Performance Evaluation at the Software Architecture Level

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

[1] Università "Ca' Foscari" di Venezia
Dipartimento di Informatica
Via Torino 155, 30172 Mestre, Italy
{balsamo, simeoni}@dsi.unive.it
[2] Università di Urbino "Carlo Bo"
Istituto di Scienze e Tecnologie dell'Informazione
Piazza della Repubblica 13, 61029 Urbino, Italy
bernardo@sti.uniurb.it

Abstract. When tackling the construction of a software system, at the software architecture design level there are two main issues related to the system performance. First, the designer may need to choose among several alternative software architectures for the system, with the choice being driven especially by performance considerations. Second, for a specific software architecture of the system, the designer may want to understand whether its performance can be improved and, if so, it would be desirable for the designer to have some diagnostic information that guide the modification of the software architecture itself. In this paper we show how these two issues can be addressed in practice by employing a methodology relying on the combined use of Æmilia — an architectural description language based on stochastic process algebra — and queueing networks — structured performance models equipped with fast solution algorithms — which allows for a quick prediction, improvement, and comparison of the performance of different software architectures for a given system. The methodology is illustrated through a case study in which a sequential architecture, a pipeline architecture, and a concurrent architecture for a compiler system are compared on the basis of typical average performance indices.

1 Introduction

Software architecture (SA) is an emerging discipline within software engineering, aiming at describing the structure and the behavior of the software systems at a high level of abstraction [43, 46]. A SA represents the structure and the behavior of a software system in an early stage of the development cycle, the phase in which basic design choices of components and interactions among components are made and clearly influence the subsequent development and deployment phases. Appropriate languages and tools are required to give precise descriptions of SAs and to support the efficient analysis of their properties in a way that provides component-oriented diagnostic information in case of malfunctioning detection.

A crucial issue in the software development cycle is that of integrating the analysis of nonfunctional system properties since the early stages, where performance is one of the most influential factors that drive the design choices. To this purpose, in the formal method research field several description and analysis techniques have been proposed in the past twenty years, like stochastic Petri nets (SPN; see, e.g., [39, 1]) and stochastic process algebras (SPA; see, e.g., [29, 31, 30, 14]). On the side of the system performance evaluation research field, various models and methods have been proposed for the quantitative evaluation of hardware and sofware systems, which were traditionally based mostly on queueing networks (QN; see, e.g., [34, 35, 37, 33, 42, 50, 8]). However, only more recently some research has been truly focused on the integration of specific performance models in the software development cycle (see, e.g., [47, 52, 3]).

In this paper, which is a full and revised version of [7] that builds on material in [16, 3], we propose a methodology to evaluate the performance of SAs, which combines formalisms and techniques developed in the two different communities in a way that can be integrated in the software development cycle. More precisely, the methodology is based on both SPA modeling and QN analysis and is realized through a transformation of SPA specifications into QN models.

On the modeling side, we choose SPAs because they are compositional languages permitting the description of functional and performance aspects, which can be enhanced to act as fully fledged architectural description languages (ADL) that elucidate the architectural notions of component and interaction and support the detection of architectural mismatches arising when assembling several components together. The specific ADL that we consider is Æmilia [16], which is based on the stochastic process algebra $\mathrm{EMPA}_{\mathrm{gr}}$ [14]. Æmilia is illustrated in Sect. 2.

On the analysis side, we choose QNs for several reasons. First, QNs are structured performance models, therefore they should support the possibility of keeping track of the correspondence between their constituent service centers and the components of the architectural specifications. Second, typical performance measures can be computed both at the level of the overall QNs and at the level of their constituent service centers. Such global and local performance indicators can then be interpreted back at the level of the overall architectural specifications and at the level of their constituent components, respectively, so that useful diagnostic information can be obtained in the case of poor global performance. Third, QNs are equipped with efficient solution techniques that do not require the construction of the underlying state space, so that scalability with respect to the number of components in the architectural specifications should be achieved. Fourth, the solution of the QNs can be expressed symbolically in the case of simple open topologies, and can be approximated through an asymptotic bound analysis. This feature is useful in the early stages of the software development cycle, since the actual values of the system parameters, as well as its complete behavior, may be unknown. The basic concepts and results about QNs are recalled in Sect. 3.

The translation of Æmilia specifications into QN models is not straightforward, because the two formalisms are quite different from each other. On the one hand, Æmilia is a component-oriented language for handling both functional and performance characteristics, in which all the details must be expressed in an action-based way. On the other hand, QNs result in a queue-oriented graphical notation for performance modeling purposes only, in which some details — notably the queueing disciplines — are described in natural language. In addition to that, the components of the Æmilia specifications cannot be mapped to QN service centers, but on finer parts that we call QN basic elements. As a consequence, the translation can be applied only to a (reasonably wide) class of Æmilia specifications that satisfy certain syntax restrictions, which ensure that each component in such specifications is a QN basic element, i.e. an arrival process, a buffer, a fork process, a join process, or a service process. The translation, whose complexity is linear in the number of components declared in the Æmilia specifications, leads to the generation of open, closed or mixed QN models with phase-type interarrival and service time distributions, queueing disciplines with noninterruptable service, fork and join nodes for handling parallelism and synchronization, and arbitrary topologies. Depending on the type of QN model, various solution algorithms, either exact or approximate, can be applied to efficiently evaluate some average performance indices that are eventually interpreted back at the Æmilia specification level. The translation is defined in Sect. 4.

Based on the above translation of Æmilia specifications into QN models, we develop a practical multi-phase methodology to quickly predict, improve, and compare the performance of different architectures for the same software system. In the proposed methodology, the first step is to model with Æmilia all the architectural alternatives devised by a designer for the same system. Such Æmilia specifications may then need to be manipulated in such a way that they satisfy the syntax restrictions that make it possible to derive QN models out of them. Once the QN models for the architectural alternatives are obtained by applying the above translation, they are in turn manipulated so that some typical average performance measures can efficiently be computed in several scenarios of interest. The previous approximations, both at the Æmilia level and at the QN level, are justified at the architectural level of abstraction by the fact that we are more interested in rapidly getting an indication about the performance of the architectural alternatives, rather than in their precise evaluation. On the basis of the computed average performance measures, the designer gets a feedback that can be used to guide the modification of the Æmilia specifications of some architectural alternatives in order to ameliorate their performance. Once the predict-improve cycle is terminated, the architectural alternatives are compared on the basis of the values of the average performance measures obtained in the considered scenarios, in order to single out the best one. Because of the approximations that might have been performed in the previous phases, and the fact that the considered average performance measures are not necessarily related to the performance requirements of the system under study, the exact Æmilia specification of the selected architectural design is finally checked against the specific performance requirements. The methodology is presented in Sect. 5.

The application of the methodology and the translation of Æmilia specifications into QN models are clarified in Sect. 6 by means of a case study in which three different architectures — a sequential one, a pipeline one, and a concurrent one — for a compiler system are compared in different scenarios on the basis of average performance indices like the mean number of programs compiled per unit of time, the mean fraction of time during which the compiler is being used, the mean number of programs in the compiler system, and the mean compilation time.

Finally, in Sect. 7 we report some concluding remarks about future perspectives.

2 Æmilia: A SPA-Based ADL

In this section we present the main ingredients of Æmilia [16], a performance-oriented ADL. Æmilia is the result of the integration of two earlier formalisms: PADL [15, 17, 18, 2] and EMPA_{gr} [14]. The former is a process-algebra-based ADL, which is equipped with some architectural checks for the detection of deadlock-related architectural mismatches within families of software systems called architectural types. The latter is an expressive SPA, which allows for both the functional verification and the performance evaluation of concurrent and distributed systems. Below we recall through a running example how PADL and EMPA_{gr} have been combined together in order to give rise to the syntax, the semantics, and the analysis support for Æmilia.

2.1 Textual and Graphical Notation

A description in Æmilia represents an architectural type. An architectural type is an intermediate abstraction between a single SA and an architectural style [46], which results in a family of software systems sharing certain constraints on the component observable behavior as well as on the architectural topology [15, 17, 18].

As shown in Table 1, the description of an architectural type starts with the name of the architectural type and its formal parameters, which can represent variables as well as exponential rates, priorities, and weights for EMPA_{gr} actions. Each architectural type is defined as a function of its architectural element types (AETs) and its architectural topology. An AET, whose description starts with its name and its formal parameters, is defined as a function of its behavior, specified either as a list of sequential EMPA_{gr} defining equations or through an invocation of a previously defined architectural type, and its interactions, specified as a set of EMPA_{gr} action types occurring in the behavior that act as interfaces for the AET.

`ARCHI_TYPE`	⟨name and formal parameters⟩
`ARCHI_ELEM_TYPES`	⟨architectural element types: behaviors and interactions⟩
`ARCHI_TOPOLOGY`	
`ARCHI_ELEM_INSTANCES`	⟨architectural element instances⟩
`ARCHI_INTERACTIONS`	⟨architectural interactions⟩
`ARCHI_ATTACHMENTS`	⟨architectural attachments⟩
`END`	

Table 1. Structure of an Æmilia textual description

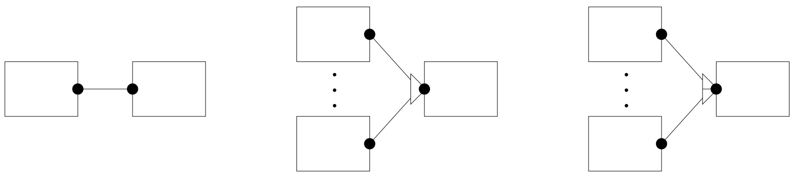
A sequential EMPA_{gr} defining equation specifies a possibly recursive behavior in the following way:

behavior_id(*formal_parameter_list*; *local_variable_list*) = *sequential_term*

where a sequential EMPA_{gr} term is written according to the following syntax:

sequential_term ::= `stop`
  | <*action_type*, *action_rate*> . *sequential_term_1*
  | `choice` { *sequential_term_2_list* }
sequential_term_1 ::= *sequential_term*
  | *behavior_id*(*actual_parameter_list*)
sequential_term_2 ::= *sequential_term*
  | `cond`(*boolean_guard*) -> *sequential_term*

Every behavior is given an identifier, a possibly empty list of comma-separated formal parameters, and a possibly empty list of comma-separated local variables. The admitted data types for parameters and variables are boolean, integer, bounded integer interval, real, list, array, and record. The sequential term `stop` cannot execute any action. The sequential term <*action_type*, *action_rate*> . *sequential_term_1* can execute an action having a certain type and a certain rate and then behaves as specified by *sequential_term_1*, which can be in turn a sequential term or a behavior invocation with a possibly empty list of comma-separated actual parameters. The action type can be a simple identifier (unstructured action), an identifier followed by the symbol "`?`" and a list of comma-separated variables enclosed in parentheses (input action), or an identifier followed by the symbol "`!`" and a list of comma-separated expressions enclosed in parentheses (output action). The action rate can be the identifier or the numeric value for the rate of an exponential distribution (exponentially timed action), the keyword `inf` followed by the identifiers or the numeric values of a priority level and a weight enclosed in parentheses (immediate action), or the symbol "`*`" followed by the identifiers or the numeric values of a priority level and a weight enclosed in parentheses (passive action). Finally, the sequential term `choice` { *sequential_term_2_list* } expresses a choice among at least two comma-separated alternatives, each of which may be subject to a boolean guard. If all the alternatives with a true guard start with an exponentially timed action, then the race policy applies: each involved action is selected with a probability proportional to its rate. If some of the alternatives with a true guard start with an immediate action, then such immediate actions take precedence over the 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 some of the alternatives with a true guard start with a passive action, then the reactive preselection policy applies to them: 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 passive actions of different types is nondeterministic).



(a) uni−uni *(b) uni−and* *(c) uni−or*

Fig. 1. Legal attachments

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. Alternatively, the architectural topology can be specified through the Æmilia graphical notation inspired by flow graphs [38], in which 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.

Every interaction is declared to be an input interaction or an output interaction and every attachment must go from an output interaction to an input interaction of two different AEIs. In addition, every interaction is declared to be a uni-interaction, an and-interaction, or an or-interaction. As shown in Fig. 1, the only legal attachments are those between two uni-interactions, an and-interaction and a uni-interaction, and an or-interaction and a uni-interaction. An and-interaction and an or-interaction can be attached to several uni-interactions. In the case of execution of an and-interaction, it synchronizes with all the uni-interactions attached to it. In the case of execution of an or-interaction, instead, it synchronizes with only one of the uni-interactions attached to it. An AEI can have different types of interactions (input/output, uni/and/or, local/architectural). Every local interaction must be involved in at least one attachment, while every architectural interaction must not be involved in any attachment. No isolated groups of AEIs are admitted in the architectural topology. 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 infinite with the same priority level, or passive with the same priority level). In order to comply with the generative-reactive synchronization discipline of $\mathrm{EMPA_{gr}}$, which establishes that two nonpassive actions cannot synchronize, every set of attached interactions must contain at most one interaction whose associated rate is exponential or infinite.

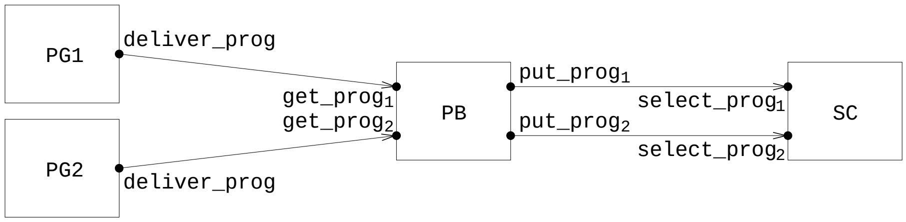


Fig. 2. Graphical description of `SeqCompSys`

As an example, we show in Table 2 an Æmilia textual description for an architectural type representing a compiler system. The compiler we consider is a sequential monolithic compiler that carries out all the phases (lexical analysis, parsing, type checking, code optimization, and code generation), with each phase introducing an exponentially distributed delay. For the sake of performance evaluation, the description of the compiler system comprises a generator of programs to be compiled, where the program interarrival times are assumed to follow an exponential distribution, as well as un unbounded buffer in which such programs wait before being compiled one at a time. We suppose that there are two different classes of programs: those whose code must be optimized and those whose code must not. As can be noted, the description of the architectural type `SeqCompSys` is parametrized with respect to the arrival rates of the two classes of programs (λ_1, λ_2) and the service rates of the five compilation phases $(\mu_{\mathtt{l}}, \mu_{\mathtt{p}}, \mu_{\mathtt{c}}, \mu_{\mathtt{o}}, \mu_{\mathtt{g}})$. The omitted values for the priority levels and the weights of the infinite and passive rates in the specification are taken to be 1. The same sequential compiler system is depicted in Fig. 2 through the Æmilia graphical notation.

