A Fluid Model for Layered Queueing Networks

Mirco Tribastone

Abstract—Layered queueing networks are a useful tool for the performance modeling and prediction of software systems that exhibit complex characteristics such as multiple tiers of service, fork/join interactions, and asynchronous communication. These features generally result in nonproduct form behavior for which particularly efficient approximations based on mean value analysis (MVA) have been devised. This paper reconsiders the accuracy of such techniques by providing an interpretation of layered queueing networks as *fluid* models. Mediated by an automatic translation into a stochastic process algebra, PEPA, a network is associated with a set of ordinary differential equations (ODEs) whose size is insensitive to the population levels in the system under consideration. A substantial numerical assessment demonstrates that this approach significantly improves the quality of the approximation for typical performance indices such as utilization, throughput, and response time. Furthermore, backed by established theoretical results of asymptotic convergence, the error trend shows monotonic decrease with larger population sizes—a behavior which is found to be in sharp contrast with that of approximate mean value analysis, which instead tends to increase.

Index Terms—Modeling and prediction, Markov processes, PEPA, ordinary differential equations, queueing networks, mean value analysis

1 Introduction

Enhanced queueing networks are the subject of a large body of literature concerned with the performance modeling and prediction of software systems. The *method of surrogate delays* by Jacobson is an early approach to the description of simultaneous resource possession [1]. The solution technique transforms the original model, which presents nonproduct form features, into two distinct networks which do enjoy a product form solution. These are analyzed iteratively until the difference between the two is less than a threshold. *Mutatis mutandis*, such a scheme is featured by other variants such as the stochastic rendezvous network model [2] and Rolia's method of layers [3], both concerned with the analysis of software systems with synchronicity where a resource is blocked while it is waiting for services from other resources.

More recently, the layered queueing network (LQN) model has been presented in a unified manner and has been shown to incorporate all of the aforementioned features, in addition to being able to describe other forms of behavior which typically arise in the modeling of distributed computer systems, such as fork/join interaction and asynchronous communication [4].

A common feature of enhanced networks is that the analytical solution methods may present two orthogonal forms of approximation. One arises from replacing the recursive scheme of mean value analysis (MVA) [5] with a fixed-point iterative algorithm which is fundamentally insensitive to the customer population levels in the network [6], [7], [8]. The other source of approximation is due to the heuristic modifications applied to the MVA algorithm in order to characterize and analyze nonproduct form behavior. Establishing exactly the quality of the accuracy for both kinds of error has proven to be difficult. Apart from a theoretical study concerned with the former kind [9], the most typical route is to perform thorough validation studies which compare the approximate estimates against simulation. For layered models, the studies reported in the literature have shown good accuracy in general, with errors within a few percent on average [3], [10], [11], [12].

The purpose of the present paper is to reconsider the error behavior in LQNs in light of analogous developments regarding scalable analytical techniques witnessed in the context of stochastic process algebra, specifically PEPA [13]. Here, the approximation is derived by shifting from a discrete-state characterization in terms of a Markov process to a continuous-state representation based on a system of ordinary differential equations (ODEs) [14]. Despite the drastically different reasoning behind the nature of the approximation, there are two points in common with approximate mean value analysis (AMVA). The first is that the problem size is independent of the population levels in the system, as those only affect the initial condition of an underlying initial value problem to be solved. The second point in common is that the ODE solution can be interpreted as an approximation to the expected value of the original stochastic process [15].

On the other hand, there are characteristics which make the ODE approach substantially different from AMVA. First, the ODE solution gives an approximation to the entire time-course evolution of the stochastic process under scrutiny, thereby readily enabling transient analysis as well as steady-state analysis, when the equilibrium point of the ODE system is examined. Second, and most importantly, in the case of PEPA the approximated stochastic process is a *population model* where the state descriptor is a vector of nonnegative integers which gives the number of components in each of the possible local states of the system.

• *The author is with the Department for Informatics, Ludwig-Maximilians University of Munich, Oettingenstrasse 67, D-80538 Munich, Germany. E-mail: tribastone@pst.ifi.lmu.de.*

Manuscript received 11 Oct. 2011; revised 25 July 2012; accepted 26 Aug. 2012; published online 26 Sept. 2012.
Recommended for acceptance by C.M. Woodside.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-10-0291.
Digital Object Identifier no. 10.1109/TSE.2012.66.

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

A powerful result by Kurtz guarantees uniform convergence in probability to the ODE solution as the initial population levels in the model go to infinity [16]. In practice, extensive experimental studies have shown that the quality of the approximation increases quickly with larger population sizes [14]. Using the same framework, suitable functions of Markov chains may be shown to converge to deterministic estimates, and typical performance indices such as utilization, throughput, and average response time have been shown to be encoded as such functions [17].

The peculiar nonproduct form features of LQNs, which necessitate ad hoc modifications to the AMVA algorithms, find instead a more congenial modeling environment in stochastic process algebra since these patterns of behavior can be described naturally using standard compositional operators. The main observation which motivates this paper is that endowing an LQN-like description with a process algebraic population-oriented semantics may improve the accuracy of the estimation because the analysis technique does not involve heuristic corrections.

Cross-fertilization between queueing networks and process algebras has been exploited by other authors (e.g., see [18] for an account). Hillston and Thomas find a class of PEPA models that admits a product form solution [19]; a general framework for product forms for PEPA is that of Harrison's *Reversed Compound Agent Theorem* [20]; Thomas and Zhao define a class of queueing-network type PEPA models that are amenable to MVA [21]. Interestingly, they also provide an approximate solution based on a system of ODEs that correspond to those that would be generated by the encoding presented in this paper. However, as with all the other aforementioned contributions, the PEPA model must satisfy certain syntactical conditions which, in general, are not met when a translation from LQNs is required.

This author has presented a preliminary study on the relationship between PEPA and LQNs in [22] and [23]. The present paper is a considerably extended version which makes the following novel contributions:

- We develop a formal model of the LQN model (cf., Section 2.2). This will form the basis for a systematic translation into a PEPA model that is convenient for a population-oriented interpretation (cf., Section 3). This is a significant deviation from [22], where the translation was less formal, being tailored to a case study.
- The accuracy of the approximation is thoroughly studied. Unlike [22], which presented a numerical assessment on a small number of cases, the error behavior is analyzed on a large set of randomly generated model instances for a more unbiased evaluation. The results indicate that the ODE approximation generally behaves better than AMVA (cf., Section 4.2).
- Further, we test the hypothesis that the ODE approximation yields a more *controlled* error behavior: Layered queueing networks with increasingly larger population sizes enjoy more precise ODE estimates. Instead, the AMVA approximation error generally shows a tendency to increase as a function of the system size.

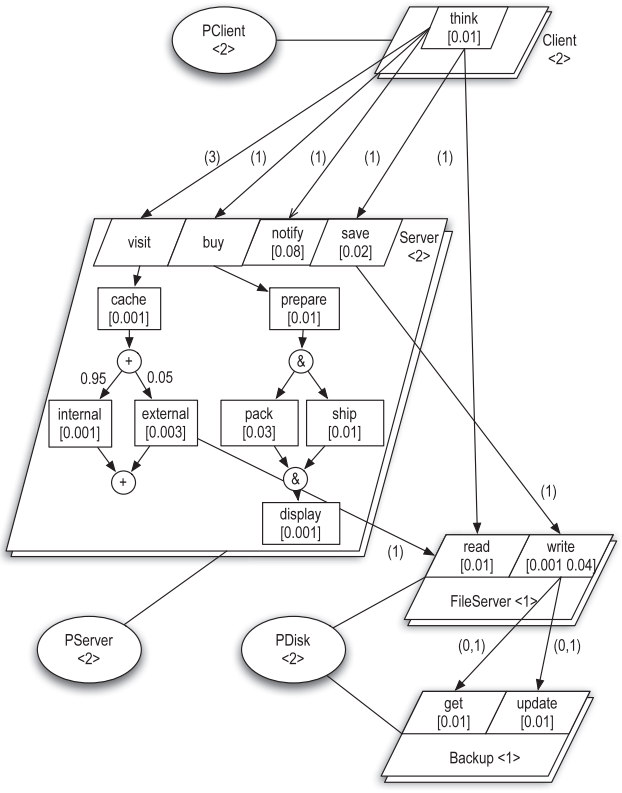

Fig. 1. LQN model of a distributed application.

For the sake of self-containment, the paper is completed by an overview of the LQN model, in Section 2.1, and an introduction to PEPA, in Section 2.3, with emphasis on the notions and notation that will be used throughout the remainder.

2 Background

2.1 Overview of Layered Queuing Networks

Fig. 1 shows a sample LQN of a distributed application. The intent is not to provide a realistic model of a concrete system; rather it serves the purpose of acting as the case study that features all of the LQN elements considered in this paper. The reader is referred to [4] (and the rich bibliography therein) for a more detailed treatment. Servers (called *tasks*) are drawn as stacked parallelograms and their multiplicity is indicated within angular brackets alongside the task's name. For instance, *File Server* <1> denotes one single thread of execution for the file server. A task is deployed onto a *processor*, depicted as a circle connected to the task. Concurrency levels for processors are denoted similarly to tasks.

