorithms for detecting branching patterns in workflow traces
wf_loop_detector	wf	Implements algorithms for detecting loop and iterative patterns in workflow
wf_parallel_detector	wf	Implements algorithms for detecting parallel execution patterns in workflow
wf_pattern_updater	wf	Implements dynamic pattern updating algorithms for workflow analysis
wf_sequence_detector	wf	Implements algorithms for detecting sequential patterns in workflow traces

Bibliography

[1] M. Ajmone Marsan, G. Conte, and G. Balbo. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. Comput. Syst.*, 2(2):93–122, May 1984.

[2] Ian F Akyildiz and Gunter Bolch. Approximate analysis of load dependent general queueing networks. *IEEE Transactions on Software Engineering*, 14(11):1537–1545, 1988.

[3] D. Anick, D. Mitra, and M. M. Sondhi. Stochastic theory of a data handling system with multiple sources. *The Bell System Technical Journal*, 61:1871–1894, 1982.

[4] J. Anselmi and P. Cremonesi. On the property of product-form queueing networks with load-dependent stations. *Performance Evaluation*, 65(11–12):858–874, 2008.

[5] J. Anselmi, B. D'Auria, and N. Walton. Closed queueing networks under congestion: non-bottleneck independence and bottleneck convergence. accepted in Mathematics of Operations Research, 2013.

[6] A. Asanjarani, B. Liquet, and Y. Nazarathy. Estimation of semi-Markov multi-state models: a comparison of the sojourn times and transition intensities approaches. arXiv:2005.14462, 2020.

[7] S. Asmussen and J. R. Møller. Calculation of the steady state waiting time distribution in GI/PH/c and MAP/PH/c queues. *Queueing Systems*, 37(1):9–29, 2001.

[8] Simonetta Balsamo, Gian-Luca Dei Rossi, and Andrea Marin. A numerical algorithm for the solution of product-form models with infinite state spaces. In *European Performance Engineering Workshop*, volume 6342 of *LNCS*, pages 191–206. Springer, 2010.

[9] C. Bandi, D. Bertsimas, and N. Youssef. Robust queueing theory. *Operations Research*, 63(3):676–700, 2015.

[10] Y. Bard. Some extensions to multiclass queueing network analysis. In *Proc. of the 3rd Int'l Symp. on Model. and Performance Evaluation of Comp. Syst.*, pages 51–62, 1979.

[11] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed and mixed networks of queues with different classes of customers. *J. Assoc. Comput. Mach.*, 22:248–260, 1975.

[12] N. G. Bean, M-M. O'Reilly, and P. G. Taylor. Algorithms for return probabilities for stochastic fluid flows. *Stochastic Models*, 21:149–184, 2005.

[13] M. Bertoli, G. Casale, and G. Serazzi. The JMT simulator for performance evaluation of non-product-form queueing networks. In *Proc. of the 40th Annual Simulation Symposium (ANSS)*, pages 3–10, 2007.

[14] A. L. Bertozzi and J. McKenna. Multidimensional residues, generating functions, and their application to queueing networks. *SIAM Review*, 35(2):239–268, 1993.

[15] D. Bertsimas, D. Gamarnik, and J. N. Tsitsiklis. Performance of multiclass markovian queueing networks via piecewise linear lyapunov functions. *The Annals of Applied Probability*, 11(4):1384–1428, 2001.

[16] D. Bertsimas, I. Ch. Paschalidis, and J. N. Tsitsiklis. Optimization of multiclass queueing networks: Polyhedral and nonlinear characterizations of achievable performance. *The Annals of Applied Probability*, 4(1):43–75, 1994.

[17] D. Bini, B. Meini, S. Steffé, J. F. Pérez, and B. Van Houdt. Smcsolver and q-mam: tools for matrix-analytic methods. *SIGMETRICS Performance Evaluation Review*, 39(4):46, 2012.

[18] Alexander Birman and Yaakov Kogan. Asymptotic evaluation of closed queueing networks with many stations. *Communications in Statistics. Stochastic Models*, 8(3):543–563, 1992.

[19] A. Bobbio, A. Horváth, M. Scarpa, and M Telek. Acyclic discrete phase type distributions: properties and a parameter estimation algorithm. *Perform. Eval.*, 54(1):1–32, 2003.

[20] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains*. Wiley, 2006.

[21] A. B. Bondi and W. Whitt. The influence of service-time variability in a closed network of queues. *Perform. Eval.*, 6:219–234, 1986.

[22] L. Bortolussi and M. Tribastone. Fluid limits of queueing networks with batches. In *Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*, ICPE '12, pages 45–56, New York, NY, USA, 2012. ACM.

[23] L. Bright and P. G. Taylor. Calculating the equilibrium distribution in level dependent quasi-birth-and-death processes. *Stochastic Models*, 11(3):497–525, 1995.

