# Enabling Fluid Analysis for Queueing Petri Nets via Model Transformation

Christoph Müller[1] Piotr Rygielski [2] Simon Spinner[2]
Samuel Kounev[2]

*Chair of Software Engineering*
*Institute of Computer Science, University of Würzburg*
*Am Hubland, 97074 Würzburg, Germany*

**Abstract**

Due to the growing size of modern IT systems, their performance analysis becomes an even more challenging task. Existing simulators are unable to analyze the behavior of large systems in a reasonable time, whereas analytical methods suffer from the state space explosion problem. Fluid analysis techniques can be used to approximate the solution of high-order Markov chain models enabling time efficient analysis of large performance models. In this paper, we describe a model-to-model transformation from queueing Petri nets (QPN) into layered queueing networks (LQN). Obtained LQN models can benefit from three existing solvers: LINE, LQNS, LQSIM. LINE internally utilize fluid limits approximation to speed up the solving process for large models. We present the incentives for developing the automated model-to-model transformation and present a systematic approach that we followed in its design. We demonstrate the transformations using representative examples. Finally, we evaluate and compare the performance predictions of existing analytical, simulation and fluid analysis solvers. We analyze solvers' limitations, solving time, and memory consumption.

*Keywords:* Queueing Petri Nets, Layered Queueing Networks, Model transformation, fluid analysis

## 1 Introduction

The complexity of today's IT systems is increasing due to the emergence of new computing paradigms, such as cloud computing, big data analytics, or cyber-physical systems. The computing resources, such as CPUs, cannot be scaled-up vertically effectively anymore. Instead, horizontal scaling of resources (replication) provides the required power and addresses the growing needs of the users. As the complexity of the systems usually grows with their size, the performance analysis becomes even more challenging.

In order to enable the performance analysis of such systems, efficient and accurate solution techniques for performance models are necessary. Simulation techniques generally require long simulation runs to achieve the required accuracy. On

[1] Email: christoph@raytracer.me

[2] Email: {piotr.rygielski, simon.spinner, samuel.kounev}@uni-wuerzburg.de

*This paper is electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* www.elsevier.nl/locate/entcs

the other hand, exact analytical models suffer from the state space explosion problem [34], severely limiting the size of models that can be analyzed in practice. Fluid analysis is an approximate solution technique for continuous-time Markov chains that works especially well for models with a large state space while reducing the computational effort significantly [32]. Thus, fluid analysis techniques promise a trade-off solution for the two extremes.

According to [4], fluid analysis techniques have been developed to avoid the state space explosion by approximating the state space with a set of time-varying real variables and describes their evolution by a set of differential equations. In contrast to the well-known approach of analyzing via continuous time Markov chains, Hillston [14] proposed an underlying mathematical representation based on a set of coupled ordinary differential equations. This allows efficient performance analysis of the systems with large numbers of replicated components and users. We provide a brief overview of fluid analysis in Section 2.1 and 3.1.

### *1.1 Motivation*

Queuing Petri Nets (QPN) [1] are a powerful and expressive performance modeling formalism which are a combination of classic Queueing Networks (QNs) [3] and Colored Generalized Stochastic Petri Nets (CGSPN) [8]. It has been shown, that even relatively small architecture-level models representing a data center infrastructure and the software (e.g., as shown in [22]) may result in hundreds of places, thousands of transitions and millions of tokens when transformed into QPNs. Unfortunately, existing analytical solution techniques cannot be applied to QPN models of this complexity. Only time-inefficient discrete-event simulation can be used in these cases.

In this work, we leverage layered queueing networks (LQNs) formalism and its solvers. LQNs can be solved using LQNS which is the standard solver for LQNs [11], LQSIM which is a discrete-event simulation, or LINE [25] that leverages fluid-limit approximation to accelerate the solving. We provide more background on LQN and QPN formalisms in Section 2.3.

We use the power of model-to-model transformations to transform existing QPN models into LQN models which can be later solved using LQNS, LQSIM, and LINE. We transform QPN models systematically enabling the users without QPN or LQN expertise to profit from the LQN representation and the features of LQN solvers that are unavailable to the QPN solvers (e.g., SimQPN [18]). Without the automated transformation, the ability to manually transform QPNs into LQNs would be limited to experts in both fields. Moreover, the manual transformation of big models would be time inefficient and error prone.

Finally, the third incentive is the variety of currently existing QPN models. There exist high-level models for which automated transformations to QPN have been developed. The examples are: Palladio Component Model (PCM) [2], Descartes Modeling Language (DML) [6], and Descartes Network Infrastructures (DNI) [27]. DML and DNI support transformation to QPN, but are currently not compatible with solvers that leverage fluid analysis. We elaborate more on the capabilities of existing transformations for PCM, DML, and DNI in Section 3.2.

The goal of this paper is to enable fluid analysis for currently existing QPN models. We provide a concept of model-to-model transformation that converts any valid QPN model into an equivalent LQN instance that can be solved using LINE, LQNS, or LQSIM. LINE solver internally leverages fluid analysis [25].

The main contribution of this work is the concept of automated model-to-model transformation that translates QPN models into LQNs. We characterize the transformation, its features, and limitations. Additionally, we present the rules of the transformation by demonstrating which QPN patterns are translated into which LQN constructs. We characterize the semantic gaps between the QPN and LQN formalisms. Moreover, we state which LQN models are not supported by LINE solver but can be solved with other existing tools (e.g., LQNS or LQSIM which does not support fluid analysis). Finally, based on two representative examples, we demonstrate the transformation in practice and evaluate the performance prediction capabilities, solving time, and memory consumption of SimQPN, LINE, LQNS, and LQSIM.

### *1.2 Organization*

This paper is organized as follows. In Section 2, we provide background of the QPN and LQN formalisms and describe the LINE solver and its specifics regarding the support for LQN models. Later, in Section 3, we analyze the existing works on fluid analysis in performance prediction and existing model-to-model transformations involving QPNs and LQNs. Section 4 is devoted to describe the concept of the contributed transformation, whereas in Section 5, we present two examples that demonstrate the transformation using a simple and a complex case. Then, we evaluate the models using four solvers and quantify the prediction accuracy and solving time of them. Finally, in Section 6, we conclude and propose directions for future work.

## 2 Background

### *2.1 Fluid Analysis*

In this paper, we focus on fluid analysis (a.k.a, fluid limits) which is a deterministic real-valued process which approximates the evolution of a given stochastic process in which all state variables are approximated by continuous variables [14].

Fluid analysis techniques have been developed to cope with the state-space explosion problem. According to Tribastone et al. [32], if the model is represented as a Markov chain, the performance metrics (e.g., utilization, throughput, response time) are modeled as real functions of the chain called reward models. The complexity of their analysis grows with the increasing order of the Markov chain making the analysis infeasible for large scale systems. Hilston [14] showed that high order[3] continuous-time Markov chains can be approximated ($X_n(t) \approx nx(t)$) by substituting the real-valued stochastic process $X_n(t)/n$ with $x(t)$, where $X_n(t)$ is continuous-time Markov chain of a system's parameter $n$, and $x(t)$ is an ordinary

[3] In a Markov chain of order $m$, the future state depends on the past $m$ states.

differential equation. The parameter $n$ describes the scale of a model (e.g., number of users or components). The approximation is better for higher values of $n$ [32]. This approximation has been applied for PEPA (Performance Evaluation Process Algebra) [14] stochastic algebra models which is an underlying internal mechanism of LINE.