Distinct kinds of services (called *entries*) exposed by a task are represented by small parallelograms drawn inside the task. Each entry is associated with an execution graph consisting of atomic units of computation called *activities*, drawn as rectangles. Activities are arranged through operators for precedence (directed arrows), decision/merge nodes (small circles with the + symbol, also called OR-fork and OR-join, respectively) and fork/join synchronization (small circles with the & symbol, also called AND-fork and AND-join, respectively). Each activity is characterized by a

service time demand on the processor with which the task is associated, indicated within square brackets. For the sake of graphical convenience, execution graphs which consist of a single activity are not explicitly drawn, and the activity's execution demand is directly shown within the associated entry. In Fig. 1, only the execution graphs of entry *visit* and *buy* are drawn. The former models an activity which accesses some cached information, after which it performs an *internal* activity with probability 0.95 or a more expensive *external* activity with probability 0.05. In the entry *buy*, after *prepare* is performed the two activities *pack* and *ship* are executed in parallel. When they both finish, *display* is executed.

Layering of services is modeled by means of *requests* made from an activity to an entry in another task in the network. Requests are indicated by directed arrows and may be of two kinds: *synchronous*, with closed arrowheads, and *send-no-reply*, with open arrowheads. The semantics of the latter is that there is synchronization during the send action, but the callee does not await a response. This is also called *asynchronous* in the remainder because the caller progresses independently from the callee after the send has occurred.

Each request is labeled with a number between parentheses which gives the number of requests per execution. This can be interpreted deterministically or as the mean of a geometric distribution. The total number of requests performed by an activity determines the distribution of its execution demand. The total demand is divided into *slices* whose duration is drawn from independent exponential distributions with mean equal to the ratio between total execution demand and total number of requests. The execution of one slice is interposed between successive requests to other entries. For example, the total demand for *save* is divided into two slices with mean duration 0.01 time units, between which is interposed a synchronous call to *write*. *Reference tasks* are tasks which do not accept requests; they are used to model system workload.

For entries which accept synchronous requests, their overall behavior may be subdivided into two *phases*. The first phase models the computation carried out from the receipt of the request until the reply to the caller. Such a reply is denoted as a dashed arrow pointing to the activity's entry. To reduce clutter in the graphical notation of LQN, synchronous entries which do not explicitly show a return arrow are assumed to reply after the whole control flow has been executed; thus, *visit* returns after *internal* or *external*, while *buy* returns after *display*. All the activities in the execution graph that follow the replying entry are part of the second phase, indicating an autonomous continuation during which the caller is not blocked. Execution graphs consisting of two activities such that each represents the behavior of one phase can be conveniently drawn in a compact form, as illustrated by *write* in Fig. 1. The execution demand for each phase is drawn inside the entry within square brackets. The requests from multiphase entries are labeled with pairs in which the ith element represents the number of requests made by the activity in the ith phase.

By convention, we assume that this compact notation for activities with second phases is systematically expanded into an equivalent one with two distinct activities joined by a precedence relation. For instance, *write* is transformed into two activities, $write'$ and $write''$, as shown in Fig. 2. In practice, this is no restriction as it can capture situations of instantaneous duration, as in the LQN model, with excellent accuracy (cf., Section 4.1). In addition, it offers more flexibility in contexts where nonzero duration times are desirable.

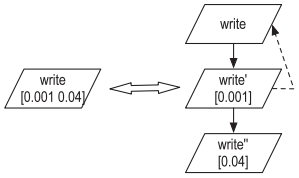

Fig. 2. Treatment of activities with second phases. Recalling the semantics of implicit activity invocation, $write$ represents two distinct activities, here denoted by $write'$ and $write''$.

2.2 A Formal Model of a Layered Queuing Network

Although the LQN model is provided in [4] with a UML metamodel which could be employed as the basis for the transformation into PEPA, for the purposes of this paper we find it more convenient to work with a more formal LQN specification. The most notable peculiarity of such a setting is the treatment of execution graphs. Each task entry is associated with a single-rooted directed acyclic graph (DAG) which models the *main control flow*. There may be more than one control flow if the model contains fork/join nodes. In this case, the main control flow resumes with the behavior occurring after the fork.

For example, in Fig. 1, the main control flow of *buy* executes *prepare*, then forks two other control flows, and resumes after joining with *display*. Each of the forked flows is modeled as a distinct DAG. The forking/forked relationship is formally captured by having a *fork* node in the main flow which is labeled with the set of DAGs which are forked at that node. Any of such DAGs may contain other fork nodes, which in turn are labeled with further DAGs, with an arbitrary nesting. This scheme allows for rich forms of interaction where any flow may fork further flows.

In the remainder, sets are indicated with calligraphic letters and are ranged over by the corresponding Roman upper case letters, e.g., A, D, E are elements of $\mathcal{A}, \mathcal{D}, \mathcal{E}$, respectively. All sets are assumed to be pairwise disjoint. Our LQN model is defined by the following elements:

- A set of vertices $\mathcal{V} = \mathcal{A} \cup \mathcal{D} \cup \mathcal{F} \cup \mathcal{M}$ where
  - $\mathcal{A}$ is the set of activities, e.g., $\mathcal{A} = \{buy, save, \ldots\}$;
  - $\mathcal{D}$ is the set of decision nodes;
  - $\mathcal{F}$ is the set of fork nodes; each node $F \in \mathcal{F}$ is labeled with a set of S_F DAGs, $\{\Gamma_F^i \mid 1 \leq i \leq S_F\}$, each giving a distinct control flow forked by F; let $\mathcal{V}_F^i \subseteq \mathcal{V}$ be the set of vertices of Γ_F^i;
  - $\mathcal{M}$ is the set of merge nodes.
- $\mathcal{E}$ is the set of entries, e.g., $\mathcal{E} = \{visit, read, \ldots\}$; it is partitioned into two sets, denoted by $\mathcal{E}^s$ and $\mathcal{E}^a$, of synchronous and asynchronous entries, respectively. Let Γ_E be the DAG associated with entry E, with vertices denoted by $\mathcal{V}_E \subseteq \mathcal{V}$. Fig. 3 shows the formal interpretation of entry *buy*.

(a) Γ_{buy} (b) Γ^1_{fork} (c) Γ^2_{fork}

Fig. 3. The entry *buy* is modeled with three DAGs, with $\{prepare, display, pack, ship\} \subseteq \mathcal{A}$ and $fork \in \mathcal{F}$. The vertex which is linked by the $fork$ node, i.e., $display$, gives the behavior of the process after all forked flows have completed.

- $\mathcal{P}$ is the set of processors, e.g.,

$$\mathcal{P} = \{PClient, PServer, PDisk\}.$$

  Let $M_P \in \mathbb{N}$ denote the multiplicity of processor P.
- $\mathcal{T}$ is the set of tasks, e.g.,

$$\mathcal{T} = \{Client, Server, FileServer, Backup\}.$$

  Let $N_T \in \mathbb{N}$ denote the multiplicity of task T.

The single-rooted DAGs Γ_E, with $E \in \mathcal{E}$, and Γ^i_F, for $F \in \mathcal{F}$ and $1 \le i \le S_F$, have arcs with labels in $\{l \in \mathbb{R} : 0 < l \le 1\} \cup \{\top\}$. The usual notation $V_1 \xrightarrow{l} V_2$ indicates a directed arc from vertex V_1 to vertex V_2 labeled with l. Labels will be used for the treatment of decision nodes, to assign a probability to each outgoing arc. For other edges, the *don't-care* symbol $\top$ is used; however, $V_1 \xrightarrow{\top} V_2$ is more conveniently written $V_1 \to V_2$ (as is the case in Fig. 3). The DAG roots are denoted by $\mathrm{root}(\Gamma_E)$ (or $\mathrm{root}(\Gamma^i_F)$), and the leaves by $\mathrm{leaves}(\Gamma_E)$ (or $\mathrm{leaves}(\Gamma^i_F)$).

Definition 1. *Given an entry E, a node V is said to belong to E if $E \in \mathcal{V}_E$, or if it is a vertex in any of the DAGs which are defined through fork nodes in $\mathcal{V}_E$, with arbitrary nesting. Similarly, we say that V belongs to T, where T is the task which exposes E.*

The following functions on these sets will also be used.

- $\mathrm{act} : \mathcal{P} \to 2^{\mathcal{A}}$ gives the set of activities executed on a processor. (The obvious requirement is that at least one activity be executed on a processor.)
- $\mathrm{act}^{-1} : \mathcal{A} \to \mathcal{P}$ gives the processor on which an activity is executed.
- $\mathrm{ctx} : \mathcal{P} \to \mathbb{R}_{>0}$ gives the rate of a *context switch* between to successive requests to a processor. Here, we assume that $\mathrm{ctx}(P) = \nu$ for all $P \in \mathcal{P}$. In the comparison with the LQN analysis by AMVA, ν will be set to a large value so as to approximate instantaneous switches. This is discussed in more detail in Section 4.1.
- $\mathrm{del} : \mathcal{E} \to \mathbb{R}_{>0}$ gives the rate of propagation of an entry invocation, i.e., the time it takes a task to receive a service invocation. Similarly to $\mathrm{ctx}(\cdot)$, we assume that $\mathrm{del}(E) = 1/\nu$ for all $E \in \mathcal{E}$. For large ν this approximates instantaneous transfer of messages.
- $\mathrm{dem} : \mathcal{V} \to \mathbb{R}_{\ge 0}$ gives the total mean execution rate of a node in an execution graph (e.g., $\mathrm{dem}(prepare) = 1/0.01$). As with $\mathrm{del}(\cdot)$, we assume for simplicity that $\mathrm{dem}(V) = \nu$ if $V \in \mathcal{F} \cup \mathcal{M}$.
- $\mathrm{ent} : \mathcal{T} \to 2^{\mathcal{E}}$ gives the set of entries of a task.
- $\mathrm{frk} : \mathcal{T} \to 2^{\mathcal{F}}$ returns the set of fork nodes which belong to T.
- $\mathrm{prc} : \mathcal{E} \to \mathcal{P}$ gives the process on which the activities of an entry are executed.
- $\mathrm{rep} : \mathcal{A} \cup \mathcal{F} \cup \mathcal{M} \to \mathcal{E}$ gives the entry to which a node replies (may be $\emptyset$). An extension of this model to allow replies at decision nodes is straightforward, but it would unnecessarily complicate the translation.
- $\mathrm{req} : \mathcal{A} \to 2^{\mathcal{E} \times \mathbb{N}}$ gives the set of requests made by an activity with their multiplicity. The elements of this set may be denoted by (E_i, N_i), with $1 \le i \le |\mathrm{req(A)}|$ (cf., Template 2).
- $N(A) = \sum_{(E,N) \in \mathrm{req}(A)} N$ is the total number of requests made during the execution of an activity $A \in \mathcal{A}$.

