Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis

Simonetta Balsamo[1] Marco Bernardo[2] Marta Simeoni[1]

[1]Dipartimento di Informatica
Università "Ca' Foscari" di Venezia - Italy
[2]Centro per l'Applicazione delle Scienze e Tecnologie dell'Informazione
Università di Urbino - Italy

Abstract

In this paper 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 that is equipped with suitable checks for the detection of architectural mismatches. Although performance analysis could be carried on directly by Æmilia, it requires the construction and solution of the underlying Markov Chain (MC), whose state space explosion soon makes the analysis unfeasible. Moreover, since the MC model does not keep track of the structure of the specified SA, it is difficult to interpret the performance analysis results at the architectural description level.

Therefore, we propose a systematic approach to derive QN models from Æmilia specifications, 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 the architectural components directly fall into one of the three classes. Having a QN model allows us to evaluate performance indices possibly by exact product form solutions or by well known approximate methods. Furthermore, the high level of abstraction of the QN model makes it easier the interpretation of the performance results at the architectural description level.

1 Introduction

Software Architecture (SA) is an emerging discipline within software engineering aiming at describing both the structure and the behavior of software systems at a high level of abstraction [14]. The static and behavioral descriptions characterize, at an early stage of development, the basic design choices on the system being modeled, 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 [15, 16]. In particular, SAs have been devised as the appropriate design level to conduct early predictive performance analysis, allowing then 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. [8, 7, 6]) is a well known formal specification technique for concurrent and distributed systems, whose main features, i.e. composability – which allows the building of modular and hierarchical system descriptions – and abstraction – which allows the hiding of internal details of a specification – make them suited to work with at the architectural level of design. Besides the purely functional aspects, with SPA it is possible to express activity durations by using random variables. This allows the quantitative analysis of the modeled system through the construction and solution of the underlying stochastic process.

We start by introducing Æmilia, an architectural description language (ADL) based on an expressive SPA called $\mathrm{EMPA}_{\mathrm{gr}}$ [6]. Æmilia provides a formal specification language for the compositional, graphical, and hierarchical modeling of software systems, which is equipped with suitable checks for the detection of possible architectural mismatches inspired by [3, 4]. The timing of the durational actions is 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 which does not reflect the structure of the specified SA, thus making it difficult 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, buffer queues, and service processes – and on some restrictions to the Æmilia specifications, so that the architectural components directly fall into one of the three classes. Such restrictions can be easily verified at the syntax level.

QNs have been widely applied as system performance models [11, 10, 9]. Classical QNs represent resource sharing systems and can be solved by efficient algorithms. 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 [11, 10, 9]. Some approximate solution techniques have been defined for several types of extended QNs.

Our translation of Æmilia specifications leads to classical open or closed QNs, general (or phase-type) arrival and service time distribution 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 makes it easier 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 concurrently. We present their Æmilia graphical representation and formal specification, and the mapping to the corresponding QN models. The MC models associated with the Æmilia specification of these three SAs are infinite and require numeric approximate solutions. On the other hand, the translation of their Æmilia specification leads to the definition of simple QN models admitting either product form or approximate solutions.

The SAs modeling the sequential, pipeline and concurrent 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 which avoids the explicit construction of the underlying transition system.

The paper is organized as follows. In Sect. 2 we introduce the syntax, semantics and the 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 representation and formal specification and by showing their corresponding QNs. Concluding remarks and indications concerning future work are given in Sect. 5.

2 Æmilia: A Stochastic Process Algebra Based ADL

In this section we introduce the syntax, the semantics, and the architectural checks for Æmilia, an architectural description language (ADL) for the compositional, graphical, and hierarchical modeling of software systems. Æmilia is the result of the integration of two formalisms: PADL [3, 4] and EMPA_{gr} [6]. The former is a process algebra based ADL equipped with some architectural checks for the detection of architectural mismatches leading to system blocks. The latter is an expressive process algebra allowing for the performance modeling and analysis of concurrent and distributed systems. The integration of the two formalisms aims at developing a performance oriented ADL supporting suitable architectural checks. [1]

2.1 Syntax and Translation Semantics

Æmilia is based on the stochastic process algebra EMPA_{gr}, which we briefly recall 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] $L, S \subseteq \mathit{AType} - \{\tau\}$, 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. ■

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. The semantics for EMPA_{gr} is operationally defined through inference rules that map every term $E \in \mathcal{G}$ to a state transition graph $\mathcal{I}[\![E]\!]$ whose states are represented by process terms and whose transitions are labeled with actions. From the integrated semantic model $\mathcal{I}[\![E]\!]$ is it possible to derive, after pruning the lower priority

[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.

transitions, a purely functional semantic model $\mathcal{F}[\![E]\!]$ by removing action rates from the transitions, and a purely 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 Markov chain depending on whether $\mathcal{I}[\![E]\!]$ has exponentially timed transitions or not. A notion of equivalence called strong Markovian bisimulation is defined on the integrated semantic models of EMPA$_{\rm gr}$ terms and used to compositionally reduce the state space of such models. However, the notion of equivalence that we consider for EMPA$_{\rm gr}$ in this paper is the weak bisimulation equivalence [12], denoted $\approx_{\rm B}$, 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.

A description in Æmilia represents an architectural type (AT). The description of an AT starts with the name of the AT and its numeric parameters, which are values for exponential rates and weights. Each AT 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 AT, and its interactions, specified as a set of EMPA$_{\rm gr}$ action types occurring in the behavior. The architectural topology is specified through the declaration of a set of architectural element instances (AEIs), a set of architectural interactions (AIs) given by some interactions of the AEIs that act as interfaces for the whole AT, and a set of directed architectural attachments (DAAs) among the interactions of the AEIs. Every interaction is declared to be an input interaction or an output interaction and the DAAs must respect such a classification: every DAA must involve an output interaction and an input interaction of two different AEIs. Every interaction that is not an AI must be involved in at least one DAA. In order to allow for multi AEI synchronizations, every interaction can be involved in several DAAs, provided that no autosynchronization arises, i.e. no chain of DAAs is created that starts from an interaction of an AEI and terminates on an interaction of the same AEI. On the performance side, we require that, for the sake of modeling consistency, all the occurrences of an action type in the behavior of an AET have the same kind of rate (exponential or immediate with the same priority level or passive with the same priority level) and that, to comply with the synchronization discipline of EMPA$_{\rm gr}$, every chain of DAAs contains at most one interaction whose associated rate is exponential or immediate.

We show in Table 1 an Æmilia textual description for a performance closed AT representing a compiler system. We assume that programs to be compiled arrives 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, which is based on flow graphs [12].