[24] S. C. Bruell, G. Balbo, and P. V. Afshari. Mean value analysis of mixed, multiple class BCMP networks with load dependent service stations. *Performance Evaluation*, 4:241–260, 1984.

[25] Peter Buchholz, Jan Kriege, and Iryna Felko. *Input Modeling with Phase-Type Distributions and Markov Models: Theory and Applications*. Springer, Cham, 2014.

[26] J. P. Buzen. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM*, 16(9):527–531, 1973.

[27] G. Casale. CoMoM: Efficient class-oriented evaluation of multiclass performance models. *IEEE Trans. Software Engineering*, 35(2):162–177, 2009.

[28] G. Casale. Exact analysis of performance models by the method of moments. *Perform. Eval.*, 68(6):487–506, 2011.

[29] G. Casale. Accelerating performance inference over closed systems by asymptotic methods. In *Proc. of ACM SIGMETRICS*. ACM Press, 2017.

[30] G. Casale. Integrated Performance Evaluation of Extended Queueing Network Models with Line. In *2020 Winter Simulation Conference (WSC)*, pages 2377–2388. IEEE, dec 2020.

[31] G. Casale, V. De Nitto Personé, and E. Smirni. QRF: An optimization-based framework for evaluating complex stochastic networks. *ACM Transactions on Modeling and Computer Simulation*, 26(3):15:1–15:24, 2016.

[32] G. Casale and P. G. Harrison. AutoCAT: Automated product-form solution of stochastic models. In *Matrix-Analytic Methods in Stochastic Models*, volume 27 of *Springer Proceedings in Mathematics & Statistics*, pages 57–85. Springer, 2013.

[33] G. Casale, P.G. Harrison, and O.W. Hong. Facilitating load-dependent queueing analysis through factorization. *Perform. Eval.*, 2021.

[34] G. Casale, Richard R. Muntz, and Giuseppe Serazzi. Geometric bounds: A noniterative analysis technique for closed queueing networks. *IEEE Trans. Computers*, 57(6):780–794, 2008.

[35] G. Casale, J. F. Pérez, and W. Wang. QD-AMVA: Evaluating systems with queue-dependent service requirements. In *Proceedings of IFIP PERFORMANCE*, 2015.

[36] G. Casale and M. Tribastone. Fluid analysis of queueing in two-stage random environments. In *Proceedings of QEST 2011*, 2011.

[37] G. Casale, M. Tribastone, and P. G. Harrison. Blending randomness in closed queueing network models. *Perform. Eval.*, 82:15–38, 2014.

[38] G. Casale, E. Z. Zhang, and E. Smirni. KPC-Toolbox: Best recipes for automatic trace fitting using Markovian arrival processes. *Performance Evaluation*, 67(9):873–896, 2010.

[39] K. M. Chandy, U. Herzog, and L. Woo. Parametric analysis of queuing networks. *IBM Journal of Research and Development*, 19(1):36–42, 1975.

[40] K. M. Chandy and D. Neuse. Linearizer: a heuristic algorithm for queueing network models of computing systems. *Commun. ACM*, 25(2):126–134, 1982.

[41] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[42] G. L. Choudhury, K. K. Leung, and W. Whitt. Calculating normalization constants of closed queuing networks by numerically inverting their generating functions. *J. ACM*, 42(5):935–970, 1995.

[43] W.-M. Chow. Approximations for large scale closed queueing networks. *Performance Evaluation*, 3(1):1–12, 1983.

[44] W.-M. Chow. Approximations for large scale closed queueing networks. *Perform. Eval*, 3(1):1–12, 1983.

[45] C. Comte and J.-P. Dorsman. Pass-and-swap queues. *Queueing Systems*, 98(3):275–331, 2021.

[46] A. E. Conway. *Fast Approximate Solution of Queueing Networks with Multi-Server Chain-Dependent FCFS Queues*, pages 385–396. Springer US, Boston, MA, 1989.

[47] A. E. Conway and N. D. Georganas. RECAL - A new efficient algorithm for the exact analysis of multiple-chain closed queueing networks. *J. ACM*, 33(4):768–791, 1986.

[48] P. J. Courtois. *Decomposability: queueing and computer system applications*. Academic Press, New York, 1977.

[49] P.J. Courtois. Decomposability, instabilities, and saturation in multiprogramming systems. *Commun. ACM*, 18(7):371–377, 1975.

[50] J. Coury and P. G. Harrison. Asymptotic properties of queuing networks. *IEE Proceedings - Computers and Digital Techniques*, 144(5):247–254, 1997.

[51] P. Cremonesi, P. J. Schweitzer, and G. Serazzi. A unifying framework for the approximate solution of closed multiclass queuing networks. *IEEE Trans. Computers*, 51:1423–1434, 2002.

[52] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):143–152, 1990.

[53] M. H. A. Davis. Piecewise-deterministic Markov processes: a general class of non-diffusion stochastic models. *Journal of the Royal Statistical Society. Series B*, 46(3):353–388, 1984.