2.3 PEPA

A PEPA model consists of a composition of entities which can perform actions sequentially (*sequential components*). Actions may be performed autonomously (*independent actions*) or in synchronization with other sequential components of the system (*shared actions*). Similarly to the reference book for the language [13], throughout this paper process names are strings with uppercase initials, whereas action types have lowercase initials. The language supports the following operators.

Prefix $(\alpha, r).P$ constitutes the atomic unit of computation of a PEPA model. It is a sequential component which may perform an action of type α, subsequently behaving as P, which is said to be a *local derivative* (or *local state*) of the component. The action duration is exponentially distributed with mean $1/r$ time units. Every behavior that involves precedence between two distinct activities will be captured by a prefix in the remainder.

Choice $P + Q$ indicates that the sequential component may behave as P or Q. For instance, $(\alpha, r).P + (\beta, s).Q$ is said to *enable* actions α and β, which are executed with probabilities $r/(r+s)$ and $s/(r+s)$, respectively. Choice between distinct action types will be used to model different entries within the same task. With a slight abuse, a PEPA choice between prefixes will be described using the following *sigma notation*:

$$\sum \{(\alpha, r).P \mid \mathbb{P}(\alpha, r, P)\},$$

where $\mathbb{P}(\alpha, r, P)$ is a predicate of the choice's constituting prefixes, e.g.,

$$\sum \{(\alpha, i \cdot r).P_i \mid 1 \le i \le 2)\} := (\alpha, r).P_1 + (\alpha, 2r).P_2.$$

Constant $A \stackrel{def}{=} P$*:* is used to model cyclic behavior. For instance, $A \stackrel{def}{=} (\alpha, r).(\beta, s).A$ is a sequential component with two local derivatives which performs sequences of α- and β-activities forever. For instance, in the treatment of LQNs, constants will be fundamental in modeling the cyclic behavior of a thread which finishes work for a client and goes back to its initial state where another client may be served.

Cooperation $P \overset{\bowtie}{_L} Q$*:* is the synchronization operator of the language. The processes P and Q are required to synchronize over the action types in the set L. All the other actions are performed autonomously. For instance,

$(\alpha, r).(\beta, s).P \overset{\bowtie}{_{\{\alpha\}}} (\alpha, t).(\gamma, u).Q$ is a cooperation between two sequential components which may perform a shared activity of type α, subsequently behaving as $(\beta, s).P \overset{\bowtie}{_{\{\alpha\}}} (\gamma, u).Q$. Then, actions β and γ are carried autonomously. By contrast, in the cooperation $(\alpha, r).P \overset{\bowtie}{_{\{\alpha\}}} (\beta, s).Q$ the process $(\alpha, r).P$ does not progress because α is not available in the right hand side of the cooperation. The set of all shared action types between P and Q will be denoted by the symbol $*$. Cooperation will be used in the translation of LQNs into PEPA for two main modeling situations: 1) the definition of synchronization point in forks and joins; 2) passing the focus of control from one sequential component to another, e.g., a request from an activity to an entry.

Let us introduce two compact notations for replicas of many sequential components. We write $C[N]$ for many independent copies of C, i.e.,

$$C[N] := \begin{cases} C, & \text{if } N = 1, \\ \underbrace{C \overset{\bowtie}{_{\emptyset}} C \overset{\bowtie}{_{\emptyset}} \cdots \overset{\bowtie}{_{\emptyset}} C}_{N \text{ times}}, & \text{if } N > 1. \end{cases}$$

The following *product notation* is used for many distinct sequential components. It is written in the form

$$\prod_{L} \{C \mid \mathbb{P}(C)\},$$

where L is an action set, C is a process term, and $\mathbb{P}(C)$ is a predicate on it. For instance,

$$\prod_{L} \{C_i[N] \mid 1 \le i \le 3\} := C_1[N] \overset{\bowtie}{_{L}} C_2[N] \overset{\bowtie}{_{L}} C_3[N],$$
$$\prod_{L} \{C_i[N] \mid i = 1\} := C_1[N].$$

For completeness, we point out that PEPA also features another operator, *hiding*, which is, however, not used in the remainder of the paper.

It will be useful to indicate a set that represents all the possible action types that a component P may engage in. This is denoted by $\mathcal{A}ct(P)$ and is defined recursively as

$$\mathcal{A}ct(P) = \begin{cases} \{\alpha\} \cup \mathcal{A}ct(Q), & \text{if } P = (\alpha, r).Q, \\ \mathcal{A}ct(Q), & \text{if } P = A, \text{ with } A \overset{def}{=} Q, \\ \mathcal{A}ct(Q_1) \cup \mathcal{A}ct(Q_2), & \text{if } P = Q_1 + Q_2, \\ \mathcal{A}ct(Q_1) \cup \mathcal{A}ct(Q_2), & \text{if } P = Q_1 \overset{\bowtie}{_{L}} Q_2. \end{cases}$$

3 PEPA Interpretation of LQNs

The main rationale behind the PEPA interpretation of LQNs is to model replicated tasks and processors as copies of identical sequential components. Thus, if T is the sequential component which describes the behavior of a task thread, then the whole server will be described as $T[N]$, where N is the multiplicity of the server in the LQN model. The empty cooperation set between two copies of the same component represents a reasonable assumption of independence between the behavior of two distinct threads of execution. Analogously, two distinct copies of the same processor will be assumed to behave independently from each other. This is consistent with the LQN interpretation of a processor as a multiserver station in the network. Here, the main benefit in using this form of replication of behavior is that the model has a convenient population-based representation, which makes the fluid approximation independent of the replica sizes, but only dependent on the local states of the replicated process.

3.1 Processor

Template 1. Translation of a processor.

$$P_1 \overset{def}{=} (get_P, \mathrm{ctx}(P)).P_2$$

$$P_2 \overset{def}{=} \sum_{\substack{A \in \mathrm{act}(P), \\ \mathrm{dem}(A) > 0}} (A, s(A)).P_1$$

The template for the translation of a processor P is illustrated in Template 1, showing a cyclic two-state sequential component. The state P_1 models a *context-switch* activity which grants exclusive access to the processor. The second state P_2 enables all the actions corresponding to the activities which are executed on P, by means of the choice operator. Each activity phase is mapped onto a distinct action type in PEPA and the rate of execution reflects the fragmentation of the computation into slices. For any activity A, the rate of execution of a slice is

$$s(A) = (N(A) + 1)\mathrm{dem}(A).$$

Notice that this interpretation produces a concise description for a processor, whose number of sequential components—which is always equal to two—does not depend upon the distinct classes of service enabled. For example, the translation of *PDisk* is shown in Example 1.

Example 1. Processor *PDisk*.

$$\begin{aligned} PDisk_1 &\overset{def}{=} (get_{PDisk}, \nu).PDisk_2 \\ PDisk_2 &\overset{def}{=} (read, 1/0.01).PDisk_1 \\ &\quad + (write', 1/0.001).PDisk_1 \\ &\quad + (write'', 3/0.04).PDisk_1 + (get, 1/0.01).PDisk_1 \\ &\quad + (update, 1/0.01).PDisk_1 \end{aligned}$$

3.2 Nodes of an Execution Graph

3.2.1 Activity and Request

An LQN activity subsumes a PEPA prefix with a sequence of action types whose length is determined by the number of outgoing requests and their synchronicity. A synchronous call is described by two activities which model the request and the reply. The PEPA action type for the request has the form $req_{A,E}$, where A is the activity from which the request originates and E is the entry called by A. Similarly, the action type for the reply has the form $rep_{A,E}$. An asynchronous call is represented with a single prefix of type $req_{A,E}$.

