Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis

Simonetta Balsamo
Dipartimento di Informatica
Univ. "Ca' Foscari" di Venezia
balsamo@dsi.unive.it

Marco Bernardo
Centro per l'Appl. delle S.T.I.
Univ. di Urbino
bernardo@sti.uniurb.it

Marta Simeoni
Dipartimento di Informatica
Univ. "Ca' Foscari" di Venezia
simeoni@dsi.unive.it

ABSTRACT

We propose an integrated approach to the functional and performance analysis of Software Architectures (SAs) based on Stochastic Process Algebras (SPAs) and Queueing Networks (QNs), in order to combine their main advantages: formal techniques for the verification of functional properties of systems for SPAs, and efficient performance analysis for QNs. We first introduce Æmilia, a SPA based architectural description language for the compositional, graphical and hierarchical modeling of SAs, which is equipped with suitable checks for the detection of architectural mismatches. Then we present a systematic approach to derive QN models from Æmilia specifications. This is based on the identification of three different classes of QN basic elements – arrival processes, buffers, and service processes – and on syntactic restrictions to be imposed to Æmilia specifications, so that each architectural component directly falls into one of the three classes. Although performance analysis could be carried out directly on the Markov chain (MC) underlying an Æmilia specification, having a QN model allows performance indices to be evaluated possibly by exact product form solutions or by well known approximate methods. Furthermore, unlike the underlying MC, the high level of abstraction of the QN model should ease the interpretation of the performance results at the architectural description level.

1. INTRODUCTION

Software Architecture (SA) is an emerging field within software engineering aiming at describing both the structure and the behavior of software systems at a high level of abstraction [18, 17]. The static and behavioral descriptions characterize, at an early stage of development, the basic design choices on the system under consideration, which clearly influence the subsequent development and deployment phases. Appropriate languages and tools are then required, in order to support SA with a suitable formalization of the architecture descriptions and the automatic analysis of their functional properties.

There is moreover a growing interest in quantitative analysis of software systems, and it has been recognized in the last years that performance analysis should be integrated in the software development life cycle since the early stages (see, e.g., [20, 21]). In particular, SAs have been devised as the appropriate design level to conduct early predictive performance analysis, thus allowing for a choice among alternative architectures on the basis of quantitative aspects.

In this paper we propose an integrated approach to the functional and performance analysis of SAs based on Stochastic Process Algebras (SPAs) and Queueing Networks (QNs). The idea is to combine the main advantages of the two frameworks: formal techniques for system specification and verification of functional properties for SPAs, and efficient performance analysis for QNs.

SPA (see, e.g., [11, 10, 6]) is a well known formal specification technique for concurrent and distributed systems. Its main features, i.e. composability – which allows system descriptions to be built in a modular and hierarchical way – and abstraction – which allows the internal details of a system description to be hidden at analysis time – make SPA suited to work with at the architectural level of design. Compared to classical process algebra (see, e.g., [16]), besides the purely functional aspects with SPA it is possible to express activity durations by using random variables. In addition to functional verification (e.g. via model checking [7]), this permits the quantitative analysis of the modeled system through the construction and solution of the underlying stochastic process.

QNs (see, e.g., [15, 14, 13]) have been widely applied as system performance models. Classical QNs represent resource sharing systems and can be solved by efficient algorithms, which do not require the construction of the underlying stochastic process. Moreover, extensions of classical QNs have been introduced in order to represent other interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queue and memory constraints, and simultaneous resource possession. Some approximate solution techniques have been defined for several types of extended QNs.

The presentation of our integrated approach starts with the introduction of Æmilia, an architectural description language (ADL) based on an expressive SPA called EMPA$_{\mathrm{gr}}$ [6]. Æmilia provides a formal framework for the compositional, graphical, and hierarchical modeling of software systems, which is equipped with some checks inspired by [3, 4] for the detection of possible architectural mismatches. The timing of the durational actions is mainly expressed by exponential random variables, so that the underlying stochastic process yields a Markov Chain (MC). Although performance analysis could be carried on directly by Æmilia, this requires the construction and solution of the underlying MC, whose state space explosion soon makes the analysis unfeasible. Moreover the MC is a flat model that does not reflect the structure of the specified SA, thus hampering the interpretation of the performance analysis results at the architectural description level.

In order to overcome these two drawbacks related to the efficient evaluation of performance indices and with the possibility of getting some feedback at the architectural description level in case of poor performance, we propose an approach to derive QN models from Æmilia specifications. More precisely, we define a mapping from Æmilia to QNs based on the identification of three different classes of QN basic elements – arrival processes, buffers, and service processes – and on some restrictions to the Æmilia specifications, so that each architectural component directly falls into one of the three classes. Such restrictions can easily be verified at the syntax level. Our translation of Æmilia specifications leads to classical open or closed QNs, with general (or phase-type) arrival and service time distributions and FIFO buffers with finite or infinite capacity. This allows us to apply efficient performance evaluation methods possibly by exact product form solutions or by well known approximate methods. Furthermore, the high level of abstraction of the QN model should ease the interpretation of the performance results at the architectural description level.

We illustrate an example of application of the combined approach to a compiler system. More precisely, we consider three different SAs allowing the compilation phases to work sequentially, in a pipeline fashion, and in parallel, respectively. We present their Æmilia graphical representations and formal specifications, and the mapping to the corresponding QN models. The MC models associated with the Æmilia specification of these three SAs are infinite state and require numeric approximate solutions. On the other hand, the translation of the Æmilia specifications leads to simple QN models admitting either product form or approximate solutions. The SAs modeling the three variants of the compiler system have been already presented in [1], where the authors propose a methodology to derive a QN model from a labeled transition system describing the behavior of a SA. The QN models obtained for the three considered SAs by using the methodology in [1] and the ones obtained by using our mapping coincide. Using transition systems as a SA formal specification model allows the independence from any specific architectural description language. In our approach, however, by considering Æmilia as specification language we can define a direct mapping to QNs that avoids the explicit construction of the underlying transition system.

The paper is organized as follows. In Sect. 2 we introduce syntax, semantics, and architectural checks for Æmilia. In Sect. 3 we describe a QN semantics for Æmilia by providing a mapping from Æmilia specifications to QN models. The example about the pipeline and concurrent SAs for the compiler system are described in Sect. 4 by providing their Æmilia graphical representations and formal specifications and by showing their corresponding QNs. Concluding remarks about related and future work are given in Sect. 5.

2. ÆMILIA: A SPA BASED ADL

In this section we introduce the syntax, the semantics, and the architectural checks for Æmilia, a performance oriented ADL. Æmilia is the result of the integration of two formalisms: PADL [3, 4] and EMPA_{gr} [6]. [1] The former is a process algebra based ADL equipped with some architectural checks for the detection of deadlock related architectural mismatches. The latter is an expressive process algebra allowing for the performance modeling and analysis of concurrent and distributed systems.

In essence, with Æmilia the description of a complex system can be built compositionally and hierachically through a graphical support. First, we have to define the behavior of the types of components in the system and their interactions with the other components. The functional and performance aspects of the behavior are described through a family of EMPA_{gr} terms, while the interactions are described through actions occurring in such terms. Then, we have to declare the instances of each type of component present in the system and the way in which their interactions are attached to each other in order to allow the instances to communicate. For the sake of ease, this can be accomplished with the support of a flow graph [16], where the nodes represent the component instances and the links represent the interaction attachments. Whenever some component is in turn made out of several subcomponents, the procedure is simply repeated at the subcomponent level. Afterwards, the consistency of the attachments w.r.t. deadlock freedom (in case the building components are deadlock free) and the completeness of the performance related information is verified through some architectural checks. Finally, the whole behavior of the Æmilia description is a family of EMPA_{gr} terms automatically obtained by composing in parallel the behavior of the declared instances according to the specified attachments. From the whole behavior, state transition based models can be automatically derived on which functional verification and performance evaluation can be carried out.

2.1 Syntax

Æmilia is based on the stochastic process algebra EMPA_{gr}, whose syntax is briefly recalled below.

Definition 2.1. *The set of terms of EMPA_{gr} is generated by the following syntax*

$$E ::= \underline{0} \mid <a, \tilde{\lambda}>.E \mid E/L \mid E[\varphi] \mid E + E \mid E \parallel_S E \mid A$$

where a belongs to a set AType of action types including a distinguished action τ for unobservable activities, $\tilde{\lambda}$ belongs to a set ARate of action rates including generative exponential rates $\lambda \in \mathbb{R}_+$, immediate rates of the form $\infty_{l,w}$ where $l \in \mathbb{N}_+$ is a generative priority level and $w \in \mathbb{R}_+$ is a generative weight, and passive rates of the form $*_{l,w}$ where $l \in \mathbb{N}_+$ is a reactive priority level and $w \in \mathbb{R}_+$ is a reactive weight, [2] *$L, S \subseteq AType - \{\tau\}$, φ belongs to a set ATRFun of action type relabeling functions preserving observability (i.e., $\varphi^{-1}(\tau) = \{\tau\}$), and A belongs to a set Const of constants each possessing a (possibly recursive) defining equation of the form $A \stackrel{\Delta}{=} E$. We denote by Act the set of actions and by $\mathcal{G}$ the set of closed and guarded terms.* ■

[1] An earlier, not fully integrated version of Æmilia, called ÆMPA, can be found in [2].