2.2 Formal Semantics and Analysis Support

The semantics of an Æmilia specification is given by translation into $\mathrm{EMPA_{gr}}$. This translation is carried out in two steps. In the first step, the semantics of each AEI is defined to be the behavior of the corresponding AET — in which the formal rates, priority levels, and weights are replaced by the corresponding actual ones — projected onto its interactions. Such a projected behavior is obtained from the list of sequential $\mathrm{EMPA_{gr}}$ defining equations 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

```
ARCHI_TYPE                  SeqCompSys(rate λ1, λ2, μ1, μp, μc, μo, μg)

 ARCHI_ELEM_TYPES

  ELEM_TYPE                 ProgGenT(rate λ)
   BEHAVIOR                 ProgGen(void; void) =
                              <generate_prog, λ>.<deliver_prog, inf>.ProgGen()
   INPUT_INTERACTIONS
   OUTPUT_INTERACTIONS      UNI deliver_prog

  ELEM_TYPE                 ProgBufferT(integer h1, h2)
   BEHAVIOR                 ProgBuffer(integer h1, h2; void) =
                              choice
                              {
                                <get_prog1, *>.ProgBuffer(h1 + 1, h2),
                                <get_prog2, *>.ProgBuffer(h1, h2 + 1),
                                cond(h1 > 0) -> <put_prog1, *>.ProgBuffer(h1 - 1, h2),
                                cond(h2 > 0) -> <put_prog2, *>.ProgBuffer(h1, h2 - 1)
                              }
   INPUT_INTERACTIONS       UNI get_prog1; get_prog2
   OUTPUT_INTERACTIONS      UNI put_prog1; put_prog2

  ELEM_TYPE                 SeqCompT(rate μ1, μp, μc, μo, μg)
   BEHAVIOR                 SeqComp(void; void) =
                              choice
                              {
                                <select_prog1, inf>.<recognize_tokens, μ1>.
                                  <parse_phrases, μp>.<check_phrases, μc>.
                                  <optimize_code, μo>.<generate_code, μg>.SeqComp(),
                                <select_prog2, inf>.<recognize_tokens, μ1>.
                                  <parse_phrases, μp>.<check_phrases, μc>.
                                  <generate_code, μg>.SeqComp()
                              }
   INPUT_INTERACTIONS       UNI select_prog1; select_prog2
   OUTPUT_INTERACTIONS

 ARCHI_TOPOLOGY

  ARCHI_ELEM_INSTANCES      PG1 : ProgGenT(λ1);
                            PG2 : ProgGenT(λ2);
                            PB : ProgBufferT(0, 0);
                            SC : SeqCompT(μ1, μp, μc, μo, μg)

  ARCHI_INTERACTIONS

  ARCHI_ATTACHMENTS         FROM PG1.deliver_prog TO PB.get_prog1;
                            FROM PG2.deliver_prog TO PB.get_prog2;
                            FROM PB.put_prog1 TO SC.select_prog1;
                            FROM PB.put_prog2 TO SC.select_prog2

END
```

Table 2. Textual description of SeqCompSys

behavior of the AEI. In addition, the projected behavior must reflect the fact that an or-interaction can result in several distinct synchronizations. Therefore, every or-interaction is rewritten as a choice among as many indexed instances of uni-interactions as there are attachments involving the or-interaction. Recalled that in EMPA_{gr} the hiding operator is denoted by the symbol "/", for our compiler system example we have:

$$
\begin{aligned}
[\![\mathtt{PG}_1]\!] &= \mathtt{ProgGen}_1 \,/\, \{\mathtt{generate\_prog}\}\\
[\![\mathtt{PG}_2]\!] &= \mathtt{ProgGen}_2 \,/\, \{\mathtt{generate\_prog}\}\\
[\![\mathtt{PB}]\!] &= \mathtt{ProgBuffer}(0,0)\\
[\![\mathtt{SC}]\!] &= \mathtt{SeqComp} \,/\, \{\mathtt{recognize\_tokens}, \mathtt{parse\_phrases}, \mathtt{check\_phrases},\\
&\qquad\qquad\qquad \mathtt{optimize\_code}, \mathtt{generate\_code}\}
\end{aligned}
$$

where $\mathtt{ProgGen}_1$ (resp. $\mathtt{ProgGen}_2$) is obtained from $\mathtt{ProgGen}$ by replacing each occurrence of λ with λ_1 (resp. λ_2).

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, after relabeling to the same fresh action type all the interactions attached to each other. This relabeling is required by the synchronization mechanism of EMPA_{gr}, which establishes that only actions with the same type can synchronize. Recalled that in EMPA_{gr} the relabeling operator is denoted by the symbols "[" and "]" and that the left-associative parallel composition operator is denoted by the symbol "$\|_{\mathtt{S}}$" where $\mathtt{S}$ is the set of action types on which the synchronization is enforced, for our compiler system example we have:

$$
\begin{aligned}
[\![\mathtt{SeqCompSys}]\!] = {} & [\![\mathtt{PG}_1]\!][\mathtt{deliver\_prog} \mapsto \mathtt{a}_1] \;\|_{\emptyset}\\
& [\![\mathtt{PG}_2]\!][\mathtt{deliver\_prog} \mapsto \mathtt{a}_2] \;\|_{\{\mathtt{a}_1,\mathtt{a}_2\}}\\
& [\![\mathtt{PB}]\!][\mathtt{get\_program}_1 \mapsto \mathtt{a}_1,\, \mathtt{get\_program}_2 \mapsto \mathtt{a}_2,\\
& \quad \mathtt{put\_program}_1 \mapsto \mathtt{b}_1,\, \mathtt{put\_program}_2 \mapsto \mathtt{b}_2] \;\|_{\{\mathtt{b}_1,\mathtt{b}_2\}}\\
& [\![\mathtt{SC}]\!][\mathtt{select\_prog}_1 \mapsto \mathtt{b}_1,\, \mathtt{select\_prog}_2 \mapsto \mathtt{b}_2]
\end{aligned}
$$

Given the translation above, Æmilia inherits the semantic models of EMPA_{gr}. More precisely, the semantics of an Æmilia specification is a state-transition graph called the integrated semantic model, whose states are represented by EMPA_{gr} terms and whose transitions are labeled with EMPA_{gr} actions together with the related guards arising from the use of the `choice` operator. This graph is finite state and finitely branching unless variables taking values from infinite domains are used (like in the buffer of the compiler system example), in which case a symbolic representation of the state space is employed in accordance with [13]. After pruning the lower priority transitions from the integrated semantic model, it is possible to derive a functional semantic model, by removing the action rates from the transitions, and a performance semantic model, by removing the action types from the transitions. The performance semantic model, which is defined only if the integrated semantic model has neither passive transitions nor transitions with a guard different from `true`, is a continuous-time or a discrete-time Markov chain [48] depending on whether the integrated semantic model has exponentially timed transitions or not.

On the analysis side, Æmilia inherits from EMPA_{gr} standard techniques to assess functional properties as well as performance measures. Among such techniques we mention model checking [23], equivalence checking [24], Markovian analysis [48] based on rewards [32] as described in [14], and discrete event simulation [51], all of which are available in the Æmilia-based software tool TwoTowers 3.0 [12]. In addition to these capabilities, Æmilia comes equipped with some specific checks for the detection of architectural mismatches — and the provision of related diagnostic information — that may arise when assembling the components together. The first group of checks ensures that deadlock freedom is preserved when building a system from deadlock-free components [15, 2]. The second group of checks makes sure that assembling components with partially specified performance details (i.e., with passive actions occurring in their behaviors) results in a system with fully specified performance aspects [16]. Finally, the third group of checks comes into play in case of hierarchical modeling, i.e. whenever the description of a component behavior contains an architectural type invocation. Such checks guarantee that the actual parameters of an invocation of an architectural type conform to the formal parameters of the definition of the architectural type, in the sense that the actual components have the same observable behavior as the corresponding formal components [15] and that the actual topology is a legal extension of the formal topology [17, 18].

3 Queueing Networks

QN models have been widely applied as system performance models to represent and analyze resource sharing systems [34, 35, 37, 33, 42, 50, 8]. In essence, a QN model is a collection of interacting service centers, representing system resources, and a set of customers, representing the users sharing the resources. The customers' competition for the resource service corresponds to queueing into the service centers. The popularity of QN models for system performance evaluation is due to their relatively high accuracy in performance results and their efficiency in model analysis and evaluation. In this section we briefly recall the basic notions and properties of QN models. In particular, we shall focus on the class of product form QNs, which admit fast solution techniques.

3.1 Queueing Systems

As depicted in Fig. 3(a), the simplest case of QN is the one in which there is a single service center together with a source of arrivals, which is referred to as a queueing system (QS). Every QS is completely described by the customer interarrival time distribution, the customer service time distribution, the number of independent servers, the queue capacity, the customer population size, and the queueing discipline. The first five parameters are summarized by using the Kendall's notation A/B/m/c/p, with A and B ranging over a set of probability distributions — 'M' for memoryless distributions, 'D' for deterministic values, 'PH' for phase-type distributions, and 'G' for general distributions — and m, c, and p being natural numbers. If c and p are unspecified, they are assumed to be ∞, i.e. to describe an unlimited queue capacity and an unlimited population.

Every customer needing a certain service arrives at the QS, waits in the queue for a while, is served by one of the servers, and finally leaves the QS.

The queueing discipline is an algorithm that determines the order in which the customers in the queue are served. Such a scheduling algorithm may depend on the order in which the customers arrive at the QS, the priorities assigned to the customers, or the amounts of service already provided to the customers. Here are some commonly adopted queueing disciplines:

– First come first served (FCFS): the customers are served in the order of their arrival.
– Last come first served (LCFS): the customers are served in the reverse order of their arrival.
– Service in random order (SIRO): the next customer to be served is chosen probabilistically, with equal probabilities assigned to all the waiting customers.
– Nonpreemptive priority (NP): the customers are assigned fixed priorities; the waiting customer with the highest priority is served first; if several waiting customers have the same highest priority, they are served in the order of their arrival; once begun, a service cannot be interrupted by the arrival of a higher priority customer.
– Preemptive priority (PP): same as NP, but each arriving higher priority customer interrupts the current service, if any, and begins to be served; a customer whose service was interrupted resumes service at the point of interruption when there are no higher priority customers to be served.
– Last come first served preemptive resume (LCFS-PR): same as LCFS, but each arriving customer interrupts the current service, if any, and begins to be served; the interrupted service of a customer is resumed when all the customers that arrived later than that customer have departed.
– Round robin (RR): each customer is given continuous service for a maximum interval of time called a quantum; if the customer's service demand is not satisfied during the quantum, the customer reenters the queue and waits to receive an additional quantum, repeating this process until its service demand is satisfied; the waiting customers are served in the order in which they last entered the queue.
– Processor sharing (PS): all the waiting customers receive service simultaneously with equal shares of the service rate.
– Infinite server (IS): no queueing takes place as each arriving customer always find an available server.

If the queueing discipline is omitted in the QS notation, it is assumed to be FCFS.

The QS behavior can be analyzed either during a given time interval (transient analysis) or by assuming that it reaches a stationary condition (steady-state analysis). The analysis of the QS is based on the definition of an underlying continuous-time Markov chain. The QS steady-state analysis usually evaluates a set of four average performance indices after computing the queue length distribution, i.e. the distribution of the number of customers in the QS. The four

(a) A QS *(b) A simple closed QN*

Fig. 3. QN graphical representation

average performance indices are the throughput (mean number of customers leaving the system per unit of time), the utilization (average fraction of time during which the servers are used), the mean number of customers in the QS, and the mean response time experienced by the customers visiting the QS.

For instance, let us consider the simplest case of QS M/M/1 with arrival rate λ and service rate μ [34]. Although the stochastic process underlying the QS M/M/1 is an infinite-state continuous-time Markov chain, where each state represents the number of customers in the system, the particular structure of this Markov chain allows us to easily derive that the distribution of the number N_1 of customers in the system — on the basis of which the four average measures above are defined — is geometrical with parameter given by the traffic intensity $\rho_1 = \lambda/\mu$. The steady-state analysis of this QS requires that the stability condition $\rho_1 < 1$ holds, i.e., that the customer arrival rate is less than the service rate. In this case we can easily derive the four average perfomance indices as follows:

- The throughput is given by the probability that there is at least one customer in the system multiplied by the service rate, i.e. $\overline{X}_1 = \Pr\{N_1 > 0\} \cdot \mu = \rho_1 \cdot \mu = \lambda$.
- The utilization is given by the probability that there is at least one customer in the system, i.e. $\overline{U}_1 = \Pr\{N_1 > 0\} = \rho_1$.
- The mean number of customers in the system is the expected value of the geometrical distribution describing the number of customers in the system, i.e. $\overline{N}_1 = \rho_1/(1 - \rho_1)$.
- The mean response time is obtained from Little's law as the ratio of the mean number of customers in the system to the arrival rate, i.e. $\overline{R}_1 = \overline{N}_1/\lambda = 1/[\mu \cdot (1 - \rho_1)]$.

It can be shown that all the queueing disciplines with noninterruptable, nonprioritized service — like FCFS, LCFS, and SIRO — together with PS — which is a good approximation of RR — and LCFS-PR are equivalent with respect to the four average performance measures above for a QS M/M/1.

In the more general case of a QS M/M/m with arrival rate λ, which has m identical servers that operate independently and in parallel each with service rate μ, the traffic intensity is defined by $\rho_m = \lambda/(m \cdot \mu)$ and, under the stability condition $\rho_m < 1$, the four average performance indices are given by:

$$
\begin{aligned}
\overline{X}_m &= \sum_{i=1}^{m-1} i \cdot \mu \cdot \Pr\{N_m = i\} + \sum_{i=m}^{\infty} m \cdot \mu \cdot \Pr\{N_m = i\} = \lambda \\
\overline{U}_m &= 1 - \Pr\{N_m = 0\} \\
\overline{N}_m &= m \cdot \rho_m + \frac{\Pr\{N_m=0\} \cdot \rho_m \cdot (m \cdot \rho_m)^m}{m! \cdot (1-\rho_m)^2} \\
\overline{R}_m &= \frac{1}{\mu} \cdot \left(1 + \frac{\Pr\{N_m=0\} \cdot \rho_m \cdot (m \cdot \rho_m)^{m-1}}{m! \cdot (1-\rho_m)^2}\right)
\end{aligned}
$$

where:

$$
\Pr\{N_m = 0\} = \left(\sum_{i=0}^{m-1} \frac{(m \cdot \rho_m)^i}{i!} + \frac{(m \cdot \rho_m)^m}{m! \cdot (1 - \rho_m)} \right)^{-1}
$$

3.2 Networks of Queueing Systems

A QN is composed of a set of interconnected service centers. When describing a QN, which can be represented — as shown in Fig. 3(b) — through a directed graph whose nodes are the service centers and whose edges represent the behavior of the customers' service requests, it is necessary to specify for each service center the service time distribution, the number of servers, the queue capacity, the queueing discipline, and the routing probabilities for the customers leaving the service center. A QN can be open or closed, depending on whether external arrivals and departures are allowed or not, or mixed. In an open QN, a customer that completes service at a service center immediately enters another service center, reenters the same service center, or departs from the network. In a closed QN, instead, a fixed number of customers circulate indefinitely among the service centers.

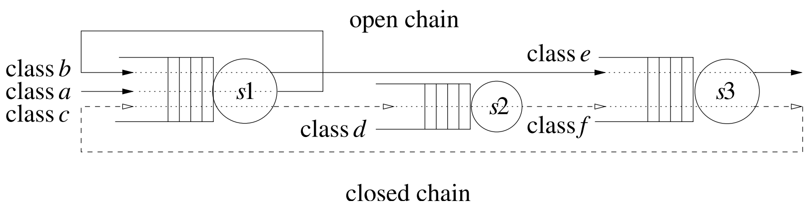


Fig. 4. A mixed network with three service centers, an open chain with classes a, b, e, and a closed chain with classes c, d, f.

Different types of customers in the QN model can be used to represent different behaviors. This in fact allows various types of external arrival process, different service demands, and different types of network routing to be modeled. A chain gathers the customers of the same type. A chain consists then of a set of classes that represent different phases of processing in the system for a given type of customers. Classes are partitioned within the service centers and each customer in a chain moves between the classes. A chain can be used to represent a customer routing behavior dependent on the past history. For example, two classes of the same chain in a service center can represent the customer requirement of two successive services. Each chain can be open or closed depending on whether external arrivals and departures are allowed or not. Multiclass or multichain networks can be open or closed if all the chains are open or closed, respectively. A mixed network has both open and closed chains. Fig. 4 shows an example of a multiclass network with two chains and six classes. The open chain describes the routing behavior of the type 1 customers: two successive visits to the service center $s1$ followed by a visit to service center $s3$. Chain 2 is closed and there is a constant number of type 2 customers circulating between service centers $s1$, $s2$, and $s3$.

Evaluating a QN model means obtaining a quantitative description of its behavior by computing a set of figures of merit, such as the four average performance indices considered for a single QS. The analysis of a QN model provides information both on the local and on the global performance, i.e. the performance of each service center and the overall system performance. A QN can be analyzed by defining and solving the underlying stochastic process, which under general assumptions is a countinous-time Markov chain. Unfortunately, its solution can often become unfeasible since its state space size grows exponentially with the number of components of the QN model. However, some efficient solution algorithms can be defined for the special subclass of product form QNs, which we briefly introduce in the next section. Such algorithms provide a powerful tool for performance analysis based on QNs.