The PEPA process corresponding to the LQN activity interposes executions of slices of A between requests. The following snippets of PEPA syntax will be useful for the translation of an activity:

$$Acq_A := \begin{cases} (get_P, \mathrm{ctx}(P)).(A, s(A)), & \text{if } s(A) > 0, \\ \epsilon, & \text{if } s(A) = 0, \end{cases}$$
$$Syc_{A,E} := \big(req_{A,E}, \mathrm{del}(E)\big).\big(rep_{A,E}, \mathrm{del}(E)\big).Acq_A,$$
$$Asc_{A,E} := \big(req_{A,E}, \mathrm{dem}(E)\big).Acq_A,$$

TABLE 1
Evaluation of the Syntactic Term Rep_V for a Vertex V Which Belongs to Entry $E \in \mathrm{ent}(T)$

Case	$\mathrm{rep}(V) = \emptyset$			$\mathrm{rep}(V) \neq \emptyset$		
Children	$V \in \mathrm{leaves}(\Gamma_E)$	$V \to V'$	$V \in \mathrm{leaves}(\Gamma_F^i)$	$V \in \mathrm{leaves}(\Gamma_E)$	$V \to V'$	$V \in \mathrm{leaves}(\Gamma_F^i)$
Rep_V	T_1	V_1'	$(join_F, \mathrm{dem}(F)).Fork_{\mathrm{root}(\Gamma_F^i)}$	Suc_{V,T_1}	$Suc_{V,V_1'}$	$(join_F, \mathrm{dem}(F)).Suc_{V,Fork_{\mathrm{root}(\Gamma_F^i)}}$

$$Suc_{V,P} = \sum \left\{ (rep_{A,\mathrm{rep}(V)}, \mathrm{del}(\mathrm{rep}(V))).P \mid \forall A \in \mathcal{A} : \exists (E,N) \in \mathrm{req}(A) \text{ with } E = \mathrm{rep}(V) \right\}$$

with $P = \mathrm{act}^{-1}(A)$. We treat ϵ as the empty string, with the understanding that $\epsilon.Q \equiv Q$ for all process terms Q.

The string Acq_A models the access to a processor and the execution of a slice of activity A. It is an empty string ϵ if the activity has no execution demand. $Syc_{A,E}$ and $Asc_{A,E}$ model the sequences of prefixes for synchronous and asynchronous requests (followed by slice executions), respectively.

We also make use of Rep_V, which links the behavior of a vertex V in the execution graph to that of its potential children. This abbreviation is defined in Table 1 and differs according to the nature of the successor node and whether a vertex performs a reply action. If the vertex is a leaf with no replies (for instance, the leaf for an entry with asynchronicity, such as *notify*), then the successor behaves as T_1. This is the initial state of a task which makes all its possible entries available to other tasks (cf., Section 3.3). If V has a successor, then the process simply behaves as the initial state of the successor node. If the vertex is a leaf of a DAG which models a forked control flow, then the behavior of a synchronizing join activity is performed and then the component behaves as the initial state in that flow, denoted by the process definition $Fork_{\mathrm{root}(\Gamma_F^i)}$. The cases when $\mathrm{rep}(V) \neq \emptyset$ are similar except that the process performs the reply actions (cf., $Rep_{V,P}$) before behaving as its successor node.

The translation of an LQN activity is shown in Template 2.

Template 2. Translation of an activity.
Let (E_i, N_i) be the ith element of $\mathrm{req}(A)$.

$$A_1 \stackrel{def}{=} Acq_a.A_2$$

$$A_{i+1} \stackrel{def}{=} \begin{cases} \underbrace{Syc_{A,E_i}.\cdots.Syc_{A,E_i}}_{N_i \text{ times}}.A_{i+2} & \text{if } E_i \in \mathcal{E}^s, \\ \underbrace{Asc_{A,E_i}.\cdots.Asc_{A,E_i}}_{N_i \text{ times}}.A_{i+2} & \text{if } E_i \in \mathcal{E}^a, \end{cases}$$

$$\text{for all } 1 \leq i \leq |\mathrm{req}(A)|,$$

$$A_{|\mathrm{req}(A)|+2} \stackrel{def}{=} Rep_A$$

The first process definition A_1 models the first slice execution. If the activity replies to a synchronous request, then the last constant models the replies. The corresponding action types are given by all LQN activities which make requests to the entry in which A is executed.

Example 2. Activity *write*.
FIRST PHASE

$$Write_1' \stackrel{def}{=} (get_{PDisk}, \nu).(write', 1/0.001).Write_2'$$
$$Write_2' \stackrel{def}{=} (rep_{save,write'}, \nu).Write_1''$$

SECOND PHASE

$$Write_1'' \stackrel{def}{=} (get_{PDisk}, \nu).(write'', 3/0.04).Write_2''$$
$$Write_2'' \stackrel{def}{=} (req_{write'',get}, \nu).(rep_{write'',get}, \nu).(get_{PDisk}, \nu).(write'', 3/0.04).Write_3''$$
$$Write_3'' \stackrel{def}{=} (req_{write'',update}, \nu).(rep_{write'',update}, \nu).(get_{PDisk}, \nu).(write'', 3/0.04).Write_4''$$
$$Write_4'' \stackrel{def}{=} FileServer_1$$

As a concrete application, the translation of *write* is given in Example 2, recalling the pretransformation applied to activities with second phases shown in Fig. 2. The process definition $Write_2'$ is obtained by evaluating $Rep_{write'}$ in the case when $\mathrm{rep}(write') \neq \emptyset$ and $write' \to write''$. Instead, $Write_4''$ is obtained as the leftmost case of Table 1, i.e., when the node is a leaf which does not reply. The latter fact captures the property that this phase is autonomous with respect to the caller's behavior.

3.2.2 Decision/Merge Nodes

The case of probabilistic branching $D \in \mathcal{D}$ is treated as a PEPA choice between prefixes with action type or_D. As shown in Template 3, the total rate for a process in state D_1 is $\mathrm{dem}(D)$ since the sum across the weights of the outgoing arcs from D must be equal to 1. A merge node is treated similarly to the final state of an activity, via the definition Rep_M.

Template 3. Translation of decision/merge nodes.

DECISION: $D_1 \stackrel{def}{=} \sum \{ (or_D, p\,\mathrm{dem}(D)).V_1 \mid D \stackrel{p}{\to} V \}$

MERGE: $M_1 \stackrel{def}{=} Rep_M$

Example 3. Decision/merge nodes in Fig. 1.

$$O_1 \stackrel{def}{=} (or_O, 0.95\,\nu).Internal_1 + (or_O, 0.05\,\nu).External_1$$

3.2.3 Fork/Join Nodes

Fork/join synchronization is rendered by assigning a distinct sequential component to each concurrent control flow. Such flows perform the activities autonomously and synchronize over the action types corresponding to fork and join nodes in the execution graph. Notice that activities in distinct control flows may be not completely independent since they may access the same processor on which the task is deployed, which is shared.

Template 4 treats fork/join nodes.

Template 4. Translation of fork/join nodes.

$$F_1 \stackrel{def}{=} (fork_F, \mathrm{dem}(F)).J_1$$
$$J_1 \stackrel{def}{=} (join_F, \mathrm{dem}(F)).Rep_F$$
$$Fork_{V^i} \stackrel{def}{=} (fork_F, \mathrm{dem}(F)).V_1^i$$

with $V^i = \mathrm{root}(\Gamma_F^i)$, for all $1 \leq i \leq S_F$.

The *forking* component in local state F_1 starts the forked processes, each of which is in state $Fork_V$. This is obtained by a suitable composition between the sequential component of F_1 and all $Fork_V$ in the system equation (cf., Section 3.4). The forking process goes into state J_1 where it waits for all the forked processes to end their computations. The other synchronizing activities are obtained by the evaluation of Rep_V for all nodes V that are leaves in the DAG which labels a fork node (cf., Table 1).

Example 4. Fork/join nodes in Fig. 1.

$$\begin{aligned} F_1 &\stackrel{def}{=} (fork_F, \nu).J_1 \\ J_1 &\stackrel{def}{=} (join_F, \nu).Display_1 \\ Fork_{pack} &\stackrel{def}{=} (fork_F, \nu).Pack_1 \\ Fork_{ship} &\stackrel{def}{=} (fork_F, \nu).Ship_1 \end{aligned}$$

Example 4 shows the process definitions that arise from the treatment of the fork and join nodes in the running example. To give more insight into the overall algebraic interpretation of this situation, it is interesting to consider the complete set of definitions for one of the forked flows, e.g., *pack*. These are, according to Template 2 and Table 1,

$$\begin{aligned} Pack_1 &\stackrel{def}{=} (get_{PServer}, \nu).(pack, 1/0.03).Pack_2, \\ Pack_2 &\stackrel{def}{=} (join_F, \nu).Fork_{pack}. \end{aligned}$$

Overall, a compound model of kind

$$F_1 \stackrel{\bowtie}{L} Fork_{Pack}, \qquad L = \{fork_F, join_F\}, \tag{1}$$

gives the desired behavior as it leads to the following computation path, according to the semantics of PEPA:

$$\begin{aligned} F_1 \stackrel{\bowtie}{L} Fork_{Pack} &\xrightarrow{(fork_F,\nu)} J_1 \stackrel{\bowtie}{L} Pack_1 \\ &\xrightarrow{(get_{PServer},\nu)} J_1 \stackrel{\bowtie}{L} (pack, 1/0.03).Pack_2 \\ &\xrightarrow{(pack,1/0.03)} J_1 \stackrel{\bowtie}{L} Pack_2 \\ &\xrightarrow{(join_F,\nu)} Display_1 \stackrel{\bowtie}{L} Fork_{Pack}. \end{aligned}$$

This models that J_1 does not change its local state until a $join_F$ action is observed. After this action, the control flow which initiated F behaves as $Display_1$, which is the first activity after the join.

3.3 Task

A reference task is a task that describes the workload in the model. It is identified by not having activities within other network tasks which make requests to it. The following definition formalizes this situation.