### 2.2 Short introduction to QPN and LQN

Queueing Petri Nets (QPNs) [1] are a combination of classic Queueing Networks (QNs) [3] and Colored Generalized Stochastic Petri Nets (CGSPN) [8]. While CGSPNs are a powerful formalism to describe the synchronization and timing behavior of software programs, they lack the expressiveness to easily describe the scheduling of jobs at hardware resources. In addition to ordinary places and transitions known in CGSPNs, QPNs therefore introduce queueing places consisting of a queue and a depository. The queues correspond to those in a traditional QN, including a scheduling strategy and a service time distribution. Incoming tokens are first served in the queue and then put into the depository where they become available to outgoing transitions. Using QPNs, it is possible to model both software and hardware contention of software systems in a single model [16]. For solving QPNs, we use SimQPN discrete-event simulator [17]. The QPN graphical notation is explained in Figure 1.

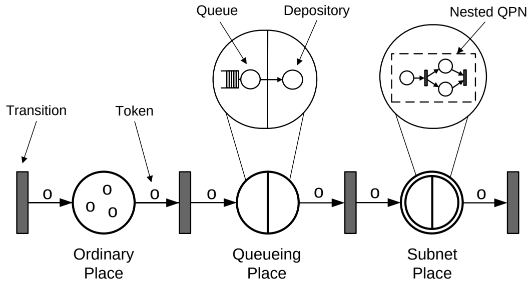

Fig. 1. Notation used in QPN diagrams.

Layered Queuing Networks (LQN) [10] are performance models that are an extension of regular Queuing Networks (QN). Compared to ordinary QNs, LQNs introduce the concept of layers, software servers, and they allow the modeling of simultaneous resource possession. LQNs are usually used to model software and hardware contention in a uniform way, as well as scheduling disciplines, simultaneous resource possession, synchronization, and blocking [36]. LQNs have been developed as a domain-specific language (DSL) covering a wide range of computer systems with a special focus on software and hardware systems. In contrast to that, QPNs are general-purpose models and are not bound to a given domain.

### 2.3 Semantic Gaps between LQN and QPN

Woodside et al. [36] claim that “LQNs have a great advantage over the competing models (Petri nets, Markov chains, timed process algebras) that they scale up to large systems with dozens or hundreds of cooperating processes.” Achieving such speed-ups in the solving is usually connected with abstracting selected data or limiting the modeling capabilities. In this Section, we analyze the limitations and differences with respect to QPNs.

Heimburger analyzed in [13] the differences between the solvers for QPNs (SimQPN [17]) and LQNs (LQNS [11]) in the context of the performance prediction of Java EE based software. We extend the comparison to the general level of the formalism and briefly summarize the key differences in Table 1.

Table 1
Comparison of the selected key differences of QPN and LQN formalism.

| Feature | QPN | LQN |
|---|---|---|
| Unit of flow | Colored tokens | Calls |
| Workload | Open and closed | Open and closed |
| Hierarchy | Yes, subnets, can be flattened | Yes, layers, cannot be flattened |
| Direction of flow | Any place with any transition | An activity to an entry, higher layer to lower layer only |
| Support of loops | Yes, any type (including infinite), loop iterations can be modeled probabilistically or deterministically | Yes, most of deterministically modeled loops (for exceptions see Section 4.3.1), number of loop iterations must be known (no infinite loops) |
| Starting point | No explicit starting place or transition. Transitions that fire first can be calculated | Top layer |

In QPNs, the colored tokens are the elements modeling the behavior—they are deposited in places and are moved from place to place by firing the transitions. In LQNs, this function is realized by calls denoted normally as arrows pointing to an entry. Both formalisms support modeling of open and closed workflows, however LQNs can be claimed to provide less support for closed workloads due to the limitations concerning loops spanning multiple layers. Layers are used to represent the hierarchy in LQNs, whereas in QPNs, nets can be nested using subnet places. QPN tokens can be moved from a place to another place when a transition fires. A transition can connect any two places at a given level in the hierarchy. The tokens, however, can be forwarded (via input and output places of subnets) to any place or transition disregarding the level in the hierarchy.

In contrast to QPNs, the LQN calls can connect only the layers that are non-higher than the layer from which the call originates. This limits the direction of the calls and narrows the modeling capabilities. The hierarchy of LQN layers cannot be flattened, whereas QPN does. Another difference is the way the loops are modeled. The LQN formalism allows to explicitly model simple loops where the loop iterations need to be specified by a constant, finite value. QPNs do no support loops directly, however loops can be built easily using few places, tokens, and transitions. Loops built in this way can iterate over a defined number of times (also infinite) or the number of iterations can be specified using probability distribution. Finally, QPNs do not have a predefined single starting point, whereas LQNs have a so-called top layer where the execution starts. In Section 4, we describe transformation rules that transform QPN models into their LQN equivalents.

### *2.4 Solvers for QPNs and LQNs and their Limitations*

In this paper, we analyze four solvers: SimQPN [17] for QPNs, LINE [25], LQNS [11], and LQSIM for LQNs. We briefly characterize the main known limitation of the solvers in their current versions.

SimQPN is a tool for steady-state analysis of QPNs. It is based on discrete-event simulation of a QPN and can yield throughput, utilization and response time statistics as a result (including confidence interval and histograms). Its capabilities are limited by the amount of free memory to a simulation of few millions ($\times 10^6$) of tokens (tokes can be created and destroyed during the analysis) on a commodity hardware.

LINE solver leverages the benefits of fluid analysis techniques for solving the LQNs. Currently, its coverage of LQNs is still limited, e.g., it does not support the $<$*and*$>$ node in the activity graphs, limiting the set of models that can be solved efficiently. While support for this functionality is planned, no concrete release date is available yet. According to the developers of LINE, the $<$*or*$>$ node is supported.

LQNS (analytical) and LQSIM (simulation) are two state-of-the-art solvers for LQNs. The LQNS solver implements several analytical solving techniques such as mean value analysis (MVA) and combines the advantages of other existing solvers, namely SRVN [36] and the Method of Layers (MOL). According to [11], LQNS and LQSIM do not support recursive calls (a task calling its own entries) and provide only limited support of replication on subsystems (details on the limitations were explained in [24]). LQNS cannot handle activity graphs whose fork is located in one task and join in another. Moreover, LQNS has troubles in solving models with exclusively external arrival flows.

The analysis of PCM models using QPNs and LQNs has been evaluated by Brosig et al. in [7]. Compared to LQNS, SimQPN was evaluated to provide full support of response time distributions, flexible parameter characterizations, and blocking behavior. On the other hand, the analyzed LQN models were more compact and the solving using LQNS was faster than the respective QPN models solved in SimQPN.

## 3 Related Work

We divide the related work into two domains: performance modeling and model-based software design (i.e., model-to-model transformations). First, we analyze the applications of fluid analysis in solving of performance models, whereas later, we briefly describe the applications of model-to-model transformations in performance analysis.