[54] E. de Souza e Silva, S. S. Lavenberg, and R. R. Muntz. A clustering approximation technique for queueing network models with a large number of chains. *IEEE Trans. Computers*, C-35(5):419–430, 1986.

[55] E. de Souza e Silva and R. R. Muntz. Approximate solutions for a class of non-product form queueing network models. *Performance Evaluation*, 7(3):221–242, August 1987.

[56] E. de Souza e Silva and R. R. Muntz. A note on the computational cost of the linearizer algorithm for queueing networks. *IEEE Trans. Computers*, 39(6):840–842, 1990.

[57] Peter J. Denning and Jeffrey P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, September 1978.

[58] R.-A. Dobre, Z. Niu, and G. Casale. Approximating fork-join systems via mixed model transformations. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 273–280, New York, NY, USA, 2024. Association for Computing Machinery.

[59] L. W. Dowdy, B. M. Carlson, A. T. Krantz, and S. K. Tripathi. Single-class bounds of multi-class queuing networks. *Journal of the ACM*, 39(1):188–213, 1992.

[60] L. W. Dowdy, D. L. Eager, K. D. Gordon, and L. V. Saxton. Throughput concavity and response time convexity. *Information Processing Letters*, 19(4):209–212, 1984.

[61] D. L. Eager. *Bounding Algorithms for Queueing Network Models of Computer Systems*. Tech. rept. csrg-156, University of Toronto, Toronto, Ontario, Canada, 1984.

[62] D. L. Eager and J. N. Lipscomb. The AMVA priority approximation. *Perform. Eval.*, 8(3):173–193, 1988.

[63] D. L. Eager and K. C. Sevcik. Performance bound hierarchies for queueing networks. *ACM Transactions on Computer Systems*, 1(2):99–115, 1983.

[64] D. L. Eager and K. C. Sevcik. Bound hierarchies for multiple-class queuing networks. *Journal of the ACM*, 33(1):179–206, 1986.

[65] M. Fidler and A. Rizk. A guide to the stochastic network calculus. *IEEE Communications Surveys and Tutorials*, 17(1):92–105, 2015.

[66] A. J. Field and P. G. Harrison. An approximate compositional approach to the analysis of fluid queue networks. *Perform. Eval.*, 64(9-12):1137–1152, October 2007.

[67] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi. Enhanced modeling and solution of layered queueing networks. *IEEE Trans. Software Engineering*, 35(2):148–161, 2009.

[68] G. Franks, P. Maly, C. M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz. *Layered Queueing Network Solver and Simulator User Manual*, 2012.

[69] G. Franks, P. Maly, M. Woodside, D. C. Petriu, Alex Hubbard, and Martin Mroz. *Layered Queueing Network Solver and Simulator User Manual*. Carleton University, January 2013.

[70] G. Franks and M. Woodside. Multiclass multiservers with deferred operations in layered queueing networks, with software system applications. In *Proceedings of the 12th IEEE MASCOTS*, 2004.

[71] R. G. Franks. *Performance Analysis of Distributed Server Systems*. PhD thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, December 1999.

[72] Anshul Gandhi, Mor Harchol-Balter, and Ivo Adan. Server farms with setup costs. *Performance Evaluation*, 67(11):1123–1138, 2010.

[73] N. Gast. Expected values estimated via mean-field approximation are 1/n-accurate. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):17:1–17:26, 2017.

[74] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. *Queueing Syst*, 83(3-4):293–328, 2016.

[75] N. Gast and B. Van Houdt. A refined mean field approximation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(2):33:1–33:28, 2017.

[76] Alexander I. Gerasimov. On normalizing constants in multiclass queueing networks. *Operations Research*, 43(4):704–711, 1995.

[77] Reinhard German. On synchronous approximations of non-markovian processes with dual bounds. In *Proc. of the 21st European Performance Engineering Workshop (EPEW)*, Catania, Italy, 2025.

[78] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[79] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[80] W. J. Gordon and G. F. Newell. Closed queuing systems with exponential servers. *Operations Research*, 15(2):254–265, 1967.

[81] V. Grassi, R. Mirandola, and A. Sabetta. From design to analysis models: a kernel language for performance and reliability analysis of component-based systems. In *Proc. 5th International Workshop on Software and Performance (WOSP)*, 2004.

[82] M. C. Guenther, A. Stefanek, and J. T. Bradley. Moment closures for performance models with highly non-linear rates. In *Computer Performance Engineering (EPEW/UKPEW 2012)*, volume 7587 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2013.

[83] V. Gupta and P. G. Harrison. Fluid level in a reservoir with an on-off source. *SIGMETRICS Perform. Eval. Rev.*, 36(2):128–130, August 2008.

[84] L. Gurvits. Hyperbolic polynomials approach to Van der Waerden/Schrijver-Valiant like conjectures: Sharper bounds, simpler proofs and algorithmic applications. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 417–426, 2006.

[85] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*, volume 14 of *Springer Series in Computational Mathematics*. Springer, 2nd edition, 1996.