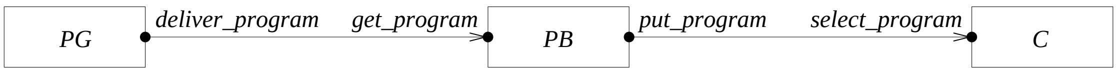


Figure 1: Flow graph of *CompilerSystem*

The semantics of an Æmilia specification is given by translation into EMPA$_{\rm gr}$ 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.

Definition 2.2 Given an AT, let $\mathcal{C}$ be an AET with behavior E and interaction set $\mathcal{I}$. The semantics of $\mathcal{C}$ and its instances is defined by

$$[\![\mathcal{C}]\!] = E / (AType - \{\tau\} - \mathcal{I})$$ ■

In our compiler system example we have

$$\begin{array}{rcrcl} [\![ProgramGeneratorT]\!] & = & [\![PG]\!] & = & ProgramGenerator / \{generate\_program\} \\ [\![ProgramBufferT]\!] & = & [\![PB]\!] & = & ProgramBuffer_0 \\ [\![CompilerT]\!] & = & [\![C]\!] & = & Compiler / \{recognize\_tokens, parse\_phrases, check\_phrases, \\ & & & & \qquad opt\_yes, opt\_no, optimize, generate\_code\} \end{array}$$

archi_type	$\mathit{CompilerSystem}(\textbf{exp\_rates}\ \lambda_{prog}, \mu_{lexer}, \mu_{parser}, \mu_{checker}, \mu_{optimizer}, \mu_{generator};$ $\textbf{weights}\ \mathit{prob}_{optimizer})$
archi_elem_types	
elem_type	$\mathit{ProgramGeneratorT}(\textbf{exp\_rates}\ \lambda;)$
behavior	$\mathit{ProgramGenerator} \stackrel{\Delta}{=} <\mathit{generate\_program}, \lambda>.$ $<\mathit{deliver\_program}, \infty>.\mathit{ProgramGenerator}$
interactions	**output** $\mathit{deliver\_program}$
elem_type	$\mathit{ProgramBufferT}$
behavior	$\mathit{ProgramBuffer}_0 \stackrel{\Delta}{=} <\mathit{get\_program}, *>.\mathit{ProgramBuffer}_1$
	$\mathit{ProgramBuffer}_i \stackrel{\Delta}{=} <\mathit{get\_program}, *>.\mathit{ProgramBuffer}_{i+1} +$ $<\mathit{put\_program}, *>.\mathit{ProgramBuffer}_{i-1} \quad i \geq 1$
interactions	**input** $\mathit{get\_program}$
	output $\mathit{put\_program}$
elem_type	$\mathit{CompilerT}(\textbf{exp\_rates}\ \mu_l, \mu_p, \mu_c, \mu_o, \mu_g;$ $\textbf{weights}\ p_o)$
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** $\mathit{select\_program}$
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** $\mathit{PG.deliver\_program}$ **to** $\mathit{PB.get\_program}$
	from $\mathit{PB.put\_program}$ **to** $\mathit{C.select\_program}$
end	

Table 1: Textual description of *CompilerSystem*

In the second step, the semantics of an AT is obtained by composing in parallel the semantics of its AEIs according to the specified DAAs. In our compiler system example we have

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

where the use of the relabeling operator is necessary to make the AEIs interact. In general, let $C_1, \dots, C_n$ be AEIs of an AT, with interaction sets $\mathcal{I}_{C_1}, \dots, \mathcal{I}_{C_n}$ containing the AI sets $\mathcal{AI}_{C_1}, \dots, \mathcal{AI}_{C_n}$, respectively. Let i, j, k range over $\{1, \dots, n\}$. We say that $C_i.a_1$ is connected to $C_j.a_2$ iff either there is a DAA between them, or there exists an interaction a_3 of C_k such that $C_i.a_1$ is connected to $C_k.a_3$ and there is a DAA between $C_k.a_3$ and $C_j.a_2$. We say that a subset of interactions of $C_1, \dots, C_n$ is connected iff they are pairwise connected via DAAs involving interactions of $C_1, \dots, C_n$ only and the subset is maximal. Since the actions of a connected subset of interactions must be identically relabeled in order to result in a synchronization at the semantic level, denoted by $\mathcal{I}_{C_i;C_1,\dots,C_n} \subseteq \mathcal{I}_{C_i}$ the subset of interactions of C_i attached to $C_1, \dots, C_n$, let $\mathcal{S}(C_1, \dots, C_n)$ be a set of as many fresh actions as there are connected subsets of interactions among the considered AEIs, let $\varphi_{C_i;C_1,\dots,C_n} : \mathcal{I}_{C_i;C_1,\dots,C_n} \longrightarrow \mathcal{S}(C_1, \dots, C_n)$ be injective relabeling functions such that $\varphi_{C_i;C_1,\dots,C_n}(a_1) = \varphi_{C_j;C_1,\dots,C_n}(a_2)$ iff $C_i.a_1$ is connected to $C_j.a_2$, and let $\mathcal{S}(C_i; C_1, \dots, C_n) = \varphi_{C_i;C_1,\dots,C_n}(\mathcal{I}_{C_i;C_1,\dots,C_n})$ and $\mathcal{S}(C_i, C_j; C_1, \dots, C_n) = \mathcal{S}(C_i; C_1, \dots, C_n) \cap \mathcal{S}(C_j; C_1, \dots, C_n)$.

Definition 2.3 Let $C_1, \dots, C_n$ be AEIs of an AT. The closed and the open interacting semantics of C_i restricted to $C_1, \dots, C_n$ are defined by

$$\begin{aligned}[\![C_i]\!]^{\mathrm{c}}_{C_1,\dots,C_n} &= [\![C_i]\!] \;/\; (\mathit{AType} - \{\tau\} - \mathcal{I}_{C_i;C_1,\dots,C_n}) \;[\varphi_{C_i;C_1,\dots,C_n}] \\ [\![C_i]\!]^{\mathrm{o}}_{C_1,\dots,C_n} &= [\![C_i]\!] \;/\; (\mathit{AType} - \{\tau\} - (\mathcal{I}_{C_i;C_1,\dots,C_n} \cup \mathcal{AI}_{C_i})) \;[\varphi_{C_i;C_1,\dots,C_n}]\end{aligned}$$