3.3 Product Form QNs

Product form QNs (see [6] for a complete survey) avoids the state space explosion problem because they can be solved compositionally. Given that the state of a QN is a tuple consisting of the number of customers in the various service centers, the probability of a product form QN state is simply obtained as the product of the probabilities of its constituent service center states, up to a normalizing constant in the case of closed QNs. An important characterization of product form QNs is given by the BCMP theorem [10], which defines the BCMP class of product form open, closed and mixed QNs with multiple classes of customers, Poisson arrivals (i.e. exponentially distributed interarrival times) with rates possibly depending on the total population of the QN or on the population of a chain, and arbitrary Markovian routing. According to the BCMP theorem, each multiclass service center can have one combination of the following queueing disciplines and service time distributions:

- FCFS with exponentially distributed service times, with the same rate for all the classes of customers;
- PS, LCFS-PR, or IS with phase-type distributed service times, possibly different for the various classes of customers.

In the second case, only the expected values of the phase-type service time distributions affect the QN solution in terms of the four average performance indices, so when computing such indices the phase-type distributions can be replaced with exponential distributions having the same expected values.

In the case of an open product form QN, the four average performance measures can easily be obtained at the global level and at the local level from the analysis of the constituent service centers, when considered as isolated QSs with Poisson arrivals, by exploiting the two groups of formulas at the end of Sect. 3.1. The arrival rates are derived by solving the linear system of the traffic equations defined by the routing probabilities among the service centers. The same average indices can be obtained at the global level and at the local level for a closed or mixed product form QN by applying one of the following algorithms: the convolution algorithm [19], the mean-value analysis algorithm (MVA) [44], the local balance algorithm for normalizing constants (LBANC) [20], and the recursion-by-chain algorithm (RECAL) [25]. These algorithms also provide the basis for most approximate analytical methods that need to be applied whenever the QN model under consideration does not belong to the class of product form QNs (see, e.g., [36]).

An important property of product form QNs is exact aggregation, which allows replacing a subnetwork with a single service center, in such a way that the new aggregated QN has the same behavior in terms of the four average performance indices. Thus, exact aggregation can be used to represent and evaluate a system at different levels of abstraction. Moreover, exact aggregation for product form QNs provides a basis for approximate solution methods of more general QNs that are not product form (see, e.g., [36]).

Various extensions of the class of BCMP product form QNs have been derived. They include QNs with other queueing disciplines, QNs with state dependent routing, some special cases of QNs with finite capacity queues, subnetwork popolation constraints, and blocking, and QNs with batch arrivals and batch services. Another extension of QNs networks with product form is the class of G-networks [28], which comprise both positive and negative customers.

3.4 QN Extensions

Extensions of classical QN models, named extended QN (EQN) models, have been introduced in order to represent several interesting features of real systems, such as synchronization and concurrency constraints, finite capacity queues, memory constraints, and simultaneous resource possession.

In particular, concurrency and synchronization among tasks are represented in an EQN model by fork and join nodes. A fork node starts the parallel execution on distinct service centers of the different tasks in which a customer's request can be split, while a join node represents a synchronization point for the termination of all such tasks. A few cases of QNs with forks and joins have been solved with exact and approximate analytical techniques (see, e.g., [40, 4, 9]).

QNs with finite capacity queues and blocking have been introduced as more realistic models of systems with finite resources and population constraints. When a customer arrives at a finite capacity queue that is full, the customer cannot enter the queue and it is blocked. Various blocking mechanisms can be defined — like blocking after or before service and repetitive service — that specify the behavior of the customers blocked in the network. Except for a few cases that admit product form solutions, QNs with blocking are solved through approximate techniques (see, e.g., [42, 8]).

Another extension of the QN model is given by the layered queueing network (LQN) model, which allows client-server communication patterns to be modeled in concurrent and/or distributed software systems. LQN models can be solved by analytic approximation methods based on standard methods for EQNs with simultaneous resource possession and MVA (see, e.g., [45, 52, 27]).

The exact and approximate analytical methods for solving EQNs require that a set of assumptions and constraints are satisfied. Should this not be the case, EQN models can be analyzed via simulation, at the cost of higher development and computational times to obtain accurate results.

Examples of performance evaluation tools based on QNs and their extensions are RESQ [21], QNAP2 [49], HIT [11], and LQNS [26].

4 Translating Æmilia Specifications into QN Models

In this section we provide a translation that maps an Æmilia specification into a QN model to be used to predict and improve the performance of the described SA. As mentioned in Sect. 1, there are several good reasons for resorting to QN models at the SA level of design, instead of the flat state-transition graphs used as semantic models for Æmilia. First, QNs are structured performance models whose constituent service centers can be put in correspondence with groups of AEIs of the Æmilia specifications. Second, typical average performance measures can be computed at the level of the overall QNs and interpreted at the level of the groups of AEIs of the Æmilia specifications corresponding to their constituent service centers, thus providing a useful feedback. Third, QNs do not suffer from the state space explosion problem, as they are equipped with efficient solution techniques that avoid the construction of the state space. Finally, QNs can sometimes be solved symbolically, without having to instantiate the values of the corresponding parameters in the Æmilia specifications.

To carry out the translation, first of all we observe that the two formalisms that we are considering are quite different from each other. On the one hand, Æmilia is a component-oriented language for handling both functional and performance characteristics, in which all the details must be expressed in an action-based way. On the other hand, QNs result in a queue-oriented graphical notation for performance modeling purposes only, in which some details — notably the queueing disciplines — are described in natural language. As a consequence, there will be Æmilia specifications that cannot be converted into QN models, either because they do not follow a queue-oriented pattern, or because it is hard to understand — by looking at their process algebraic defining equations — the queueing disciplines that they encode. Therefore, we shall impose some general syntax restrictions that single out a reasonably wide class of Æmilia specifications for which a QN model may be derived.

Within the class of Æmilia specifications that obey the general syntax restrictions, given a specification we try to map each of its constituent AEIs into a part of a QN model. In principle, it would seem to be natural to map each AEI into a QS PH/PH/$m/c/p$. However, this is not always possible because the AEIs are usually finer than the QSs. As a consequence, we identify five classes of QN basic elements — which we call arrival processes, buffers, fork processes, join processes, and service processes, respectively, and graphically represent through an extension of the traditional notation used for QNs — and we impose some further specific syntax restrictions to single out those AEIs that fall into one of the five classes. For each Æmilia specification obeying both the general and the specific syntax restrictions, the translation is accomplished by first mapping each of its constituent AEIs into the corresponding QN basic element and then composing the previously obtained QN basic elements according to the attachments declared in the Æmilia specification. The translation will be illustrated by means of the sequential compiler system example introduced in Sect. 2.

4.1 General Syntax Restrictions: Benefits and Limitations

The general syntax restrictions helps identifying the Æmilia specifications for which it is possible to derive an open, closed or mixed QN model comprising arrival processes, buffers, fork processes, join processes, and service processes. The general restrictions are mainly based on the observation that an AEI describes a sequential software component, which thus runs on a single computational resource.

The first general restriction is that every AEI of an Æmilia specification must be an arrival process, a buffer, a fork process, a join process, or a service process, and must be properly connected to the other AEIs in order to obtain a well-formed QN. This is achieved through specific syntax restrictions depending on the particular QN basic element, which will be introduced in the next sections.

The second general restriction aims at easing the identification of those AETs that represent arrival or service processes, which are built around exponentially timed actions describing the relevant delays. The second general restriction establishes that the interactions of an Æmilia specification cannot be exponentially timed, i.e. they must be immediate or passive.

The third general restriction aims at avoiding the unnatural application of the race policy to several distinct activities within the same (sequential) AEI, thus causing the various arrival and service processes to be modeled separately with different AEIs. The third general restriction establishes that, within the behavior of the AETs of an Æmilia specification, no exponentially timed action can be alternative to another exponentially timed action.

The fourth general restriction aims at allowing interarrival and service times to be characterized through precisely defined phase-type distributions. The fourth general restriction establishes that, within the behavior of the AETs of an Æmilia specification, no exponentially timed action can be alternative to an immediate or passive action, no immediate action can be alternative to a passive action, and no interaction can be alternative to a local action.

The last three general restrictions, as well as the specific restrictions illustrated in the next sections that implement the first general restriction, can automatically be checked at the syntax level, without constructing the underlying state space of the entire Æmilia specification. They preserve much of the modeling power that Æmilia inherits from $\mathrm{EMPA_{gr}}$, without hampering the description of typical situations like parallel executions, synchronization constraints, probabilistic/prioritized choices, and activities whose duration is or can be approximated with a phase-type distribution. It is straightforward to verify that `SeqCompSys` defined in Table 2 satisfies the last three general restrictions.

The four general restrictions, together with the specific syntax restrictions accompanying the first general one, introduce two main limitations. First, due to the fourth general restriction, the Æmilia specifications modeling preemption cannot be dealt with, as it is not possible to express the fact that the service of a customer of a certain class is interrupted by the arrival of a customer of another class having higher service priority. Second, as we shall see when presenting the specific syntax restrictions for the buffers, we only address queueing disciplines with noninterruptable service for a fixed number of servers, like FCFS, LCFS, SIRO, and NP, thus excluding those policies in which the service of a customer can be interrupted (PP, LCFS-PR) or divided into several rounds (RR, PS), as well as those policies in which no queueing takes place as every incoming customer always finds an available server (IS).

4.2 Modeling Phase-Type Distributions

Since the interarrival times and the service times are allowed to follow phase-type distributions, before proceeding with the translation it is worth recalling how the phase-type distributions can be modeled in a language like Æmilia, where only exponentially distributed delays can directly be specified. A continuous phase-type distribution [41] describes the time to absorption in a finite-state continuous-time Markov chain having exactly one absorbing state. Well known examples of phase-type distributions are the exponential distribution, the hypoexponential distribution, the hyperexponential distribution, and combinations thereof, which are characterized in terms of time to absorption in a finite-state continuous-time Markov chain with one absorbing state as depicted in Fig. 5, where the numbers labeling the states describe the initial state probability functions.

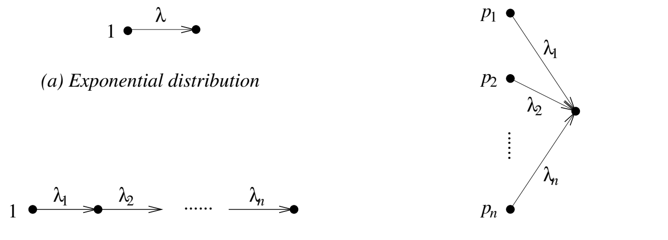
Observed that an absorbing state can be modeled by term `stop`, the three phase-type distributions above can easily be modeled through a suitable interplay of exponentially timed actions and immediate actions as follows. An exponential distribution with rate λ can be modeled through the following equation:

$$\mathtt{Exp}_{\lambda}(\mathtt{void};\mathtt{void}) = <\mathtt{phase},\lambda>.\mathtt{stop}$$

An n-stage hypoexponential distribution with rates $\lambda_1,\dots,\lambda_n$ can be modeled through the following equation:

$$\mathtt{Hypoexp}_{\lambda_1,\dots,\lambda_m}(\mathtt{void};\mathtt{void}) = <\mathtt{phase},\lambda_1>.\dots.<\mathtt{phase},\lambda_n>.\mathtt{stop}$$

An n-stage hyperexponential distribution with rates $\lambda_1,\dots,\lambda_n$ and branching



(a) Exponential distribution

(b) Hypoexponential distribution

(c) Hyperexponential distribution

Fig. 5. Typical phase-type distributions

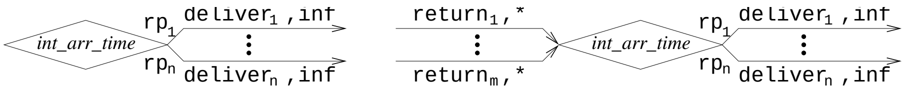
probabilities $\mathtt{p}_1, \ldots, \mathtt{p}_\mathtt{n}$ can be modeled through the following equation:

$$
\begin{array}{l}
\mathtt{Hyperexp}_{\lambda_1,\ldots,\lambda_\mathtt{n},\mathtt{p}_1,\ldots,\mathtt{p}_\mathtt{n}}(\mathtt{void};\mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\quad\quad <\mathtt{branch}, \mathtt{inf}(1, \mathtt{p}_1)>.<\mathtt{phase}, \lambda_1>.\mathtt{stop}, \\
\quad\quad \vdots \\
\quad\quad <\mathtt{branch}, \mathtt{inf}(1, \mathtt{p}_\mathtt{n})>.<\mathtt{phase}, \lambda_\mathtt{n}>.\mathtt{stop} \\
\quad \}
\end{array}
$$

In the arrival processes and in the service processes with phase-type distributed delays, the occurrences of `stop` will be replaced by suitable invocations of the behaviors that must take place after the delays have elapsed.

4.3 Arrival Processes

An arrival process is a generator of arrivals of customers of a certain class, whose interarrival times follow a phase-type distribution. As depicted in Fig. 6, we distinguish between two different kinds of arrival processes depending on whether the related customer population is unbounded or finite.



(a) Arrival process for unbounded population

(b) Arrival process for single customer of finite population

Fig. 6. Graphical representation of the arrival processes

In the case of an unbounded customer population, the customer interarrival time distribution refers to the whole population, so there is no need to explicitly model the return of a customer after its service termination. As an example, the behavior of an AEI, which acts as an arrival process for an unbounded population of customers whose interarrival time is exponentially distributed with rate λ, where each customer has a set of $\mathtt{n}$ different forks or service centers as destinations chosen according to the intraclass routing probabilities $\mathtt{rp_1}, \ldots, \mathtt{rp_n}$, must be equivalent [3] to the following one:

```
UnboundedPopArrProc(void; void) =
   <generate, λ>.UnboundedPopArrProc'()
UnboundedPopArrProc'(void; void) =
   choice
   {
      <choose_1, inf(1, rp_1)>.<deliver_1, inf>.UnboundedPopArrProc(),
      ⋮
      <choose_n, inf(1, rp_n)>.<deliver_n, inf>.UnboundedPopArrProc()
   }
```

with $\mathtt{deliver_1}, \ldots, \mathtt{deliver_n}$ being output interactions attached to input interactions of buffers (not related to join processes), fork processes with no buffer, or service processes with no buffer. The specific syntax restriction requires that, in order for an AEI to be classified as an arrival process for an unbounded population of customers, its behavior and interactions must be equivalent to the previous ones, with: the exponentially timed action possibly replaced by a term describing a more general phase-type distribution where `UnboundedPopArrProc'()` is substituted for each occurrence of `stop`; the destination choice actions omitted if there is only one possible destination; the delivery actions possibly having specific priority levels and specific weights if the related destinations are service processes with no buffer.