[86] A. Harel, S. Namn, and J. Sturm. Simple bounds for closed queueing networks. *Queueing Systems*, 31(1-2):125–135, 1999.

[87] J. M. Harrison. *Brownian Motion and Stochastic Flow Systems*. Wiley, 1985.

[88] P. G. Harrison and T. T. Lee. A new recursive algorithm for computing generating functions in closed queueing networks. In *Proc. of IEEE MASCOTS Symposium*, pages 223–230. IEEE Press, 2004.

[89] P. Heidelberger and K. Trivedi. Queueing network models for parallel processing with asynchronous tasks. *IEEE Trans. Computers*, 100(11):1099–1109, 1982.

[90] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[91] G. Horton, V.G. Kulkarni, D.M. Nicol, and K.S. Trivedi. Fluid stochastic Petri nets: Theory, applications, and solution techniques. *European Journal of Operational research*, 105:184–201, 1998.

[92] G. Horváth. Measuring the distance between MAPs and some applications. In *Proc. of ASMTA 2015*, volume 9081 of *LNCS*, pages 95–109. Springer, 2015.

[93] G. Horváth and M. Telek. Sojourn times in fluid queues with independent and dependent input and output processes. *Performance Evaluation*, 79:160–181, 2014.

[94] G. Horváth and M. Telek. Butools 2: A rich toolbox for markovian performance evaluation. In *Proc. of VALUETOOLS*, pages 137–142, ICST, Brussels, Belgium, Belgium, 2017. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[95] C. H. Hsieh and S. Lam. Two classes of performance bounds for closed queueing networks. *Performance Evaluation*, 7(1):3–30, 1987.

[96] C. T. Hsieh and S. S. Lam. PAM - a noniterative approximate solution method for closed multichain queueing networks. *ACM SIGMETRICS Performance Evaluation Review*, 16(1):261–269, 1988.

[97] M. Huber and J. Law. Fast approximation of the permanent for very dense problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 681–689, 2008.

[98] J. R. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[99] A. Jensen. Markoff chains as an aid in the study of markoff processes. *Scandinavian Actuarial Journal*, 1953(sup1):87–91, 1953.

[100] K. Kant. MVA approximations for SJN scheduling. *Performance Evaluation*, 15(1):41–61, 1992.

[101] F. P. Kelly. *Reversibility and Stochastic Networks*. Cambridge University Press, New York, NY, USA, 1979.

[102] F. P. Kelly. Loss networks. *Annals of Applied Probability*, 1(3):319–378, 1991.

[103] T. Kerola. The composite bound method for computing throughput bounds in multiple class environments. *Performance Evaluation*, 6(1):1–9, 1986.

[104] S. Kijima and T. Matsui. Approximate/perfect samplers for closed Jackson networks. In *Proceedings of the Winter Simulation Conference*, pages 862–868, 2005.

[105] C. Knessl and C. Tier. Asymptotic expansions for large closed queueing networks with multiple job classes. *IEEE Trans. Computers*, 41(4):480–488, 1992.

[106] Y. M. Ko and J. Pender. Diffusion limits for the $(\mathrm{MAP}_t/\mathrm{Ph}_t/\infty)^n$ queueing network. *Operations Research Letters*, 45(3):248–253, 2017.

[107] H. Kobayashi. Application of the diffusion approximation to queueing networks I: equilibrium queue distributions. *J. ACM*, 21(2):316–328, 1974.

[108] J. R. Koury, D. F. McAllister, and W. J. Stewart. Iterative methods for computing stationary distributions of nearly completely decomposable Markov chains. *SIAM Journal on Algebraic Discrete Methods*, 5(2):164–186, 1984.

[109] D.D. Kouvatsos. Entropy maximisation and queueing network models. *Annals of Operations Research*, 48:63–126, 1994.

[110] H. Koziolek. Performance evaluation of component-based software systems: a survey. *Perform. Eval.*, 67:634–658, 2010.

[111] W. Krämer and M. Langenbach-Belz. Approximate formulae for general single server systems with single and batch arrivals. *Angewandte informatik*, 9, 1978.

[112] A. E. Krzesinski. Multiclass queueing networks with state-dependent routing. *Performance Evaluation*, 7(2):125–143, 1987.

[113] J. Kuck, T. Dao, H. Rezatofighi, A. Sabharwal, and S. Ermon. Approximating the permanent by sampling from adaptive partitions. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[114] V. G. Kulkarni. Fluid models for single buffer systems. In Jewgeni H. Dshalalow, editor, *Frontiers in queueing*, pages 321–338. CRC Press, Inc., Boca Raton, FL, USA, 1997.

[115] T. G. Kurtz. Solutions of ordinary differential equations as limits of pure jump Markov processes. *Journal of Applied Probability*, 7:49–58, 1970.

[116] T. G. Kurtz. *Approximation of Population Processes*. Society for Industrial and Applied Mathematics, 1981.