### *3.1 Application of Fluid Analysis in Performance Models*

Acknowledging the works treating about fluid queueing [23] , and fluid stochastic Petri nets [33] , we focus on the fluid analysis as defined in Section 2.1 and work [14]. Fluid limits for approximating Markov chain models were first introduced by Kurtz in 1971 in work [20]. Since then, the fluid limits were used for approximation in performance models consisting of high order Markov chains.

Fluid limits were applied to performance modeling by extending stochastic process algebra PEPA [14]. The authors of [25] leveraged the fluid limits by enabling LQNs to be solved internally using PEPA. This connection (*fluid-limits approximation* → *PEPA* → *LQN*) allowed solving the software and hardware performance models (which are the domain of LQNs) using ordinary differential equations as approximation for the analysis of the underlying Markov chains.

There are number of applications of LQNs and stochastic algebras for performance predictions, for example [31,37,7,19]. All LQN performance models can benefit from the fluid limit approximation as long as the LINE solver [25] can be applied (see LINE limitations in Section 2.4). To the best of our knowledge, LINE is the only LQN solver that leverages fluid analysis techniques so far. Further, we analyze other performance models that are transformable to LQNs and QPNs, so that they can benefit from the transformation contributed in this paper.

### *3.2 Existing Transformations of Performance Models*

A meta-model describes the allowed elements in a model instance as well as the relationships between them. Model transformations are used to automatically transform between models of different meta-models. For performance modeling and analysis, transformations have mainly been used to translate from a high-level architecture-level performance model into a lower-level prediction model. Koziolek and Reussner [19] describe a transformation from the Palladio Component Model (PCM)—which is a meta-model supporting quality-of-service analyses of software architectures at design-time—into Layered Queueing Networks (LQNs) for their analytical solution. Meier et al. [22] describe a transformation from PCM into Queueing Petri Nets (QPNs) and show that it has a higher accuracy and better coverage of PCM elements than the transformation into LQNs.

Another, descriptive meta-model for performance modeling is Descartes Modeling Language (DML) [5,15], which is aimed at online performance and resource management scenarios. Brosig [5] implements different transformations from DML into Queueing Networks (QNs), LQNs and QPNs and proposes an algorithm to automatically decide which transformation shall be used depending on the required prediction accuracy and speed.

Descartes Network Infrastructures (DNI) [27] fills the gaps left by DML and PCM in the area of data center networks. DNI models can be transformed into OMNeT++ Simulation and two QPNs—with finer and coarser level of details [26,27,28]. Even small DNI models may result in large QPNs so the effective and timely performance analysis is difficult without abstracting some data in the DNI's transformations.

In Figure 2 we depict some existing models, solvers, and model transformations.

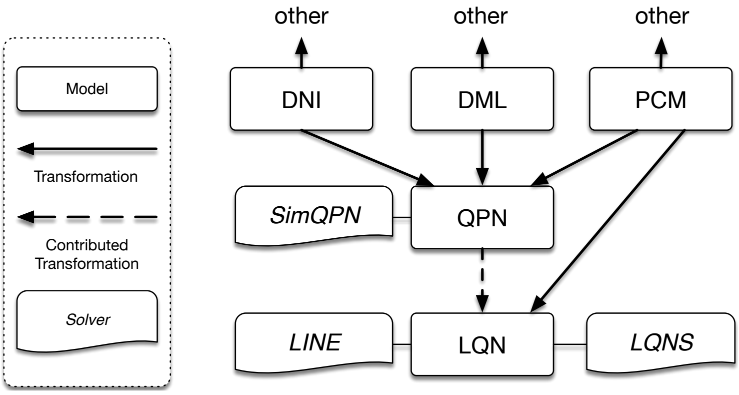

Fig. 2. Selected domain-specific and general purpose models, their available transformations and solvers.

The three considered architecture level models (PCM, DML, DNI) support solving with SimQPN using corresponding QPN model transformations. Only PCM supports wider variety of solvers, including LINE and LQNS. After transforming QPN models into LQN, we expect obtaining more compact models than the QPN equivalent and faster solving time thanks to the LQN analytical solvers (similar phenomena were observed by Brosig et al. in [7]).

## 4 Transformation Concept

In this section, we present the systematical approach that we followed in designing a transformation from QPN into LQN models in order to enable the fluid analysis of the former. A transformation consists of rules that are executed for each matching element of the source model (QPN) and that produces respective elements in the destination model (LQN).

In Section 4.1, we describe our overall approach for the QPN-to-LQN model transformation. In Section 4.2, we describe the individual transformation rules for mapping QPN elements to LQN ones taking into consideration the context in which they are used. We describe the limitations of the transformation in Section 4.3.

### *4.1 Approach*

Our approach is based on a model-to-model transformation mapping QPN models to equivalent LQN ones, so that existing fluid analysis solvers for LQNs can be used to also analyze QPNs. Such a transformation requires us to define a set of rules translating QPN elements into equivalent LQN ones. In the simplest case, transformation rules are context-free, injective functions mapping single QPN elements to corresponding LQN ones. However, when comparing the two formalisms, one can quickly see that this is not the case for our QPN-to-LQN transformation: certain behaviors (e.g., loops, forks, etc.) are explicit model elements in LQNs, while the same behavior is modeled in QPNs using a combination of places and transitions. In order to identify such combinations of places and transitions (in the following we call this a *pattern*), transformation rules also need to consider the context of a model element. An example of such context information may be the neighboring places and transitions or a topology of the QPN. As a result, there may be several, context-sensitive transformation rules that apply to the same model element in a QPN.

To determine which of a set of context-sensitive transformation rules to use for a certain model element, we need to analyze the graph structure of a QPN first. The transformation searches for certain patterns (e.g., loops, forks, joins, etc.) in the QPN model. In general, graph pattern matching is an NP-complete problem [12], but many efficient pattern matching algorithms exist (e.g., [9]) assuming that any colored Petri net can be unfolded into a single-colored one [21].

The LQN formalism requires us to explicitly model the starting point of requests as top layers. In QPNs, we need to determine these starting points first, because a net can have the form of an arbitrary graph. In order to determine the starting places, the reachability of places withing the QPN needs to be calculated and open or closed workload places can be identified (e.g., using the approach described in [35]). In case of a closed workload, the cycle around the complete net is removed and included in the LQN as a special top layer with a user population. The starting places are also the places from which the search for the other patterns begins.

### *4.2 Transformation Rules*

Table 2 gives an overview of our transformation rules. The rules are described in detail and accompanied with examples in Sections 4.2.1– 4.2.7.

Table 2
Key rules used in the QPN-to-LQN transformation.

| QPN element/pattern | LQN representation |
|---|---|
| Queues | Processors |
| Queueing places | Task with entry and assigned processor |
| Ordinary places | Depends on context. See Section 4.2.3 |
| Token colors | Individual entries for each color in the respective task |
| Modes of transitions | Activity Graphs for every input color that resemble the mode wiring (see Fig. 5) |
| Fork and join pattern | Fork and join nodes in activity graphs |
| Loop pattern | Loop notation (see Fig. 10) |
| Critical sections | Critical sections are created by a layer that marks the entrance to the section, has limited resources and uses a processor with a FCFS scheduling strategy |

#### *4.2.1 Queues and Queuing Places*

In QPNs, we distinguish between queueing places and queues. A queueing place consists of a queue and a depository. The queue may be shared between different queueing places. Queues are used to describe scheduling behavior in QPNs (e.g., at hardware resources). In LQNs, the same scheduling behavior can be described using processors. The transformation directly maps queues to processors. The associated queueing places are mapped to tasks in the LQN that use the corresponding processor. In case of shared queues, each queuing place that references the queue will be mapped to separate tasks using the same underlying processor. Figure 3 illustrates the mapping for the different cases.