[2] If omitted, the value of a priority level or weight is intended to be 1.

In the syntax above, "$\underline{0}$" is the term that cannot execute any action. Term $<a, \tilde{\lambda}>.E$ can execute action $<a, \tilde{\lambda}>$ and then behaves as term E. Term E/L behaves as term E with each executed action $<a, \tilde{\lambda}>$ turned into $<\tau, \tilde{\lambda}>$ whenever $a \in L$. Term $E[\varphi]$ behaves as term E with each executed action $<a, \tilde{\lambda}>$ turned into $<\varphi(a), \tilde{\lambda}>$. Term $E_1 + E_2$ behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed. If the choice involves exponentially timed actions, the race policy applies and each involved action is selected with a probability proportional to its rate. If the choice involves immediate actions, they take precedence over exponentially timed ones and the generative preselection policy applies: each involved immediate action with the highest priority level is selected with a probability proportional to its weight. If the choice involves passive actions, the reactive preselection policy applies: for every action type, each involved passive action of that type with the highest priority level is selected with a probability proportional to its weight (the choice among involved passive actions of different types is nondeterministic). Term $E_1 \parallel_S E_2$ asynchronously executes actions of E_1 or E_2 not belonging to S and synchronously executes equal actions of E_1 and E_2 belonging to S provided that one of them is passive. In case of synchronization, the resulting action has the same type as the two original actions, while its rate is given by the rate of the original nonpassive action multiplied by the reactive execution probability of the original passive action (or is passive in case of synchronization of two passive actions). The action prefix operator and the alternative composition operator are called dynamic operators, whereas the hiding operator, the relabeling operator, and the parallel composition operator are called static operators. A term is called sequential if it is composed of dynamic operators only.

A description in Æmilia represents an architectural type. The description of an architectural type starts with the name of the architectural type and its numeric parameters, which often are values for exponential rates and weights. Each architectural type is defined as a function of its architectural element types (AETs) and its architectural topology. An AET is defined as a function of its behavior, specified either as a family of EMPA$_{\rm gr}$ sequential terms or through an invocation of a previously defined architectural type, and its interactions, specified as a set of EMPA$_{\rm gr}$ action types occurring in the behavior that act as interfaces for the AET. The architectural topology is specified through the declaration of a set of architectural element instances (AEIs) representing the system components, a set of architectural (as opposed to local) interactions given by some interactions of the AEIs that act as interfaces for the whole architectural type, and a set of directed architectural attachments among the interactions of the AEIs. Every interaction is declared to be an input interaction or an output interaction and the attachments must respect such a classification: every attachment must involve an output interaction and an input interaction of two different AEIs. An AEI can have different types of interactions (input/output, local/architectural); it must have at least one local interaction. Every local interaction must be involved in at least one attachment, while every architectural interaction must not be involved in any attachment. In order to allow several AEIs to synchronize, every local interaction can be involved in several attachments provided that no autosynchronization arises, i.e. no chain of attachments is created that starts from a local interaction of an AEI and terminates on a local interaction of the same AEI. On the performance side, we have two additional requirements. For the sake of modeling consistency, all the occurrences of an action type in the behavior of an AET must have the same kind of rate (exponential, or immediate with the same priority level, or passive with the same priority level). In order to comply with the synchronization discipline of EMPA$_{\rm gr}$, every chain of attachments must contain at most one interaction whose associated rate is exponential or immediate.

We show in Table 1 an Æmilia textual description for an architectural type representing a compiler system. We assume that programs to be compiled arrive according to a Poisson process, that the compiler examines one source program at a time taken from a FIFO buffer, that every compilation phase (lexical analysis, parsing, type checking, optimization, code generation) has an exponentially distributed duration, and that the optimization is carried out with a certain probability. The same compiler system is depicted in Fig. 1 through the Æmilia graphical notation. In a flow graph, the boxes denote the AEIs, the black circles denote the local interactions, the white squares denote the architectural interactions, and the directed edges denote the attachments.

2.2 Translation Semantics

The semantics of an Æmilia specification is given by translation into EMPA$_{\rm gr}$. Since the semantics for EMPA$_{\rm gr}$ yields state transition based models, an Æmilia specification is eventually equipped with such models, which can be used to assess functional properties, via model checking [7] and equivalence/preorder checking [8], as well as performance properties, via Markovian/simulation analysis, with the tool TwoTowers [5]. The semantics of $E \in \mathcal{G}$ is a state transition graph $\mathcal{I}[\![E]\!]$ called the integrated semantic model, whose states are represented by terms and whose transitions are labeled with actions. After pruning the lower priority transitions from $\mathcal{I}[\![E]\!]$, it is possible to derive a functional semantic model $\mathcal{F}[\![E]\!]$ by removing action rates from the transitions, and a performance semantic model $\mathcal{M}[\![E]\!]$ by essentially removing action types from the transitions. $\mathcal{M}[\![E]\!]$, which is defined only if $\mathcal{I}[\![E]\!]$ has no passive transitions, is a continuous time or a discrete time MC depending on whether $\mathcal{I}[\![E]\!]$ has exponentially timed transitions or not.

This section provides the detailed definition of the translation semantics from Æmilia to EMPA$_{\rm gr}$; it can be safely skipped by the reader interested in the queueing network semantics only. The translation semantics proceeds in two steps. In the first step, the semantics of all the instances of each AET is defined to be the behavior of the AET projected onto its interactions. Such a projected behavior is obtained from the family of sequential EMPA$_{\rm gr}$ terms representing the behavior of the AET by applying a hiding operator on all the actions that are not interactions. In this way, we abstract from all the internal details of the behavior of the instances of the AET.

DEFINITION 2.2. *Let $\mathcal{C}$ be an AET with behavior E and interaction set $\mathcal{IS}$. The semantics of $\mathcal{C}$ and its instances is defined by $[\![\mathcal{C}]\!] = E/(AType - \{\tau\} - \mathcal{IS})$.* ■

For our compiler system example we have

$$[\![ProgramGeneratorT]\!] = [\![PG]\!] = ProgramGenerator/ \{generate\_program\}$$

archi_type	$\mathit{CompilerSystem}(\textbf{exp\_rate}\ \lambda_{prog}, \mu_{lexer}, \mu_{parser}, \mu_{checker}, \mu_{optimizer}, \mu_{generator};$ $\textbf{weight}\ \mathit{prob}_{optimizer})$
archi_elem_types	
elem_type	$\mathit{ProgramGeneratorT}(\textbf{exp\_rate}\ \lambda)$
behavior	$\mathit{ProgramGenerator} \stackrel{\Delta}{=} {<}\mathit{generate\_program}, \lambda{>}.$ ${<}\mathit{deliver\_program}, \infty{>}.\mathit{ProgramGenerator}$
interactions	**output** *deliver_program*
elem_type	$\mathit{ProgramBufferT}$
behavior	$\mathit{ProgramBuffer}_0 \stackrel{\Delta}{=} {<}\mathit{get\_program}, *{>}.\mathit{ProgramBuffer}_1$
	$\mathit{ProgramBuffer}_i \stackrel{\Delta}{=} {<}\mathit{get\_program}, *{>}.\mathit{ProgramBuffer}_{i+1} +$ ${<}\mathit{put\_program}, *{>}.\mathit{ProgramBuffer}_{i-1} \qquad i \geq 1$
interactions	**input** *get_program*
	output *put_program*
elem_type	$\mathit{CompilerT}(\textbf{exp\_rate}\ \mu_l, \mu_p, \mu_c, \mu_o, \mu_g;$ $\textbf{weight}\ p_o)$
behavior	$\mathit{Compiler} \stackrel{\Delta}{=} {<}\mathit{select\_program}, \infty{>}.{<}\mathit{recognize\_tokens}, \mu_l{>}.{<}\mathit{parse\_phrases}, \mu_p{>}.$ ${<}\mathit{check\_phrases}, \mu_c{>}.({<}\mathit{opt\_yes}, \infty_{1,p_o}{>}.{<}\mathit{optimize}, \mu_o{>}.$ ${<}\mathit{generate\_code}, \mu_g{>}.\mathit{Compiler} +$ ${<}\mathit{opt\_no}, \infty_{1,1-p_o}{>}.$ ${<}\mathit{generate\_code}, \mu_g{>}.\mathit{Compiler})$
interactions	**input** *select_program*
archi_topology	
archi_elem_instances	$\mathit{PG} : \mathit{ProgramGeneratorT}(\lambda_{prog})$
	$\mathit{PB} : \mathit{ProgramBufferT}$
	$C : \mathit{CompilerT}(\mu_{lexer}, \mu_{parser}, \mu_{checker}, \mu_{optimizer}, \mu_{generator}; \mathit{prob}_{optimizer})$
archi_interactions	
archi_attachments	**from** $\mathit{PG}.\mathit{deliver\_program}$ **to** $\mathit{PB}.\mathit{get\_program}$
	from $\mathit{PB}.\mathit{put\_program}$ **to** $C.\mathit{select\_program}$
end	