Definition 2. *A task $T \in \mathcal{T}$ in an LQN model is said to be a reference task if there is no $A \in \mathcal{A}$ such that there exists $(E, N) \in \mathrm{req}(A)$ such that $E \in \mathrm{ent}(T)$. A task which is not a reference task is said to be a* layered task.

Template 5 shows the translation of tasks.

Template 5. Translation of a task.
REFERENCE TASK

$$T_1 \stackrel{def}{=} V_1, \quad V = \mathrm{root}(\Gamma_E), \quad \text{with } E = \mathrm{ent}(T).$$

LAYERED TASK

$$T_1 \stackrel{def}{=} \sum\{(req_{A,E}, 1/\mathrm{del}(E)).R_1 \mid \forall E \in \mathrm{ent}(T),$$

$$\forall A \in \mathcal{A} : \exists (E, N) \in \mathrm{req}(A), \text{with } R = \mathrm{root}(\Gamma_E)\}$$

In case of a reference task, T_1 is the initial state of its unique entry. A layered task is a PEPA component which initially enables the activities corresponding to the invocations of all its entries, modeled as an initial choice process. When one of these activities is chosen, the process behaves as the initial state of the main flow of the execution graph corresponding to that entry.

An excerpt of the transformation of task *Server* is shown in Example 5.

Example 5. Translation of task *Server*.

$$\begin{aligned} Server_1 \stackrel{def}{=}\ & (req_{think,visit}, \nu).Visit_1 + (req_{think,buy}, \nu).Buy_1 \\ & + (req_{think,notify}, \nu).Notify_1 \\ & + (req_{think,save}, \nu).Save_1 \\ & \vdots \end{aligned}$$

$$\begin{aligned} Display_1 &\stackrel{def}{=} (get_{PServer}, \nu).(display, 1/0.001).Display_2 \\ Display_2 &\stackrel{def}{=} (rep_{think,buy}, \nu).Server_1 \\ Notify_1 &\stackrel{def}{=} (get_{PServer}, \nu).(notify, 1/0.08).Notify_2 \\ Notify_2 &\stackrel{def}{=} Server_1 \end{aligned}$$

The state $Server_1$ shows that all requests come from activity *think*. For a synchronous entry, e.g., *buy*, the final activity *display* ends with a reply action, as per Table 1. Instead, in the case of *notify* the component returns to state $Server_1$ without executing the single slice on *PServer*.

It is interesting to note that the overall definitions of a task describe a single thread of execution. When the process is not in state $Server_1$, none of the synchronizing request actions may be observed in the system. Therefore, other clients have to wait before accessing the task. Multithreading levels in PEPA are simply obtained by defining multiple copies of the same sequential component in the system equation, as discussed next.

3.4 Network

Template 6 Translation of an LQN.
Define for $T \in \mathcal{T}$, with $\mathrm{frk}(T) = \emptyset$:

$$C_T := T_1[N_T]$$

Otherwise, if $\mathrm{frk}(T) \neq \emptyset$, define

$$T_1[N_T] \stackrel{\bowtie}{L_1} \prod_{L_1} \big\{Fork_V[N_T] \mid V = \mathrm{root}(\Gamma_F^i), \text{ for all } F \in \mathrm{frk}(T), \text{ with } 1 \leq i \leq S_F\big\},$$

where $L_1 = * \setminus (\bigcup_{p \in \mathcal{P}} \{get_p\} \cup \{rep_{A,E} \mid \forall A, E\})$.

$$LQN: = \prod_{L_2} \{C_T \mid T \in \mathcal{T}\} \stackrel{\bowtie}{*} \prod_{\emptyset} \{P_1[M_P] \mid P \in \mathcal{P}\},$$

with $L_2 = * \setminus \bigcup_{p \in \mathcal{P}} \{get_p\}$.

The complete LQN is obtained according to Template 6. It is a composition between the tasks, defined by the compound processes C_T, and the processors, defined by the processes P_1.

Each C_T gives the overall behavior of a multientry multithreaded task with concurrency level N_T. The process T_1 subsumes the main control flow of all its entries, as discussed in Section 5. It may be composed of a number of other components with initial states $Fork_V$, each collecting the behavior of control flows arising from all vertices V which are immediate successors of a fork node in each execution graph within that task. These constants are defined as per Template 4. This is the general case of the situation illustrated in (1). Note that these flows inherit the concurrency levels of the tasks in which they are deployed.

Example 6. Translation of the LQN in Fig. 1.

$$C_{Client} := Client_1[2]$$
$$C_{Server} := Server_1[2] \underset{L_1}{\bowtie} Fork_{prepare}[2] \underset{L_1}{\bowtie} Fork_{ship}[2],$$
$$L_1 = \{fork_F, join_F\}$$
$$C_{FileServer} := FileServer_1[1]$$
$$C_{Backup} := Backup_1[1]$$
$$LQN := \left(C_{Client} \underset{L_2}{\bowtie} C_{Server} \underset{L_3}{\bowtie} C_{FileServer} \underset{L_4}{\bowtie} C_{Backup}\right) \underset{L_5}{\bowtie} \left(PClient[2] \underset{\emptyset}{\bowtie} PServer[2] \underset{\emptyset}{\bowtie} PDisk[2]\right),$$
$$L_2 = \{req_{think,visit}, rep_{think,visit}, req_{think,buy}, rep_{think,buy}, req_{think,notify}, req_{think,save}, rep_{think,save}, req_{think,read}\},$$
$$L_3 = \{rep_{think,read}, req_{external,read}, rep_{external,read}, rep_{save,write}, req_{save,write}\},$$
$$L_4 = \{rep_{write'',get}, rep_{write'',get}, rep_{write'',update}, req_{write'',update}\}$$
$$L_5 = \{think, cache, internal, external, prepare, pack, ship, display, read, write', write'', get, update\}.$$

It is important to recall that $*$ is a function of all the possible actions performed by the operands of the synchronization operator in which it appears, e.g.,

$$A \underset{*}{\bowtie} B \underset{*}{\bowtie} C \equiv A \underset{K_1}{\bowtie} B \underset{K_2}{\bowtie} C,$$

with

$$K_1 = \mathcal{A}ct(A) \cap \mathcal{A}ct(B), \qquad K_2 = \mathcal{A}ct(A \underset{K_1}{\bowtie} B) \cap \mathcal{A}ct(C).$$

Therefore, in general it is not the case that $K_1 \neq K_2$ (although, clearly, it holds that $K_2 \subseteq K_1$). In the action sets identified by L_1 and L_2, however, the actions get_P are excluded to avoid a form of *multiway synchronization* for the access to a processor whereby three or more constituting components perform the same action simultaneously. In other words, with respect to the model component LQN, each of the sequential components in the left-hand operand of $\underset{*}{\bowtie}$ are independent processes which compete for the shared processors, collectively represented as the right-hand operand. Finally, all processors are composed over empty synchronization sets because all the current actions must be independent—one activity runs on one and only one processor.

3.5 Properties of the Translation

This section is ended with a list of properties which are enjoyed by the translation patterns herein examined.

3.5.1 Multiway Synchronization for Fork/Join Actions

The definition of the cooperation set L_1 in Template 6 guarantees that the model does not feature undesired forms of synchronization. By the previous templates, the possible action types that any sequential component may perform are as follows:

- get_P, for some $P \in \mathcal{P}$; notice that for sequential components related to the same task, there is only one such action, namely, that related to the processor on which the task is deployed. Thus, if L_1 contained get_P, then a synchronization could arise between the main flow T_1 and one or more secondary flows $Fork_V$. This does not correspond to the intended behavior of the system, whereby the processor implements a kind of mutual exclusion amongst all competing flows.
- $rep_{A,E}$ for some activity A and entry E; the same action may be performed by two distinct flows, e.g., when one performs an early reply and the other one does not. As in the previous case, these actions are intended to be independent within the same task.
- $req_{A,E}$ for some activity A and entry E; these actions are all distinct because of the uniqueness of the originating activity within the overall network. Therefore, $req_{A,E}$ is executed by only one kind of sequential component in the overall PEPA model.
- A, with $A \in \mathcal{A}$; this case does not give rise to unintended synchronized behavior because the activity names are all distinct. A similar argument applies to the or_D actions arising from decision nodes and to the actions $req_{A,E}$ because at least A will be distinct in the components.
- $fork_F$ and $join_F$, with $F \in \mathcal{F}$; the action set L_1 does enforce synchronization between these actions. Each sequential component must synchronize with all flows such that the same $fork_F$ or $join_F$ may be executed. This is indeed a multiway synchronization because more than two flows may be spawned from a fork node.

3.5.2 Binary Communication for Requests and Replies

The action set L_2 in Template 6 does permit synchronization for the actions $req_{A,E}$ and $rep_{A,E}$ because activities within one task may make calls to activities in other tasks. It is possible to show that all such sets are pairwise disjoint, thereby proving that at most two components C_T synchronize, i.e., the communication is binary.