[117] G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling*. Society for Industrial and Applied Mathematics, 1999.

[118] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice-Hall, 1984.

[119] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: A framework for stochastic simulation in Java. In *Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. IEEE, 2002.

[120] C. Li, T. Altamimi, M. H. Zargari, G. Casale, and D. C. Petriu. Tulsa: A tool for transforming UML to layered queueing networks for performance analysis of data intensive applications. In *Quantitative Evaluation of Systems - 14th International Conference, QEST 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, pages 295–299, 2017.

[121] Z. Li and G. Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks. In *Proc. of ACM/SPEC ICPE*, pages 77–88, 2024.

[122] Z. Li and G. Casale. Matrix network analyzer: A new decomposition algorithm for phase-type queueing networks (work in progress paper). In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24 Companion, page 34–39, New York, NY, USA, 2024. Association for Computing Machinery.

[123] J. D. C. Little. A proof for the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[124] Q. Liu and D. A. Pierce. A note on Gauss-Hermite quadrature. *Biometrika*, 81(3):624–629, 1994.

[125] J. Lüthi and G. Haring. Mean value analysis for queueing network models with intervals as input parameters. *Performance Evaluation*, 32(3):185–215, 1998.

[126] S. Majumdar and C. M. Woodside. Robust bounds and throughput guarantees for closed multiclass queueing networks. *Performance Evaluation*, 32(2):101–136, 1998.

[127] Andrea Marin and Samuele Rota Bulò. A general algorithm to compute the steady-state solution of product-form cooperating Markov chains. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–10, London, UK, 2009. IEEE.

[128] Andrea Marin, Samuele Rota Bulò, and Simonetta Balsamo. A numerical algorithm for the decomposition of cooperating structured Markov processes. In *Proc. of the 20th IEEE MASCOTS*, pages 401–410. IEEE, 2012.

[129] KT Marshall. Some relationships between the distributions of waiting time, idle time and interoutput time in the gi/g/1 queue. *SIAM Journal on Applied Mathematics*, 16(2):324–327, 1968.

[130] J. McKenna and D. Mitra. Asymptotic expansions and integral representations of moments of queue lengths in closed markovian networks. *J. ACM*, 31(2):346–360, April 1984.

[131] Carl D. Meyer. Stochastic complementation, uncoupling markov chains, and the theory of nearly reducible systems. *SIAM Review*, 31(2):240–272, 1989.

[132] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *Proc. of ICATPN, LNCS 1639*, pages 6–25, 1999.

[133] A. S. Miner, G. Ciardo, and S. Donatelli. Using the exact state space of a Markov model to compute approximate stationary measures. In *Proc. of ACM SIGMETRICS*, pages 207–216, 2000.

[134] D. Mitra and J. McKenna. Asymptotic expansions for closed markovian networks with state-dependent service rates. *J. ACM*, 33(3):568–592, July 1986.

[135] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Trans. Computers*, 31(9):913–917, September 1982.

[136] Christoph Müller, Piotr Rygielski, Simon Spinner, and Samuel Kounev. Enabling fluid analysis for queueing petri nets via model transformation. *Electr. Notes Theor. Comput. Sci.*, 327:71–91, 2016.

[137] R. R. Muntz and J. W. Wong. Asymptotic properties of closed queueing network models. In *Proceedings of the 8th Princeton Conference on Information Sciences and Systems*, pages 348–352, Princeton, NJ, 1974. Department of Electrical Engineering and Computer Science, Princeton University.

[138] T. Murata. Petri nets: Properties, analisys and applications. *Proc. IEEE*, 77:541–579, 1989.

[139] D. Neuse and K. M. Chandy. SCAT: A heuristic algorithm for queueing network models of computing systems. *ACM SIGMETRICS Perform. Eval. Rev.*, 10(3):59–79, 1981.

[140] M. F. Neuts. *Matrix-geometric solutions in stochastic models: an algorithmic approach*. Johns Hopkins University Press, 1981.

[141] M. F. Neuts. *Structured Stochastic Matrices of M/G/1 Type and Their Applications*. Marcel Dekker, 1989.

[142] M. Nuyens and A. Wierman. The foreground-background queue: A survey. *Performance Evaluation*, 65(3-4):286–307, 2008.

[143] J. F. Pérez and G. Casale. Assessing SLA compliance from Palladio component models. In *Proceedings of the 2nd MICAS*, 2013.

[144] J. F. Pérez and G. Casale. Line: Evaluating software applications in unreliable environments. *IEEE Trans. Reliability*, 66(3):837–853, Sept 2017.

[145] Juan F. Pérez, Giuliano Casale, and Sergio Pacheco-Sanchez. Estimating computational requirements in multi-threaded applications. *IEEE Trans. Software Eng.*, 41(3):264–278, 2015.