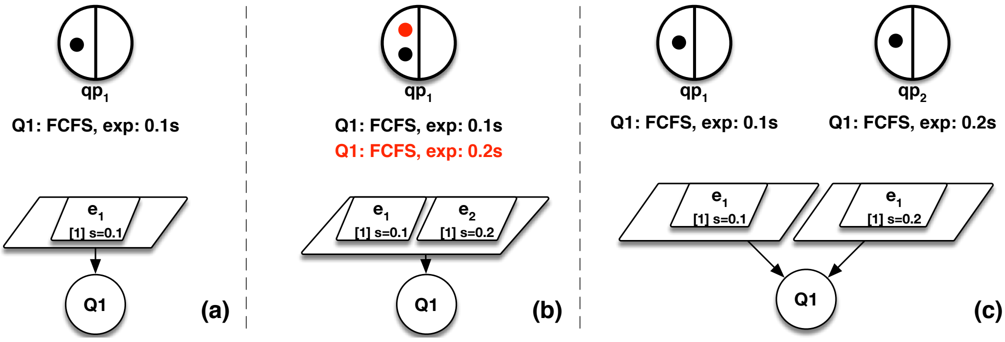

Fig. 3. Transformation of queueing places with: (a) single place, queue, and color; (b) single place, queue, and two colors; (c) two places, single queue and single color. Processing times are modeled with the exponential distribution with a given mean value in seconds.

### *4.2.2 Colors in Places*

Tokens in QPN may represent a single request, a resource (e.g., database connection in the pool), or a user. Each token has an associated color. Colors are usually used to model the routing of requests (different colors are traversing different path) or to represent various classes of requests (e.g., separate colors for read and write requests). While colors help to reduce modeling efforts, they do not increase the modeling power of QPNs. Using replication of parts of the net, every colored petri net can be transformed into non-colored one without loss of information [21].

The calls in a LQN are identical and cannot be distinguished by different types (or colors). In order to distinguish different types of calls to a task in LQN, we map each color to a separate entry. An example is presented in Figure 3b. In case of queueing places, the entries are parameterized with the service time specified in the QPN. In case of ordinary places, the service time is always set to zero.

### *4.2.3 Ordinary Places*

Ordinary places play a specific role in QPNs. They accumulate tokens but have limited influence on the time aspect of the network. We transform ordinary places based on the context in which they appear. We distinguish the following cases.

First, an ordinary place is a part of a pattern, for example a critical section and represents the limited resources (see *pool* place in Fig. 11). This case is covered by the critical section pattern described in Section 4.2.7.

Second, an ordinary place can be reduced if it does not influence the execution (e.g., it was used only for the convenience of the modeler). An ordinary place can be reduced—i.e., the neighboring transitions can be merged—only if the place is the only successor of the preceding transition and the only predecessor of the succeeding transition. An example is depicted in Figure 4a.

Third, an ordinary place can be used also as a synchronization point. This happens when a succeeding transition consumes multiple tokens and the tokens are held in the ordinary place until the required amount is deposited. According to LQNS documentation [11], LQN supports this case using the *calls-mean* parameter that can be specified as a real variable. An ordinary place followed by a transition that consumes $n$ and produces $m$ tokens will result in *calls-mean*= $\frac{m}{n}$ in LQN. An example is depicted in Figure 4b.

Finally, an ordinary place (the same applies to a queueing place) can precede a branch where the deposited token is consumed by one of the succeeding transitions. We depict it in Figure 4c, where the token in place $p_1$ has the equal probability $= 0.5$ to be consumed by transition $t_1$ or $t_2$. The probabilities can be calculated based on the priorities of the transition (by default all transitions have equal priority). The QPN transition $t_1$ is transformed into LQN's activity $a_1$ and the task directly connected to it (here $t_1$). The contents of the LQN tasks $t_1$ and $t_2$ depends on the successors of the QPN's transitions $t_1$ and $t_2$ which are abstracted in the Figure.

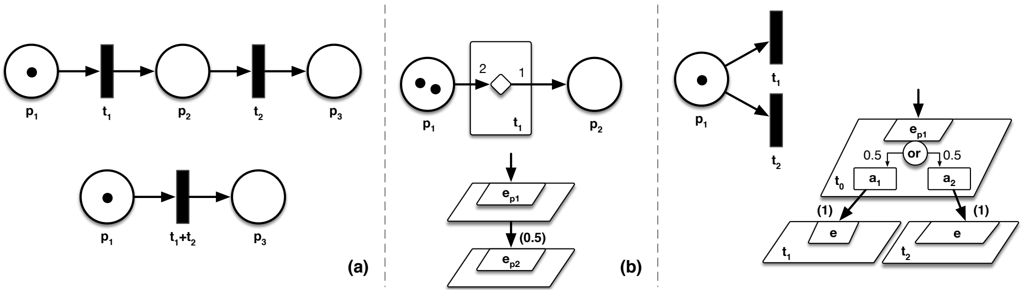

Fig. 4. Transformation of ordinary places depending on context: (a) redundant place reduction; (b) reduction of tokens; (c) branch in workflow.

### 4.2.4 Transitions and Modes

In QPNs, transitions consume tokens from incoming places and produce new tokens in outgoing places. Transitions can fire in different modes (also known as colors), to model different dynamic behavior. The incidence function defines the number and color of tokens consumed and produced by a firing mode. Multiple incoming places connected to the same mode are a synchronization point or a join (for a single mode). Multiple outgoing places from the same mode represent a fork. In LQN, the transitions are mapped to activity graphs. Fork and joins are represented by <*and*> nodes in the activity graph. Transitions containing multiple modes can be theoretically decomposed into multiple transitions each with a single mode. As a result, they can be treated as independent calls to the same entry of a task. Figure 5 depicts the possible transition configurations.

### 4.2.5 Fork and Join Pattern

The fork and join pattern (presented in Fig. 6) in QPNs is built by defining a mode in a transition that consumes a token and forwards tokens to multiple succeeding places. In LQNs, forks are modeled with activity graphs. The <*and*> nodes are used to execute calls in parallel and to join (synchronize) them after they are finished. We depict a simple fork-join pattern in QPN in Figure 6 and the transformed LQN model in Figure 7.

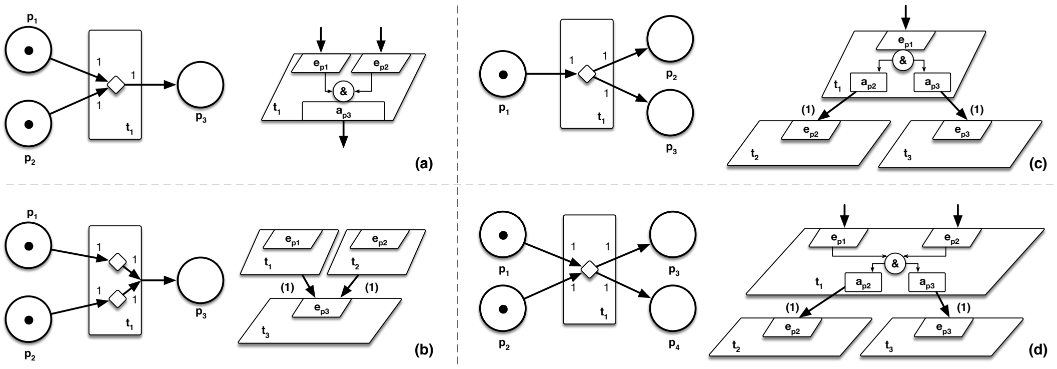

Fig. 5. Transformation of transitions with: (a) join-and; (b) join-or; (c) fork; (d) join-fork. The QPN transition (denoted $t_1$) contain modes (diamonds) that consume and produce a given number of tokens when firing. LQNs representation is simplified (no processors) for brevity.