Table 1: Textual description of *CompilerSystem*

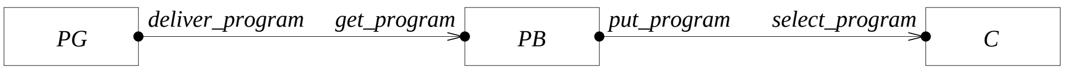


Figure 1: Flow graph of *CompilerSystem*

$$\begin{aligned}
[\![\mathit{ProgramBufferT}]\!] &= [\![\mathit{PB}]\!] = \mathit{ProgramBuffer}_0 \\
[\![\mathit{CompilerT}]\!] &= [\![C]\!] = \mathit{Compiler}/ \\
&\quad \{\mathit{recognize\_tokens}, \mathit{parse\_phrases}, \mathit{check\_phrases}, \mathit{opt\_yes}, \mathit{opt\_no}, \mathit{optimize}, \mathit{generate\_code}\}
\end{aligned}$$

In the second step, the semantics of an architectural type is obtained by composing in parallel the semantics of its AEIs according to the specified attachments, possibly after relabeling to the same type the interactions whose types are involved in the same chain of attachments. For our compiler system example we have

$$\begin{aligned}
[\![\mathit{CompilerSystem}]\!] = {} & [\![\mathit{PG}]\!][\mathit{deliver\_program} \mapsto a] \parallel_{\{a\}} \\
& [\![\mathit{PB}]\!][\mathit{get\_program} \mapsto a, \\
& \qquad \mathit{put\_program} \mapsto b] \parallel_{\{b\}} \\
& [\![C]\!][\mathit{select\_program} \mapsto b]
\end{aligned}$$

In order to define the semantics of an arbitrary architectural type, first we have to determine the number of fresh action types that we need in order to make the AEIs interact according to the attachments. To achieve that, we have to single out all the chains of attachments, as each of them corresponds to a maximal set of synchronizing interactions, all of which must therefore be relabeled to the same fresh action. Given an architectural type $\mathcal{A}$, let $C_1, \ldots, C_n$ be some of its AEIs and let i, j, k range over $\{1, \ldots, n\}$. For each AEI C_i, let $\mathcal{IS}_{C_i}$ be the set of its interactions, $\mathcal{AI}_{C_i} \subseteq \mathcal{IS}_{C_i}$ be the set of its interactions declared as being architectural, and $\mathcal{LI}_{C_i;C_1,\ldots,C_n} \subseteq \mathcal{IS}_{C_i} - \mathcal{AI}_{C_i}$ be the set of its local interactions attached to local interactions of $C_1, \ldots, C_n$. We say that a set $\mathcal{LI}$ of local interactions of $C_1, \ldots, C_n$ is connected if it is maximal w.r.t. the following property: for each pair $(C_i.a_1, C_j.a_2)$ of interactions of $\mathcal{LI}$, either there is an attachment between them, or there exists an interaction $C_k.a_3$ of $\mathcal{LI}$ such that there is an attachment between

$C_i.a_1$ and $C_k.a_3$ and $C_k.a_3$ is connected to $C_j.a_2$. Once we have identified the connected sets of local interactions, we construct a set $\mathcal{S}(C_1, \ldots, C_n)$ composed of as many fresh action types as there are connected sets of local interactions. Then we relabel all the local interactions in the same connected set to the same fresh action type. This is achieved by defining a set of injective action type relabeling functions of the form $\varphi_{C_i;C_1,\ldots,C_n} : \mathcal{LI}_{C_i;C_1,\ldots,C_n} \longrightarrow \mathcal{S}(C_1, \ldots, C_n)$ in such a way that $\varphi_{C_i;C_1,\ldots,C_n}(a_1) = \varphi_{C_j;C_1,\ldots,C_n}(a_2)$ iff $C_i.a_1$ and $C_j.a_2$ belong to the same connected set. Based on these relabeling functions that prepare the AEIs to interact, we now define two semantics for C_i restricted to its local interactions attached to local interactions of $C_1, \ldots, C_n$. The closed semantics will be used in the definition of the architectural checks. It abstracts from the architectural interactions of C_i as these must not come into play when checking for deadlock related architectural mismatches. Since the open semantics will be used instead in the definition of the semantics of an architectural type, it does not abstract from the architectural interactions of C_i as these must be observable. If C_i has no architectural interactions, then the two semantics coincide.

Definition 2.3. *The closed and the open interacting semantics of C_i restricted to $C_1, \ldots, C_n$ are defined by*

$$\begin{array}{rcl} [\![C_i]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} & = & [\![C_i]\!]/(AType - \{\tau\} - \mathcal{LI}_{C_i;C_1,\ldots,C_n}) \\ & & [\varphi_{C_i;C_1,\ldots,C_n}] \\ [\![C_i]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} & = & [\![C_i]\!]/(AType - \{\tau\} - \mathcal{LAI}_{C_i;C_1,\ldots,C_n}) \\ & & [\varphi_{C_i;C_1,\ldots,C_n}] \end{array}$$

where $\mathcal{LAI}_{C_i;C_1,\ldots,C_n} = \mathcal{LI}_{C_i;C_1,\ldots,C_n} \cup \mathcal{AI}_{C_i}$. ■

If we compare Def. 2.2 and Def. 2.3, we observe that the latter gives rise to a further projection on the local interactions attached to local interactions of $C_1, \ldots, C_n$ and relabel such local interactions in order to make it possible the synchronization among $C_1, \ldots, C_n$. Finally, we define the closed and the open interacting semantics of $C_1, \ldots, C_n$ by putting in parallel the closed and the open interacting semantics of each of the considered AEIs, respectively. To do that, we need to define the synchronization sets. Let us preliminarily define for each AEI and pair of AEIs in $C_1, \ldots, C_n$ the subset of fresh action types to which their local interactions are relabeled:

$$\begin{array}{rcl} \mathcal{S}(C_i; C_1, \ldots, C_n) & = & \varphi_{C_i;C_1,\ldots,C_n}(\mathcal{LI}_{C_i;C_1,\ldots,C_n}) \\ \mathcal{S}(C_i, C_j; C_1, \ldots, C_n) & = & \mathcal{S}(C_i; C_1, \ldots, C_n) \cap \\ & & \mathcal{S}(C_j; C_1, \ldots, C_n) \end{array}$$

Recalled that the parallel composition operator is left associative, the synchronization set between the interacting semantics of C_1 and C_2 is given by $\mathcal{S}(C_1, C_2; C_1, \ldots, C_n)$, the synchronization set between the interacting semantics of C_2 and C_3 is given by the union of $\mathcal{S}(C_1, C_3; C_1, \ldots, C_n)$ and $\mathcal{S}(C_2, C_3; C_1, \ldots, C_n)$, and so on.

Definition 2.4. *The closed and the open interacting semantics of $C_1, \ldots, C_n$ are defined by*

$$\begin{array}{rcl} [\![C_1, \ldots, C_n]\!]^{\mathrm{c}} & = & [\![C_1]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C_1,C_2;C_1,\ldots,C_n)} \\ & & [\![C_2]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \parallel_{\cup_{i=1}^{2}\mathcal{S}(C_i,C_3;C_1,\ldots,C_n)} \cdots \\ & & \cdots \parallel_{\cup_{i=1}^{n-1}\mathcal{S}(C_i,C_n;C_1,\ldots,C_n)} [\![C_n]\!]^{\mathrm{c}}_{C_1,\ldots,C_n} \\ [\![C_1, \ldots, C_n]\!]^{\mathrm{o}} & = & [\![C_1]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \parallel_{\mathcal{S}(C_1,C_2;C_1,\ldots,C_n)} \\ & & [\![C_2]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \parallel_{\cup_{i=1}^{2}\mathcal{S}(C_i,C_3;C_1,\ldots,C_n)} \cdots \\ & & \cdots \parallel_{\cup_{i=1}^{n-1}\mathcal{S}(C_i,C_n;C_1,\ldots,C_n)} [\![C_n]\!]^{\mathrm{o}}_{C_1,\ldots,C_n} \end{array}$$ ■

Definition 2.5. *The semantics of an architectural type $\mathcal{A}$ with AEIs $C_1, \ldots, C_n$ is $[\![\mathcal{A}]\!] = [\![C_1, \ldots, C_n]\!]^{\mathrm{o}}$.* ■

2.3 Architectural Checks

Æmilia is equipped with some architectural checks that the designer can use to verify the well formedness of the architectural types and, in case a mismatch is detected, to identify the components that cause it. Most of such checks are based on the weak bisimulation equivalence [16], denoted $\approx_{\mathrm{B}}$, which captures the ability of the integrated semantic models of two terms to simulate each other behaviors up to τ actions when ignoring action rates.

This section provides the detailed description of the architectural checks; it can be safely skipped by the reader interested in the queueing network semantics only. The first two checks, which are a variant of two checks defined for PADL, take care of verifying whether the deadlock free AEIs of an architectural type fit together well, i.e. do not lead to system blocks. The first check (compatibility) is concerned with architectural types whose topology is acyclic. For an acyclic architectural type, if we take an AEI K and we consider all the AEIs $C_1, \ldots, C_n$ attached to it, we can observe that they form a star topology whose center is K, as the absence of cycles prevents any two AEIs among $C_1, \ldots, C_n$ from communicating via an AEI different from K. It can easily be recognized that an acyclic architectural type is just a composition of star topologies. In the case of PADL, an efficient compatibility check based on the weak bisimulation equivalence $\approx_{\mathrm{B}}$ ensures the absence of deadlock within a star topology whose center K is deadlock free, and this check scales to the whole acyclic architectural type. The basic condition to check is that every C_i is compatible with K, i.e. the parallel composition of their closed interacting semantics is weakly bisimulation equivalent to the closed interacting semantics of K itself. Intuitively, this means that attaching C_i to K does not alter the behavior of K, i.e. K is designed in such a way that it suitably coordinates with C_i.