If the customer population is finite, instead, then the customer interarrival time distribution for the whole population varies proportionally to the number of customers that are not requesting any service, hence the return of a customer after its service termination must explicitly be modeled. In this case, the customers are represented separately through independent instances of the same AET with the same individual interarrival time distribution, in order to easily achieve the global interarrival time distribution scaling. For instance, the behavior of an AEI, which acts as an arrival process for a single customer belonging to a finite population of customers whose individual interarrival time is exponentially distributed with rate λ, where the customer has a set of $\mathtt{n}$ different forks or service centers as destinations chosen according to the intraclass routing probabilities $\mathtt{rp_1}, \ldots, \mathtt{rp_n}$ and can return from $\mathtt{m}$ distinct joins or service processes, must be equivalent to the following one:

[3] In our framework, equivalence can formally be checked on the basis of the notion of strong extended Markovian bisimulation [14].

```
SingleCustArrProc(void; void) =
   <generate, λ>.SingleCustArrProc'()
SingleCustArrProc'(void; void) =
   choice
   {
      <choose_1, inf(1, rp_1)>.<deliver_1, inf>.SingleCustArrProc''(),
      ⋮
      <choose_n, inf(1, rp_n)>.<deliver_n, inf>.SingleCustArrProc''()
   }
SingleCustArrProc''(void; void) =
   choice
   {
      <return_1, *>.SingleCustArrProc(),
      ⋮
      <return_m, *>.SingleCustArrProc()
   }
```

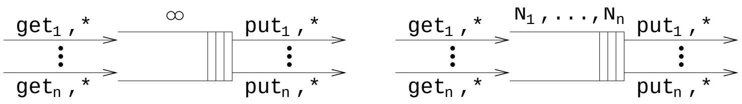
with: $\texttt{deliver}_1, \dots, \texttt{deliver}_\texttt{n}$ being output interactions attached to input or-interactions of buffers (not related to join processes), fork processes with no buffer, or service processes with no buffer; $\texttt{return}_1, \dots, \texttt{return}_\texttt{m}$ being input interactions attached to output or-interactions of join processes or service processes. The specific syntax restriction requires that, in order for an AEI to be classified as an arrival process for a single customer belonging to a finite population of customers, its behavior and interactions must be equivalent to the previous ones, with the remaining constraints similar to those for the arrival processes for unbounded populations. In addition, all the AEIs modeling the customers of the same finite population must be instances of the same AET characterized by the same individual interarrival time distribution and must be attached to the same input or-interactions of buffers (not related to join processes), fork processes with no buffer, or service processes with no buffer as well as to the same output or-interactions of join processes or service processes.

To conclude, for the sequential compiler system of Sect. 2 we observe that $\texttt{PG}_1$ and $\texttt{PG}_2$ are arrival processes for unbounded populations of customers of two different classes, each having a single destination.

4.4 Buffers

A buffer is a repository of customers of different classes that are waiting to be served. As depicted in Fig. 7, we distinguish between two different kinds of buffers depending on their capacity.

In the case of an unbounded buffer, the incoming customers can always be accommodated within the buffer. The specific syntax restriction requires that, in order for an AEI to be classified as an unbounded buffer for n classes of customers, it must have a behavior equivalent to the following one:



(a) Unbounded buffer *(b) Finite capacity buffer*

Fig. 7. Graphical representation of the buffers

$$
\begin{array}{l}
\mathtt{UnboundedBuffer}(\mathtt{integer}\, \mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n}; \mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\qquad <\mathtt{get}_1, *>.\mathtt{UnboundedBuffer}(\mathtt{h}_1 + 1, \dots, \mathtt{h}_\mathtt{n}), \\
\qquad \vdots \\
\qquad <\mathtt{get}_\mathtt{n}, *>.\mathtt{UnboundedBuffer}(\mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n} + 1), \\
\qquad \mathtt{cond}(\mathtt{h}_1 > 0) \text{ -> } <\mathtt{put}_1, *>.\mathtt{UnboundedBuffer}(\mathtt{h}_1 - 1, \dots, \mathtt{h}_\mathtt{n}), \\
\qquad \vdots \\
\qquad \mathtt{cond}(\mathtt{h}_\mathtt{n} > 0) \text{ -> } <\mathtt{put}_\mathtt{n}, *>.\mathtt{UnboundedBuffer}(\mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n} - 1) \\
\quad \}
\end{array}
$$

with: $\mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n}$ initially set to nonnegative integers; $\mathtt{get}_1, \dots, \mathtt{get}_\mathtt{n}$ being input interactions attached to output interactions of arrival processes, fork processes, join processes, or service processes; $\mathtt{put}_1, \dots, \mathtt{put}_\mathtt{n}$ being output interactions attached to input interactions of fork processes with buffer, join processes with buffers, or service processes with buffer; $\mathtt{get}_\mathtt{i}$ being an input or-interaction if the customers of class $\mathtt{i}$ belong to a finite population and come directly from their arrival processes.

If the buffer capacity is finite, instead, then the incoming customers can be accommodated only if the buffer capacity is not exceeded. The specific syntax restriction requires that, in order for an AEI to be classified as a finite capacity buffer for $\mathtt{n}$ classes of customers, where the customers of class $\mathtt{i}$ can occupy up to $\mathtt{N}_\mathtt{i}$ positions in the buffer, it must have a behavior equivalent to the following one:

$$
\begin{array}{l}
\mathtt{FiniteCapBuffer}(\mathtt{integer}(0..\mathtt{N}_1)\, \mathtt{h}_1, \dots, \mathtt{integer}(0..\mathtt{N}_\mathtt{n})\, \mathtt{h}_\mathtt{n}; \mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\qquad \mathtt{cond}(\mathtt{h}_1 < \mathtt{N}_1) \text{ -> } <\mathtt{get}_1, *>.\mathtt{FiniteCapBuffer}(\mathtt{h}_1 + 1, \dots, \mathtt{h}_\mathtt{n}), \\
\qquad \vdots \\
\qquad \mathtt{cond}(\mathtt{h}_\mathtt{n} < \mathtt{N}_\mathtt{n}) \text{ -> } <\mathtt{get}_\mathtt{n}, *>.\mathtt{FiniteCapBuffer}(\mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n} + 1), \\
\qquad \mathtt{cond}(\mathtt{h}_1 > 0) \text{ -> } <\mathtt{put}_1, *>.\mathtt{FiniteCapBuffer}(\mathtt{h}_1 - 1, \dots, \mathtt{h}_\mathtt{n}), \\
\qquad \vdots \\
\qquad \mathtt{cond}(\mathtt{h}_\mathtt{n} > 0) \text{ -> } <\mathtt{put}_\mathtt{n}, *>.\mathtt{FiniteCapBuffer}(\mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n} - 1) \\
\quad \}
\end{array}
$$

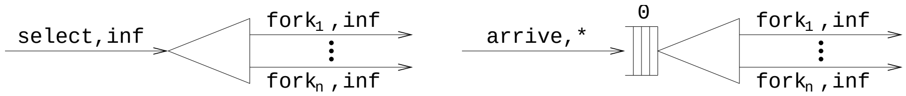
with the remaining constraints equal to those for the unbounded buffers, except for the fact that now the initial values of $\mathtt{h}_1, \dots, \mathtt{h}_\mathtt{n}$ cannot exceed the corresponding capacities.

It is worth observing that the buffers outlined above do not make any assumption about the order in which the customers of the same class are taken from the buffer with respect to the order in which they arrive at the buffer. Therefore, from the point of view of the four average performance indices introduced in Sect. 3.1, such buffers can be used to support any queueing discipline with noninterruptable service, like FCFS, LCFS, SIRO, and NP. On the contrary, the buffers above cannot be used to describe those queueing disciplines in which the service of a customer can be interrupted (PP, LCFS-PR), or can be divided into several rounds (RR, PS), or can immediately take place (IS).

To conclude, for the sequential compiler system of Sect. 2 we observe that `PB` is an unbounded buffer for two classes of customers.

4.5 Fork Processes

A fork process handles the splitting of each request of the customers of a certain class into several subrequests to be served in parallel by different service centers. As depicted in Fig. 8, we distinguish between two different kinds of fork processes depending on the presence or the absence of a buffer — modeled by another AEI — where the customers can wait before being split.



(a) Fork process with buffer *(b) Fork process with no buffer*

Fig. 8. Graphical representation of the fork processes

In the case of a fork process equipped with a buffer, the description of the fork process starts with the selection of the next customer to be split from the buffer. The specific syntax restriction requires that, in order for an AEI to be classified as a fork process equipped with a buffer, where the subrequests are forwarded to $\mathtt{n}$ different forks or service centers, it must have a behavior equivalent to the following one:

$$\begin{array}{l}\mathtt{ForkProcWithBuffer}(\mathtt{void};\mathtt{void}) = \\ \quad <\mathtt{select}, \mathtt{inf}>.<\mathtt{fork}_1, \mathtt{inf}>.\,\ldots\,.<\mathtt{fork}_\mathtt{n}, \mathtt{inf}>.\mathtt{ForkProcWithBuffer}()\end{array}$$

with: `select` being an input interaction attached to the output interaction of a buffer; $\mathtt{fork}_1, \ldots, \mathtt{fork}_\mathtt{n}$ being output interactions attached to input interactions of buffers (not related to join processes), fork processes with no buffer, or service processes with no buffer; the fork actions possibly having specific priority levels and specific weights if the related destinations are service processes with no buffer.

In the case of a fork process with no buffer, instead, the description of the fork process starts with the arrival of the next customer to be split directly from an arrival process, a fork, a join, or a service center. The specific syntax restriction requires that, in order for an AEI to be classified as a fork process with no buffer, where the subrequests are forwarded to n different forks or service centers, it must have a behavior equivalent to the following one:

$\mathtt{ForkProcNoBuffer(void;void)} =$

$\quad <\mathtt{arrive}, *>.<\mathtt{fork_1}, \mathtt{inf}>. \ldots .<\mathtt{fork_n}, \mathtt{inf}>.\mathtt{ForkProcNoBuffer()}$

with arrive being an input interaction — or an input or-interaction if the customers belong to a finite population and come directly from their arrival processes — attached to an output interaction of an arrival process, a fork process, a join process, or a service process and the remaining constraints equal to those for the fork processes equipped with a buffer.

4.6 Join Processes

A join process handles the merging of the subrequests of the customers of a certain class after they have been served in parallel by different service centers. As depicted in Fig. 9, we distinguish between two different kinds of join processes depending on the presence or the absence of buffers — modeled by other AEIs — where the subrequests can wait before being merged.

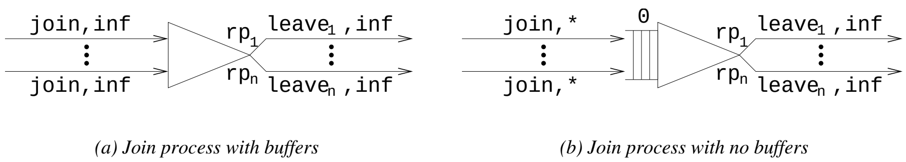


Fig. 9. Graphical representation of the join processes

In the case of a join process equipped with buffers, the description of the join process starts with the selection of the next subrequests to be merged from the buffers. The specific syntax restriction requires that, in order for an AEI to be classified as a join process equipped with buffers, where the subrequests are forwarded by several different joins or service centers and the result of the merging has a set of n different finite population arrival processes, forks, joins, or service centers as destinations chosen according to the intraclass routing probabilities $\mathtt{rp_1}, \ldots, \mathtt{rp_n}$, it must have a behavior equivalent to the following one:

$\mathtt{JoinProcWithBuffer(void;void)} =$

$\quad <\mathtt{join}, \mathtt{inf}>.\mathtt{JoinProcWithBuffer'()}$

$\mathtt{JoinProcWithBuffer'(void;void)} =$

$\quad \mathtt{choice}$

$\quad \{$

$\qquad <\mathtt{choose_1}, \mathtt{inf(1, rp_1)}>.<\mathtt{leave_1}, \mathtt{inf}>.\mathtt{JoinProcWithBuffer()},$

$\qquad \vdots$

$\qquad <\mathtt{choose_n}, \mathtt{inf(1, rp_n)}>.<\mathtt{leave_n}, \mathtt{inf}>.\mathtt{JoinProcWithBuffer()}$

$\quad \}$

with: `join` being an input and-interaction attached to the output interaction of each buffer; $\texttt{leave}_1, \ldots, \texttt{leave}_n$ being output interactions attached to input interactions of arrival processes for finite populations, buffers, fork processes with no buffer, join processes with no buffers, or service processes with no buffer; the destination choice actions omitted if there is only one possible destination; the departure actions possibly having specific priority levels and specific weights if the related destinations are service processes with no buffer; the departure actions omitted if the related destinations are arrival processes for unbounded populations; $\texttt{leave}_\texttt{i}$ being an output or-interaction if destination `i` is an arrival process for a finite population.

In the case of a join process with no buffers, instead, the description of the join process starts with the arrival of the subrequests to be merged directly from a join or a service center. The specific syntax restriction requires that, in order for an AEI to be classified as a join process with no buffers, with the same characteristics as in the previous example, it must have a behavior equivalent to the following one:

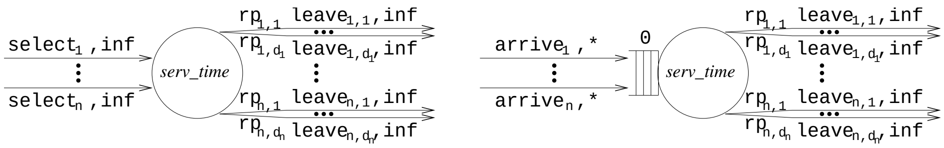
$$
\begin{array}{l}
\texttt{JoinProcNoBuffer}(\texttt{void};\texttt{void}) = \\
\quad <\texttt{join}, *>.\texttt{JoinProcNoBuffer}'() \\
\texttt{JoinProcNoBuffer}'(\texttt{void};\texttt{void}) = \\
\quad \texttt{choice} \\
\quad \{ \\
\qquad <\texttt{choose}_1, \texttt{inf}(1, \texttt{rp}_1)>.<\texttt{leave}_1, \texttt{inf}>.\texttt{JoinProcNoBuffer}(), \\
\qquad \vdots \\
\qquad <\texttt{choose}_n, \texttt{inf}(1, \texttt{rp}_n)>.<\texttt{leave}_n, \texttt{inf}>.\texttt{JoinProcNoBuffer}() \\
\quad \}
\end{array}
$$

with `join` being an input and-interaction attached to output interactions of join processes or service processes and the remaining constraints equal to those for the join processes equipped with buffers.

4.7 Service Processes

A service process is a server for customers of different classes, whose service times follow a phase-type distribution. As depicted in Fig. 10, we distinguish between two different kinds of service processes depending on the presence or the absence of a buffer — modeled by another AEI — where the customers can wait before being served.

In the case of a service process equipped with a buffer, the description of the service process starts with the selection of the next customer to be served from the buffer. As an example, the behavior of an AEI, which acts as a service process equipped with a buffer that serves customers of `n` different classes, where each class `i` has priority $\texttt{prio}_\texttt{i}$ to be selected, probability $\texttt{prob}_\texttt{i}$ to be selected among the classes with the same priority, exponentially distributed service time with rate $\mu_\texttt{i}$, and a set of $\texttt{d}_\texttt{i}$ different finite population arrival processes, forks, joins, or service centers as destinations chosen according to the intraclass routing probabilities $\texttt{rp}_{\texttt{i},1}, \ldots, \texttt{rp}_{\texttt{i},\texttt{d}_\texttt{i}}$, respectively, must be equivalent to the following



(a) Service process with buffer

(b) Service process with no buffer

Fig. 10. Graphical representation of the service processes

one:

$$
\begin{array}{l}
\mathtt{ServProcWithBuffer}(\mathtt{void};\mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\qquad <\mathtt{select_1}, \mathtt{inf}(\mathtt{prio_1}, \mathtt{prob_1})>.\mathtt{ServProcWithBuffer}'_1(), \\
\qquad \vdots \\
\qquad <\mathtt{select_n}, \mathtt{inf}(\mathtt{prio_n}, \mathtt{prob_n})>.\mathtt{ServProcWithBuffer}'_n() \\
\quad \} \\
\mathtt{ServProcWithBuffer}'_\mathtt{i}(\mathtt{void};\mathtt{void}) = \\
\quad <\mathtt{serve_i}, \mu_\mathtt{i}>.\mathtt{ServProcWithBuffer}''_\mathtt{i}() \\
\mathtt{ServProcWithBuffer}''_\mathtt{i}(\mathtt{void};\mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\qquad <\mathtt{choose_i}, \mathtt{inf}(1, \mathtt{rp_{i,1}})>.<\mathtt{leave_{i,1}}, \mathtt{inf}>.\mathtt{ServProcWithBuffer}(), \\
\qquad \vdots \\
\qquad <\mathtt{choose_i}, \mathtt{inf}(1, \mathtt{rp_{i,d_i}})>.<\mathtt{leave_{i,d_i}}, \mathtt{inf}>.\mathtt{ServProcWithBuffer}() \\
\quad \}
\end{array}
$$

with: $\mathtt{select_1}, \dots, \mathtt{select_n}$ being input interactions attached to the output interactions of a buffer; $\mathtt{leave_{1,1}}, \dots, \mathtt{leave_{n,d_n}}$ being output interactions attached to input interactions of arrival processes for finite populations, buffers, fork processes with no buffer, join processes with no buffers, or service processes with no buffer. The specific syntax restriction requires that, in order for an AEI to be classified as a service process equipped with a buffer, its behavior and interactions must be equivalent to the previous ones, with: the exponentially timed actions possibly replaced for certain classes of customers by terms describing more general phase-type distributions where $\mathtt{ServProcWithBuffer}''_\mathtt{i}()$ is substituted for each occurrence of `stop`; the destination choice actions omitted for those classes of customers for which there is only one possible destination; the departure actions possibly having specific priority levels and specific weights if the related destinations are service processes with no buffer; the departure actions omitted if the related destinations are arrival processes for unbounded populations; the departure actions being output or-interactions if the related destinations are arrival processes for finite populations.