[146] Tuan Phung-Duc, Hiroyuki Masuyama, Shoji Kasahara, and Yutaka Takahashi. A simple algorithm for the rate matrices of level-dependent qbd processes. In *Proc. of QTNA*, pages 46–52. ACM, 2010.

[147] J. G. Propp and D. B. Wilson. Exact sampling with coupled Markov chains and applications to statistical mechanics. *Random Structures and Algorithms*, 9(1-2):223–252, 1996.

[148] M. Reiser. Mean-value analysis and convolution method for queue-dependent servers in closed queueing networks. *Perform. Eval.*, 1:7–18, 1981.

[149] M. Reiser and S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, April 1980.

[150] A. Riska and E. Smirni. MAMSolver: A matrix analytic methods tool. In *Proc. of the 12th Int. Conf. on Modelling Techniques and Tools (TOOLS)*, volume 2324 of *LNCS*, pages 205–211, 2002.

[151] A. Riska and E. Smirni. ETAQA solutions for infinite Markov processes with repetitive structure. *INFORMS Journal on Computing*, 19(2):215–228, 2007.

[152] J. A. Rolia. *Predicting the Performance of Software Systems*. PhD thesis, University of Toronto, Toronto, Ontario, Canada, January 1992. Computer Systems Research Institute, Technical Report CSRI-260.

[153] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Trans. Software Engineering*, 21(8):689–700, August 1995.

[154] Keith W. Ross. *Multiservice Loss Models for Broadband Telecommunication Networks*. Springer, London, 1995.

[155] J. Ruuskanen, T. Berner, K.-E. Årzén, and A. Cervin. Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing. *Perform. Evaluation*, 151:102231, 2021.

[156] H. J. Ryser. *Combinatorial Mathematics*. Number 14 in Carus Mathematical Monographs. Mathematical Association of America, 1963.

[157] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[158] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[159] R. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, 1996.

[160] R. A. Sahner and K. S. Trivedi. Performance and reliability analysis using directed acyclic graphs. *IEEE Transactions on Software Engineering*, 13(10):1105–1114, 1987.

[161] C. H. Sauer. Computational algorithms for state-dependent queueing networks. *ACM Trans. Comput. Syst.*, 1(1):67–92, 1983.

[162] W. Scheinhardt. *Markov-modulated and feedback fuid queues*. PhD thesis, University of Twente, 1998.

[163] Rolf Schmidt. An approximate MVA algorithm for exponential, class-dependent multiple servers. *Performance Evaluation*, 29(4):245–254, 1997.

[164] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[165] P. J. Schweitzer. Approximate analysis of multiclass closed networks of queues. In *Proc. of the Int'l Conf. on Stoch. Control and Optim.*, pages 25–29, Amsterdam, 1979.

[166] P. J. Schweitzer, G. Serazzi, and M. Broglia. A queue-shift approximation technique for product-form queueing networks. In *Computer Performance Evaluation (Tools'98)*, volume 1469 of *Lecture Notes in Computer Science*, pages 267–279. Springer, 1998.

[167] K. Sevcik. Priority scheduling disciplines in queuing network models of computer systems. In *IFIP Congress*, 1977.

[168] M. Sheldon and G. Casale. Taussa: Simulating markovian queueing networks with tau leaping. *SIGMETRICS Perform. Eval. Rev.*, 49(4):70–75, jun 2022.

[169] Matthew Sheldon, Daphne Tuncer, and Giuliano Casale. TBI: Transient hierarchical modeling of large-scale vehicle sharing systems. *IEEE Transactions on Intelligent Transportation Systems*, 2025. In submission.

[170] R. Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *The Annals of Mathematical Statistics*, 35(2):876–879, 1964.

[171] Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating approaches to resource demand estimation. *Performance Evaluation*, 92:51–71, 2015.

[172] V. Srinivasan. Successively improving bounds on performance measures for single class product form queueing networks. *IEEE Transactions on Computers*, C-34(11):1076–1082, 1985.

[173] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[174] R. Suri and Y. Dallery. Load-dependent stations and bounds for closed queueing networks. *Performance Evaluation Review*, 14(1):203–212, 1986.

[175] R. Suri, S. K. Sahu, and M. Vernon. Approximate mean value analysis for closed queuing networks with multiple-server stations. In *Proc. of the Industrial Engineering Research Conference*, pages 1–6, 2007.

[176] H. Takagi. Queuing analysis of polling models. *ACM Computing Surveys*, 20(1):5–28, 1988.

[177] Y. Takahashi. A lumping method for numerical calculations of stationary distributions of Markov chains. Technical Report B-18, Department of Information Sciences, Tokyo Institute of Technology, Tokyo, Japan, June 1975.

[178] Y. C. Tay and R. Suri. Error bounds for performance prediction in queuing networks. *ACM Transactions on Computer Systems*, 3(4):227–254, 1985.

[179] M. Tribastone. A fluid model for layered queueing networks. *IEEE Trans. Software Engineering*, 39(6):744–756, June 2013.