■

Definition 2.4 Let $C_1, \dots, C_n$ be AEIs of an AT. The closed and the open interacting semantics of the set of AEIs are defined by

$$\begin{aligned}[\![C_1, \dots, C_n]\!]^{\mathrm{c}} = \; & [\![C_1]\!]^{\mathrm{c}}_{C_1,\dots,C_n} \;\|_{\mathcal{S}(C_1,C_2;C_1,\dots,C_n)} \\ & [\![C_2]\!]^{\mathrm{c}}_{C_1,\dots,C_n} \;\|_{\mathcal{S}(C_1,C_3;C_1,\dots,C_n) \cup \mathcal{S}(C_2,C_3;C_1,\dots,C_n)} \cdots \\ & \cdots \;\|_{\cup_{i=1}^{n-1} \mathcal{S}(C_i,C_n;C_1,\dots,C_n)} \; [\![C_n]\!]^{\mathrm{c}}_{C_1,\dots,C_n} \\ [\![C_1, \dots, C_n]\!]^{\mathrm{o}} = \; & [\![C_1]\!]^{\mathrm{o}}_{C_1,\dots,C_n} \;\|_{\mathcal{S}(C_1,C_2;C_1,\dots,C_n)} \\ & [\![C_2]\!]^{\mathrm{o}}_{C_1,\dots,C_n} \;\|_{\mathcal{S}(C_1,C_3;C_1,\dots,C_n) \cup \mathcal{S}(C_2,C_3;C_1,\dots,C_n)} \cdots \\ & \cdots \;\|_{\cup_{i=1}^{n-1} \mathcal{S}(C_i,C_n;C_1,\dots,C_n)} \; [\![C_n]\!]^{\mathrm{o}}_{C_1,\dots,C_n}\end{aligned}$$

■

Definition 2.5 The semantics of an AT $\mathcal{A}$ with AEIs $C_1, \dots, C_n$ is defined by

$$[\![\mathcal{A}]\!] = [\![C_1, \dots, C_n]\!]^{\mathrm{o}}$$

■

Given such a translation of an Æmilia AT $\mathcal{A}$ into an EMPA$_{\mathrm{gr}}$ term $[\![\mathcal{A}]\!]$, $\mathcal{A}$ is equipped with an integrated semantic model $\mathcal{I}[\![[\![\mathcal{A}]\!]]\!]$, a functional semantic model $\mathcal{F}[\![[\![\mathcal{A}]\!]]\!]$, and a performance semantic model $\mathcal{M}[\![[\![\mathcal{A}]\!]]\!]$ (if $\mathcal{I}[\![[\![\mathcal{A}]\!]]\!]$ has no passive transitions). Such models can be used to predict functional and performance properties of the software system via functional verification (model checking, equivalence/preorder checking) and Markovian/simulation analysis with the TwoTowers tool [5].

2.2 Architectural Checks

Æmilia is equipped with two checks inherited from PADL for the detection of architectural mismatches resulting in deadlocks when combining deadlock free AEIs. The first check (compatibility) is concerned with the well formedness of acyclic ATs, while the second check (interoperability) is concerned with the well formedness of sets of AEIs forming a cycle. The validity of such checks for PADL is based on the fact that $\approx_{\mathrm{B}}$, besides preserving deadlock freedom, is a congruence for the static operators of a classical process algebra. $\approx_{\mathrm{B}}$ still is a congruence for the static operators when applied to the integrated semantics for EMPA$_{\mathrm{gr}}$ because it retains the lower priority transitions. However, in the functional semantics of an Æmilia description deadlocks can actually arise in the presence of highest priority transitions corresponding to nonpassive actions that are not interactions, as they always prevent their alternative interactions from taking place. In order to restore the validity of the compatibility and interoperability checks in the case of the functional semantics for Æmilia, it suffices to assume that the behavior of each AET is given by a sequential $\mathrm{EMPA}_{\mathrm{gr}}$ that is correctly prioritized w.r.t. the actions whose type belongs to a set S including at least all the action types that are not interactions, which means that in every alternative composition $\sum_{h \in H} <a_h, \tilde{\lambda}_h>.E_h$ of this term there is no summand whose initial action has type in S, nonpassive rate, and priority greater than the priority of the initial action of another summand, if the latter action is a nonpassive interaction whose type is not in S, or the priority of a nonpassive interaction of another AET that is attached to the initial action of another summand, if the latter action is a passive interaction. Note that such a constraint can be syntactically checked, and is reasonable from the modeling viewpoint as far as the actions that are not interactions are concerned.

Definition 2.6 Given an acyclic AT, let $C_1, \dots, C_n$ be the AEIs attached to AEI K. C_i is said to be compatible with K iff

$$[\![K]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n} \parallel_{\mathcal{S}(K;K,C_1,\dots,C_n)} [\![C_i]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n} \approx_{\mathrm{B}} [\![K]\!]^{\mathrm{c}}_{K,C_1,\dots,C_n}$$ ■

Theorem 2.7 Given an acyclic AT, let $C_1, \dots, C_n$ be the AEIs attached to AEI K. If the behavior of $K, C_1, \dots, C_n$ is correctly prioritized w.r.t. the actions whose type is not an interaction or is an interaction not attached to the considered AEIs, 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

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

is deadlock free. ■

Corollary 2.8 Given an acyclic AT, if the behavior of each AET is correctly prioritized w.r.t. the actions whose type is not an interaction, the functional semantics of every restricted closed interacting semantics of each AEI is deadlock free, and every AEI is compatible with each AEI attached to it, then the functional semantics of the AT is deadlock free. ■

Definition 2.9 Given an AT, let $C_1, \dots, C_n$ be AEIs forming a cycle. C_i is said to interoperate with $C_1, \dots, C_{i-1}, C_{i+1}, \dots, C_n$ iff

$$[\![C_1, \dots, C_n]\!]^{\mathrm{c}} / (\mathit{AType} - \{\tau\} - \mathcal{S}(C_i; C_1, \dots, C_n)) \approx_{\mathrm{B}} [\![C_i]\!]^{\mathrm{c}}_{C_1,\dots,C_n}$$ ■

Theorem 2.10 Given an AT, let $C_1, \dots, C_n$ be AEIs forming a cycle. If the behavior of $C_1, \dots, C_n$ is correctly prioritized w.r.t. the actions whose type is not an interaction or is an interaction not attached to the considered AEIs 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 new check for the detection of architectural mismatches resulting in performance underspecification. This check (performance closure) ensures that the performance semantic model underlying an AT exists in the form of a continuous or discrete time Markov chain.