In the case of a service process with no buffer, instead, the description of the service process starts with the arrival of the next customer to be served directly from arrival processes, forks, joins, or service centers. As an example, the behavior of an AEI, which acts as a service process with no buffer that serves customers of $\mathtt{n}$ different classes, where each class $\mathtt{i}$ has the same characteristics as in the previous example, must be equivalent to the following one:

$$
\begin{array}{l}
\mathtt{ServProcNoBuffer}(\mathtt{void};\mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\quad\quad <\mathtt{arrive}_{\mathtt{1}}, *>.\mathtt{ServProcNoBuffer}'_{\mathtt{1}}(), \\
\quad\quad \vdots \\
\quad\quad <\mathtt{arrive}_{\mathtt{n}}, *>.\mathtt{ServProcNoBuffer}'_{\mathtt{n}}() \\
\quad \} \\
\mathtt{ServProcNoBuffer}'_{\mathtt{i}}(\mathtt{void};\mathtt{void}) = \\
\quad <\mathtt{serve}_{\mathtt{i}}, \mu_{\mathtt{i}}>.\mathtt{ServProcNoBuffer}''_{\mathtt{i}}() \\
\mathtt{ServProcNoBuffer}''_{\mathtt{i}}(\mathtt{void};\mathtt{void}) = \\
\quad \mathtt{choice} \\
\quad \{ \\
\quad\quad <\mathtt{choose}_{\mathtt{i}}, \mathtt{inf}(\mathtt{1}, \mathtt{rp}_{\mathtt{i},\mathtt{1}})>.<\mathtt{leave}_{\mathtt{i},\mathtt{1}}, \mathtt{inf}>.\mathtt{ServProcNoBuffer}(), \\
\quad\quad \vdots \\
\quad\quad <\mathtt{choose}_{\mathtt{i}}, \mathtt{inf}(\mathtt{1}, \mathtt{rp}_{\mathtt{i},\mathtt{d}_{\mathtt{i}}})>.<\mathtt{leave}_{\mathtt{i},\mathtt{d}_{\mathtt{i}}}, \mathtt{inf}>.\mathtt{ServProcNoBuffer}() \\
\quad \}
\end{array}
$$

with: $\mathtt{arrive}_{\mathtt{1}}, \dots, \mathtt{arrive}_{\mathtt{n}}$ being input interactions attached to output interactions of arrival processes, fork processes, join processes, or service processes; $\mathtt{leave}_{\mathtt{1},\mathtt{1}}, \dots, \mathtt{leave}_{\mathtt{n},\mathtt{d}_{\mathtt{n}}}$ being output interactions attached to input interactions of arrival processes for finite populations, buffers, fork processes with no buffer, join processes with no buffers, or service processes with no buffer. The specific syntax restriction requires that, in order for an AEI to be classified as a service process with no buffer, its behavior and interactions must be equivalent to the previous ones, with the arrival actions being input or-interactions if the related customers belong to a finite population and come directly from their arrival processes, and the remaining constraints similar to those for the service processes equipped with a buffer.

It is worth observing that the service processes above allow for classes of customers with different service priorities (NP) and with specific service frequencies among classes with the same service priority (variants of SIRO). This is realized in two different ways for the two kinds of service processes. For the service processes equipped with a buffer, the service priorities and the service frequencies are expressed through the priority levels and the weights associated with the immediate selection actions. This is not possible in the case of the service processes with no buffer, because the arrival actions are passive, hence their priority levels and weights are reactive, i.e. their scope is limited to passive actions of the same type, whereas the arrival actions for different classes of customers have different types. This drawback is overcome by expressing the service priorities and the service frequencies through the priority levels and the weigths of the immediate output interactions of the arrival processes, fork processes, join processes, and service processes that forward customers to the service processes without buffer.

The case of a service center composed of several identical and independent servers is regulated by an additional specific syntax restriction. It requires first of all that the service processes constituting the multi-server service center are instances of the same AET with the same individual service time distribution and the interactions attached to the same AEIs. Three cases then arise. In the first case, the service processes share a buffer, from which they take all of their customers. In this case, the $\mathtt{put_1}, \dots, \mathtt{put_n}$ actions of the buffer must be output or-interactions. In the second case, the service processes have no buffer and receive some of their customers directly from arrival processes for unbounded populations, fork processes, join processes, or service processes. Similarly to the previous case, the output interactions of the upstream arrival processes for unbounded populations, fork processes, join processes, or service processes that are related to the multi-server service center, must be output or-interactions. In the third case, the service processes have no buffer and receive some of their customers from arrival processes for finite populations. For each such upstream arrival process, the action among $\mathtt{deliver_1}, \dots, \mathtt{deliver_n}$ that is related to the multi-server service center must be replaced in the specification of the arrival process by as many alternative copies of it as there are service processes in the multi-server service center.

To conclude, for the sequential compiler system of Sect. 2 we observe that SC is a service process equipped with a buffer, which serves two different classes of customers — returning to the unbounded populations to which they belong — according to two different hypoexponential distributions.

4.8 Translating AEIs into QN Basic Elements

Given an Æmilia specification that satisfies both the general and the specific syntax restrictions introduced in the previous sections, the translation of its constituent AEIs into their corresponding QN basic elements is carried out by applying a set of functions that provide the attributes that label the resulting QN basic elements, as depicted in Fig. 6, 7, 8, 9, and 10.

There are two groups of functions. The functions of the first group play a documental role and are subsequently used to assemble the QN basic elements according to the attachments declared in the Æmilia specification. The functions of the first group are *qnbe*, *name*, *input*, and *output*. When applied to an AEI, *qnbe* determines whether it is an arrival process for an unbounded population or a single customer belonging to a finite population, a buffer with unlimited or finite capacity, a fork process with or without buffer, a join process with or without buffers, or a service process with or without buffer. As an example, for the sequential compiler system of Sect. 2 we have:

$$
\begin{aligned}
qnbe(\mathtt{PG_1}) &= \text{arrival process for an unbounded population} \\
qnbe(\mathtt{PG_2}) &= \text{arrival process for an unbounded population} \\
qnbe(\mathtt{PB}) &= \text{unbounded buffer} \\
qnbe(\mathtt{SC}) &= \text{service process equipped with a buffer}
\end{aligned}
$$

The other three functions, instead, associate the name of the AEI with the corresponding QN basic element and label the incoming and outgoing arrow-headed arcs of the QN basic element with the corresponding input and output interactions of the AEI, respectively. As an example:

$$\begin{aligned}
name(\mathtt{PG_1}) &= \mathtt{PG_1}\\
name(\mathtt{PG_2}) &= \mathtt{PG_2}\\
name(\mathtt{PB}) &= \mathtt{PB}\\
name(\mathtt{SC}) &= \mathtt{SC}\\
input(\mathtt{PG_1}) &= \emptyset\\
input(\mathtt{PG_2}) &= \emptyset\\
input(\mathtt{PB}) &= \{<\mathtt{get_prog_1}, *>, <\mathtt{get_prog_2}, *>\}\\
input(\mathtt{SC}) &= \{<\mathtt{select_prog_1}, \mathtt{inf}>, <\mathtt{select_prog_2}, \mathtt{inf}>\}\\
output(\mathtt{PG_1}) &= \{<\mathtt{deliver_prog}, \mathtt{inf}>\}\\
output(\mathtt{PG_2}) &= \{<\mathtt{deliver_prog}, \mathtt{inf}>\}\\
output(\mathtt{PB}) &= \{<\mathtt{put_prog_1}, *>, <\mathtt{put_prog_2}, *>\}\\
output(\mathtt{SC}) &= \emptyset
\end{aligned}$$

$$\begin{aligned}
pt_distr(\mathtt{stop}) &= \emptyset\\
pt_distr(<\mathtt{phase}, \lambda>.\mathtt{E}) &= hypoexp(exp(\lambda), pt_distr(\mathtt{E}))\\
pt_distr(\mathtt{choice}\ \{ <\mathtt{branch_1}, \mathtt{inf}(1, \mathtt{w_1})>.\mathtt{E_1}, \ldots, <\mathtt{branch_n}, \mathtt{inf}(1, \mathtt{w_n})>.\mathtt{E_n} \}) &= hyperexp\Big(\tfrac{\mathtt{w_1}}{\mathtt{w_1}+\ldots+\mathtt{w_n}}, pt_distr(\mathtt{E_1}); \ldots; \tfrac{\mathtt{w_n}}{\mathtt{w_1}+\ldots+\mathtt{w_n}}, pt_distr(\mathtt{E_n})\Big)\\
pt_distr(\mathtt{A}(\underline{\mathtt{e}})) &= pt_distr(\mathtt{E}) \qquad \text{if } \mathtt{A}(\underline{\mathtt{x}}; \underline{\mathtt{y}}) = \mathtt{E}
\end{aligned}$$

Table 3. Recursive definition of function *pt_distr*

The functions of the second group are *int_arr_time*, *capacity*, *queueing_disc*, *serv_time*, and *intra_routing_prob*:

- Function *int_arr_time* indicates the phase-type distribution governing the interarrival times of the arrival processes. In the case of an AEI acting as an arrival process for an unbounded population of customers (resp. for a single customer belonging to a finite population), *int_arr_time* is the result of the application of function *pt_distr* of Table 3 to the term equivalent to `UnboundedPopArrProc()` (resp. `SingleCustArrProc()`), with each occurrence of the term equivalent to `UnboundedPopArrProc'()` (resp. `SingleCustArrProc'()`) replaced by `stop`. As an example:

$$int\_arr\_time(\mathtt{PG_1}) = exp(\lambda_1)$$
$$int\_arr\_time(\mathtt{PG_2}) = exp(\lambda_2)$$

- Function *capacity* determines the capacity of the buffers. In the case of an AEI acting as an unbounded buffer, the application of *capacity* yields ∞. In the case of an AEI acting as a finite capacity buffer for $\mathtt{n}$ classes of customers, where the customers of class $\mathtt{i}$ can occupy up to $\mathtt{N_i}$ positions in the buffer as specified by the parameters of the AEI behavior, the application of *capacity* yields $\mathtt{N_1}, \dots, \mathtt{N_n}$. As an example:

$$capacity(\mathtt{PB}) = \infty$$

- Function *queueing_disc* defines the queueing discipline of the buffers based on the priority levels of the input interactions of the service processes to which the buffers are attached. If all the input interactions of the service process to which a buffer is attached have the same priority level, then the application of *queueing_disc* to the buffer yields FCFS, otherwise NP. As an example:

$$queueing\_disc(\mathtt{PB}) = \mathrm{FCFS}$$

- Function *serv_time* establishes the phase-type distribution governing the service times of the service processes. In the case of an AEI acting as a service process equipped with a buffer (resp. with no buffer) for $\mathtt{n}$ classes of customers, *serv_time* for class $\mathtt{i}$ is the result of the application of function *pt_distr* of Table 3 to the term equivalent to $\mathtt{ServProcWithBuffer}'_{\mathtt{i}}()$ (resp. $\mathtt{ServProcNoBuffer}'_{\mathtt{i}}()$), with each occurrence of the term equivalent to $\mathtt{ServProcWithBuffer}''_{\mathtt{i}}()$ (resp. $\mathtt{ServProcNoBuffer}''_{\mathtt{i}}()$) replaced by $\mathtt{stop}$. As an example:

$$serv\_time(\mathtt{SC}, 1) = hypoexp(exp(\mu_{\mathtt{l}}), exp(\mu_{\mathtt{p}}), exp(\mu_{\mathtt{c}}), exp(\mu_{\mathtt{o}}), exp(\mu_{\mathtt{g}}))$$
$$serv\_time(\mathtt{SC}, 2) = hypoexp(exp(\mu_{\mathtt{l}}), exp(\mu_{\mathtt{p}}), exp(\mu_{\mathtt{c}}), exp(\mu_{\mathtt{g}}))$$

- Function *intra_routing_prob* reports the intraclass routing probabilities for the customers of a certain class leaving an arrival process, a join process, or a service process. It is simply derived from the weights of the choice actions of the QN basic element from which the customers of the considered class depart. It is worth observing that, in the case of a join process or a service process, this function returns a value also for a destination given by an arrival process for an unbounded customer population, and that, for a complete graphical representation of the considered QN basic element, such a value must label the join process or service process despite of the absence of the related outgoing arrow-headed arc. As an example:

$$intra\_routing\_prob(\mathtt{PG}_1, 1, \mathtt{PB}) = 1$$
$$intra\_routing\_prob(\mathtt{PG}_2, 2, \mathtt{PB}) = 1$$
$$intra\_routing\_prob(\mathtt{SC}, 1, -) = 1$$
$$intra\_routing\_prob(\mathtt{SC}, 2, -) = 1$$

We conclude by showing in Fig. 11 the QN basic elements associated with the AEIs constituting the sequential compiler system.

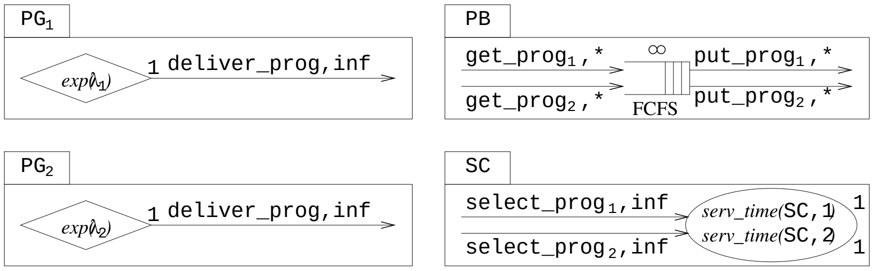


Fig. 11. QN basic elements of `SeqCompSys`

4.9 Attachment Driven Composition of QN Basic Elements

Given an Æmilia specification that satisfies both the general and the specific syntax restrictions introduced in the previous sections, once each of its constituent AEIs has been mapped to its corresponding QN basic element, the translation is completed by connecting the resulting QN basic elements according to the attachments declared in the Æmilia specification. Graphically, this amounts to superposing the arrow-headed arcs of the QN basic elements corresponding to interactions attached to each other. The obtained QN is closed if there are no arrival processes, in which case the QN population is given by the summation of the initial number of customers in each buffer.

It is worth observing that the specific syntax restrictions ensure the correct composition of the QN basic elements obtained from the translation of the AEIs declared in the Æmilia specification, because the restrictions impose that:

- The input interactions of an arrival process for a finite population cannot be attached to output interactions of other arrival processes, buffers, and fork processes.
- The output interactions of an arrival process cannot be attached to input interactions of other arrival processes for finite populations, buffers related to join processes, and join processes with no buffers.
- The input interactions of a buffer cannot be attached to output interactions of other buffers.
- The output interactions of a buffer cannot be attached to input interactions of arrival processes for finite populations and other buffers.
- The output interactions of a fork process cannot be attached to input interactions of arrival processes for finite populations, buffers related to join processes, and join processes with no buffers.
- The input interactions of a join process cannot be attached to output interactions of arrival processes and fork processes.
- Suitable or-interactions are used in the case of arrival processes for finite populations as well as multi-server service centers.
- Suitable and-interactions are used in the case of join processes.