[180] M. Tribastone, J. Ding, S. Gilmore, and J. Hillston. Fluid rewards for a stochastic process algebra. *IEEE Trans. Software Engineering*, 38(4):861–874, 2012.

[181] K. S. Trivedi and A. Bobbio. *Reliability and Availability Engineering: Modeling, Analysis, and Applications*. Cambridge University Press, 2017.

[182] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[183] P. O. Vontobel. The Bethe permanent of a nonnegative matrix. *IEEE Transactions on Information Theory*, 59(3):1866–1901, 2013.

[184] H. Wang and K. C. Sevcik. Experiments with improved approximate mean value analysis algorithms. *Perform. Eval.*, 39(1-4):189–206, 2000.

[185] W. Wang, G. Casale, and C. A. Sutton. A bayesian approach to parameter inference in queueing networks. *ACM Trans. Model. Comput. Simul.*, 27(1):2:1–2:26, 2016.

[186] Weikun Wang, Giuliano Casale, Ajay Kattepur, and Manoj K. Nambiar. Maximum likelihood estimation of closed queueing network demands from queue length data. In *Proc. of ACM/SPEC ICPE*, pages 3–14. ACM, 2016.

[187] W. Whitt. The queueing network analyzer. *Bell System Technical Journal*, 62(9):2779–2815, 1983.

[188] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with respect to unfairness in an M/GI/1. In *Proc. ACM SIGMETRICS*, pages 238–249, 2003.

[189] J. Zahorjan, D. L. Eager, and H. M. Sweillam. Accuracy, speed, and convergence of approximate mean value analysis. *Perform. Eval.*, 8(4):255–270, 1988.

[190] John Zahorjan, Kenneth C. Sevcik, Derek L. Eager, and Bruce Galler. Balanced job bound analysis of queueing networks. *Communications of the ACM*, 25(2):134–141, 1982.

[191] S. Zhou. A new approximation for multiserver waiting time, for layered queueing systems. M.A.Sc. thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, 2021.

[192] S. Zhou and M. Woodside. A multiserver approximation for cloud scaling analysis. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, ICPE '22, pages 129–136, New York, NY, USA, 2022. Association for Computing Machinery.

Index

achievable region, 22
AMVA, 55
API packages
    cache, 123
    da, 125
    dqsys, 125
    fes, 126
    fj, 126
    infer, 126
    lossn, 127
    lqn, 127
    lsn, 127
    lti, 126
    mam, 123
    mam.m3pp, 127
    mapqn, 130
    mc, 124
    measures, 132
    npfqn, 134
    pfqn, 127
    pfqn.lcfs, 136
    pfqn.ld, 134
    pfqn.mva, 134
    pfqn.nc, 135
    pfqn.sens, 138
    polling, 138
    qsys, 139
    sn, 140
    snc, 142
    spn, 142
    trace, 134
    wf, 143
Arrival rate (`ArvR`), 6
arrival theorem, 56
asymptotic bound analysis, 19
`AUTO` solver, 14, 114

`BA` solver, 18
balanced job bounds, 19
Bard-Schweitzer, 57
Bayesian model averaging, 98
BiCGSTAB, 31
blending method, 86
bound hierarchy, 20

`Cache` node, 119
`cache_mva`, 124
Che approximation, 59
CoMoM, 67
convolution algorithm, 67
convolutional bound hierarchy, 20
coupling from the past, 75
Courtois decomposition, 30
`CTMC` solver, 29
`ctmc_solve`, 125

`defaultOptions`, 111
direct method, 80
discrete-event simulation, 44
distributions
    `MAPt`, 39
    `Prior`, 98
`dtmc_solve`, 126

`ENV` solver, 86
Erlang fixed-point approximation, 71
Euler-Maruyama, 39

feature envelope, 114
feature set, 14

file formats
    JMVA, 103
    JSIM, 103
    JSON, 44
    LQNX, 106
fixed-point iteration, 57
`FLD` solver, 37
fork-join, 60
future-event list, 44

geometric bounds, 19
`getAvgTable`, 4
`getFeatureSet`, 111
`getInterval`, 100
`getLayers`, 122
`getProbSysMarg`, 73
`getStruct`, 119
GMRES, 30

importance sampling, 70
index spaces
    absolute index, 121
    call index, 121
    host index, 121
    index shift, 121
    node index, 119
    relative index, 121
    stateful node index, 119
    station index, 119
infinitesimal generator, 29
interlock correction, 94
`isstateful`, 119
`isstation`, 119

`jline.api`, 123
`JMT` solver, 103
JMVA format, 103
JSIM format, 103
JSON format, 44

Ko-Pender limits, 39

layer, 121
layer decomposition, 90
layered queueing network, 90
`LayeredNetwork` class, 106, 121
LayeredNetworkStruct
    static properties, 121
`LDES` solver, 44
`line_citations`, 113
`line_method_type`, 113
Linearizer, 57
`listValidMethods`, 111
`LN` solver, 90, 122
logistic expansion, 68
Looping, 20
loss network, 71
`LQNS` solver, 106
LQNX format, 106
`lqsim`, 106