Definition 2.11 An AT $\mathcal{A}$ is said to be performance closed iff $\mathcal{I}[\![[\![\mathcal{A}]\!]]\!]$ has no passive transitions. ■

Theorem 2.12 An AT $\mathcal{A}$ is performance closed iff no AET contains in its behavior a passive action whose type is not an interaction and every chain of DAAs contains one interaction whose associated rate is exponential or immediate. ■

An Æmilia description represents a family of software architectures called an AT. 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 AT can be obtained by invoking the AT and passing actual AETs preserving the observable functional behavior of the formal AETs, actual names for the AIs, and actual values for the numeric parameters. Æmilia thus inherits from PADL a check to verify whether an AT invocation conforms to an AT definition, in such a way that compatibility, interoperability, and performance closure are preserved. The conformity check is strengthen w.r.t. PADL as we additionally require the rate preservation between corresponding interactions of actual and formal AETs.

Definition 2.13 Let $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l; \lambda'_1, \ldots, \lambda'_h; w'_1, \ldots, w'_k)$ be an invocation of the AT $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$, formal AIs $a_1, \ldots, a_l$, and formal numeric parameters $\lambda_1, \ldots, \lambda_h, w_1, \ldots, w_k$. $\mathcal{C}'_i$ is said to conform 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

$$[\![\mathcal{C}'_i]\!][\varphi'_i] \approx_{\mathrm{B}} [\![\mathcal{C}_i]\!]\{\lambda'_1/\lambda_1, \ldots, \lambda'_h/\lambda_h, w'_1/w_1, \ldots, w'_k/w_k\}[\varphi_i]$$

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 $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l; \lambda'_1, \ldots, \lambda'_h; w'_1, \ldots, w'_k)$ be an invocation of the AT $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$, formal AIs $a_1, \ldots, a_l$, and formal numeric parameters $\lambda_1, \ldots, \lambda_h, w_1, \ldots, w_k$. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1, \ldots, m$, then the semantics of the AT invocation is defined by

$$\begin{aligned}&[\![\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l; \lambda'_1, \ldots, \lambda'_h; w'_1, \ldots, w'_k)]\!] = \\ &\qquad [\![\mathcal{A}]\!]\{\lambda'_1/\lambda_1, \ldots, \lambda'_h/\lambda_h, w'_1/w_1, \ldots, w'_k/w_k\}[a_1 \mapsto a'_1, \ldots, a_l \mapsto a'_l]\end{aligned}$$

■

Theorem 2.15 Let $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l; \lambda'_1, \ldots, \lambda'_h; w'_1, \ldots, w'_k)$ be an invocation of the AT $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$, formal AIs $a_1, \ldots, a_l$, and formal numeric parameters $\lambda_1, \ldots, \lambda_h, w_1, \ldots, w_k$. Let $C'_1, \ldots, C'_n$ be the AEIs of the AT invocation. If $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$ for all $i = 1, \ldots, m$, then there exist an injective relabeling function φ' for the interactions of the AT invocation and an injective relabeling function φ for the interactions of the AT definition such that

$$[\![C'_1, \ldots, C'_n]\!]^{\mathrm{o}}[\varphi'] \approx_{\mathrm{B}} [\![\mathcal{A}]\!]\{\lambda'_1/\lambda_1, \ldots, \lambda'_h/\lambda_h, w'_1/w_1, \ldots, w'_k/w_k\}[\varphi]$$

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

Theorem 2.16 Let $\mathcal{A}(\mathcal{C}'_1, \ldots, \mathcal{C}'_m; a'_1, \ldots, a'_l; \lambda'_1, \ldots, \lambda'_h; w'_1, \ldots, w'_k)$ be an invocation of the AT $\mathcal{A}$ defined with formal AETs $\mathcal{C}_1, \ldots, \mathcal{C}_m$, formal AIs $a_1, \ldots, a_l$, and formal numeric parameters $\lambda_1, \ldots, \lambda_h, w_1, \ldots, w_k$. If for all $i = 1, \ldots, m$ $\mathcal{C}'_i$ conforms to $\mathcal{C}_i$, $\mathcal{C}'_i$ is correctly prioritized w.r.t. the actions whose type is not an interaction 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 AT invocation and the AT 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 AT is provided with a Markov chain model on which we can compute the performance metrics for the system under consideration. Unfortunately, such a Markov chain model has two drawbacks. First, it suffers from the state space explosion problem, i.e. the fact that the number of states of the model 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 reinterpretation 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 reinterpretation 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 DAAs.

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:

$$Arrival \stackrel{\Delta}{=} <generate, \lambda>.<deliver, \infty>.Arrival$$

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:

$$\begin{array}{rcl} Queue_0 & \stackrel{\Delta}{=} & <arrive, *>.Queue_1 \\ Queue_i & \stackrel{\Delta}{=} & <arrive, *>.Queue_{i+1} + <depart, *>.Queue_{i-1}, \;\; 0 < i < n \\ Queue_n & \stackrel{\Delta}{=} & <depart, *>.Queue_{n-1} \end{array}$$

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:

$$Server \stackrel{\Delta}{=} <select, \infty>.<serve, \mu>.<leave, \infty>.Server$$

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:

$$Server \stackrel{\Delta}{=} <select, *>.<serve, \mu>.<leave, \infty>.Server$$

As can be noted in the examples above, 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.

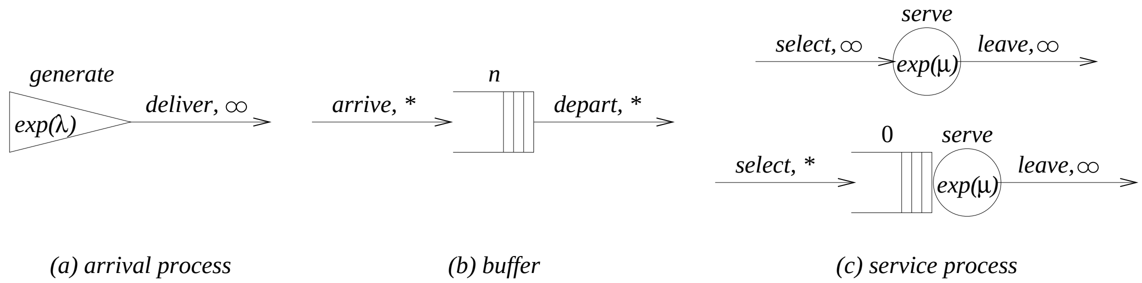


Figure 2: Graphical representation of the exemplified QN basic elements

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, and given the fact that an AET models a sequential software component running on a single computational resource, we impose the following restrictions so that every AET described in Æmilia falls into one of the three classes:

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.