We conclude by showing in Fig. 12 the QN associated with the Æmilia specification of the sequential compiler system of Sect. 2.

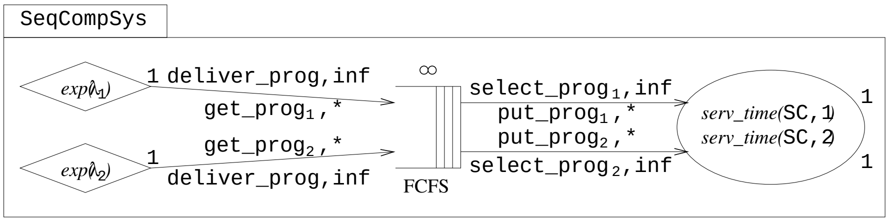


Fig. 12. QN associated with `SeqCompSys`

5 A Practical Methodology

When tackling the construction of a software system, at the SA design level there are two main issues related to the system performance. First, the designer may need to choose among several alternative SAs for the system under study, with the choice being driven especially by performance considerations. Second, for a specific SA of the system under study, the designer may want to understand whether its performance can be improved and, if so, it would be desirable for the designer to have some diagnostic information that guide the modification of the SA itself. In this section we show how these two issues can be addressed in practice by employing a methodology based on the translation of Æmilia specifications into QN models, which allows for a quick prediction, improvement, and comparison of the performance of different SAs for the system under study.

Before illustrating the methodology, it is worth recalling that the use of an ADL like Æmilia in the methodology is due to the fact that Æmilia comes equipped with an analysis machinery that supports both functional verification and performance evaluation, together with some SA level checks. On the other hand, the use of QNs in the methodology is motivated by their capability of providing performance indices both at their constituent service center level and at the overall network level. Another advantage is that the local performance indices can be interpreted back on the components of the architectural specification and used as a feedback to ameliorate the performance of the architectural specification.

The methodology focuses on the four average performance measures mentioned in Sect. 3.1, which will be computed for groups of software components forming a QN service center:

- The throughput is a measure of the productivity of the service centers, so it can provide information about those components that are bottlenecks, i.e. those components that are responsible for degrading the system performance.
- The utilization is the fraction of time during which a service center is being used. In software component terms, this amounts to the fraction of time during which the code of a group of components is being executed, so it supplies useful information, which may be exploited at deployment time, about the relative usage of computational resources by different software components.

- The mean number of customers present in a service center is an indicator to be used for a reasonable dimensioning of buffers and data repositories in general, in order to avoid performance degradation due to code execution blocking (under-sized buffers) and waste of memory (over-sized buffers).
- The mean response time is the time spent on average by a customer within a service center. In software component terms, this essentially amounts to the expected running time of a group of software components for a complete execution of their code. In other terms, it is a measure of the quality of service perceived by a generic user of the software system.

The four average performance measures considered above, although generally useful for the feedback they quickly provide, are not necessarily connected in a specific way to the performance requirements of the system under study. In addition to that, as we shall see they are usually computed after applying some approximations at the Æmilia specification level or at the QN level. As a consequence, the methodology must be complemented by an additional phase, in which the exact Æmilia specification of the chosen SA is checked against the specific performance requirements.

The various phases of the methodology are depicted in Fig 13. Given a set of (functional and performance) requirements characterizing the software system under study, the designer can devise multiple alternative SAs that should meet the requirements. Such SAs are typically expressed in an informal way, e.g. in natural language or through box-and-line diagrams (phase 1). Then the designer works on each SA separately.

First of all, since the SA must be analyzed, its informal description must be converted by the designer into a component-oriented formal representation supporting both functional verification and performance evaluation. This is carried out by the designer using Æmilia (phase 2).

The Æmilia specification produced for the SA does not necessarily satisfy the syntax restrictions that make it possible to translate the specification into a QN model. In such a case, the original Æmilia specification must be approximated with another Æmilia specification that meets both the general restrictions and the specific restrictions, so that it can be converted into a QN (phase 3). The approximation must be conducted in a way that every original AEI becomes an arrival process, a buffer with a noninterruptable queueing discipline, a fork process, a join process, or a service process, with the appropriate attachments. This may require that the behavior of some AEI is modified. This happens e.g. when some scheduling algorithm based on PP, LCFS-PR, RR, PS, or IS is adopted within an AEI, which needs to be approximated with a noninterruptable discipline. [4] In addition, the approximation may require adding new AEIs or deleting existing AEIs. This is the case e.g. when an AEI contains several alternative exponentially timed actions, which means that the AEI is the combination of several components running in parallel that must therefore be

[4] According to the results mentioned at the end of Sect. 3.1, this approximation may be exact with respect to the four average performance indices of interest.

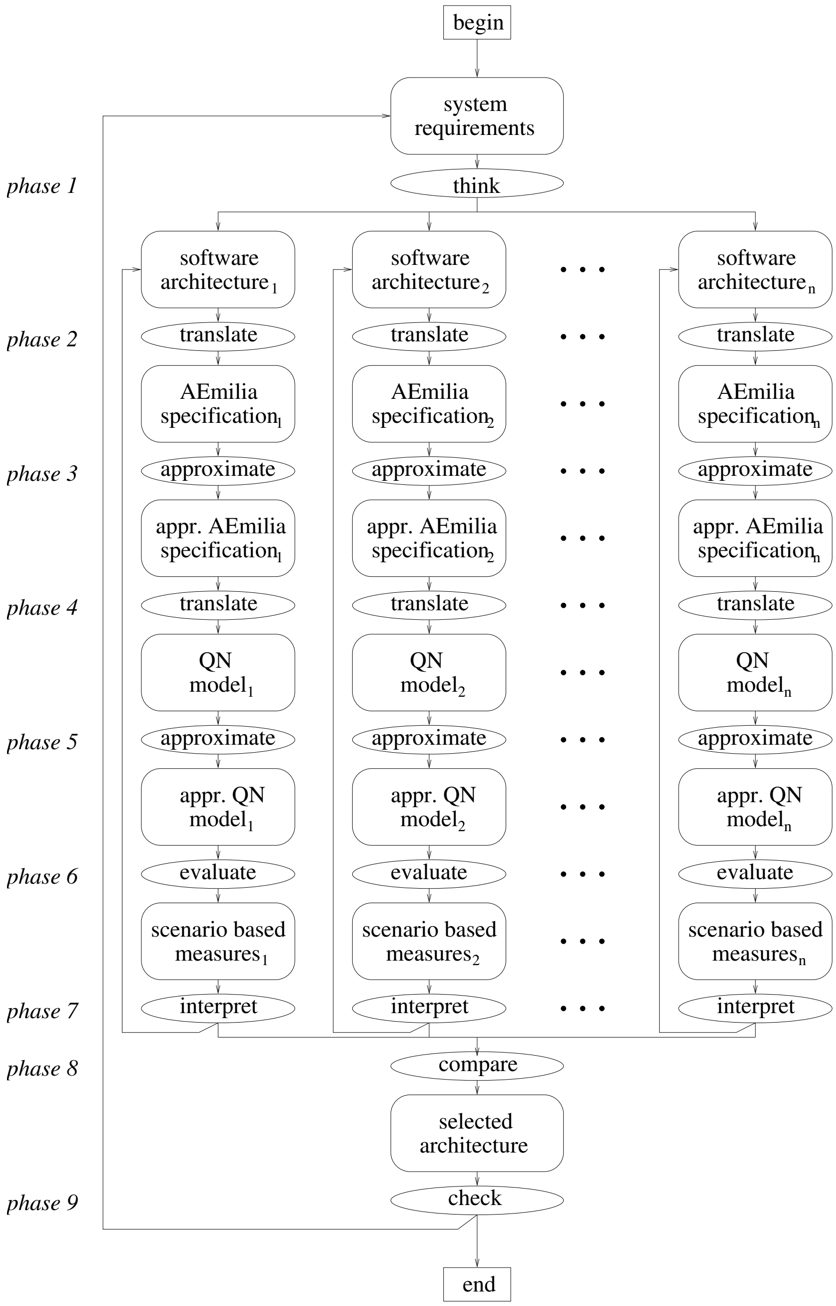


Fig. 13. Phases of the methodology

represented separately. The approximation is justified at this level of abstraction by the fact that we are more interested in getting a quick indication about the performance of the SA, rather than in its precise evaluation.

The Æmilia specification is then automatically translated into a QN based performance model, in accordance with the guidelines provided in Sect. 3.1 (phase 4).

The QN model obtained for the SA is not necessarily product form, which may hamper a quick computation of the four average performance measures of interest at the component level as well as at the overall system level. In such a case, the original QN model must be approximated with another QN model that hopefully is product form, with the approximation aiming at transforming every service center of the QN into a QS M/M/1 or a QS M/M/m with possibly variable arrival rates (phase 5):

- Finite, nonzero capacity buffers must be transformed into unbounded buffers, and similarly NP must be transformed into FCFS. This is not a problem at this stage as we are more interested in finding a reasonable size and queueing discipline for the buffers, rather than working with a size and a queueing discipline fixed a priori. However, zero capacity buffers cannot be approximated as seen before, because their performance would be significantly altered.
- Since we are considering only noninterruptable, nonprioritized queueing disciplines, phase-type distributed interarrival times and service times must be approximated with exponentially distributed interarrival times and service times having the same expected values as the original ones, respectively.
- For each multi-class service center, the classes must be approximated with a single class whose service time distribution is the convex combination of the original (exponential) service time distributions, with the coefficients being given by the interclass routing probabilities of the original classes. The resulting hyperexponential service time distribution must then be approximated with an exponential service time distribution with the same expected value. [5]
- As far as the arrival processes are concerned, we recall that the overall arrival rate for an unbounded customer population (represented by a single arrival process) is constant, whereas the overall arrival rate for a finite customer population (represented by several arrival processes) is variable. Such overall arrival rates convey useful information to be exploited when computing the interclass routing probabilities needed for the approximation of multi-class service centers with single-class service centers. We also recall that the total arrival rate for a service center having several external arrivals is the sum of the overall arrival rates of the different populations of customers that can get to the service center.

Although the perturbation of the four average performance measures introduced by the approximations above cannot easily be quantified, it is worth reminding

[5] If all the classes have the same (exponential) service time distribution, then the overall service time distribution coincides with the original one and no further approximation is needed.

that, as observed in [37], QN models are in general robust, i.e. their approximate analysis is in any case useful to get some indications about the performance of the systems they represent. With respect to the architectural level of abstraction, the approximations above are justified in the framework of the proposed methodology by the fact that we are more interested in getting a quick feedback about how to improve the average performance of a specific SA or making a rapid comparison of the average performance of architectural alternatives, rather than in a precise performance evaluation. This is conducted anyhow, but only in the last phase of the methodology, in order to make sure that the exact Æmilia specification of the best architectural design, selected with respect to the four average performance indices possibly after some approximations, actually meets the specific performance requirements.

The QN model for the SA is subsequently evaluated in order to compute the throughput, the utilization, the mean number of customers, and the mean response time for each service center, as well as the corresponding measures for the overall QN, in different scenarios (phase 6):

– Such an evaluation preliminarily requires the parameterization of the QN and the characterization of its workload. Since the Æmilia specifications are already parameterized and their translation into QN models preserves the parameterization, the QN model for the SA is parameterized by construction. As far as the characterization of the workload is concerned, we have to include in the Æmilia specification suitable arrival processes for those service centers with external arrivals and to establish the number of customers initially present in each buffer.
– The evaluation of the QN then proceeds in accordance with some scenarios of interest, which are derived by playing with the arrival rates and the service rates. As an example, the four average performance measures can be computed under light and heavy load, by making the interrival rates vary from small values to values close to those of the service rates (without violating stability), or by changing the numbers of customers initially in the buffers from small values (close to zero) to large values (close to the buffer capacities). As another example, it is useful to assess how the four average performance measures vary in the case in which all the service centers have service rates of the same order of magnitude, in the case in which there is one service center whose service rate is some orders of magnitude smaller than the service rates of the other service centers, and in the more general case in which the rates of all the service centers range in an interval between a minimum rate and a maximum rate that are some orders of magnitude apart.
– The evaluation of the QN can be accomplished on the basis of the selected scenarios in three different ways. The most convenient way is symbolic analysis, which is possible only if the QN is open and product form and has a simple topology. In this case the four average performance measures are expressed through a suitable combination of the formulas at the end of Sect. 3.1, which is particularly desirable at the architectural level of design, as usually

the actual values of the arrival rates and the service rates are not known yet in this early stage. If the QN is product form but it is not open or it has not a simple topology, then the four average performance measures are calculated after solving the traffic equations or by applying some algorithm like MVA, which require the specification of the values of the arrival rates, the service rates, and the intraclass routing probabilities. In this case, the specification of the parameter values will be driven by the selected scenarios. If the QN is not product form, as may happen in the case in which some buffers have zero capacity or there are forks and joins, then we resort to approximation algorithms, which again require the scenario driven specification of the parameter values.

Once the values of the four average performance indices for the SA are available in the selected scenarios, they are interpreted back on the Æmilia specification of the SA at the level of the groups of components forming the QN service centers (phase 7). On the basis of such a component-oriented feedback, the designer can make some modification on the SA to improve its performance and return to phase 2, or proceed with the next phase.

When the predict-improve cycle is terminated for every SA devised for the system under study, all the alternative SAs are compared on the basis of the four average performance measures in different scenarios, in order to single out the best one (phase 8). Of course, the scenario driven comparison should be fair, which means that all the alternative SAs should be given comparable workloads in each scenario. In addition, we note that the outcomes of the comparison in different scenarios may be different. In this case, the best SA must be selected by taking into account the frequency with which every considered scenario can arise in practice.

Finally, the chosen SA is checked against the specific performance requirements of the system under study (phase 9). As explained at the beginning of this section, this is needed because the four average performance measures used to choose among the alternative SAs are not necessarily connected in a specific way to the performance requirements, so we do not know whether the best SA selected on the basis of the four average performance measures actually meets the performance requirements. Moreover, we have to take into account that the Æmilia specification or the QN model of the chosen SA might have been approximated in the previous phases, while now we have to consider the exact Æmilia specification of the chosen SA. The check of such an exact Æmilia specification can be accomplished by formally specifying the performance requirements through reward structures and temporal logic formulas [14, 22, 5]. If the outcome of the check is positive, then the application of the methodology terminates, otherwise the designer has to reconsider the performance requirements — as they may turn out to be impossible to meet — and apply the methodology again.

6 Comparing Three Different Compiler Architectures

The compiler shown in Table 2, whose QN model is reported in Fig. 12, examines one source program at a time, i.e. it is a completely sequential compiler. In this section we consider two different architectures realizing a pipeline compiler and a concurrent compiler, respectively, and we apply the methodology described in Sect. 5 to compare the three alternative architectures. This requires specifying the pipeline compiler and the concurrent compiler in Æmilia, building their associated QN models as described in Sect. 4, and computing the four average performance measures on the three QN models in some scenarios of interest.

6.1 Æmilia Specification of the Pipeline Compiler

The architecture for the pipeline compiler allows the various compilation phases to work on different programs. This is achieved by splitting the various phases into different AETs — one for the lexer, one for the parser, one for the type checker, one for the code optimizer, and one for the code generator — and by providing each such AET with its own buffer.

The pipeline compiler system, which includes the arrival sources for the two classes of programs, is graphically represented in Fig. 14, while its Æmilia specification is given in Tables 4, 5, and 6. Each compilation phase is modeled by a specific AET. In addition, there are two further AETs that model unbounded FCFS buffers accepting one class of programs — for the optimizer — or two classes of programs — for all the other compilation phases. The declared AEIs and their attachments ensure that the compilation phases are combined in the correct order and that each phase is provided with its own buffer, so making it possible the simultaneous compilation of several programs at different stages. `PipeCompSys` is the output of phase 2 of the methodology for the pipeline architecture. Since it satifies all the syntax restrictions of Sect. 4, it is also the output of phase 3.