`MAM` solver, 49
MAP fitting, 51
`MAPt` distribution, 39
Markov-modulated fluid queue, 38
Markov-modulated process, 87
Matrix Network Analyzer, 50
matrix permanent, 71
matrix-geometric solution, 49
mean-field limit, 37
method of layers, 90
`methodFeatureSet`, 111
MMAP decomposition, 49
model classes
    `LayeredNetwork`, 106, 121
model features
    fork-join, 60
    polling, 59
    random environment, 86
    state-dependent routing, 74
model marshalling, 103
`ModelAnalyzer`, 14
multi-valued decision diagram, 32
multiserver approximation, 107
`MVA` solver, 55

`NC` solver, 66

NetworkSolver, 111
NetworkStruct
  computed properties, 119
  index conversion, 120
  node parameters, 120
  static properties, 119
NetworkStruct, 4
next reaction method, 80
node types
  Cache, 119
  Queue, 119
  Sink, 119
  Source, 119
nodeparam, 120
nodeToStateful, 120
nodeToStation, 120
normalizing constant, 66

ODE integration, 37
options
  container, 106
  cutoff, 29
  iter_tol, 56
  level, 20
  method, 5
  multiserver, 107
  samples, 45
  seed, 82

Panacea, 69
parseOptions, 114
passage time, 39
performance bound hierarchy, 20
performance metrics
  Arrival rate, 6
  Queue length, 6
  Residence time, 6
  Response time, 6
  Throughput, 6
  Utilization, 6
pfqn_ca, 135
pfqn_comom, 135
pfqn_linearizer, 136
pfqn_mva, 137
Pollaczek-Khinchine formula, 59
polling, 59
population vector, 6
Prior distribution, 98
product form, 55

QBD, 49
  level-dependent, 51
QD-AMVA, 57
QNA, 59
qnsolver, 106
Quadratic Reduction Framework, 21
Queue length (QLen), 6
Queue node, 119

random environment, 86
random number streams, 47
RCAT, 50
RECAL, 103
recursion by generating functions, 68
refined mean-field, 38
refreshStruct, 119
Residence time (ResidT), 6
resolveMethod, 111
Response time (RespT), 6
Robust Queueing Theory, 59
runAnalyzer, 111
Ryser formula, 72

saddle point, 70
semi-Markov process, 87
series-parallel reduction, 91
Sink node, 119
solution methods
  AMVA, 55
  asymptotic bound analysis, 19
  balanced job bounds, 19
  Bard-Schweitzer, 57
  Bayesian model averaging, 98
  BiCGSTAB, 31
  blending method, 86
  Che approximation, 59

CoMoM, 67
convolution algorithm, 67
convolutional bound hierarchy, 20
coupling from the past, 75
Courtois decomposition, 30
direct method, 80
Erlang fixed-point approximation, 71
Euler-Maruyama, 39
geometric bounds, 19
GMRES, 30
importance sampling, 70
Ko-Pender limits, 39
Linearizer, 57
logistic expansion, 68
Looping, 20
Matrix Network Analyzer, 50
method of layers, 90
MMAP decomposition, 49
next reaction method, 80
Panacea, 69
performance bound hierarchy, 20
Pollaczek-Khinchine formula, 59
QD-AMVA, 57
QNA, 59
Quadratic Reduction Framework, 21
RCAT, 50
RECAL, 103
recursion by generating functions, 68
refined mean-field, 38
Robust Queueing Theory, 59
Ryser formula, 72
series-parallel reduction, 91
stochastic complementation, 30
stochastic network calculus, 23
stochastic simulation algorithm, 80
Takahashi method, 30
tau-leaping, 81
trajectory-based iteration, 37
uniformization, 31
solver extension
analyzer dispatch, 112
citation registry, 113
method-type registry, 111
new method, 110
new solver, 110
option whitelist, 111
solver contract, 111
solvers
`AUTO`, 14, 114
`BA`, 18
`CTMC`, 29
`ENV`, 86
`FLD`, 37
`JMT`, 103
`LDES`, 44
`LN`, 90, 122
`LQNS`, 106
`MAM`, 49
`MVA`, 55
`NC`, 66
`SSA`, 80
`UQ`, 98
`Source` node, 119
SRVN layering, 122
`SSA` solver, 80
state space
generation, 29
tractability, 15
state-dependent routing, 74
`statefulToNode`, 120
`stationToNode`, 120
`stationToStateful`, 120
stochastic complementation, 30
stochastic network calculus, 23
stochastic simulation algorithm, 80
`supports`, 111
surrogate delay, 92, 121

Takahashi method, 30
tau-leaping, 81
Throughput (`Tput`), 6
trajectory-based iteration, 37

uncertainty quantification, 98
uniformization, 31

UQ solver, 98
Utilization (Util), 6

warm-up removal, 81