Finding the start and end of forking process is challenging. While the start (the fork) is marked by a mode taking a token and forwarding it to multiple places, the matching end (the join) must be found by processing the graph. Since colors can change on the way through the graph it is non-trivial how to match a fork with the respective join.

To address this problem, we utilize the following possibilities. First, we try to fit the fork-join pattern in a single LQN's task, so that more solvers can be used to solve such model (see solvers limitations in Section 2.4). The analysis of non-trivial fork-join patterns in QPN (e.g., with colors changing between fork and join) is conducted using algorithms for graph analysis (e.g., [29]). Second, we may skip processing the QPN topology to find the matching transitions and ignore the transformation rule. In this way, the fork and join pair may be separated and placed on different tasks. Although this limits the compatible set of solvers (LQNS does not support separated fork-join), the output of the transformation is a valid LQN model and can be solved by the solvers.

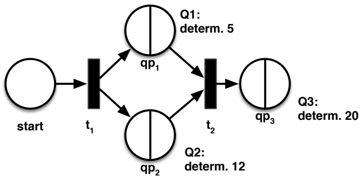

Fig. 6. Exemplary QPN containing the fork and join pattern.

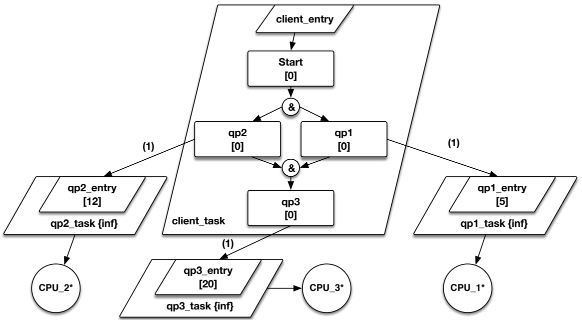

Fig. 7. Exemplary LQN containing the fork and join representation of the QPN shown in Fig. 6. Transitions $t_1$ and $t_2$ are represented here as the fork and join elements (&).

### *4.2.6 Loop Pattern*

The QPN formalism does not support modeling of loops directly but a loop can be modeled indirectly. Examples of loops modeled in QPN are presented in Fig. 8 and 9. The loop presented in Fig. 8 iterates based on the probability defined in the incidence function of the *Loop-Exit* transition. The expected number of iterations needs to be calculated in the transformation, as LQN requires exact number of iterations to be specified. In Fig. 9, the number of iterations is defined deterministically by the number of tokens produced by the *1-to-num-loop-iter* transition. LQN supports loops directly, so once the loop pattern is recognized correctly and the number of iterations is calculated, the transformation is trivial. The graphical representation of a loop in LQN is shown in Fig. 10.

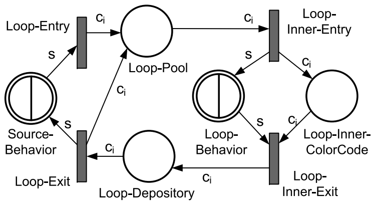

Fig. 8. Example of a QPN loop representation with probabilistically modeled number of iterations. Excerpted from [7].

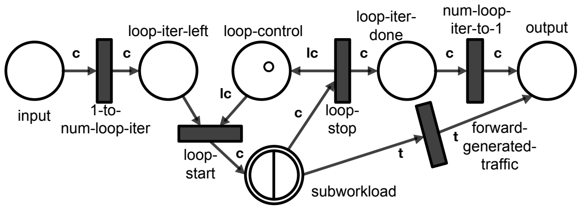

Fig. 9. Example of a QPN loop representation with deterministically modeled number of iterations. Excerpted from [26].

The QPN representations of loops are treated as patterns that need to be discovered by the transformation (or a transformation pre-processing library) in order to be transformed. In case of an unsupported loop pattern (there may exist other patterns than the two presented in Fig. 8 and 9), the transformation of a loop will be covered by the remaining transformation rules, however, the compact notation of LQN loop (as in Fig. 10) will not be used.

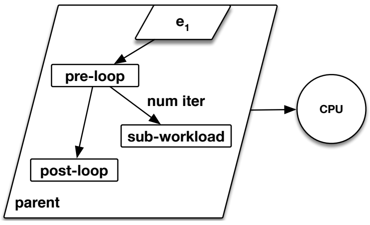

Fig. 10. LQN loop representation with deterministically modeled number of iterations.

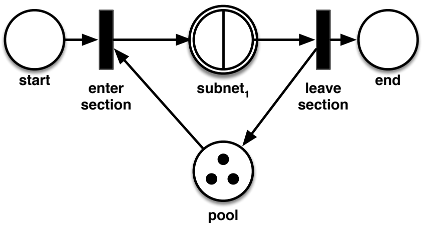

Fig. 11. Example QPN containing a critical section. The pool contains, so maximally three tokens can enter the subnet.

### *4.2.7 Critical Section Pattern*

A critical section is a region which can simultaneously handle only limited number of objects. Both LQN and QPN can model critical sections. Figure 11 shows a critical section in QPN. It is modeled with the *enter section* transition that consumes a token from the *start* and second from the *pool* place. The amount of initial tokens in the pool defines the number of tokens that enter the section at the same time. At the end, the *leave section* transition passes the token further to the *end* place and at the same time deposits another token back into the pool, so that the next token from *start* can enter the section.

LQNs represents a critical section with a layer that contains a defined number of FCFS queues. The number of FCFS quques in LQN corresponds to the QPN's pool tokens that limit the maximum number of tokens in the critical section. Every task in every queue will execute a synchronous call to perform the work in the critical section. Only when this call finishes, the next element will be dequeued and processed. Graphically, we depict LQN critical section in Figure. 12. The size of the pool is denoted with the quantity of the task *critical_section[3]*.

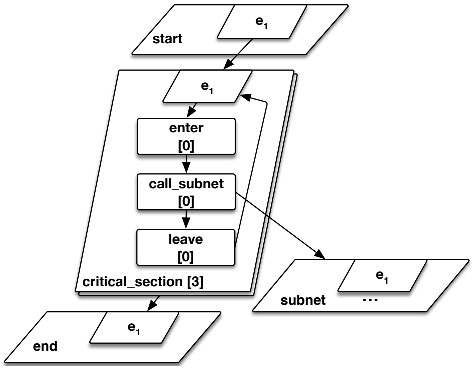

Fig. 12. LQN representation of the critical section corresponding to the QPN in Figure 11.