The compatibility result for PADL stems from the fact that $\approx_{\mathrm{B}}$, besides preserving deadlock freedom, is a congruence for the static operators of a classical process algebra. In order to get a similar result in the framework of Æmilia, where priorities come into play, we observe that $\approx_{\mathrm{B}}$ still is a congruence for the static operators when applied to the integrated semantics for EMPA$_{\mathrm{gr}}$, because this semantics retains the lower priority transitions. However, to make sure that no deadlock arises, we have to get rid of lower priority transitions, i.e. we have to reason on the functional semantics of an Æmilia description. Here deadlocks can actually arise in the presence of choices among nonpassive actions having different priority levels within the behavior of an AEI. The reason is that, when we consider the AEI in isolation, the functional semantics of the AEI contains only the transitions related to the highest priority actions occurring in the choices. Instead, the functional semantics of the AEI in parallel with an attached AEI may expose some transitions related to lower priority actions occurring in the choices, as the highest priority ones may be prevented from being executed in the case they are interactions due to some synchronization constraint. In order to be able to define a compositional compatibility check on the functional semantics, it suffices to assume that every sequential EMPA$_{\mathrm{gr}}$ term representing the behavior of an AEI is equally prioritized. This means that, in every alternative composition of the form $\sum_{h=1}^{n} {<}a_h, \tilde{\lambda}_h{>}.E_h$ occurring in the behavior of an AEI:

- the initial nonpassive actions are all either exponential or immediate with the same priority level;
- every initial passive interaction can synchronize with a nonpassive interaction of another AEI whose rate is of the same kind as the rates of the initial nonpassive actions.

The constraint above can simply be checked at the syntax level (no state space construction is required).

DEFINITION 2.6. *Given an acyclic architectural type, let $C_1, \dots, C_n$ be the AEIs attached to AEI K. C_i is compatible with K iff $[\![K]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n} \,\|_{\mathcal{S}(K;K,C_1,\dots,C_n)}\, [\![C_i]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n} \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n}$.* ■

THEOREM 2.7. *Given an acyclic architectural type, let $C_1, \dots, C_n$ be the AEIs attached to AEI K. If the behavior of $K, C_1, \dots, C_n$ is equally prioritized, the functional semantics of $[\![K]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n}$ is deadlock free, and C_i is compatible with K for all $i = 1, \dots, n$, then the functional semantics of*

$$\begin{aligned}[\![K; C_1, \dots, C_n]\!] \;=\; & [\![K]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n} \,\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \\ & [\![C_1]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n} \,\|_{\mathcal{S}(K;K,C_1,\dots,C_n)} \cdots \\ & \cdots \,\|_{\mathcal{S}(K;K,C_1,\dots,C_n)}\, [\![C_n]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n}\end{aligned}$$

is deadlock free. ■

COROLLARY 2.8. *Given an acyclic architectural type, if the behavior of each AET is equally prioritized, the functional semantics of each AET with the architectural interactions being hidden is deadlock free, and every AEI is compatible with each AEI attached to it, then the functional semantics of the architectural type is deadlock free.* ■

Since the compatibility check is not sufficient for cyclic architectural types, the second check (interoperability) deals with cycles. In the case of PADL, a suitable interoperability check based on $\approx_{\mathrm{B}}$ ensures the absence of deadlock within a cycle $C_1, \dots, C_n$ of AEIs in the case that at least one of such AEIs is deadlock free. The basic condition to check is that at least one deadlock free C_i interoperates with the other AEIs in the cycle, i.e. the parallel composition of the closed interacting semantics of the AEIs in the cycle projected on the interactions with C_i only is weakly bisimulation equivalent to the closed interacting semantics of C_i. Intuitively, this means that inserting C_i into the cycle does not alter the behavior of C_i, i.e. that the behavior of the cycle assumed by C_i matches the actual behavior of the cycle.

DEFINITION 2.9. *Given an architectural type, let $C_1, \dots, C_n$ be AEIs forming a cycle. C_i interoperates with $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_n$ iff $[\![C_1, \dots, C_n]\!]^{\mathrm{c}} / (AType - \{\tau\} - \mathcal{S}(C_i; C_1, \dots, C_n)) \approx_{\mathrm{B}} [\![C_i]\!]^{\mathrm{c}}_{C_1,\dots,C_n}$.* ■

THEOREM 2.10. *Given an architectural type, let $C_1, \dots, C_n$ be AEIs forming a cycle. If the behavior of $C_1, \dots, C_n$ is equally prioritized, and there exists C_i such that the functional semantics of $[\![C_i]\!]^{\mathrm{c}}_{C_1,\dots,C_n}$ is deadlock free and C_i interoperates with $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_n$, then the functional semantics of $[\![C_1, \dots, C_n]\!]^{\mathrm{c}}$ is deadlock free.* ■

On the performance side, we introduce for Æmilia a third check to detect architectural mismatches resulting in performance underspecification. This check (performance closure) ensures that the performance semantic model underlying an architectural type exists in the form of a continuous or discrete time MC.

DEFINITION 2.11. *An architectural type $\mathcal{A}$ is performance closed iff $\mathcal{I}[\![[\![\mathcal{A}]\!]]\!]$ has no passive transitions.* ■

THEOREM 2.12. *An architectural type $\mathcal{A}$ is performance closed iff no AET behavior contains a passive action whose type is not an interaction, and every connected set of local interactions contains one interaction whose associated rate is exponential or immediate.* ■

An Æmilia description represents a family of software architectures called an architectural type. All the members of the family must have the same observable functional behavior and topology, while the internal behavior and the performance characteristics can vary. An instance of an architectural type can be obtained by invoking the architectural type and passing actual AETs preserving the observable functional behavior of the formal AETs, actual names for the architectural interactions, and actual values for the numeric parameters. Similarly to PADL, Æmilia is equipped with an efficient, $\approx_{\mathrm{B}}$ based check to verify whether an architectural type invocation conforms to an architectural type definition, in the sense that the architectural type invocation and the architectural type definition have the same observable semantics up to some relabeling. This ensures that all the correct instances of an architectural type possess the same compatibility, interoperability, and performance closure properties. The Æmilia conformity check actually strengthens the PADL one, since it additionally requires the rate preservation between corresponding interactions of actual and formal AETs.

DEFINITION 2.13. *Let $\mathcal{A}(\mathcal{C}'_1, \dots, \mathcal{C}'_m; a'_1, \dots, a'_l; v'_1, \dots, v'_h)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \dots, \mathcal{C}_m$, formal architectural interactions $a_1, \dots, a_l$, and formal numeric parameters $v_1, \dots, v_h$. $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ iff there exist an injective relabeling function φ'_i for the interactions of $\mathcal{C}'_i$ and an injective relabeling function φ_i for the interactions of $\mathcal{C}_i$ such that*

$$[\![\mathcal{C}'_i]\!][\varphi'_i] \approx_{\mathrm{B}} [\![\mathcal{C}_i]\!]\{v'_1/v_1, \dots, v'_h/v_h\}[\varphi_i]$$

where the curly braces enclose a syntactical substitution and the interactions equally relabeled by φ'_i and φ_i occur with the same kind of rate in the behavior of both $\mathcal{C}'_i$ and $\mathcal{C}_i$. ■

DEFINITION 2.14. *Let $\mathcal{A}(\mathcal{C}'_1, \dots, \mathcal{C}'_m; a'_1, \dots, a'_l; v'_1, \dots, v'_h)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \dots, \mathcal{C}_m$, formal architectural interactions $a_1, \dots, a_l$, and formal numeric parameters $v_1, \dots, v_h$. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1, \dots, m$, then the semantics of the architectural type invocation is defined by*

$$\begin{aligned}&[\![\mathcal{A}(\mathcal{C}'_1, \dots, \mathcal{C}'_m; a'_1, \dots, a'_l; v'_1, \dots, v'_h)]\!] = \\ &\qquad [\![\mathcal{A}]\!]\{v'_1/v_1, \dots, v'_h/v_h\}[a_1 \mapsto a'_1, \dots, a_l \mapsto a'_l]\end{aligned}$$ ■

THEOREM 2.15. *Let $\mathcal{A}(\mathcal{C}'_1, \dots, \mathcal{C}'_m; a'_1, \dots, a'_l; v'_1, \dots, v'_h)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \dots, \mathcal{C}_m$, formal architectural interactions $a_1, \dots, a_l$, and formal numeric parameters $v_1, \dots, v_h$. Let $C'_1, \dots, C'_n$ be the AEIs of the architectural type invocation. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1, \dots, m$, then there exist an injective relabeling function φ' for the interactions of the architectural type invocation and a relabeling function φ for the interactions of the architectural type definition, with φ being injective at least on the local interactions, such that*

$$[\![C'_1, \dots, C'_n]\!]^{\mathrm{o}}[\varphi'] \approx_{\mathrm{B}} [\![\mathcal{A}]\!]\{v'_1/v_1, \dots, v'_h/v_h\}[\varphi]$$

where the interactions equally relabeled by φ' and φ occur with the same kind of rate in the semantics of both the architectural type invocation and definition. ■

COROLLARY 2.16. *Let $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l; v'_1, \ldots, v'_h)$ be an invocation of the architectural type $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$, formal architectural interactions $a_1, \ldots, a_l$, and formal numeric parameters $v_1, \ldots, v_h$. If, for all $i = 1, \ldots, m$, $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$, $\mathcal{C}'_i$ is equally prioritized iff so is $\mathcal{C}_i$, and $\mathcal{C}'_i$ contains in its behavior a passive action whose type is not an interaction iff so does $\mathcal{C}_i$, then the architectural type invocation and the architectural type definition have the same compatibility, interoperability and performance closure properties.* ■

The proofs of the results presented in this section are simple reworking of those for PADL.