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 making it possible a 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 the correct modeling of 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 the correct modeling of 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, where 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, i.e. they do not require the construction of the state space of the AETs. For instance in our compiler system example 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, we observe that such syntax restrictions do not allow for preemption, i.e. the fact that the service of a request of a certain type can be interrupted due to the arrival of a request of another type having higher service priority.

3.3 Mapping AEIs to QN Basic Elements

The first step of the translation of a performance closed AT 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 $\mathcal{G}_{\rm seq,buf}$ 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 function, $input : \mathcal{G}_{\rm seq} \mathrel{-\!\!\circ\!\!\!\rightarrow} 2^{Act}$, 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, $input(E)$ 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

$$\begin{array}{rcl} input(ProgramBuffer_0) & = & \{<get\_program, *>\} \\ input(Compiler) & = & \{<select\_program, \infty>\} \end{array}$$

The second function, $s\_policy : \mathcal{G}_{\rm seq} \mathrel{-\!\!\circ\!\!\!\rightarrow} STG$, 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, $s\_policy(E)$ 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 $s\_policy(Compiler)$ is as depicted below:

$$\bullet \xrightarrow{get\_program,\ \infty} \bullet$$

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

The third function, $process : \mathcal{G}_{\rm seq} \mathrel{-\!\!\circ\!\!\!\rightarrow} 2^{AType}$, 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, $process(E)$ 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

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

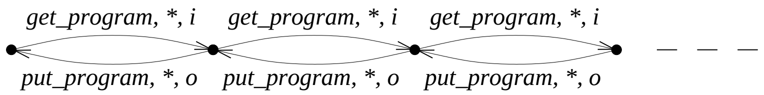
The fourth function, $ptdistr : \mathcal{G}_{\mathrm{seq}} \;{-\!\!\circ\!\!\!\rightarrow}\; PTDistr$, computes the phase type distribution associated with the generation or service of a request. If we consider Fig. 2, the application of function *ptdistr* results in the probability distribution labeling 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, $ptdistr(E)$ is defined as the smallest phase type distribution satisfying the following equalities:

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

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

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

The fifth function, $b\_policy : \mathcal{G}_{\mathrm{seq,buf}} \longrightarrow STG'$, describes the policy according to which the requests are accepted and delivered by a buffer. Given an AEI representing a buffer with behavior E, $b\_policy(E)$ 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. In the compiler system example of Table 1 we have that $b\_policy(ProgramBuffer_0)$ 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 function, $b\_capacity : \mathcal{G}_{\mathrm{seq,buf}} \times AType \;{-\!\!\circ\!\!\!\rightarrow}\; \mathbb{N} \cup \{\infty\}$, 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}, $b\_capacity(E, a_{in})$ is computed on the basis of $b\_policy(E)$ as follows. If $b\_policy(E)$ contains a cycle composed of only input interactions among which a_{in}, then $b\_capacity(E, a_{in}) = \infty$. If this is not the case, we consider every maximal simple [3] path in $b\_policy(E)$ 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). $b\_capacity(E, a_{in})$ is then given by the maximum of such values. In the compiler system example of Table 1 we have

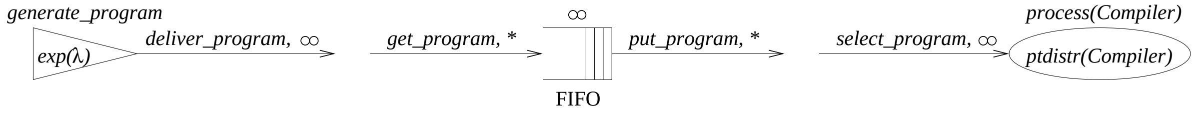
$$b\_capacity(ProgramBuffer_0, get\_program) = \infty$$

[3] Traversing each state at most once.

The seventh function, $output : \mathcal{G}_{\text{seq}} \longrightarrow 2^{Act}$, 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, $output(E)$ 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

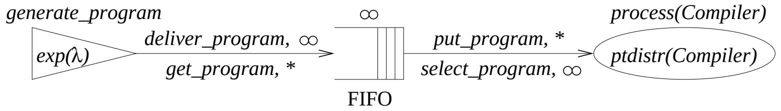
$$\begin{array}{rcl} output(ProgramGenerator) & = & \{<deliver\_program, \infty>\} \\ output(ProgramBuffer_0) & = & \{<put\_program, *>\} \\ output(Compiler) & = & \emptyset \end{array}$$

We conclude this section by showing below the three QN basic elements resulting from the first step of the translation of the performance closed AT representing the compiler system of Table 1:



3.4 Connecting QN Basic Elements

The second step of the translation of a performance closed AT into a QN simply consists of connecting the basic QN elements stemming from the translation of the AEIs according to the DAAs. 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 below:



4 Modeling Two Alternative Architectures for the Compiler System

In this section we propose an application example of the proposed approach to two alternative architectures for the compiler system, similar to the one presented in [1]. We describe the Æmilia specification for two possible SAs and we apply the mapping described in Sect. 3 to derive the associated QN model.

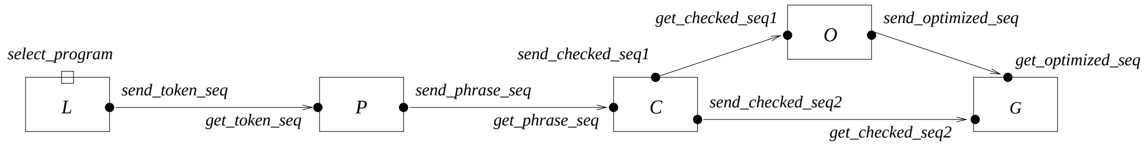


Figure 3: Flow graph of *PipelineCompiler*

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. With respect to the compiler system depicted in Table 1 we replace the *CompilerT* AET with the following one, whose behavior is ruled by the *PipelineCompiler* AT:

elem_type	$CompilerT(\textbf{exp\_rates}\ \mu_l, \mu_p, \mu_c, \mu_o, \mu_g; \textbf{weights}\ p_o)$
behavior	$Compiler \triangleq PipelineCompiler(; select\_program; \mu_l, \mu_p, \mu_c, \mu_o, \mu_g; p_o)$
interactions	**input** *select_program*

where the absence of actual AETs in the AT invocation means that they coincide with the formal ones. The Æmilia graphical representation of the *PipelineCompiler* AT is shown in Fig. 3, while its Æmilia specification is shown in Table 2. The pipeline connection between the different compiler phases is realized by directly connecting their corresponding AETs. By applying to each AEI of the *PipelineCompiler* AT the mappings defined in Sect. 3 we get their corresponding QN basic elements, and by connecting them according to the DAAs we obtain the QN model depicted in Fig. 4 for the whole compiler system with the pipeline compiler. 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. 3, and this makes it easier to interpret at the SA level the performance results obtained by the analysis of the QN model.