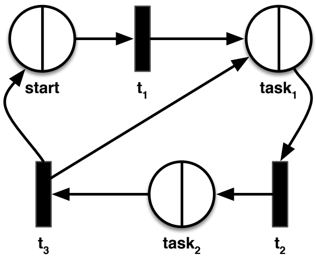

Fig. 13. Example of QPN containing a second internal loop.

### 4.3 Limitations of the Transformation

In this section we describe the limitations of the *QPN-to-LQN* transformation. This section covers general limitations of the LQN formalism and do not focus on solver-specific limitations (the limitations of LQN and QPN solvers are presented in Section 2.4). The most challenging parts of the transformation are: loops where a higher layer in LQN needs to be called, and the problem of finding the top layers (also called reference layers). We address both problems in this Section.

#### 4.3.1 Support for Specific Loops

Currently LQN supports only a *loop* node which executes a defined number of loop iterations assuming that the number of loop iterations is known beforehand. Unfortunately, it is impossible to build a LQN model of a QPN loop with unknown number of iterations, for example, as seen in Figure 13. This limitation comes from the lack of support of LQN to call layers that lie higher in the hierarchy. Solvers like LINE will run into recursion problems (exceeding the maximum depth). The general loop that models the closed workload of a complete LQN model is a special case and its number of iterations can be infinite or unknown.

#### 4.3.2 Finding the Top Layers

LQN use special layers (called reference or top layers) to start the workload cycle. The task in the top layer will be executed periodically according to the think time parameter. QPN does not necessarily have obvious starting places. Finding the transitions that fire first or model the think time of the closed workload may be also challenging. In order to address this limitation, we analyze the input QPN and estimate which transitions will fire first. We aim to find the transition that models the beginning of a closed workload. Unfortunately, QPN allows to represent a system in many ways and it is not guaranteed that the transitions found are responsible for representing the think time of the closed workload loops. An approach to this problem was tackled by Walter et al. in [35]. For each found transition, we create a top layer that is treated specially in LQN. We construct LQN tasks that succeed the top layers by traversing the next QPN elements starting with the QPN's successor places of the first-firing transitions.

## 5 Validation [4]

### 5.1 Example #1: Simple QPN Model

We validate the transformation based on two examples. First is a simple QPN model with three queueing places as depicted in Fig. 14. Each queueing place has a separate queue with deterministic processing time. The execution is looped to represent a closed workload with no think time.

We transformed it into LQN that is graphically represented in Fig. 15. Transition $t_1$ was recognized as firing first based on the initial marking of the *start* place. Transition $t_4$ is not represented in LQN as it serves only to model the closed workload.

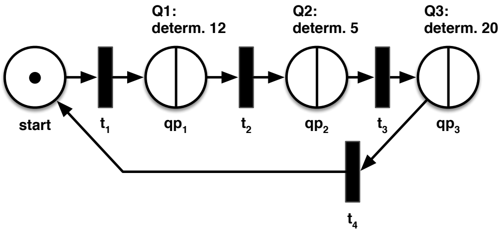

Fig. 14. Example #1 QPN representation.

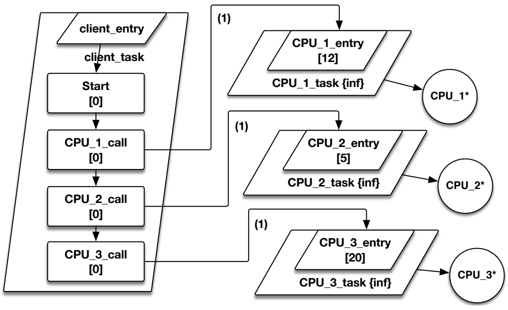

Fig. 15. Example #1 LQN representation.

The example was solved using four solvers: SimQPN for QPN, and LQNS, LQSIM, LINE for LQN. In this experiment, we want to show that the transformation behavior is correct for simple cases. We examine the utilization of queueing places/processors and throughput. The results are presented in Table 3.

Table 3
Example 1

| Utilization | $CPU_1$ | $CPU_2$ | $CPU_3$ |
|---|---|---|---|
| SimQPN | 0.324 | 0.135 | 0.541 |
| LINE | 0.32432 | 0.13513 | 0.54054 |
| LQNS | 0.32432 | 0.13513 | 0.54054 |
| LQSIM | 0.32585 | 0.13768 | 0.53646 |

| Throughput | $CPU_1$ | $CPU_2$ | $CPU_3$ |
|---|---|---|---|
| SimQPN | 0.0270 | 0.0270 | 0.0270 |
| LINE | 0.027027 | 0.027027 | 0.027027 |
| LQNS | 0.027027 | 0.027027 | 0.027027 |
| LQSIM | 0.02738 | 0.02791 | 0.0277 |

The prediction of utilization and throughput was almost identical for all examined solvers. Taking the SimQPN's prediction as a baseline, LQSIM solved the model with the highest error mispredicting the utilization by maximally 3%. The results demonstrate that the transformation is correct for the simple case.

We expected higher inaccuracy for LINE because the solving using fluid limits approximation is expected to work better for bigger models and provide higher errors for small. This issue seems to have been addressed by the authors of LINE

[4] All QPN and LQN models used in this paper are available online under the url: http://go.uni-wuerzburg.de/aux

as the results for small models are also good. We investigate a more complex model in the second example.

### *5.2 Example #2: SPECjAppServer2001*

The system represented in the second example is based on a Java Enterprise Edition (Java EE) server application benchmark (SPECjAppServer2001). The application is modeled after a business consisting of four domains: customer domain (customer orders and interactions), manufacturing domain ("just in time" manufacturing operations), supplier domain (interactions with suppliers) and corporate domain (customer, product and supplier domain). The workload is claimed to be big and complex enough to represent a real-world enterprise system [30]. In our scenario, the model is focused on the customer domain including four transaction types: NewOrder, ChangeOrder, OrderStatus and CustomerStatus. The system is deployed on two separate machines, one hosting the application logic and the other running a relational database. Besides the physical resources of the two machines (CPU and disk subsystem of the database), the model contains also logical resources, such as, the thread-pool of the application server, the connection and the process pool of the database server. A complete description of the model and its validation on a real system can be found in our previous work [16].

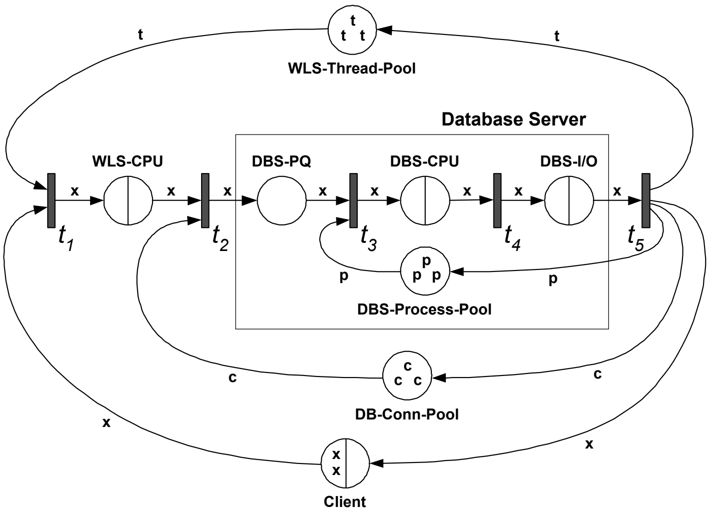

Fig. 16. QPN representation of example #2. *WLS* stands for WebLogic Server, and *DBS* for a database server. Excerpted from [16].