3. A QUEUEING NETWORK SEMANTICS FOR ÆMILIA

As seen in the previous section, each Æmilia description of a performance closed architectural type is provided with a MC model on which we can compute the performance metrics for the system under consideration. Unfortunately, such a MC model has two drawbacks. First, it suffers from the state space explosion problem, i.e. the number of states of the MC grows exponentially with the number of AEIs. Second, it is a flat model, i.e. it does not keep track of the architectural structure, thus hampering altogether the interpretation of the performance figures at the architectural level, and in particular the identification of the AEIs responsible for poor performance.

In order to overcome these two problems related to the efficient computation of performance figures and their interpretation on the architectural description, in this section we provide a mapping from Æmilia to QNs. The reason why we consider QNs as target model for the mapping is that they are endowed with efficient solution techniques that do not require the construction of the underlying state space, and that they preserve the structure of the system under study. The basic idea underlying this mapping is to identify three classes of QN basic elements – arrival processes, buffers, and service processes – and impose some restrictions on the syntax of the AETs, so that each of their instances exactly falls into one of the three classes. The mapping is then accomplished by translating each AEI into the corresponding QN basic element and composing the QN basic elements according to the attachments.

3.1 QN Basic Elements

An arrival process is a generator of requests of a certain type, whose generation times follow a certain phase type distribution (which is an exponential distribution in the simplest case). Algebraically, an arrival process is composed of several exponentially timed and immediate actions representing the various phases of the generation, followed by an immediate action representing the arrival of the request at a buffer or a service process. Below is an example of behavior of an AET that is an exponential arrival process:

$$Arrival \triangleq <generate, \lambda>.<deliver, \infty>.Arrival$$

where *deliver* is an output interaction.

A buffer is a repository of requests of different types that are waiting to be served. Algebraically, a buffer is composed only of passive actions representing the arrival or the departure of requests according to a certain discipline, where actions concerned with different types of requests are distinct. Below is an example of behavior of an AET that is a buffer with n positions for a single type of requests, following a FIFO discipline on the departure side:

$$\begin{aligned} Queue_0 &\triangleq <arrive, *>.Queue_1 \\ Queue_i &\triangleq <arrive, *>.Queue_{i+1} + \\ & <depart, *>.Queue_{i-1}, \;\; 0 < i < n \\ Queue_n &\triangleq <depart, *>.Queue_{n-1} \end{aligned}$$

where *arrive* is an input interaction whereas *depart* is an output interaction.

A service process is a server for requests of different types that are served one at a time, whose service times follow a certain phase type distribution. Algebraically, a service process is composed of a choice among several immediate actions representing the selection from some buffers of the next request to be served, followed by several exponentially timed and immediate actions representing the various phases of the service, possibly followed by an immediate action representing the leave of the served request. Below is an example of behavior of an AET that is an exponential service process for a single type of requests:

$$Server \triangleq <select, \infty>.<serve, \mu>.<leave, \infty>.Server$$

where *select* is an input interaction whereas *leave* is an output interaction. Since a service process may represent a server within a system with no buffer capacity, we also admit the case in which the algebraic description of the service process starts with a choice among several passive actions representing the selection from some arrival/service processes of the next request to be served. In such a case, the term above should be changed as follows:

$$Server \triangleq <select, *>.<serve, \mu>.<leave, \infty>.Server$$

As can be noted, exponentially timed actions are used to model the generation and service times for the requests. Immediate actions are used to model the output of the arrival processes, the input of the service processes through possible selections, the possible selection among parallel phases within the arrival and service processes, and the possible output of the service processes. Finally, passive actions are used to model the input and the output of the buffers and the input of the service processes whenever they are included in systems with no buffer capacity.

The graphical representation of the three classes of QN basic elements is shown in Fig. 2 for the examples considered above. Their graphical representation is consistent with the usual graphical representation adopted for QNs.

3.2 Æmilia Syntax Restrictions

Given the three classes of QN basic elements identified above, we have to make sure that every AET described in Æmilia falls into one of the three classes. Recalled that an AET models a sequential software component running on a single computational resource, we impose the following syntax restrictions:

1. The first restriction aims at easing the identification of those AETs that represent arrival or service processes, which are built around exponentially timed actions. It establishes that the interactions cannot be exponentially timed.

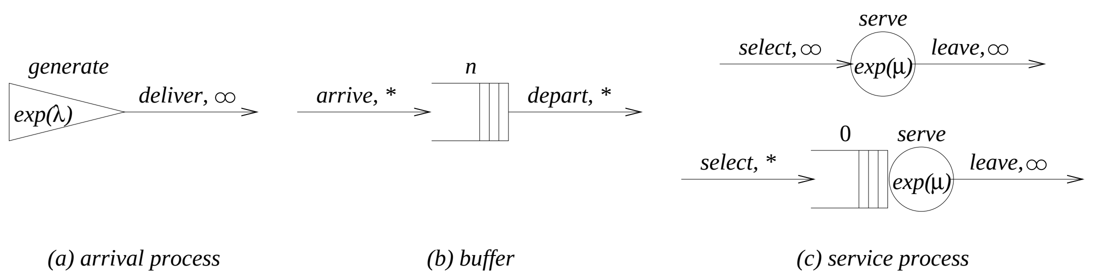


Figure 2: Graphical representation of the exemplified QN basic elements

2. The second restriction aims at avoiding the unnatural application of the race policy to several distinct activities within the same sequential AET, thus causing the architect to separately model the computational elements of the system (i.e. arrival and service processes) with different AEIs. It establishes that an exponentially timed action cannot be alternative to another exponentially timed action.

3. The third restriction aims at allowing the precise computation of the probability distribution associated with every arrival or service process. It establishes that an exponentially timed action cannot be alternative to a passive or immediate action.

4. The fourth restriction aims at easing the identification of those AETs that represent buffers or service processes with no buffer capacity. It establishes that either all the actions in the behavior of an AET are passive, or only all the actions in an initial choice are. A consequence of this restriction is that choices between passive actions and immediate actions are forbidden as well.

5. The fifth restriction establishes the structure of an arrival process, thus making it possible to correctly model the generation of a single request at a time – on a software component running on a single computational resource – according to a certain phase type distribution. The behavior of an AET that represents an arrival process must start with a possibly empty choice among several noninteracting immediate actions. Every branch must then continue with a sequence of distinct phases whose unit element is an exponentially timed action followed by a possibly empty choice among several noninteracting immediate actions and output immediate interactions. Every branch must terminate with an output immediate interaction whose following behavior must be the same as that of the whole AET. A consequence of this restriction is that AETs representing arrival processes cannot have input interactions.

6. The sixth restriction establishes the structure of a service process, thus making it possible to correctly model the service of a single request at a time – on a software component running on a single computational resource – according to a certain phase type distribution. The behavior of an AET that represents a service process must start with a choice among several input immediate interactions or several input passive interactions, with each alternative possibly followed by a choice among several noninteracting immediate actions. Every branch must then continue with a sequence of distinct phases whose unit element is an exponentially timed action followed by a possibly empty choice among several noninteracting immediate actions and output immediate interactions. The behavior after an output immediate interaction must be the same as that of the whole AET. A consequence of this restriction is that choices between noninteracting immediate actions or output immediate interactions and input immediate interactions are forbidden.

The restrictions above are easily enforceable because they can be checked at the syntax level (no state space construction is required). For instance, for our compiler system example we can strightforwardly verify that all the restrictions above are satisfied, with PG representing an arrival process, PB representing a buffer, and C representing a service process. As far as expressiveness is concerned, the syntax restrictions are reasonable, as they preserve much of the power that Æmilia inherits from EMPA$_{\rm gr}$. Additionally, they are consistent with the way systems are usually modeled, and do not hamper the description of typical situations like probabilistic/prioritized choices as well as activities whose duration is phase type. A limitation to the original modeling capabilities that such syntax restrictions introduce is that they do not allow for preemption, i.e. the fact that the service of a request of a certain type can be interrupted by the arrival of a request of another type having higher service priority. We conclude by pointing out that the restrictions above have not been imposed directly on the syntax that Æmilia inherits from EMPA$_{\rm gr}$, because this would leave out system descriptions that can be analyzed (on a MC model) without resorting to a QN model.