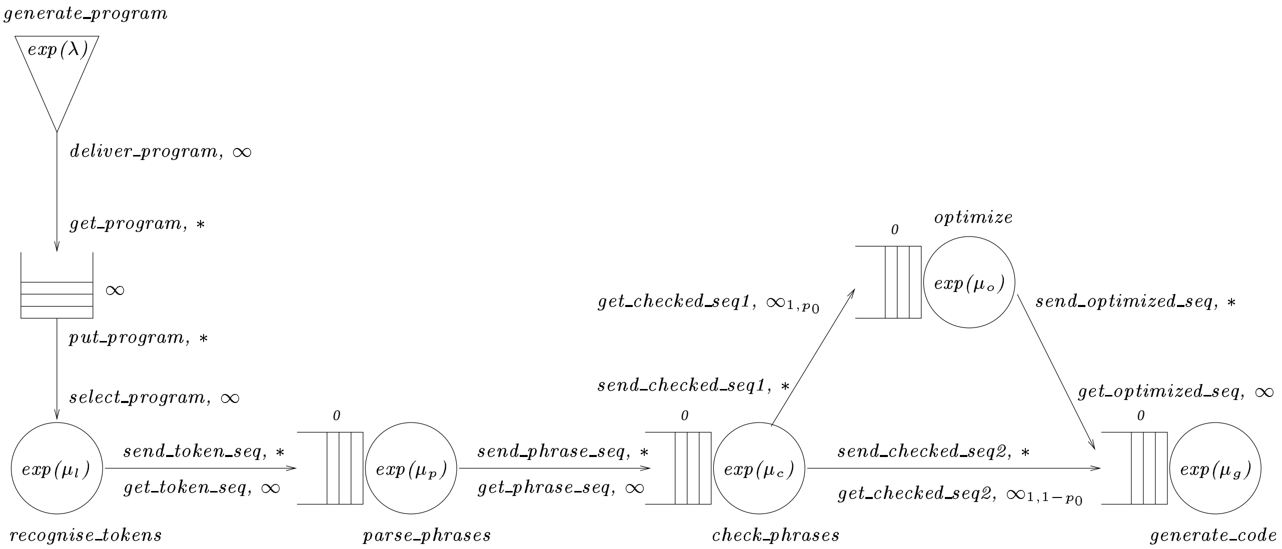


Figure 4: QN model of the compiler system with *PipelineCompiler*

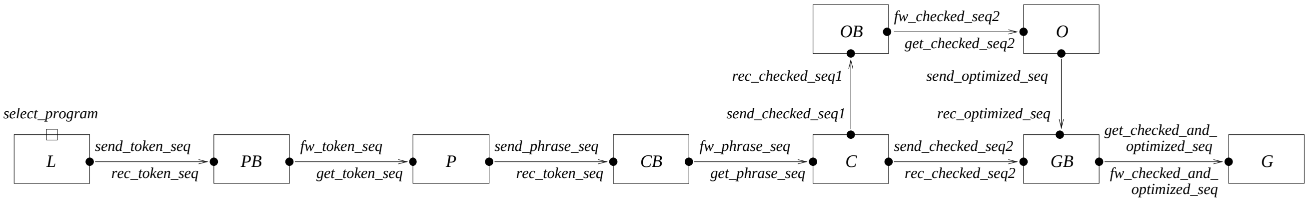


Figure 5: Flow graph of *ConcurrentCompiler*

We consider a second alternative architecture of the compiler allowing the parsing, checking, optimizing and generating phases to proceed in parallel. This is achieved by providing the *ParserT*, *CheckerT*, *OptimizerT* and *GeneratorT* AETs of the *PipelineCompiler* AT with their own (infinite) buffer, in such a way that they can concurrently work on the compilation of different source programs. In this case with respect to Table 1, we replace the *CompilerT* AET with the following one, whose behavior is ruled by the *ConcurrentCompiler* AT:

archi_type	$\mathit{PipelineCompiler}(\textbf{exp\_rates}\ \mu_{lexer}, \mu_{parser}, \mu_{checker}, \mu_{optimizer}, \mu_{generator};$ $\textbf{weights}\ \mathit{prob}_{optimizer})$
archi_elem_types	
elem_type	$\mathit{LexerT}(\textbf{exp\_rates}\ \mu_l;)$
behavior	$\mathit{Lexer} \stackrel{\Delta}{=} <\mathit{select\_program}, \infty>.<\mathit{recognize\_tokens}, \mu_l>.$ $<\mathit{send\_token\_seq}, \infty>.\mathit{Lexer}$
interactions	**input** $\mathit{select\_program}$ **output** $\mathit{send\_token\_seq}$
elem_type	$\mathit{ParserT}(\textbf{exp\_rates}\ \mu_p;)$
behavior	$\mathit{Parser} \stackrel{\Delta}{=} <\mathit{get\_token\_seq}, *>.<\mathit{parse\_phrases}, \mu_p>.$ $<\mathit{send\_phrase\_seq}, \infty>.\mathit{Parser}$
interactions	**input** $\mathit{get\_token\_seq}$ **output** $\mathit{send\_phrase\_seq}$
elem_type	$\mathit{CheckerT}(\textbf{exp\_rates}\ \mu_c; \textbf{weights}\ p_o)$
behavior	$\mathit{Checker} \stackrel{\Delta}{=} <\mathit{get\_phrase\_seq}, *>.<\mathit{check\_phrases}, \mu_c>.$ $(<\mathit{send\_checked\_seq1}, \infty_{1,p_o}>.\mathit{Checker} +$ $<\mathit{send\_checked\_seq2}, \infty_{1,1-p_o}>.\mathit{Checker})$
interactions	**input** $\mathit{get\_phrase\_seq}$ **output** $\mathit{send\_checked\_seq1}, \mathit{send\_checked\_seq2}$
elem_type	$\mathit{OptimizerT}(\textbf{exp\_rates}\ \mu_o;)$
behavior	$\mathit{Optimizer} \stackrel{\Delta}{=} <\mathit{get\_checked\_seq1}, *>.<\mathit{optimize}, \mu_o>.$ $<\mathit{send\_optimized\_seq}, \infty>.\mathit{Optimizer}$
interactions	**input** $\mathit{get\_checked\_seq1}$ **output** $\mathit{send\_optimized\_seq}$
elem_type	$\mathit{GeneratorT}(\textbf{exp\_rates}\ \mu_g;)$
behavior	$\mathit{Generator} \stackrel{\Delta}{=} <\mathit{get\_checked\_seq2}, *>.<\mathit{generate\_code}, \mu_g>.\mathit{Generator} +$ $<\mathit{get\_optimized\_seq}, *>.<\mathit{generate\_code}, \mu_g>.\mathit{Generator}$
interactions	**input** $\mathit{get\_checked\_seq2}, \mathit{get\_optimized\_seq}$
archi_topology	
archi_elem_instances	$L : \mathit{LexerT}(\mu_{lexer};)$ $P : \mathit{ParserT}(\mu_{parser};)$ $C : \mathit{CheckerT}(\mu_{checker}; \mathit{prob}_{optimizer})$ $O : \mathit{OptimizerT}(\mu_{optimizer};)$ $G : \mathit{GeneratorT}(\mu_{generator};)$
archi_interactions	$L.\mathit{select\_program}$
archi_attachments	**from** $L.\mathit{send\_token\_seq}$ **to** $P.\mathit{get\_token\_seq}$ **from** $P.\mathit{send\_phrase\_seq}$ **to** $C.\mathit{get\_phrase\_seq}$ **from** $C.\mathit{send\_checked\_seq1}$ **to** $O.\mathit{get\_checked\_seq1}$ **from** $C.\mathit{send\_checked\_seq2}$ **to** $G.\mathit{get\_checked\_seq2}$ **from** $O.\mathit{send\_optimized\_seq}$ **to** $G.\mathit{get\_optimized\_seq}$
end	