The transformed LQN model is depicted in Figure 17. The reference layer was selected based on transition $t_1$ and queueing place *Client*. The place *Client* represents the think time of the closed workload (parameter $Z = 200$ in *clien_7* task in the LQN) and the initial population of clients set to 80 (parameter [80] in *clien_7* task in the LQN). Next, we observe three layers that represent three nested critical sections that are limited by the thread pool, database connection pool, and database process pool. Once the activity *DBS-I_2* finishes, the process starts again. We have examined utilization and throughput. The results are presented in Table 4.

We take SimQPN prediction as a baseline again. The utilization results show that the *WLS-CPU* is the bottleneck of the modeled system. All solvers reported

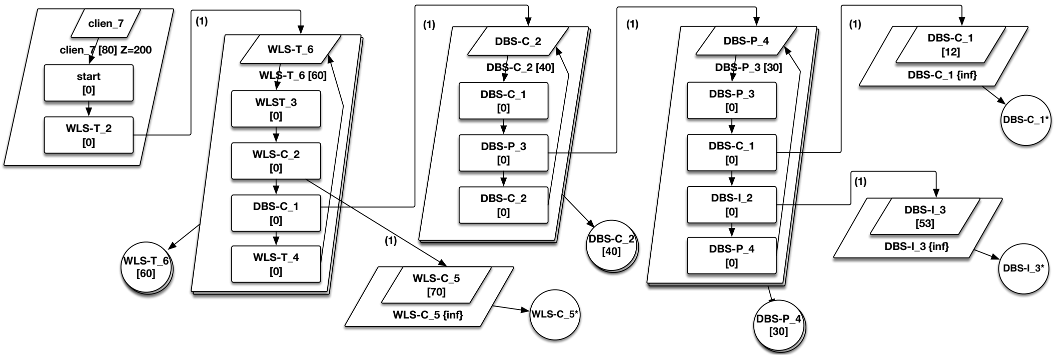

Fig. 17. LQN representation of example #2.

Table 4
Example #2—processor utilization and throughput for 80 clients.

| Utilization | DBS-CPU | DBS-I/O | WLS-CPU |
|---|---|---|---|
| SimQPN | 0.757 | 0.171 | 1 |
| LINE | 0.75714 | 0.17142 | 1 |
| LQNS | 0.75742 | 0.17149 | 1.00013 |
| LQSIM | 0.75525 | 0.17508 | 0.9828 |

| Throughput | DBS-CPU | DBS-I/O | WLS-CPU |
|---|---|---|---|
| SimQPN | 0.014 | 0.014 | 0.014 |
| LINE | 0.0142857 | 0.0142857 | 0.0142857 |
| LQNS | 0.0142934 | 0.0142934 | 0.0142877 |
| LQSIM | 0.01459 | 0.01425 | 0.01404 |

nearly 100% utilization. LQNS overestimated the utilization, probably due to a rounding error, whereas LQSIM reported the utilization as 1.8% lower than the other solvers. The predicted throughput is affected by the bottleneck resource and is similar for all the solvers. LINE and LQNS overestimated the throughput by up to 2% relatively, whereas LQSIM reported up to 4% higher throughput than the baseline.

### *5.3 Analysis of Solving Time*

Additionally to the performance prediction accuracy, we investigated the time needed to solve the model using the four examined solvers. We examined the LQN model from example #2. We varied the number of customers in the *clien_7* layer and solved the model for 1, 10, 40, and 80 clients. Then, we scaled up the modeled system and increased the quantities of the resources 2, 4, 8, 16, and 32 times with respect to the following configuration: 100 clients, 40 database connection pool size, 30 database processes, and 60 WLS threads. We denote these setups as $100 \times \{2, 4, 8, 16, 32\}$ respectively.

We executed the four solvers in Windows 7 virtual machine running on VirtualBox with assigned 2 CPUs and 4GB memory. Solving times of LQNS and LQSIM were measured using unix *time* command and included such activities like starting the solver, reading input file, solving the model and writing output. Although LQNS reports the solving time internally, we ignore them as the running time was reported as $0s$. The execution time of LINE was measured in the java code of the solver as LINE's source code is available. SimQPN reports the running wall-clock time directly in the results. The solution time measurements for LQNS and LQSIM may contain a constant additive error because the *time* command includes also the initiation of the solver in the measurement. The results are presented in Table 5.

In this experiment, we expect the analytical solvers (LINE and LQNS) to outperform the simulations (SimQPN and LQSIM). The expectation was confirmed

Table 5
Solving times of four solvers for varying number of clients in example #2.

| Clients: | SimQPN | LINE | LQNS<br>(linearizer) | LQNS<br>(exact MVA) | LQSIM |
|---:|---:|---:|---:|---:|---:|
| 1 | 0.48s | 0.44s | 0.03s | 0.05s | 06.49s |
| 10 | 0.51s | 0.54s | 0.09s | 0.94s | 1m28.95s |
| 40 | 1.09s | 0.63s | 0.06s | 1.35s | 2m49.89s |
| 80 | 1.26s | 0.72s | 0.06s | 3.75s | 4m05.62s |
| 100 × 2 | 1.12s | 0.96s | 0.07s | 9.67s | 6m56.36s |
| 100 × 4 | 2.54s | 1.34s | 0.10s | 2m7.01s | 12m23.42s |
| 100 × 8 | 7.15s | 2.09s | 0.15s | 10m8.00s | 36m08.91s |
| 100 × 16 | 12.98s | 3.54s | crash* | crash* | 100m50.59s |
| 100 × 32 | 45.78s | 6.37s | crash* | crash* | 219m22.36s |
| * out of memory ($> 4GB$) | | | | | |

experimentally, as LQSIM was the slowest of the solvers and needed 4 minutes to solve the case with 80 clients and about 3.5 hours to evaluate the $100 \times 32$ scenario. SimQPN uses *batch mean* method which observes the simulation in the steady-state (i.e., after so-called warm-up period). The simulation stops when required precision is reached. The SimQPN's solving time is low, although for big model instances, we observe a non-linear growth. LINE has outperformed the simulators and achieved linear growth of the solving time. Similar observation holds for LQNS which solved the models in a linear time and was about 10 times faster than LINE. Unfortunately, LQNS requires much more memory to solve bigger models. During the solving of the $100 \times 16$ and $100 \times 32$ models, LQNS terminated almost immediately after the start due to the lack of memory (reported error: *std::bad_alloc*).

Regarding the memory consumption, LQSIM uses a constant amount of memory during the simulation—about 75MB, 150MB, and 300MB for $100 \times 8$, $100 \times 16$, and $100 \times 32$ scenario respectively. LQNS consumes memory very fast and is unable to solve bigger models in the given configuration of the experimental machine. SimQPN has larger memory footprint due to Java virtual machine, however it scales well and can effectively handle simulations with up to several million tokens on a machine equipped with 16GB memory (for comparison, we observed $\approx 7300$ tokens in experiment #2 for the $100 \times 32$ model). LINE uses Matlab libraries for computation so there exists a memory footprint. We are unable to observe precise memory consumption for LINE due to short solving times. More experiments are needed to provide an insight into memory complexity of LINE, however, we expect low consumption as LINE uses analytical methods.