In order to do so, observe that, by construction, all the get_P action types are not contained in the set L_2. Any pair of components of type C_T does not exhibit the same action type for the execution of a basic activity since each activity belongs to only one task. The same fork/join action type cannot be exhibited because these activities are executed within the same task, and distinct fork/join nodes give rise to distinct action types in the PEPA model. Thus, the only potential elements of L_2 are the action types for message exchange $req_{A,E}$ and $rep_{A,E}$. The fact that sets with such action types are pairwise disjoint follows immediately from the uniqueness of activity and entry names in the LQN and

can be proven by structural induction. Let us consider an arbitrary composition of three components C_T, i.e.,

$$C_{T_1} \underset{L_2}{\bowtie} C_{T_2} \underset{L_2}{\bowtie} C_{T_3}.$$

Component C_{T1} may enable request/reply actions with subscripts (A', E'), $(A'', A''), \ldots$, where $E', E'', \ldots \in \mathrm{ent}(T_1)$ and $A, A', \ldots$ are basic activities. If some action with subscript (A, E) was present in both cooperation sets then it would mean that both C_{T_2} and C_{T_3} can perform the same basic activity A, which is a contradiction. Then, assuming that the property holds for a cooperation among $n > 3$ components:

$$C_{T_1} \underset{L_2}{\bowtie} C_{T_2} \underset{L_2}{\bowtie} C_{T_3} \underset{L_2}{\bowtie} \cdots \underset{L_2}{\bowtie} C_{T_n}$$

in order to prove that it holds for $n+1$ components:

$$C_{T_1} \underset{L_2}{\bowtie} C_{T_2} \underset{L_2}{\bowtie} C_{T_3} \underset{L_2}{\bowtie} \cdots \underset{L_2}{\bowtie} C_{T_n} \underset{L_2}{\bowtie} C_{T_{n+1}},$$

it suffices to prove that the cooperation set L_2 in position $\cdots C_{T_i} \underset{L_2}{\bowtie} C_{T_i+1} \cdots$ is disjoint from the cooperation set $\cdots C_{T_n} \underset{L_2}{\bowtie} C_{T_{n+1}}$, for all $1 \leq i \leq n-1$. Suppose that, for some $i, C_{Ti} \underset{L_2}{\bowtie} C_{Ti+1}$ has some action in common with the set in $C_{Tn} \underset{L_2}{\bowtie} C_{Tn+1}$. This implies that the action must be a request/reply action with subscript (A, E), with $E \in \mathrm{ent}(T_{n+1})$, because it belongs to the set $C_{T_n} \underset{L_2}{\bowtie} C_{T_{n+1}}$, and that $E \in \mathrm{ent}(T_{i+1})$, which is a contradiction because $i+1 \neq n+1$ but one entry must belong to only one task.

3.5.3 Properties of Rep_V

The expansion Rep_V in Table 1 uses constants which are not defined therein. However, observe that T_1 is always defined for every task by Template 5. The case $V \to V'$ implies the existence of a constant V'_1. In fact, the treatment of each node V is such that the initial state is always denoted by V_1. Finally, let us consider case $V \in \mathrm{leaves}(\Gamma^i_F)$ for some $F \in \mathcal{F}$ and $1 \leq i \leq S_F$. The target constant $Fork_{\mathrm{root}(\Gamma^i_F)}$ is well defined according to Template 4. Finally, the treatment of decision nodes is well defined by similar arguments to above.

3.5.4 Size of the PEPA Model

It is straightforward to see from an inspection of Templates 1-6 that the number of sequential components generated by the translation algorithm is linear with the number of activities and processors in the LQN model. Let us remark that this also applies to reference tasks; therefore each customer class adds a number of sequential components which is linear with the number of processors and activities that are needed to define that class. The size of nonreference tasks is not affected by the number of customer classes in the LQN.

The system of ODEs underlying a PEPA model is equal to the total number of sequential components defined. As discussed, task and processor multiplicities do not impact on the ODE system size, but only affect the actual starting point of the associated initial value problem.

3.6 Performance Measures

Another paper by this author and colleagues has discussed a class of performance indices, namely, *action throughput*, *capacity utilization*, and *average response time*, which can be formally defined as Markov reward structures that enjoy convergence in probability to a deterministic estimate [17]. The typical performance metrics for an LQN model can be shown to be encoded in PEPA using suitable definitions of action throughputs and average response times, as well as using the ODE solutions directly. This section provides sample performance indices for the running example. A generalization of these specifications to arbitrary LQNs is straightforward.

We use the following notation throughout this section: For each PEPA sequential component Q, we denote by $N(Q)$ the estimate of the mean number of components which exhibit the local derivative Q in the steady state. This is computed as an expected value if the analytical model is the Markov process underlying the PEPA model, or as one of the functions of the ODE problem in the fluid approximation. Similarly, the notation $f(N(Q))$ specifies a reward function f applied to the Markov process or to the fluid approximation.

3.6.1 Utilization

In the LQN model, utilization measures the mean number of busy processors at equilibrium. As such, it yields a value between zero and their multiplicity, although it may be similarly given a normalized fashion as the fraction of busy processors. More fine-grained results can be obtained by computing the distinct contributions from each of the activities. In the PEPA model, the overall utilization for a processor P, denoted by $U(P)$, is the mean number of components which are in state P_2 (cf., Template 1), i.e.,

$$U(P) := N(P_2).$$

In order to obtain the utilization due to a given activity $A \in \mathrm{act}(P)$, denoted by $U_A(P)$, it is necessary to compute the population levels of all the sequential components which perform execution slices of A as defined by the expansion Acq_A. Then, the processor utilization due to the execution of A is given as the sum across all such population levels. If an activity has two phases, the total contribution is the sum of the contributions of each phase. For instance, the utilization of processor *PDisk* due to the execution of *write* is obtained as

$$\begin{aligned} U_{write}(PDisk) = {} & N\big((write', 1/0.001).Write'_2\big) \\ & + N\big((write'', 3/0.04).Write''_2\big) \\ & + N\big((write'', 3/0.04).Write''_3\big) \\ & + N\big((write'', 3/0.04).Write''_4\big). \end{aligned}$$

It is worth noting that the estimation of processor utilization is directly obtainable from the components of the ODE solution and does not require the use of reward structures.

An analogous definition of utilization may be given to layered tasks. In this case, this is the mean number of tasks that are occupied in the steady state. For a layered task $T \in \mathcal{T}$ its utilization is written $U(T)$ and can be computed as

$$U(T) := N_T - N(T_1)$$

since the local state T_1 is intended to be the only idle state of the task. In other words, the task is assumed to be utilized even when it is waiting for service from one of the lower-layer calls.

3.6.2 Throughput

Throughput measures the frequency at which an activity is performed in the steady state. For an activity $A \in \mathcal{A}$, the throughput is computed in the PEPA model as the action throughput of the that action as defined in [17]. Specifically, due to the structure of an LQN model encoded in PEPA, it is possible to show that the throughput of $A \in \mathrm{act}(P)$, denoted by $T(A)$, can be computed as

$$T(A) := \mathrm{dem}(A)\ \min\left\{ \sum_{\substack{(B,s(B)).Q \\ B=A}} N\big((B,s(B)).Q\big), N(P_2) \right\}.$$

The summation across all sequential components which execute a slice of A captures the fact that all slice executions compete for the same processing resource, which is the right-hand side of the minimum function in the expression. For instance, it holds that

$$T(cache) = 1/0.001\ \min\left\{N((cache, 1/0.001).D_1), N(PServer)\right\}.$$

3.6.3 Average Response Time

Little's law is used for the evaluation of different performance metrics of an LQN model.

The *entry service time* is defined to be the average response time to execute an entry, including the service time incurred for the execution of lower layer calls. Therefore, it can be computed as the fraction of the population of sequential components which model the execution of the entry and the throughput of requests for that entry. For entries which feature an early reply, the sequential components which are associated with the second phase are not taken into account in this computation. For instance, the entry service time for *write*, denoted by $W(write)$, may be computed as

$$W(write) = \left\{N(Write'_1) + N\big((write', 1/0.001).Write'_2\big) + Write'_2\right\} \cdot \frac{1}{\nu\ \min\{N(FileServer_1), N(Save_2)\}},$$

where the expression in the fraction denominator gives the throughput of requests as a function of the number of callers, $N(Save_2)$, and available tasks, $N(FileServer_1)$.

The *mean delay for a join*, expressed as the time to observe a join after the corresponding fork has been executed, can be computed in a similar manner. The expression for a mean delay for a join corresponding to a fork $F \in \mathcal{F}$, denoted by $W(F)$, is

$$W(F) := \frac{N(J_1)\ \mathrm{dem}(F)^{-1}}{\min\big(N(F_1), \{N(Fork_V) \mid V = \mathrm{root}(\Gamma_F^i)\}\big)},$$

where $N(J_1)$ is the number of forking threads waiting for the join, as per Template 4.

TABLE 2
Sensitivity of Rate ν in the PEPA Model of Fig. 1

ν	$U(PServer)$	$U_{write}(PDisk)$	$W(visit)$
Reference values			
$\mathbf{1.2 \times 10^8}$	$\mathbf{1.4763}$	$\mathbf{0.3848}$	$\mathbf{0.00364}$
Relative differences			
1.2×10^4	1.6691%	1.6691%	5.2696%
1.2×10^5	0.1692%	0.1692%	0.5245%
1.2×10^6	0.0168%	0.0168%	0.0052%
1.2×10^7	0.0015%	0.0015%	0.0049%

First row: reference values. Other rows: relative differences with respect to first row.

4 Numerical Evaluation

This section studies the nature of the PEPA interpretation of an LQN model by means of an experimental assessment on the model in Fig. 1. Section 4.1 briefly discusses the treatment of the rate ν, which is used to approximate instantaneous service invocations, accesses to processing resources and tasks, and fork/join activities. Section 4.2 presents numerical results on the comparison between the fluid interpretation of the PEPA model and the approximation by AMVA.