Table 2: Textual description of *PipelineCompiler*

$\textbf{elem\_type} \quad \mathit{CompilerT}(\textbf{exp\_rates}\ \mu_l, \mu_p, \mu_c, \mu_o, \mu_g; \textbf{weights}\ p_o)$

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

$\textbf{interactions} \quad \textbf{input}\ \mathit{select\_program}$

The Æmilia graphical representation of the *ConcurrentCompiler* AT is depicted in Fig. 5, while its Æmilia specification is shown in Table 3. The QN model associated with the Æmilia specification of the compiler system with the concurrent compiler is very similar to the one representing the compiler system with the pipeline compiler. 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 specification of the sequential, pipeline and concurrent SAs of the compiler system are infinite and require numerical approximate solutions. On the other hand, the corresponding QN models can be solved either by product-form algorithms or by approximate methods.

The three SAs of this compiler example 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 SA example by using the methodology in [1] and by using our mapping coincide. The detailed evaluation and comparison of such QN models is out of the scope of this paper and a discussion can be found in [1]. In particular, 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 QN of the sequential and concurrent SA show a product-form solution and the QN for the parallel compiler can be solved by approximate methods.

In [1] labeled transition systems have been used as specification description, in order to allow the independence from any specific ADL. In our case, however, by considering Æmilia as specification language we can define a direct mapping to QNs which avoids the explicit construction of the underlying transition system and ease the interpretation of the performance results at the architectural description level.

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 ADL based on $\mathrm{EMPA}_{\mathrm{gr}}$ for the compositional, graphical and hierarchical modeling of SAs that is equipped with suitable checks inspired by [3, 4] for the detection of architectural mismatches, and by defining a translation of Æmilia specifications into QN models, with the goals of exploiting the efficient solution methods of QNs and reinterpreting the performance figures at the architectural description level.

It is worth observing that, in order for our translation be effective, we have imposed some syntactical restrictions on the Æmilia specifications. The reason is that, while SPAs allow the architect to model systems in a very liberal way, basic QNs only provide a few basic elements: arrival processes, queues, and servers. The syntactical restrictions determine a more constrained way of modeling systems that allows QNs to be extracted from Æmilia specifications.

We focus on basic QNs to take advantage of their efficient analytical methods, such as product-form solution algorithms to derive performance indices. However, we can consider extended QNs that allow more general system modeling and include some other element types such as fork and join nodes, subnetwork population constraints, finite queue capacity and blocking mechanisms, multiple classes of customers and various scheduling discipline [11, 10, 9]. However, most of extended QNs solution requires numeric approximate methods or simulation. A translation of Æmilia specifications into extended QNs models is out of the scope of this paper.

From the complexity viewpoint, our QN based translation is linear in the number of software components. Therefore, it is clearly advantageous w.r.t. the MC based translation, as this results in a state space whose size grows exponentially with the number of software components.