## 6 Conclusions

In this paper, we presented a concept of the model transformation that automatically transforms QPN models into LQNs in a systematic manner. We characterized the QPN and LQN formalisms by comparing the differences and pointing out the possible incompatibilities. We presented selected model fragments where the information could be lost due to necessary simplifications in the automated process of transformation (e.g., loops). We provided multiple examples to demonstrate the transformation and evaluated the solvers by means of performance prediction accuracy and solving time.

We showed that solving the transformed QPN models using LQN solvers is beneficial, especially using fluid approximation with solvers such as LINE as its solving times are lower than LQSIM and SimQPN. For small models LQNS provides short solving times, however, it consumes more memory than LINE, SimQPN, and LQSIM. The contributed transformation enables support for the three new solvers to already existing QPN models, in particular the models obtained in model-to-model transformations of DML [6] and DNI [27].

As a part of our future works, we plan to integrate the transformation and the LQN solvers in DML's and DNI's tool-chain. Additionally, we plan to develop a library for discovering common patterns in Petri nets to support more effective pattern matching in Petri nets and analysis of their features.

# References

[1] F. Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, pages 14–23. IEEE, 1993.

[2] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *J. Syst. Softw.*, 82(1):3–22, 2009.

[3] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications.* John Wiley & Sons, 2006.

[4] Jeremy T. Bradley, Richard Hayden, WilliamJ. Knottenbelt, and Tamas Suto. Extracting response times from fluid analysis of performance models. In S. Kounev, I. Gorton, and K. Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 29–43. Springer Berlin Heidelberg, 2008.

[5] F. Brosig. *Architecture-Level Software Performance Models for Online Performance Prediction.* PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, July 2014.

[6] F. Brosig, N. Huber, and S. Kounev. Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Elsevier Science of Computer Programming Journal (SciCo)*, Vol. 90, Part B:71–92, September 2014.

[7] F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev. Quantitative Evaluation of Model-Driven Performance Analysis and Simulation of Component-based Architectures. *IEEE Transactions on Software Engineering (TSE)*, 41(2):157–175, February 2015.

[8] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *Computers, IEEE Transactions on*, 42(11):1343–1360, Nov 1993.

[9] W. Fan, X. Wang, and Y. Wu. Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3):18:1–18:47, 2013.

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

[11] G. Franks, P. Maly, M. Woodside, D.C. Petriu, and A. Hubbard. Layered Queueing Network Solver and Simulator User Manual. Manual, Real-Time and Distributed Systems Lab, Carleton Univ., Canada, 2009.

[12] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to NP-completeness.* W. H. Freeman, 1979.

[13] F. Heimburger. Performance Modelling of Java EE Applications using LQNs and QPNs. Master's thesis, Technische Universitaet Darmstadt, Germany, 2007.

[14] J. Hillston. Fluid flow approximation of PEPA models. In *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, pages 33–42, Sept 2005.

[15] N. Huber. *Autonomic Performance-Aware Resource Management in Dynamic IT Service Infrastructures.* PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, July 2014.

[16] S. Kounev and A. Buchmann. Performance modeling of distributed e-business applications using queueing petri nets. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003*, pages 143–155, Washington, DC, USA, March 2003. IEEE Computer Society.

[17] S. Kounev and A. Buchmann. SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. *Performance Evaluation*, 63(4-5):364–394, 5 2006.

[18] S. Kounev and C. Dutz. QPME - A Performance Modeling Tool Based on Queueing Petri Nets. *ACM SIGMETRICS Performance Evaluation Review (PER), Special Issue on Tools for Computer Performance Modeling and Reliability Analysis*, 36:46–51, 3 2009.

[19] H. Koziolek and R. Reussner. A model transformation from the palladio component model to layered queueing networks. In Samuel Kounev, Ian Gorton, and Kai Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *Lecture Notes in Computer Science*, pages 58–78. Springer Berlin Heidelberg, 2008.

[20] T. G. Kurtz. Limit theorems for sequences of jump markov processes approximating ordinary differential processes. *Journal of Applied Probability*, 8(2):344–356, 1971.

[21] F. Liu, M. Heiner, and M. Yang. An efficient method for unfolding colored Petri nets. In *Proceedings of the 2012 Winter Simulation Conference (WSC)*, pages 1–12, 2012.

[22] P. Meier, S. Kounev, and H. Koziolek. Automated transformation of component-based software architecture models to queueing petri nets. In *19th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011), Singapore, July 25–27*, 2011.

[23] D. Mitra. Stochastic theory of a fluid model of producers and consumers coupled by a buffer. *Advances in Applied Probability*, 20(3):646–676, 1988.

[24] A. M. Pam. *Solving Stochastic Rendezvous Networks of Large Client-Server Systems with Symmetric Replication.* PhD thesis, Carleton University Ottawa, Ontario, Canada, 1996.

[25] J.F. Perez and G. Casale. Assessing sla compliance from palladio component models. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, pages 409–416, Sept 2013.

[26] P. Rygielski and S. Kounev. Data Center Network Throughput Analysis using Queueing Petri Nets. In *34th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2014 Workshops). 4th International Workshop on Data Center Performance, (DCPerf 2014)*, pages 100–105, June 2014.

[27] P. Rygielski, S. Kounev, and P. Tran-Gia. Flexible Performance Prediction of Data Center Networks using Automatically Generated Simulation Models. In *Proceedings of the Eighth EAI International Conference on Simulation Tools and Techniques (SIMUTools 2015)*, August 2015.

[28] P. Rygielski, S. Kounev, and S. Zschaler. Model-Based Throughput Prediction in Data Center Networks. In *Proceedings of the 2nd IEEE International Workshop on Measurements and Networking (M&N 2013)*, pages 167–172, October 2013.

[29] P. Rygielski and P. Światek. Graph-fold: an Efficient Method for Complex Service Execution Plan Optimization. *Systems Science*, 36(3):25–32, 2010.

[30] Standard Performance Evaluation Corporation (SPEC). Specjappserver2001 documentation. Technical report, Sep 2002.

[31] M. Tribastone. A fluid model for layered queueing networks. *Software Engineering, IEEE Transactions on*, 39(6):744–756, June 2013.

[32] M. Tribastone, J. Ding, S. Gilmore, and J. Hillston. Fluid rewards for a stochastic process algebra. *Software Engineering, IEEE Transactions on*, 38(4):861–874, July 2012.

[33] Kishor S. Trivedi and VidyadharG. Kulkarni. Fspns: Fluid stochastic petri nets. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 24–31. Springer Berlin Heidelberg, 1993.

[34] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.

[35] J. Walter, S. Spinner, and S. Kounev. Parallel Simulation of Queueing Petri Nets. In *Proceedings of the Eighth EAI International Conference on Simulation Tools and Techniques (SIMUTools 2015)*, August 2015.

[36] C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar. The stochastic rendezvous network model for performance of synchronous client-server-like distributed software. *Computers, IEEE Transactions on*, 44(1):20–34, Jan 1995.

[37] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy. Performance modeling and prediction of enterprise javabeans with layered queuing network templates. In *Proceedings of the 2005 Conference on Specification and Verification of Component-based Systems*, SAVCBS '05, New York, NY, USA, 2005. ACM.