4.1 Approximation of Instantaneous Activities

The instantaneous activities of an LQN model are approximated by means of a rate ν which is set to a much larger value than that of all other activities in the system. Table 2 shows the results of a sensitivity analysis on the fluid approximation conducted for different values of ν. The slowest rate considered, i.e., 1.2×10^4, is equal to 20 times the fastest individual rate in the LQN model (i.e., one slice execution of *external*). The performance metrics under observation are two utilization indices, $U(PServer)$ and $U_{write}(PDisk)$, and the average response time $W(visit)$. The results in the table are reported as the percentage relative differences with respect to the performance results of the model with $\nu = 1.2 \times 10^8$. This study has motivated the choice of 1.2×10^7 for all the tests discussed in the remainder of this section as this did not cause significant loss of accuracy up to the third decimal digit of each performance metric.

From a computational viewpoint, it should be pointed out that the presence of rates which are separated by many orders of magnitude is a very likely source of *stiffness* in the numerical integration of the underlying initial value problem of the fluid approximation [24]. In practice, however, the stiff solver `ode15s` from the Matlab ODE suite (cf., [25]) turned out to be successful in dealing with the models analyzed in this paper.

An alternative approach to dealing with the separation between fast and slow rates in the model would be to cast time-scale decomposition techniques for process algebra (cf., [26]) into the ODE analysis. This is, however, outside the scope of this paper.

4.2 Comparison of AMVA and Fluid Analysis

4.2.1 Experimental Set-Up

The comparison between the accuracy of the AMVA and that of the fluid approximation is based on a set of 1,000 models

TABLE 3
Comparison between the Fluid Approximation of PEPA (Rows Labeled with *ODE*) and the LQN Analytical Solver (Rows Labeled with *MVA*) Using 1,000 Randomly Generated Model Instances with Topology as in Fig. 1

	Metric	$U(PClient)$			$U(TBackup)$			$T(cache)$			$W(buy)$			*Maximum Error*		
		5-th	*Avg.*	95-th	5-th	*Avg.*	95-th	5-th	*Avg.*	95-th	5-th	*Avg.*	95-th	5-th	*Avg.*	95-th
1x	ODE	0.04	1.55	7.81	0.01	9.81	49.79	0.05	1.55	7.80	0.73	11.58	29.60	1.83	17.62	59.96
	MVA	0.60	16.13	51.44	0.78	29.38	84.98	0.64	16.13	51.43	0.35	18.05	57.82	4.77	37.54	85.62
2x	ODE	0.04	0.96	4.44	0.01	5.74	31.31	0.05	0.96	4.46	0.62	10.47	20.13	1.42	13.73	41.75
	MVA	0.73	18.91	62.29	0.91	30.29	89.46	0.72	18.91	62.29	0.27	18.97	62.09	4.55	39.74	89.75
5x	ODE	0.05	0.63	2.16	0.01	3.10	14.06	0.05	0.63	2.18	0.56	9.38	17.94	1.22	10.91	21.43
	MVA	1.18	25.10	74.87	1.09	35.37	93.37	1.12	25.09	74.87	0.28	22.00	75.53	4.89	47.26	93.45

Rows 1x: baseline validation set, with parameters set as discussed in Section 4.2.1; rows 2x and 5x: baseline set, with all task and processor multiplicities multiplied by 2 and 5, respectively. For each of the performance metrics the 5th and 95th percentile and the average error across all model instances are plotted.

with topology as in Fig. 1 and with randomly generated rate parameters and concurrency levels. Rate values were drawn from uniform distributions in $[0.01, 100]$; concurrency levels for processors were sampled from discrete uniform distributions in $1, \ldots, 10$, whereas tasks and client populations were sampled uniformly from $1, \ldots, 50$. The larger values for concurrency levels of tasks were motivated by the desire to obtain model instances which exhibit contention for shared processors.

The analysis of the LQN models was conducted using the *Layered Queueing Network Solver* toolkit [27]; the PEPA fluid analysis was performed using a combination of the *PEPA Eclipse Plugin* [28], for the automatic derivation of the underlying system of differential equations, and Matlab, for the numerical solution by the `ode15s` solver, as discussed above. The results obtained from the discrete-event simulation of the LQN model by the `lqsim` tool were taken to be the *exact* values. The simulations were stopped when the radii of all confidence intervals at 95 percent confidence level for the performance metrics under study were less than 1 percent of the statistical averages. The analytical solution by AMVA was obtained with the `lqns` tool enabling the option *stop-on-message-loss* to deal with the asynchronous requests at *Server*.[1] The ODE integration scheme was able to detect convergence to equilibrium based on two criteria: an absolute criterion computed as the norm of the derivatives at each integration step, and a relative one based on the norm of the difference between the solution vectors at two successive steps. In both cases, the convergence threshold was set to 1×10^{-6} for the L^1 norm.

The performance metrics on which the comparison is based are: $U(PClient)$, $U(TBackup)$, $T(cache)$, and $W(buy)$. In fact, the analyses were performed using a larger set of indices; however, a preprocessing phase showed that those reported herein are representative of analogous indices computed for other entities of the network. For instance, the error behavior for the processor utilizations of *PServer* and *PDisk* did not give statistically different results from those of *PClient*. Similar trends were observed among the error behaviors of throughputs and response times.

The accuracy of the approximation is quantified by means of the usual notion of percentage relative error with respect to the statistical mean obtained by simulation. Aggregate statistics across all model instances are given by the average error as well as the 5th and the 95th percentiles. The median error was also calculated but is not reported here to ease table layout and reading. It was found to be of the same order of magnitude as the average but consistently smaller, indicating that the error distribution is shifted toward higher accuracy.

4.2.2 Results

The aggregate error statistics are collectively reported in Table 3. The two rows labeled with *1x* refer to the randomly generated validation dataset discussed in the previous section. Rows labeled with *2x* and *5x* refer to the same validation dataset, where all rate parameters are kept fixed but each model instance is initialized with concurrency levels for processors and tasks which are two and five times larger, respectively, than the original random values. This respects the kind of scaling that is used in the framework of asymptotic convergence for PEPA [14] and allows us to assess the accuracy behavior of the approximation with respect to increasing sizes.

The results clearly show that different kinds of indices yield different qualities of the approximation, with processor utilizations and throughputs being generally more accurate than task utilizations and response times. A possible explanation for the latter may be a form of *error propagation* due to the fact that steady-state average response time is a *derived* measure. It is calculated by Little's law as the fraction of the number of users within some suitable subset of local states and the related throughput, which are both approximate estimates themselves. In the context of PEPA, this phenomenon has already been observed [17].

Let us focus first on the *1x* case. The error statistics regarding the ODE approximation are consistently smaller than those for the MVA approximation. The accuracy is generally acceptable, with percentage errors of mostly about 10 percent on average, although some particularly inconvenient cases yield unsatisfactory accuracy, e.g., over 40 percent for the 95th error percentile of $U(TBackup)$. The

1. The use of a nonstandard solver option may raise the question whether potential sources of inaccuracy may come from the treatment of asynchronous messages. To investigate this issue, an analogous numerical assessment was conducted on a slight variation of the model in Fig. 1 which replaces the asynchronous call with a synchronous one. The numerical results turned out to be not appreciably different, therefore they are not presented here for the sake of conciseness.

TABLE 4
Average Runtimes for the Fluid Analysis and the MVA Approximation of the Validation Dataset

Method	*ODE*	*MVA 1x*	*MVA 2x*	*MVA 5x*
Runtime (s)	12.20	0.58	5.31	37.24

For a given model, the solution of differential equations is independent of the scaling levels.

error behavior is more problematic for the MVA approximation, where the averages are often sensibly larger than the ODE counterparts—for instance, it is one order of magnitude larger for $U(PClient)$—and with 95 percent percentiles as high as about 85 percent.

The remaining rows of the table highlight a monotonic decrease of the error statistics for the ODE approximation as a function of the population sizes in the model. This is expected as per fluid limits theory, which guarantees uniform convergence in probability of a sample path of the stochastic process to the differential trajectory. Importantly, a similar trend could not be observed in the error behavior of MVA. In fact, the results show that in general the accuracy degrades with larger population levels. In particular, the errors for the *5x* case are on average larger than 25 percent across all the performance indices considered in this study.

4.2.3 Runtime Comparison

The computational cost of both techniques was also compared. This was done by recording the execution runtimes of the software tools employed for the analysis. For statistical significance, the average execution times over 30 independent runs were used for the comparison.

A completely fair evaluation is mainly hindered by the fact that the solution of the fluid model gives a time-course trajectory of the system's behavior as a byproduct of the numerical integration until equilibrium of the initial value problem. In this respect, the analysis is more informative in that it readily yields transient characteristics, unlike AMVA which iterates for a steady-state solution. On a more practical level, the runtime performance is also heavily dependent on the relative quality of the implementations of the supporting software tools. In order to allow for the most objective comparison possible, the two implementations were treated as *black boxes*. In particular, it was decided not to apply some optimizations on the fluid analysis, despite the availability of the source code of the toolkit for PEPA.

Table 4 shows the average runtimes, measured on an ordinary desktop computer, across the validation dataset of 1,000 models used in this section. Because of the specific scaling of the concurrency levels adopted herein, for a given model in the baseline dataset its fluid model is the same as that in the corresponding *2x* and *5x* model. Instead, the runtimes for AMVA exhibit some dependency on the initial population levels due to an increasingly larger number of iterations needed to reach convergence to a fixed point.