6.2 QN Model of the Pipeline Compiler

In order to carry out phase 4 for the pipeline architecture, we apply the functions defined in Sect. 4.8 to the AEIs of `PipeCompSys`, thus obtaining their corresponding QN basic elements. In particular, the service time of each class of programs within each service process is exponentially distributed as follows:

$$\begin{aligned}
&\mathit{serv_time}(\mathtt{L},1) = \mathit{serv_time}(\mathtt{L},2) = \mathit{exp}(\mu_{\mathtt{l}})\\
&\mathit{serv_time}(\mathtt{P},1) = \mathit{serv_time}(\mathtt{P},2) = \mathit{exp}(\mu_{\mathtt{p}})\\
&\mathit{serv_time}(\mathtt{C},1) = \mathit{serv_time}(\mathtt{C},2) = \mathit{exp}(\mu_{\mathtt{c}})\\
&\mathit{serv_time}(\mathtt{O}) = \mathit{exp}(\mu_{\mathtt{o}})\\
&\mathit{serv_time}(\mathtt{G},1) = \mathit{serv_time}(\mathtt{G},2) = \mathit{exp}(\mu_{\mathtt{g}})
\end{aligned}$$

By connecting the QN basic elements according to the attachments, we then obtain the QN model depicted in Fig. 15, where the actions labeling the arrows have been omitted for the sake of readability. Note that the QN structure closely resembles the structure of the graphical description in Fig. 14, making it easier

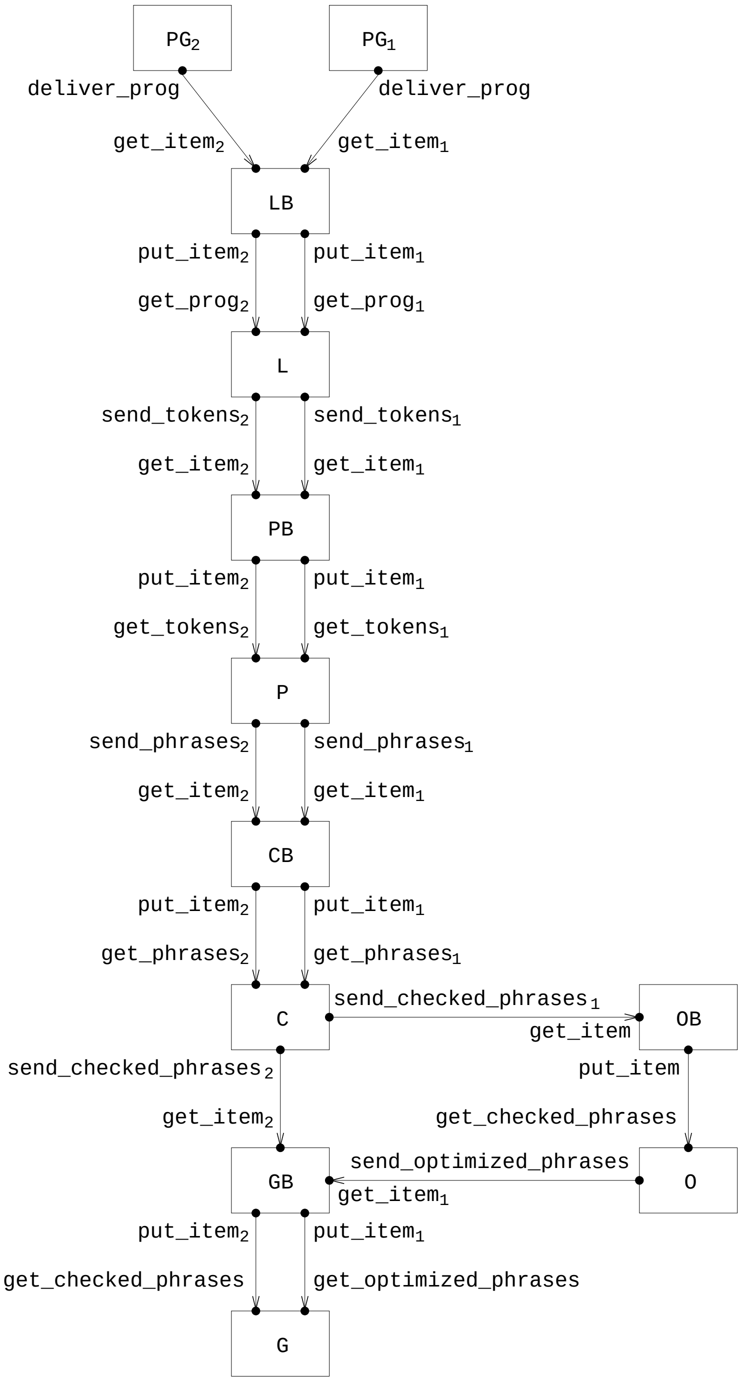


Fig. 14. Graphical description of `PipeCompSys`

```
ARCHI_TYPE                PipeCompSys(rate λ1, λ2, μ1, μp, μc, μo, μg)

 ARCHI_ELEM_TYPES

  ELEM_TYPE               ProgGenT(rate λ)
   BEHAVIOR               ProgGen(void; void) =
                            <generate_prog, λ>.<deliver_prog, inf>.ProgGen()
   INPUT_INTERACTIONS
   OUTPUT_INTERACTIONS    UNI deliver_prog

  ELEM_TYPE               OneClassBufferT(integer h)
   BEHAVIOR               OneClassBuffer(integer h; void) =
                            choice
                            {
                              <get_item, ∗>.OneClassBuffer(h + 1),
                              cond(h > 0) −> <put_item, ∗>.
                                               OneClassBuffer(h − 1)
                            }
   INPUT_INTERACTIONS     UNI get_item
   OUTPUT_INTERACTIONS    UNI put_item

  ELEM_TYPE               TwoClassesBufferT(integer h1, h2)
   BEHAVIOR               TwoClassesBuffer(integer h1, h2; void) =
                            choice
                            {
                              <get_item1, ∗>.TwoClassesBuffer(h1 + 1, h2),
                              <get_item2, ∗>.TwoClassesBuffer(h1, h2 + 1),
                              cond(h1 > 0) −> <put_item1, ∗>.
                                                TwoClassesBuffer(h1 − 1, h2),
                              cond(h2 > 0) −> <put_item2, ∗>.
                                                TwoClassesBuffer(h1, h2 − 1)
                            }
   INPUT_INTERACTIONS     UNI get_item1; get_item2
   OUTPUT_INTERACTIONS    UNI put_item1; put_item2

  ELEM_TYPE               LexerT(rate μ1)
   BEHAVIOR               Lexer(void; void) =
                            choice
                            {
                              <get_prog1, inf>.<recognize_tokens, μ1>.
                                <send_tokens1, inf>.Lexer(),
                              <get_prog2, inf>.<recognize_tokens, μ1>.
                                <send_tokens2, inf>.Lexer()
                            }
   INPUT_INTERACTIONS     UNI select_prog1; select_prog2
   OUTPUT_INTERACTIONS    UNI send_tokens1; send_tokens2
```

Table 4. Textual description of PipeCompSys — first part

```
ELEM_TYPE             ParserT(rate μp)
  BEHAVIOR            Parser(void; void) =
                        choice
                        {
                          <get_tokens1, inf>.<parse_phrases, μp>.
                            <send_phrases1, inf>.Parser(),
                          <get_tokens2, inf>.<parse_phrases, μp>.
                            <send_phrases2, inf>.Parser()
                        }
  INPUT_INTERACTIONS  UNI get_tokens1; get_tokens2
  OUTPUT_INTERACTIONS UNI send_phrases1; send_phrases2

ELEM_TYPE             CheckerT(rate μc)
  BEHAVIOR            Checker(void; void) =
                        choice
                        {
                          <get_phrases1, inf>.<check_phrases, μc>.
                            <send_checked_phrases1, inf>.Checker(),
                          <get_phrases2, inf>.<check_phrases, μc>.
                            <send_checked_phrases2, inf>.Checker()
                        }
  INPUT_INTERACTIONS  UNI get_phrases1; get_phrases2
  OUTPUT_INTERACTIONS UNI send_checked_phrases1; send_checked_phrases2

ELEM_TYPE             OptimizerT(rate μo)
  BEHAVIOR            Optimizer(void; void) =
                          <get_checked_phrases, inf>.
                            <optimize_phrases, μo>.
                            <send_optimized_phrases, inf>.Optimizer()
  INPUT_INTERACTIONS  UNI get_checked_phrases
  OUTPUT_INTERACTIONS UNI send_optimized_phrases

ELEM_TYPE             GeneratorT(rate μg)
  BEHAVIOR            Generator(void; void) =
                        choice
                        {
                          <get_optimized_phrases, inf>.
                            <generate_code, μg>.Generator(),
                          <get_checked_phrases, inf>.
                            <generate_code, μg>.Generator()
                        }
  INPUT_INTERACTIONS  UNI get_optimized_phrases; get_checked_phrases
  OUTPUT_INTERACTIONS
```

Table 5. Textual description of PipeCompSys — second part

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES $\mathtt{PG}_1$: ProgGenT(λ_1);
$\mathtt{PG}_2$: ProgGenT(λ_2);
LB : TwoClassesBufferT(0, 0);
L : LexerT(μ_1);
PB : TwoClassesBufferT(0, 0);
P : ParserT(μ_p);
CB : TwoClassesBufferT(0, 0);
C : CheckerT(μ_c);
OB : OneClassBufferT(0);
O : OptimizerT(μ_o);
GB : TwoClassesBufferT(0, 0);
G : GeneratorT(μ_g);

ARCHI_INTERACTIONS

ARCHI_ATTACHMENTS FROM $\mathtt{PG}_1$.deliver_prog TO $\mathtt{LB.get\_item}_1$;
FROM $\mathtt{PG}_2$.deliver_prog TO $\mathtt{LB.get\_item}_2$;
FROM $\mathtt{LB.put\_item}_1$ TO $\mathtt{L.get\_prog}_1$;
FROM $\mathtt{LB.put\_item}_2$ TO $\mathtt{L.get\_prog}_2$;
FROM $\mathtt{L.send\_tokens}_1$ TO $\mathtt{PB.get\_item}_1$;
FROM $\mathtt{L.send\_tokens}_2$ TO $\mathtt{PB.get\_item}_2$;
FROM $\mathtt{PB.put\_item}_1$ TO $\mathtt{P.get\_tokens}_1$;
FROM $\mathtt{PB.put\_item}_2$ TO $\mathtt{P.get\_tokens}_2$;
FROM $\mathtt{P.send\_phrases}_1$ TO $\mathtt{CB.get\_item}_1$;
FROM $\mathtt{P.send\_phrases}_2$ TO $\mathtt{CB.get\_item}_2$;
FROM $\mathtt{CB.put\_item}_1$ TO $\mathtt{C.get\_phrases}_1$;
FROM $\mathtt{CB.put\_item}_2$ TO $\mathtt{C.get\_phrases}_2$;
FROM $\mathtt{C.send\_checked\_phrases}_1$ TO OB.get_item;
FROM $\mathtt{C.send\_checked\_phrases}_2$ TO $\mathtt{GB.get\_item}_2$;
FROM OB.put_item TO O.get_checked_phrases;
FROM O.send_optimized_phrases TO $\mathtt{GB.get\_item}_1$;
FROM $\mathtt{GB.put\_item}_1$ TO G.get_optimized_phrases;
FROM $\mathtt{GB.put\_item}_2$ TO G.get_checked_phrases

END

Table 6. Textual description of PipeCompSys — third part

to interpret at the architectural description level the performance results that will be obtained at the QN level.

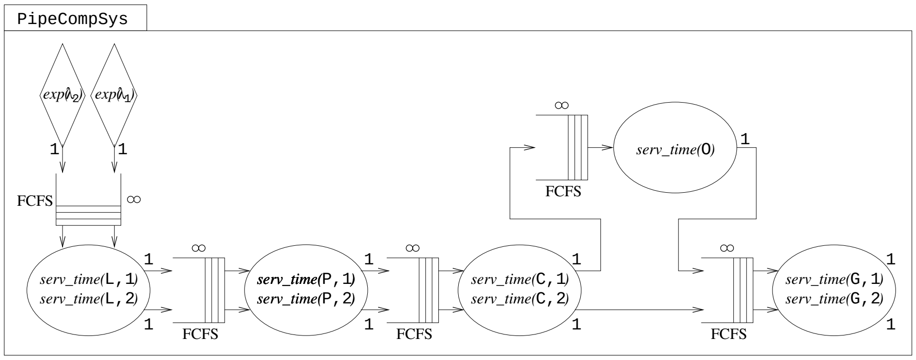


Fig. 15. QN associated with `PipeCompSys`

6.3 Æmilia Specification of the Concurrent Compiler

The architecture for the concurrent compiler consists of two sequential compilers operating in parallel and taking the programs from a shared buffer. Its graphical representation is shown in Fig. 16, while its Æmilia description is reported in Table 7. The difference with respect to `SeqCompSys` is that in `ConcCompSys` there are two instances of `SC` and the output interactions of the buffer are declared to be or-interactions, thus forwarding programs to either of the two instances of `SC`. It is easy to see that `ConcCompSys` satisfies all the syntax restrictions of Sect. 4.

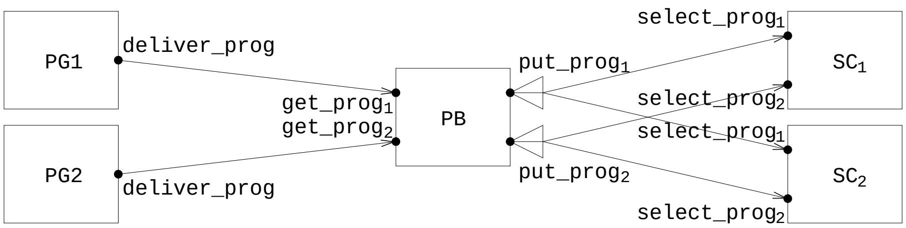


Fig. 16. Graphical description of `ConcCompSys`

6.4 QN Model of the Concurrent Compiler

The QN for `ConcCompSys` is shown in Fig. 17, where once again the action labeling the arrows have been omitted for simplicity. Differently from the QN for `SeqCompSys`, now we have a service center with two servers.

```
ARCHI_TYPE                ConcCompSys(rate λ1, λ2, μ1, μp, μc, μo, μg)
 ARCHI_ELEM_TYPES
  ELEM_TYPE               ProgGenT(rate λ)
   BEHAVIOR               ProgGen(void; void) =
                            <generate_prog, λ>.<deliver_prog, inf>.ProgGen()
   INPUT_INTERACTIONS
   OUTPUT_INTERACTIONS    UNI deliver_prog
  ELEM_TYPE               ProgBufferT(integer h1, h2)
   BEHAVIOR               ProgBuffer(integer h1, h2; void) =
                            choice
                            {
                              <get_prog1, *>.ProgBuffer(h1 + 1, h2),
                              <get_prog2, *>.ProgBuffer(h1, h2 + 1),
                              cond(h1 > 0) -> <put_prog1, *>.ProgBuffer(h1 - 1, h2),
                              cond(h2 > 0) -> <put_prog2, *>.ProgBuffer(h1, h2 - 1)
                            }
   INPUT_INTERACTIONS     UNI get_prog1; get_prog2
   OUTPUT_INTERACTIONS    OR put_prog1; put_prog2
  ELEM_TYPE               SeqCompT(rate μ1, μp, μc, μo, μg)
   BEHAVIOR               SeqComp(void; void) =
                            choice
                            {
                              <select_prog1, inf>.<recognize_tokens, μ1>.
                                <parse_phrases, μp>.<check_phrases, μc>.
                                <optimize_code, μo>.<generate_code, μg>.SeqComp(),
                              <select_prog2, inf>.<recognize_tokens, μ1>.
                                <parse_phrases, μp>.<check_phrases, μc>.
                                <generate_code, μg>.SeqComp()
                            }
   INPUT_INTERACTIONS     UNI select_prog1; select_prog2
   OUTPUT_INTERACTIONS
 ARCHI_TOPOLOGY
  ARCHI_ELEM_INSTANCES    PG1 : ProgGenT(λ1);
                          PG2 : ProgGenT(λ2);
                          PB : ProgBufferT(0, 0);
                          SC1, SC2 : SeqCompT(μ1, μp, μc, μo, μg)
  ARCHI_INTERACTIONS
  ARCHI_ATTACHMENTS       FROM PG1.deliver_prog TO PB.get_prog1;
                          FROM PG2.deliver_prog TO PB.get_prog2;
                          FROM PB.put_prog1 TO SC1.select_prog1;
                          FROM PB.put_prog1 TO SC2.select_prog1;
                          FROM PB.put_prog2 TO SC1.select_prog2;
                          FROM PB.put_prog2 TO SC2.select_prog2
END
```