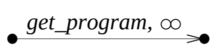
3.3 Mapping AEIs to QN Basic Elements

The first step of the translation of a performance closed architectural type into a QN consists of mapping every AEI to a QN basic element, which is carried out through several functions to be applied to the AEI behavior. In the following, we denote by STG the set of state transition graphs labeled with EMPA$_{\rm gr}$ actions, by $PTDistr$ the set of phase type distributions, by $\mathcal{G}_{\rm seq}$ the set of sequential EMPA$_{\rm gr}$ terms, by $\mathcal{G}_{\rm seq,buf}$ the set of sequential EMPA$_{\rm gr}$ terms containing only passive actions, and by STG' the set of state transition graphs labeled with EMPA$_{\rm gr}$ actions augmented with an additional piece of information specifying whether the corresponding action type is declared to be an input or an output interaction.

The first (partial) function, $\mathit{input} : \mathcal{G}_{\mathrm{seq}} \nrightarrow 2^{Act}$, documents the classes of requests accepted by a buffer or served by a service process by recording the corresponding actions. If we consider Fig. 2, the application of function *input* results in the set of actions labeling the incoming arcs of the buffer and the service process. Given an AEI representing a buffer or a service process with behavior E, $\mathit{input}(E)$ is the set of actions occurring in E whose type is declared to be an input interaction for that AEI. In the compiler system example of Table 1 we have

$$\begin{aligned} \mathit{input}(\mathit{ProgramBuffer}_0) &= \{<\mathit{get\_program}, *>\} \\ \mathit{input}(\mathit{Compiler}) &= \{<\mathit{select\_program}, \infty>\} \end{aligned}$$

The second (partial) function, $\mathit{s\_policy} : \mathcal{G}_{\mathrm{seq}} \nrightarrow STG$, describes the policy according to which the next request is selected from a buffer by a service process. Given an AEI representing a service process with behavior E, $\mathit{s\_policy}(E)$ is the initial state of $\mathcal{I}[\![E]\!]$ together with its outgoing transitions and destination states. In the compiler system example of Table 1 we have that $\mathit{s\_policy}(\mathit{Compiler})$ is as depicted below:



which means that there is no actual selection to be carried out.

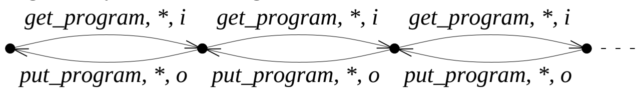
The third (partial) function, $\mathit{process} : \mathcal{G}_{\mathrm{seq}} \nrightarrow 2^{AType}$, documents the phases of the generation or service of a request by recording the corresponding action types. If we consider Fig. 2, the application of function *process* results in the set of action types labeling the triangle of the arrival process and the circle of the service process. Given an AEI representing an arrival or service process with behavior E, $\mathit{process}(E)$ is the set of action types occurring in E that are not declared to be interactions for that AEI. In the compiler system example of Table 1 we have

$$\begin{aligned} \mathit{process}(\mathit{ProgramGenerator}) &= \{\mathit{generate\_program}\} \\ \mathit{process}(\mathit{Compiler}) &= \{\mathit{recognize\_tokens}, \\ &\quad\ \mathit{parse\_phrases}, \\ &\quad\ \mathit{check\_phrases}, \\ &\quad\ \mathit{opt\_yes}, \mathit{opt\_no}, \\ &\quad\ \mathit{optimize}, \mathit{generate\_code}\} \end{aligned}$$

The fourth (partial) function, $\mathit{ptdistr} : \mathcal{G}_{\mathrm{seq}} \nrightarrow \mathit{PTDistr}$, computes the phase type distribution associated with the generation or service of a request. If we consider Fig. 2, the application of function *ptdistr* results in the probability distribution labeling the triangle of the arrival process and the circle of the service process. Given an AEI representing an arrival or service process with behavior E, $\mathit{ptdistr}(E)$ is defined as the smallest phase type distribution satisfying the equalities depicted in Table 2, where *exp* stands for an exponential distribution, *hypoexp* stands for a hypoexponential distribution, and *hyperexp* stands for a hyperexponential distribution. In the compiler system example of Table 1 we have

$$\begin{aligned} &\mathit{ptdistr}(\mathit{ProgramGenerator}) = \mathit{exp}(\lambda) \\ &\mathit{ptdistr}(\mathit{Compiler}) = \mathit{hypoexp}(\mathit{exp}(\mu_l), \\ &\qquad \mathit{exp}(\mu_p), \\ &\qquad \mathit{exp}(\mu_c), \\ &\qquad \mathit{hyperexp}(p_o, \mathit{hypoexp}(\mathit{exp}(\mu_o), \\ &\qquad\qquad \mathit{exp}(\mu_g)); \\ &\qquad\qquad 1 - p_o, \mathit{exp}(\mu_g))) \end{aligned}$$

The fifth function, $\mathit{b\_policy} : \mathcal{G}_{\mathrm{seq,buf}} \longrightarrow STG'$, describes the policy according to which the requests are accepted and delivered by a buffer. Given an AEI representing a buffer with behavior E, $\mathit{b\_policy}(E)$ is $\mathcal{I}[\![E]\!]$ where each transition label is augmented with an additional piece of information specifying whether the corresponding action type is declared to be an input or an output interaction. For the compiler system example of Table 1, $\mathit{b\_policy}(\mathit{ProgramBuffer}_0)$ is given by the following STG':



The STG' above has the typical structure of a birth-death process, thus abstracting from the identity of the individual requests. This is a typical situation with the buffers of the classical QNs, therefore we can assume that the buffer above is governed by the FIFO discipline.

The sixth (partial) function, $\mathit{b\_capacity} : \mathcal{G}_{\mathrm{seq,buf}} \times \mathit{AType} \nrightarrow \mathbb{N} \cup \{\infty\}$, describes the capacity of a buffer w.r.t. all the classes of requests that it accepts. Given an AEI representing a buffer with behavior E and an input interaction a_{in} of the AEI whose related output interaction is a_{out}, $\mathit{b\_capacity}(E, a_{in})$ is computed on the basis of $\mathit{b\_policy}(E)$ as follows. If $\mathit{b\_policy}(E)$ contains a cycle composed of only input interactions among which a_{in}, then $\mathit{b\_capacity}(E, a_{in}) = \infty$. If this is not the case, we consider every maximal simple [3] path in $\mathit{b\_policy}(E)$ starting from the initial state and containing occurrences of a_{in} and a_{out}, and we compute the maximum between the length of the longest sequence of occurrences of a_{in} with no intervening occurrences of a_{out} and the algebraic sum of the occurrences of a_{in} $(+1)$ and a_{out} (-1). $\mathit{b\_capacity}(E, a_{in})$ is then given by the maximum of such values. In the compiler system example of Table 1 we have

$$\mathit{b\_capacity}(\mathit{ProgramBuffer}_0, \mathit{get\_program}) = \infty$$

The seventh function, $\mathit{output} : \mathcal{G}_{\mathrm{seq}} \longrightarrow 2^{Act}$, documents the classes of requests that can leave a buffer or a service process by recording the corresponding actions. If we consider Fig. 2, the application of function *output* results in the set of actions labeling the outgoing arcs of the QN basic elements. Given an AEI with behavior E, $\mathit{output}(E)$ is the set of actions occurring in E whose type is declared to be an output interaction for that AEI. In the compiler system example of Table 1 we have

$$\begin{aligned} \mathit{output}(\mathit{ProgramGenerator}) &= \{<\mathit{deliver\_program}, \infty>\} \\ \mathit{output}(\mathit{ProgramBuffer}_0) &= \{<\mathit{put\_program}, *>\} \\ \mathit{output}(\mathit{Compiler}) &= \emptyset \end{aligned}$$

We conclude this section by showing in Fig. 3 the three QN basic elements resulting from the first step of the translation of the performance closed architectural type representing the compiler system of Table 1.

3.4 Connecting QN Basic Elements

The second step of the translation of a performance closed architectural type into a QN simply consists of connecting the basic QN elements (stemming from the translation of the AEIs) according to the attachments. Graphically, this amounts to superposing the arrow headed arcs corresponding to attached interactions, with the weights of the immediate interactions labeling the arcs leaving the arrival and service processes used to determine the routing probabilities. In the case of our compiler system example we obtain the QN depicted in Fig 4.