5 Conclusion

This paper has studied the performance evaluation of models of computer systems described as layered queueing networks, using a scalable fluid-limit approach mediated by an automatic translation into the stochastic process algebra PEPA. The underlying mathematical representation is a system of ordinary differential equations which can be easily solved by means of standard numerical integrators. Similarly to approximate mean value analysis, the solution may be interpreted as a deterministic estimate of the expected behavior of the stochastic process. On a side note, we remark that this intertwinement between queueing networks and process algebra goes beyond a more conventional view which has considered the two methods as being complementary but somewhat antithetic—the former being amenable to fast analytical solutions, the latter being endowed with a more expressive semantics, at the cost of a more expensive computational effort [29].

The use of fluid models inspired by process algebra is proposed here to significantly improve the accuracy of the LQN approximation. This was demonstrated by means of a numerical assessment over a large number of randomly generated models, showing that the error statistics were consistently superior to those referred to the AMVA methods available for LQNs. These findings confirm early observations on the same subject [22], [23] as well as recent work by the same author on a restricted subclass of LQNs on which numerical validation was also carried out by means of stress tests [30].

More important is the study of the error behavior with increasing problem sizes. Defining the *system size* N as a multiple of some given configuration of an LQN—i.e., the multiplicities of processors and tasks, and the client populations—the fluid trajectory is proven to be indistinguishable from a sample path of the stochastic process as $N \to \infty$. An implication of this property is that the larger the N, the smaller the error may be expected to be. Although theoretical error bounds are not practically usable at present due to a doubly exponential dependence upon time [31], this trend is confirmed more pragmatically by the numerical study presented in this paper and, most crucially, it strikingly contrasts the error behavior for AMVA, which is instead at best (statistically) insensitive with respect to increasing N, if not degrading. It is worth noting that, even with small job populations (at most 250 across all models), fluid analysis is capable of producing very accurate estimates, especially of utilizations and throughputs.

The price to pay for such increased accuracy appears to be a longer execution time on average for small-sized models. Although fluid approximations behaved quantitatively well in these specific circumstances, it is important to stress that those are not the models which are intended to be the primary target of this form of analysis, as it inherently requires large populations to yield accurate results. We observe, however, that the computational cost becomes increasingly more convenient with larger population sizes, to which the solution of the system of differential equations is insensitive as they only affect the initial condition of the numerical integration. On the other hand, we registered average runtimes which span two orders of magnitude for AMVA.

Although the focus of this paper was on comparing the approximation errors of average performance indices in the

steady state, we note that fluid techniques enable further scalable forms of analysis which are not available within a mean value analysis framework. It has already been discussed that the solution to the system of differential equations directly provides transient measures, i.e., performance indices across the whole time-course evolution of the system. These can be used on their own (cf., [17]) or to compute *cumulative rewards* (i.e., time-averaged integrals of a performance index, e.g., [32]) and passage-time distributions [33].

REFERENCES

[1] P.A. Jacobson and E.D. Lazowska, "The Method of Surrogate Delays: Simultaneous Resource Possession in Analytic Models of Computer Systems," *SIGMETRICS Performance Evaluation Rev.,* vol. 10, pp. 165-174, Sept. 1981.

[2] C.M. Woodside, "Throughput Calculation for Basic Stochastic Rendezvous Networks," *Performance Evaluation,* vol. 9, no. 2, pp. 143-160, 1989.

[3] J.A. Rolia and K.C. Sevcik, "The Method of Layers," *IEEE Trans. Software Eng.,* vol. 21, no. 8, pp. 689-700, Aug. 1995.

[4] G. Franks, T. Omari, C.M. Woodside, O. Das, and S. Derisavi, "Enhanced Modeling and Solution of Layered Queueing Networks," *IEEE Trans. Software Eng.,* vol. 35, no. 2, pp. 148-161, Mar. 2009.

[5] M. Reiser and S.S. Lavenberg, "Mean-Value Analysis of Closed Multichain Queuing Networks," *J. ACM,* vol. 27, no. 2, pp. 313-322, 1980.

[6] Y. Bard, "Some Extensions to Multiclass Queueing Network Analysis," *Proc. Third Int'l Symp. Modelling and Performance Evaluation of Computer Systems,* pp. 51-62, 1979.

[7] K.M. Chandy and D. Neuse, "Linearizer: A Heuristic Algorithm for Queueing Network Models of Computing Systems," *Comm. ACM,* vol. 25, no. 2, pp. 126-134, 1982.

[8] P. Schweitzer, "Approximate Analysis of Multiclass Closed Networks of Queues," *Proc. Int'l Conf. Stochastic Control and Optimization,* pp. 25-29, June 1979.

[9] K.R. Pattipati, M.M. Kostreva, and J.L. Teele, "Approximate Mean Value Analysis Algorithms for Queuing Networks: Existence, Uniqueness, and Convergence Results," *J. ACM,* vol. 37, no. 3, pp. 643-673, 1990.

[10] M. Woodside, J. Neilson, D. Petriu, and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software," *IEEE Trans. Computers,* vol. 44, no. 1, pp. 20-34, Jan. 1995.

[11] S. Ramesh and H. Perros, "A Multilayer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages," *IEEE Trans. Software Eng.,* vol. 26, no. 11, pp. 1086-1100, Nov. 2000.

[12] D.A. Menascé, "Simple Analytic Modeling of Software Contention," *SIGMETRICS Performance Evaluation Rev.,* vol. 29, no. 4, pp. 24-30, Mar. 2002.

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

[14] M. Tribastone, S. Gilmore, and J. Hillston, "Scalable Differential Analysis of Process Algebra Models," *IEEE Trans. Software Eng.,* vol. 38, no. 1, pp. 205-219, Jan./Feb. 2012.

[15] R.A. Hayden and J.T. Bradley, "A Fluid Analysis Framework for a Markovian Process Algebra," *Theoretical Computer Science,* vol. 411, nos. 22-24, pp. 2260-2297, 2010.

[16] T.G. Kurtz, "Solutions of Ordinary Differential Equations as Limits of Pure Markov Processes," *J. Applied Probability,* vol. 7, no. 1, pp. 49-58, Apr. 1970.

[17] M. Tribastone, J. Ding, S. Gilmore, and J. Hillston, "Fluid Rewards for a Stochastic Process Algebra," *IEEE Trans. Software Eng.,* vol. 38, no. 4, pp. 861-874, July/Aug. 2012.

[18] J. Hillston, "Exploiting Structure in Solution: Decomposing Compositional Models," *Proc. Sixth Int'l Workshop Process Algebra and Performance Modelling,* pp. 1-15, 1998.

[19] J. Hillston and N. Thomas, "Product form Solution for a Class of PEPA Models," *Performance Evaluation,* vol. 35, nos. 3/4, pp. 171-192, 1999.

[20] P.G. Harrison, "Turning Back Time in Markovian Process Algebra," *Theoretical Computer Science,* vol. 290, no. 3, pp. 1947-1986, 2003.

[21] N. Thomas and Y. Zhao, "Mean Value Analysis for a Class of PEPA Models," *Computer J.,* 2010.

[22] M. Tribastone, "Relating Layered Queueing Networks and Process Algebra Models," *Proc. First Joint WOSP/SIPEW Int'l Conf. Performance Eng.,* pp. 183-194, 2010.

[23] M. Tribastone, "Scalable Analysis of Stochastic Process Algebra Models," PhD dissertation, School of Informatics, The Univ. of Edinburgh, 2010.

[24] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems.* Springer, 1996.

[25] MATLAB, *Version 7.10.0 (R2010a),* The MathWorks Inc., 2010.

[26] J. Hillston and V. Mertsiotakis, "A Simple Time Scale Decomposition Technique for Stochastic Process Algebras," *Computer J.,* vol. 38, no. 7, pp. 566-577, 1995.

[27] Real-Time and Distributed Systems Group, Dept. of Systems and Computer Eng., Univ. of Carleton, "LQNS Software Package," http://www.sce.carleton.ca/rads/lqns, 2012.

[28] M. Tribastone, A. Duguid, and S. Gilmore, "The PEPA Eclipse Plug-In," *SIGMETRICS Performance Evaluation Rev.,* vol. 36, no. 4, pp. 28-33, Mar. 2009.

[29] U. Herzog and J.A. Rolia, "Performance Validation Tools for Software/Hardware Systems," *Performance Evaluation,* vol. 45, nos. 2/3, pp. 125-146, 2001.

[30] M. Tribastone, "Approximate Mean Value Analysis of Process Algebra Models," *Proc. 19th IEEE Int'l Symp. Modelling, Analysis and Simulation of Computer and Telecomm. Systems,* pp. 369-378, July 2011.

[31] R. Darling and J. Norris, "Differential Equation Approximations for Markov Chains," *Probability Surveys,* vol. 5, pp. 37-79, 2008.

[32] J. Meyer, "On Evaluating the Performability of Degradable Computing Systems," *IEEE Trans. Computers,* vol. 29, no. 8, pp. 720-731, Aug. 1980.

[33] R.A. Hayden, A. Stefanek, and J.T. Bradley, "Fluid Computation of Passage-Time Distributions in Large Markov Models," *Theoretical Computer Science,* vol. 413, no. 1, pp. 106-141, 2012.

Mirco Tribastone is an assistant professor at the Department of Informatics of Ludwig-Maximilians University of Munich. His principal interests include the quantitative evaluation of computer systems using analytical models. He is the lead developer of the *PEPA Eclipse Plug-In Project* which supports a range of qualitative and quantitative analysis methods for PEPA.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.