archi_type	$\mathit{ConcurrentCompiler}(\textbf{exp\_rates}\ \mu_{lexer}, \mu_{parser}, \mu_{checker}, \mu_{optimizer}, \mu_{generator};$ $\textbf{weights}\ prob_{optimizer})$
archi_elem_types	
elem_type	$\mathit{LexerT}(\textbf{exp\_rates}\ \mu_l;)$
behavior	$\mathit{Lexer} \stackrel{\Delta}{=} <\mathit{select\_program}, \infty>.<\mathit{recognize\_tokens}, \mu_l>.<\mathit{send\_token\_seq}, \infty>.\mathit{Lexer}$
interactions	**input** $\mathit{select\_program}$ **output** $\mathit{send\_token\_seq}$
elem_type	$\mathit{PBufferT}$
behavior	$\mathit{PBuffer}_0 \stackrel{\Delta}{=} <\mathit{rec\_token\_seq}, *>.\mathit{PBuffer}_1$ $\mathit{PBuffer}_i \stackrel{\Delta}{=} <\mathit{rec\_token\_seq}, *>.\mathit{PBuffer}_{i+1} + <\mathit{fw\_token\_seq}, *>.\mathit{PBuffer}_{i-1} \qquad i \geq 1$
interactions	**input** $\mathit{rec\_token\_seq}$ **output** $\mathit{fw\_token\_seq}$
elem_type	$\mathit{ParserT}(\textbf{exp\_rates}\ \mu_p;)$
behavior	$\mathit{Parser} \stackrel{\Delta}{=} <\mathit{get\_token\_seq}, \infty>.<\mathit{parse\_phrases}, \mu_p>.<\mathit{send\_phrase\_seq}, \infty>.\mathit{Parser}$
interactions	**input** $\mathit{get\_token\_seq}$ **output** $\mathit{send\_phrase\_seq}$
elem_type	$\mathit{CBufferT}$
behavior	$\mathit{CBuffer}_0 \stackrel{\Delta}{=} <\mathit{rec\_phrase\_seq}, *>.\mathit{CBuffer}_1$ $\mathit{CBuffer}_i \stackrel{\Delta}{=} <\mathit{rec\_phrase\_seq}, *>.\mathit{CBuffer}_{i+1} + <\mathit{fw\_phrase\_seq}, *>.\mathit{CBuffer}_{i-1} \qquad i \geq 1$
interactions	**input** $\mathit{rec\_phrase\_seq}$ **output** $\mathit{fw\_phrase\_seq}$
elem_type	$\mathit{CheckerT}(\textbf{exp\_rates}\ \mu_c; \textbf{weights}\ p_o)$
behavior	$\mathit{Checker} \stackrel{\Delta}{=} <\mathit{get\_phrase\_seq}, \infty>.<\mathit{check\_phrases}, \mu_c>.$ $(<\mathit{send\_checked\_seq1}, \infty_{1,p_o}>.\mathit{Checker} + <\mathit{send\_checked\_seq2}, \infty_{1,1-p_o}>.\mathit{Checker})$
interactions	**input** $\mathit{get\_phrase\_seq}$ **output** $\mathit{send\_checked\_seq1}, \mathit{send\_checked\_seq2}$
elem_type	$\mathit{OBufferT}$
behavior	$\mathit{OBuffer}_0 \stackrel{\Delta}{=} <\mathit{rec\_checked\_seq1}, *>.\mathit{OBuffer}_1$ $\mathit{OBuffer}_i \stackrel{\Delta}{=} <\mathit{rec\_checked\_seq1}, *>.\mathit{OBuffer}_{i+1} + <\mathit{fw\_checked\_seq1}, *>.\mathit{OBuffer}_{i-1} \qquad i \geq 1$
interactions	**input** $\mathit{rec\_checked\_seq1}$ **output** $\mathit{fw\_checked\_seq1}$
elem_type	$\mathit{OptimizerT}(\textbf{exp\_rates}\ \mu_o;)$
behavior	$\mathit{Optimizer} \stackrel{\Delta}{=} <\mathit{get\_checked\_seq1}, \infty>.<\mathit{optimize}, \mu_o>.<\mathit{send\_optimized\_seq}, \infty>.\mathit{Optimizer}$
interactions	**input** $\mathit{get\_checked\_seq1}$ **output** $\mathit{send\_optimized\_seq}$
elem_type	$\mathit{GBufferT}$
behavior	$\mathit{GBuffer}_0 \stackrel{\Delta}{=} <\mathit{rec\_checked\_seq2}, *>.\mathit{GBuffer}_1 + <\mathit{rec\_optimized\_seq}, *>.\mathit{GBuffer}_1$ $\mathit{GBuffer}_i \stackrel{\Delta}{=} <\mathit{rec\_checked\_seq2}, *>.\mathit{GBuffer}_{i+1} + <\mathit{rec\_optimized\_seq}, *>.\mathit{GBuffer}_{i+1} +$ $<\mathit{fw\_checked\_and\_optimized\_seq}, *>.\mathit{GBuffer}_{i-1} \qquad i \geq 1$
interactions	**input** $\mathit{rec\_checked\_seq2}, \mathit{rec\_optimized\_seq}$ **output** $\mathit{fw\_checked\_and\_optimized\_seq}$
elem_type	$\mathit{GeneratorT}(\textbf{exp\_rates}\ \mu_g;)$
behavior	$\mathit{Generator} \stackrel{\Delta}{=} <\mathit{get\_checked\_and\_optimized\_seq}, \infty>.<\mathit{generate\_code}, \mu_g>.\mathit{Generator}$
interactions	**input** $\mathit{get\_checked\_and\_optimized\_seq}$
archi_topology	
archi_elem_instances	$L : \mathit{LexerT}(\mu_{lexer};)$ $PB : \mathit{PBufferT}$ $P : \mathit{ParserT}(\mu_{parser};)$ $CB : \mathit{CBufferT}$ $C : \mathit{CheckerT}(\mu_{checker}; prob_{optimizer})$ $OB : \mathit{OBufferT}$ $O : \mathit{OptimizerT}(\mu_{optimizer};)$ $GB : \mathit{GBufferT}$ $G : \mathit{GeneratorT}(\mu_{generator};)$
archi_interactions	$L.\mathit{select\_program}$
archi_attachments	**from** $L.\mathit{send\_token\_seq}$ **to** $PB.\mathit{rec\_token\_seq}$ **from** $PB.\mathit{fw\_token\_seq}$ **to** $P.\mathit{get\_token\_seq}$ **from** $P.\mathit{send\_phrase\_seq}$ **to** $CB.\mathit{rec\_phrase\_seq}$ **from** $CB.\mathit{fw\_phrase\_seq}$ **to** $C.\mathit{get\_phrase\_seq}$ **from** $C.\mathit{send\_checked\_seq1}$ **to** $OB.\mathit{rec\_checked\_seq1}$ **from** $C.\mathit{send\_checked\_seq2}$ **to** $GB.\mathit{rec\_checked\_seq2}$ **from** $OB.\mathit{fw\_checked\_seq1}$ **to** $O.\mathit{get\_checked\_seq1}$ **from** $O.\mathit{send\_optimized\_seq}$ **to** $GB.\mathit{rec\_optimized\_seq}$ **from** $GB.\mathit{fw\_checked\_and\_optimized\_seq}$ **to** $G.\mathit{get\_checked\_and\_optimized\_seq}$
end	

Table 3: Textual description of *ConcurrentCompiler*

As far as expressiveness is concerned, 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. However, the imposed syntactical restrictions do not allow some queueing discipline, such as preemption, to be modeled.

Future work will be devoted to weaken the syntactical restrictions in such a way that the translation is still effective for a broader range of systems.

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 [16], pp. 1-11

[3] M. Bernardo, P. Ciancarini, L. Donatiello, *"On the Formalization of Architectural Types with Process Algebras"*, in Proc. of the *8th ACM Int. Symp. on the Foundations of Software Engineering (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 the *2nd Working IEEE/IFIP Conf. on Software Architecture (WICSA 2001)*, IEEE-CS Press, pp. 77-86, 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"*, Tech. Rep. UBLCS-2000-01, University of Bologna (Italy), 2000

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

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

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

[10] L. Kleinrock, *"Queueing Systems"*, Vol. 1, Theory, Wiley, New York, 1975

[11] L. L. Lavenberg, *"Computer Performance Modeling Handbook"*, Academic Press, New York, 1983

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

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

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

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

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