[3] Traversing each state at most once.

$$
\begin{array}{rcl}
\mathit{ptdistr}(\underline{0}) & = & \emptyset \\
\mathit{ptdistr}(<a,\lambda>.F) & = & \mathit{hypoexp}(\mathit{exp}(\lambda), \mathit{ptdistr}(F)) \\
\mathit{ptdistr}(<a,\infty_{l,w}>.F) & = & \begin{cases} \mathit{ptdistr}(F) & \text{if } a \text{ not interaction or input interaction} \\ \emptyset & \text{if } a \text{ output interaction} \end{cases} \\
\mathit{ptdistr}(<a_1,\infty_{l_1,w_1}>.F_1 + <a_2,\infty_{l_2,w_2}>.F_2) & = & \begin{cases} \mathit{hyperexp}(\frac{w_1}{w_1+w_2}, \mathit{ptdistr}(F_1); \frac{w_2}{w_1+w_2}, \mathit{ptdistr}(F_2)) \\ \qquad \text{if } a_1, a_2 \text{ not interactions or input interactions} \\ \mathit{hyperexp}(\frac{w_1}{w_1+w_2}, \emptyset; \frac{w_2}{w_1+w_2}, \mathit{ptdistr}(F_2)) \\ \qquad \text{if } a_1 \text{ output interaction and } a_2 \text{ not interaction} \\ \mathit{hyperexp}(\frac{w_1}{w_1+w_2}, \mathit{ptdistr}(F_1); \frac{w_2}{w_1+w_2}, \emptyset) \\ \qquad \text{if } a_1 \text{ not interaction and } a_2 \text{ output interaction} \\ \emptyset \\ \qquad \text{if } a_1, a_2 \text{ output interactions} \end{cases} \\
\mathit{ptdistr}(<a_1,*_{l_1,w_1}>.F_1 + <a_2,*_{l_2,w_2}>.F_2) & = & \mathit{hyperexp}(\frac{w_1}{w_1+w_2}, \mathit{ptdistr}(F_1); \frac{w_2}{w_1+w_2}, \mathit{ptdistr}(F_2)) \\
\mathit{ptdistr}(A) & = & \mathit{ptdistr}(F) \qquad \text{if } A \triangleq F
\end{array}
$$

Table 2: Definition of function *ptdistr*

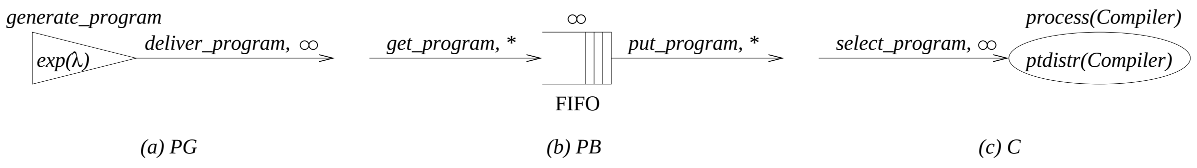


Figure 3: QN basic elements of *CompilerSystem*

4. MODELING TWO ALTERNATIVE COMPILER ARCHITECTURES

In this section we illustrate the proposed approach by describing with Æmilia two alternative architectures for the compiler system and building the associated QN models through the semantics described in Sect. 3.

The compiler component shown in Table 1 examines one source program at a time, i.e. it is a completely sequential compiler. An alternative architecture allows the compilation of a program to start before the previous one has been completed. This is achieved through a pipeline like structure obtained by splitting the various phases of the compiler into different AETs for the lexer, parser, checker, optimizer, and generator, and by connecting them in such a way that each phase can work on one single source program at a time. W.r.t. the compiler system depicted in Table 1, we replace *CompilerT* with the following one, whose behavior is ruled by the new architectural type *PipelineCompiler*:

elem_type $\mathit{CompilerT}(\textbf{exp\_rate}\ \mu_l, \mu_p, \mu_c, \mu_o, \mu_g;\ \textbf{weight}\ p_o)$

behavior $\mathit{Compiler} \triangleq \mathit{PipelineCompiler}(;\mathit{select\_program}; \mu_l, \mu_p, \mu_c, \mu_o, \mu_g; p_o)$

interactions **input** *select_program*

where the absence of actual AETs in the architectural type invocation means that they coincide with the formal ones. The Æmilia graphical representation of *PipelineCompiler* is shown in Fig. 5, while its Æmilia specification is shown in Table 3. The pipeline stream between the different compiler phases is realized by directly connecting their corresponding AETs. By applying to each AEI of Table 3 the functions defined in Sect. 3, we get the corresponding QN basic elements. By connecting such elements according to the attachments, we obtain the QN model depicted in Fig. 6 for the whole pipeline compiler system. The weights of the output immediate actions associated with the checker service process are used to determine the routing probabilities of the two following branches. Note that the QN structure closely resembles the structure of the flow graph in Fig. 5, which should make it easier to interpret at the SA level the performance results obtained at the QN level.

We now consider a second alternative architecture of the compiler system that allows the parsing, checking, optimizing and generating phases to proceed in parallel. This is achieved by providing *ParserT*, *CheckerT*, *OptimizerT*, and *GeneratorT* of *PipelineCompiler* with their own (infinite) buffers, in such a way that they can concurrently work on different source programs. W.r.t. Table 1, we replace *CompilerT* with the following one, whose behavior is ruled by the new architectural type *ConcurrentCompiler*:

elem_type $\mathit{CompilerT}(\textbf{exp\_rate}\ \mu_l, \mu_p, \mu_c, \mu_o, \mu_g;\ \textbf{weight}\ p_o)$

behavior $\mathit{Compiler} \triangleq \mathit{ConcurrentCompiler}(;\mathit{select\_program}; \mu_l, \mu_p, \mu_c, \mu_o, \mu_g; p_o)$

interactions **input** *select_program*

The Æmilia graphical representation of *ConcurrentCompiler* is shown in Fig. 7, while its Æmilia specification can easily be derived from the one of *PipelineCompiler* by adding buffer AETs to *ParserT*, *CheckerT*, *OptimizerT*, and *GeneratorT*. The QN model associated with the Æmilia specification of the concurrent compiler system is very similar to the one representing the pipeline compiler system in Fig. 6. The main difference is that the buffer capacity of the P, C, O and G service centers changes from zero to infinity.

It is easy to verify that the MCs associated with the Æmilia specifications of the three variants of the compiler system are infinite (hence they cannot be built automatically from the Æmilia specifications). On the other hand, the

archi_type	*PipelineCompiler*(**exp_rate** $\mu_{lexer}, \mu_{parser}, \mu_{checker}, \mu_{optimizer}, \mu_{generator}$; **weight** $prob_{optimizer}$)
archi_elem_types	
elem_type	*LexerT*(**exp_rate** μ_l)
behavior	$Lexer \triangleq {<}select\_program, \infty{>}.{<}recognize\_tokens, \mu_l{>}.$ ${<}send\_token\_seq, \infty{>}.Lexer$
interactions	**input** *select_program* **output** *send_token_seq*
elem_type	*ParserT*(**exp_rate** μ_p)
behavior	$Parser \triangleq {<}get\_token\_seq, *{>}.{<}parse\_phrases, \mu_p{>}.$ ${<}send\_phrase\_seq, \infty{>}.Parser$
interactions	**input** *get_token_seq* **output** *send_phrase_seq*
elem_type	*CheckerT*(**exp_rate** μ_c; **weight** p_o)
behavior	$Checker \triangleq {<}get\_phrase\_seq, *{>}.{<}check\_phrases, \mu_c{>}.$ $({<}send\_checked\_seq1, \infty_{1,p_o}{>}.Checker +$ ${<}send\_checked\_seq2, \infty_{1,1-p_o}{>}.Checker)$
interactions	**input** *get_phrase_seq* **output** *send_checked_seq1*, *send_checked_seq2*
elem_type	*OptimizerT*(**exp_rate** μ_o)
behavior	$Optimizer \triangleq {<}get\_checked\_seq1, *{>}.{<}optimize, \mu_o{>}.$ ${<}send\_optimized\_seq, \infty{>}.Optimizer$
interactions	**input** *get_checked_seq1* **output** *send_optimized_seq*
elem_type	*GeneratorT*(**exp_rate** μ_g)
behavior	$Generator \triangleq {<}get\_checked\_seq2, *{>}.{<}generate\_code, \mu_g{>}.Generator +$ ${<}get\_optimized\_seq, *{>}.{<}generate\_code, \mu_g{>}.Generator$
interactions	**input** *get_checked_seq2*, *get_optimized_seq*
archi_topology	
archi_elem_instances	$L : LexerT(\mu_{lexer})$ $P : ParserT(\mu_{parser})$ $C : CheckerT(\mu_{checker}; prob_{optimizer})$ $O : OptimizerT(\mu_{optimizer})$ $G : GeneratorT(\mu_{generator})$
archi_interactions	**input** *L.select_program*

archi_attachments	**from** *L.send_token_seq* **to** *P.get_token_seq*
	from *P.send_phrase_seq* **to** *C.get_phrase_seq*
	from *C.send_checked_seq1* **to** *O.get_checked_seq1*
	from *C.send_checked_seq2* **to** *G.get_checked_seq2*
	from *O.send_optimized_seq* **to** *G.get_optimized_seq*

end	

Table 3: Textual description of *PipelineCompiler*

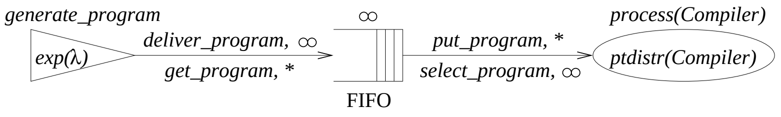


Figure 4: QN associated with *CompilerSystem*

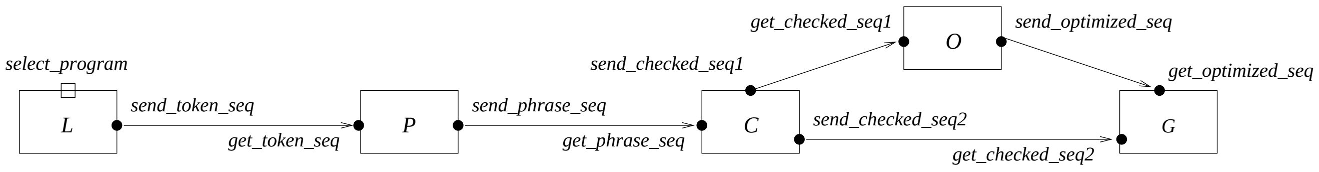


Figure 5: Flow graph of *PipelineCompiler*

corresponding QN models can be solved either by product-form algorithms or by approximate methods avoiding the construction of the MC.

The three variants of the compiler system have already been presented in [1], where the authors propose a methodology to derive a QN model from a labeled transition system describing the behavior of a SA. The QN models obtained for the three variants of the compiler system using the methodology in [1] and our semantics coincide. Steady state analysis can be performed on such QN models in order to calculate performance indices like the mean number of customers in the system, throughput, system utilization, and mean system response time. The detailed evaluation and comparison of the obtained QN models is out of the scope of this paper: a discussion can be found in [1]. We just want to point out that, under the hypothesis of exponential service time distribution at each service center, FIFO service discipline, Poisson arrivals, and independence between services and arrival times, the QNs of the sequential and concurrent compilers show a product-form solution, while the QN of the pipeline compiler can be solved by approximate methods.

5. CONCLUSION

In this paper we have proposed the integration of SPAs and QNs at the software architecture level of design, in order to take advantage of their complementary strengths in terms of functional and quantitative analysis. Such an integration has been achieved by introducing Æmilia, an EMPA$_{\rm gr}$ based ADL for the compositional, graphical and hierarchical modeling of SAs, which is equipped with some checks for the detection of architectural mismatches. For Æmilia we have defined a semantics based on QNs, with the goals of exploiting the efficient solution methods of QNs and enabling the interpretation at the SA level of the performance figures obtained at the QN level. Our QN based translation is linear in the number of components of the SA and can deal with unbounded buffers. Therefore, it is clearly advantageous w.r.t. the usual MC based translation, as this results in a state space whose size grows exponentially with the number of components of the SA.

Our semantics based on QNs is inspired by [1], where a methodology has been proposed to derive a QN model from a transition system description of a SA. With our approach centered on Æmilia, we lose the independence from the ADL, but we gain in terms of efficiency, because we avoid the construction of the transition system altogether. On the SPA side, it is worth recalling that several papers, like e.g. [9, 19, 12], have addressed the issue of characterizing at the syntax level the terms whose underlying MC admits a product-form solution. Our approach is different, in the sense that it maps an Æmilia specification to a QN independently of the satisfaction of a product-form condition. The only constraint that must be satisfied is given by the six syntax restrictions in Sect. 3.2, which do not severely limit the expressive power.

Our translation maps Æmilia specifications to open or closed QNs with an arbitrary topology, general arrival and service distributions (approximated with phase typed ones whenever necessary), and FIFO buffers with finite or infinite capacity. We have focussed on basic QNs to take advantage of their efficient analytical methods, such as product-form solution algorithms. However, in principle we may consider extended QNs [15, 14, 13], which can contain some other element types such as fork and join nodes, subnetwork population constraints, finite queue capacity and various blocking mechanisms, multiple classes of customers, and various scheduling disciplines including preemption, which we do not currently permit. A translation of Æmilia specifications into extended QN models is left for future work.

We also plan to deepen the investigation of the issue of interpreting at the SA level the performance figures calculated at the QN level. Similarly to the architectural checks for the detection of deadlock related architectural mismatches, we would like to develop suitable techniques that allow us to identify the software components that are responsible for the achievement of poor performance.

6. REFERENCES

[1] F. Aquilani, S. Balsamo, P. Inverardi, *"Performance Analysis at the Software Architecture Design Level"*, in Performance Evaluation 45:205-221, 2001

[2] M. Bernardo, P. Ciancarini, L. Donatiello, *"ÆMPA: A Process Algebraic Description Language for the Performance Analysis of Software Architectures"*, in [21], pp. 1-11

[3] M. Bernardo, P. Ciancarini, L. Donatiello, *"On the Formalization of Architectural Types with Process Algebras"*, in Proc. of FSE-8, ACM Press, pp. 140-148, San Diego (CA), 2000

[4] M. Bernardo, P. Ciancarini, L. Donatiello, *"Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems"*, in Proc. of WICSA 2001, IEEE-CS Press, pp. 77-86,

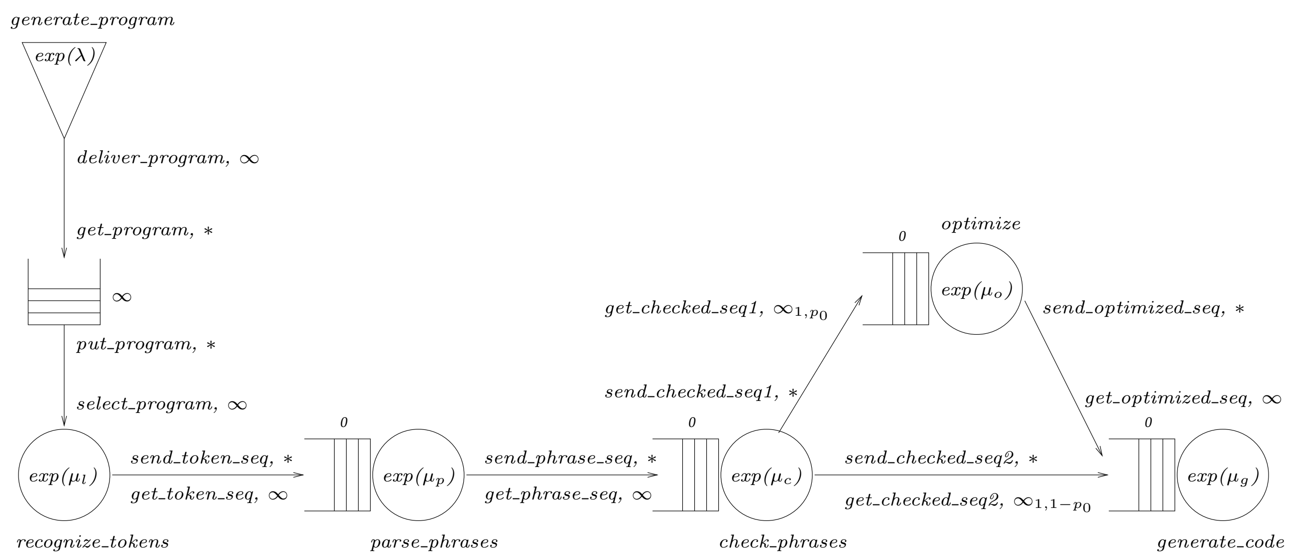


Figure 6: QN associated with *PipelineCompiler*

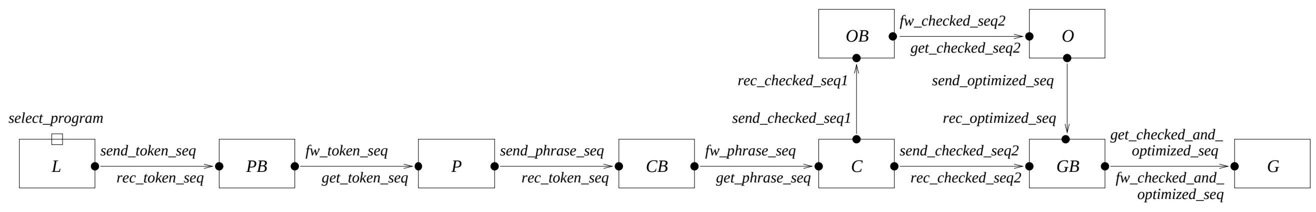


Figure 7: Flow graph of *ConcurrentCompiler*

Amsterdam (The Netherlands), 2001

[5] M. Bernardo, W.R. Cleaveland, W.S. Stewart, *"TwoTowers 1.0 User Manual"*, `http://www.sti.uniurb.it/bernardo/twotowers/`, 2001

[6] M. Bravetti, M. Bernardo, *"Compositional Asymmetric Cooperations for Process Algebras with Probabilities, Priorities, and Time"*, in Proc. of MTCS 2000, Electronic Notes in Theoretical Computer Science 39(3), State College (PA), 2000

[7] E.M. Clarke, O. Grumberg, D.A. Peled, *"Model Checking"*, MIT Press, 1999

[8] W.R. Cleaveland, J. Parrow, B. Steffen, *"The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems"*, in ACM Trans. on Programming Languages and Systems 15:36-72, 1993

[9] P.G. Harrison, J. Hillston, *"Exploiting Quasi-Reversible Structures in Markovian Process Algebra Models"*, in Computer Journal 38:510-520, 1995

[10] H. Hermanns, *"Interactive Markov Chains"*, Ph.D. Thesis, University of Erlangen (Germany), 1998

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

[12] J. Hillston, N. Thomas, *"Product Form Solution for a Class of PEPA Models"*, in Proc. of IPDS 1998, IEEE-CS Press, Durham (NC), 1998

[13] K. Kant, *"Introduction to Computer System Performance Evaluation"*, McGraw-Hill, 1992

[14] L. Kleinrock, *"Queueing Systems"*, John Wiley & Sons, 1975

[15] S.S. Lavenberg editor, *"Computer Performance Modeling Handbook"*, Academic Press, 1983

[16] R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989

[17] D.E. Perry, A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992

[18] M. Shaw, D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996

[19] M. Sereno, *"Towards a Product Form Solution for Stochastic Process Algebras"*, in Computer Journal 38:622-632, 1995

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

[21] Proc. of the *2nd Int. Workshop on Software and Performance (WOSP 2000)*, ACM Press, Ottawa (Canada), 2000