Table 7. Textual description of ConcCompSys

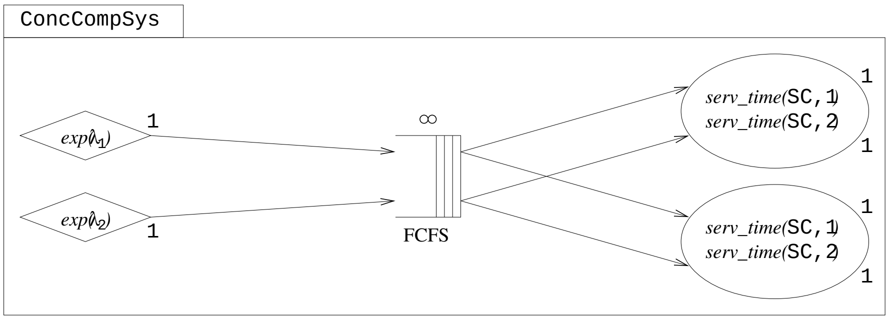


Fig. 17. QN associated with `ConcCompSys`

6.5 Analysis of the Sequential Compiler

We now return to the sequential architecture and we evaluate it with respect to the four average performance measures in a given set of scenarios of interest. Let us concentrate on a specific scenario in this set, which is characterized by certain actual values for the numeric parameters of `SeqCompSys` denoted by $\lambda_{\tt seq,1}$, $\lambda_{\tt seq,2}$, $\mu_{\tt seq,l}$, $\mu_{\tt seq,p}$, $\mu_{\tt seq,c}$, $\mu_{\tt seq,o}$, and $\mu_{\tt seq,g}$. Before proceeding with phase 6, we observe that the QN model associated with `SeqCompSys`, which is shown in Fig. 12, is closely related to a QS M/M/1. To transform it into a QS M/M/1, we perform phase 5 as follows:

- The two arrival processes are merged into a single arrival process with arrival rate $\lambda_{\tt seq} = \lambda_{\tt seq,1} + \lambda_{\tt seq,2}$. We observe that the probability that an incoming program belongs to class `1` (resp. `2`) is $\lambda_{\tt seq,1}/\lambda_{\tt seq}$ (resp. $\lambda_{\tt seq,2}/\lambda_{\tt seq}$).
- The hypoexponential service time for the first class of programs is approximated with an exponential service time with rate $\mu_{\tt seq,1}$ such that $\mu^{-1}_{\tt seq,1} = \mu^{-1}_{\tt seq,l} + \mu^{-1}_{\tt seq,p} + \mu^{-1}_{\tt seq,c} + \mu^{-1}_{\tt seq,o} + \mu^{-1}_{\tt seq,g}$.
- The hypoexponential service time for the second class of programs is approximated with an exponential service time with rate $\mu_{\tt seq,2}$ such that $\mu^{-1}_{\tt seq,2} = \mu^{-1}_{\tt seq,l} + \mu^{-1}_{\tt seq,p} + \mu^{-1}_{\tt seq,c} + \mu^{-1}_{\tt seq,g}$.
- The two classes of programs are merged into a single class, whose hyperexponential service time is approximated with an exponential service time with rate $\mu_{\tt seq}$ such that $\mu^{-1}_{\tt seq} = (\lambda_{\tt seq,1}/\lambda_{\tt seq}) \cdot \mu^{-1}_{\tt seq,1} + (\lambda_{\tt seq,2}/\lambda_{\tt seq}) \cdot \mu^{-1}_{\tt seq,2}$.

Denoted by $\rho_{\tt seq} = \lambda_{\tt seq}/\mu_{\tt seq}$ the traffic intensity of the resulting QS M/M/1 approximating the open QN model for the sequential architecture, and assumed $\rho_{\tt seq} < 1$, phase 6 is conducted symbolically by employing the first group of formulas at the end of Sect. 3.1. The results of the evaluation are reported in Table 8.

6.6 Analysis of the Pipeline Compiler

As far as the evaluation of the pipeline architecture is concerned, after denoting by $\lambda_{\tt pipe,1}$, $\lambda_{\tt pipe,2}$, $\mu_{\tt pipe,l}$, $\mu_{\tt pipe,p}$, $\mu_{\tt pipe,c}$, $\mu_{\tt pipe,o}$, and $\mu_{\tt pipe,g}$ the actual values

Compiler throughput:	$\overline{X}_{\texttt{seq}} = \lambda_{\texttt{seq}}$
Compiler utilization:	$\overline{U}_{\texttt{seq}} = \rho_{\texttt{seq}}$
Mean number of programs in the compiler:	$\overline{N}_{\texttt{seq}} = \rho_{\texttt{seq}}/(1-\rho_{\texttt{seq}})$
Mean compilation time:	$\overline{R}_{\texttt{seq}} = 1/[\mu_{\texttt{seq}} \cdot (1-\rho_{\texttt{seq}})]$

Table 8. Symbolic evaluation for the sequential architecture

for the numeric parameters of `PipeCompSys` that characterize a certain scenario, we note that the application of phase 5 to the QN of Fig. 15 simply boils down to merging the two arrival processes into a single arrival process with arrival rate $\lambda_{\texttt{pipe}} = \lambda_{\texttt{pipe},1} + \lambda_{\texttt{pipe},2}$. The multi-class service processes for the lexer, the parser, the checker, and the generator are trivially converted into single-class service processes with service rates $\mu_{\texttt{pipe},\texttt{l}}$, $\mu_{\texttt{pipe},\texttt{p}}$, $\mu_{\texttt{pipe},\texttt{c}}$, and $\mu_{\texttt{pipe},\texttt{g}}$, respectively, as the two classes of programs have the same service rate in each of the four multi-class service processes.

The resulting open QN model, which is used in phase 6, is product form and is symbolically evaluated by decomposing it into five QSs M/M/1 with the appropriate arrival rates. In particular, at equilibrium the arrival rate for the lexer, the parser, the checker, and the generator is $\lambda_{\texttt{pipe}}$, while the arrival rate for the optimizer is $\lambda_{\texttt{pipe},1}$. As a consequence, the probability that a program leaving the checker enters the optimizer (resp. the generator) is $\lambda_{\texttt{pipe},1}/\lambda_{\texttt{pipe}}$ (resp. $\lambda_{\texttt{pipe},2}/\lambda_{\texttt{pipe}}$). Another consequence is that the traffic intensity for the lexer, the parser, the checker, and the generator is $\rho_{\texttt{pipe},j} = \lambda_{\texttt{pipe}}/\mu_{\texttt{pipe},j}$ where $j \in \{\texttt{l},\texttt{p},\texttt{c},\texttt{g}\}$, while the traffic intensity for the optimizer is $\rho_{\texttt{pipe},\texttt{o}} = \lambda_{\texttt{pipe},1}/\mu_{\texttt{pipe},\texttt{o}}$. Assuming that the QN is stable, which means that each of its service centers is stable, i.e. $\lambda_{\texttt{pipe}} < \min(\mu_{\texttt{pipe},\texttt{l}}, \mu_{\texttt{pipe},\texttt{p}}, \mu_{\texttt{pipe},\texttt{c}}, \mu_{\texttt{pipe},\texttt{o}} \cdot (\lambda_{\texttt{pipe}}/\lambda_{\texttt{pipe},1}), \mu_{\texttt{pipe},\texttt{g}})$, we symbolically derive the four average performance indices both for the various compilation phases and for the overall pipeline compiler, as reported in Table 9.

Phase j throughput:	$\overline{X}_{\texttt{pipe},j} = \lambda_{\texttt{pipe}}$ for $j \in \{\texttt{l},\texttt{p},\texttt{c},\texttt{g}\}$
	$\overline{X}_{\texttt{pipe},\texttt{o}} = \lambda_{\texttt{pipe},1}$
Phase j utilization:	$\overline{U}_{\texttt{pipe},j} = \rho_{\texttt{pipe},j}$
Mean number of programs in phase j:	$\overline{N}_{\texttt{pipe},j} = \rho_{\texttt{pipe},j}/(1-\rho_{\texttt{pipe},j})$
Mean duration of phase j:	$\overline{R}_{\texttt{pipe},j} = 1/[\mu_{pipe,j} \cdot (1-\rho_{\texttt{pipe},j})]$
Compiler throughput:	$\overline{X}_{\texttt{pipe}} = \overline{X}_{\texttt{pipe},\texttt{g}}$
Compiler utilization:	$\overline{U}_{\texttt{pipe}} = 1 - \prod_j (1-\overline{U}_{\texttt{pipe},j})$
Mean number of programs in the compiler:	$\overline{N}_{\texttt{pipe}} = \sum_j \overline{N}_{\texttt{pipe},j}$
Mean compilation time:	$\overline{R}_{\texttt{pipe}} = \frac{\lambda_{\texttt{pipe},1}}{\lambda_{\texttt{pipe}}} \cdot \sum_j \overline{R}_{\texttt{pipe},j} + \frac{\lambda_{\texttt{pipe},2}}{\lambda_{\texttt{pipe}}} \cdot \sum_{j \neq \texttt{o}} \overline{R}_{\texttt{pipe},j}$

Table 9. Symbolic evaluation for the pipeline architecture

6.7 Analysis of the Concurrent Compiler

We denote by $\lambda_{\texttt{conc},1}$, $\lambda_{\texttt{conc},2}$, $\mu_{\texttt{conc},\texttt{l}}$, $\mu_{\texttt{conc},\texttt{p}}$, $\mu_{\texttt{conc},\texttt{c}}$, $\mu_{\texttt{conc},\texttt{o}}$, and $\mu_{\texttt{conc},\texttt{g}}$ the actual values for the numeric parameters of `ConcCompSys` with respect to a certain scenario. The QN model associated with `ConcCompSys`, which is shown in Fig. 17, can easily be transformed into a QS M/M/2 by performing phase 5 as in the case of `SeqCompSys`.

Denoted by $\rho_{\texttt{conc}} = \lambda_{\texttt{conc}}/(2 \cdot \mu_{\texttt{conc}})$ the traffic intensity of the resulting QS M/M/2 approximating the open QN model for the concurrent architecture, and assumed $\rho_{\texttt{conc}} < 1$, phase 6 is conducted symbolically by employing the second group of formulas at the end of Sect. 3.1 with $m = 2$. The results of the evaluation are reported in Table 10.

Compiler throughput:	$\overline{X}_{\texttt{conc}} = \lambda_{\texttt{conc}}$
Compiler utilization:	$\overline{U}_{\texttt{conc}} = 2 \cdot \rho_{\texttt{conc}}/(1 + \rho_{\texttt{conc}})$
Mean number of programs in the compiler:	$\overline{N}_{\texttt{conc}} = 2 \cdot \rho_{\texttt{conc}}/(1 - \rho_{\texttt{conc}}^2)$
Mean compilation time:	$\overline{R}_{\texttt{conc}} = 1/[\mu_{\texttt{conc}} \cdot (1 - \rho_{\texttt{conc}}^2)]$

Table 10. Symbolic evaluation for the concurrent architecture

6.8 Comparison of the Three Architectures

Due to the simplicity of the three architectures, for each of them phase 7 is skipped altogether. So, we can finally compare the sequential architecture, the pipeline architecture, and the concurrent architecture on the basis of the four average performance indices that we have symbolically computed (phase 8). In order to perform a fair comparison, we assume that in every scenario the compilation phases have the same duration for the three architectures, i.e. $\mu_{\texttt{seq},j} = \mu_{\texttt{pipe},j} = \mu_{\texttt{conc},j} \equiv \mu_j$ for all $j \in \{\texttt{l}, \texttt{p}, \texttt{c}, \texttt{o}, \texttt{g}\}$. On the contrary, the three arrival rates $\lambda_{\texttt{seq}}$, $\lambda_{\texttt{pipe}}$, and $\lambda_{\texttt{conc}}$ can freely vary provided that they preserve the frequency of each class of programs, i.e. $\lambda_{\texttt{seq},c}/\lambda_{\texttt{seq}} = \lambda_{\texttt{pipe},c}/\lambda_{\texttt{pipe}} = \lambda_{\texttt{conc},c}/\lambda_{\texttt{conc}} \equiv p_c$ for all $c \in \{1, 2\}$.

We focus on two different scenarios and we concentrate only on the mean number of programs that are compiled per unit of time, as analogous results can be derived for the other three average performance indices. In the first scenario, we assume that the three architectures undergo to a light workload. In this case, the specific architecture does not really matter, as the relations among the three throughputs directly depend on the relations among the three arrival rates: $\overline{X}_{t_1} \, \mathcal{R} \, \overline{X}_{t_2}$ if and only if $\lambda_{t_1} \, \mathcal{R} \, \lambda_{t_2}$, with $t_1, t_2 \in \{\texttt{seq}, \texttt{pipe}, \texttt{conc}\}$ and $\mathcal{R} \in \{<, =, >\}$.

In the second scenario, instead, we assume that the three architectures undergo to a heavy workload. This means that the values of the three arrival rates are such that all the architectures work close to their maximum throughputs, which can be derived from the corresponding stability conditions. In the case of the sequential architecture, $\lambda_{\texttt{seq}}$ is close to $\overline{X}_{\texttt{seq,max}} = \mu_{\texttt{seq}}$, with $\mu_{\texttt{seq}}^{-1} = p_1 \cdot \sum_j \mu_j^{-1} + p_2 \cdot \sum_{j \neq \texttt{o}} \mu_j^{-1}$. In the case of the pipeline architecture, $\lambda_{\texttt{pipe}}$ is close to $\overline{X}_{\texttt{pipe,max}} = \min(\mu_{\texttt{l}}, \mu_{\texttt{p}}, \mu_{\texttt{c}}, \mu_{\texttt{o}}/p_1, \mu_{\texttt{g}})$. In the case of the concurrent architecture, $\lambda_{\texttt{conc}}$ is close to $\overline{X}_{\texttt{conc,max}} = 2 \cdot \mu_{\texttt{conc}}$, with $\mu_{\texttt{conc}}^{-1} = p_1 \cdot \sum_j \mu_j^{-1} + p_2 \cdot \sum_{j \neq \texttt{o}} \mu_j^{-1}$. In this scenario, for an accurate comparison it is worth considering the three following sub-scenarios:

- In the first sub-scenario, the five compilation phases have approximatively the same average duration, i.e. $\mu_{\texttt{l}} \cong \mu_{\texttt{p}} \cong \mu_{\texttt{c}} \cong \mu_{\texttt{o}} \cong \mu_{\texttt{g}} \equiv \mu$. In this case $\overline{X}_{\texttt{seq,max}} \cong (4 + p_1)^{-1} \cdot \mu$, $\overline{X}_{\texttt{pipe,max}} \cong \mu$, and $\overline{X}_{\texttt{conc,max}} \cong 2 \cdot (4 + p_1)^{-1} \cdot \mu$. It follows that:
$$\begin{aligned} \overline{X}_{\texttt{pipe,max}} / \overline{X}_{\texttt{seq,max}} &\cong 4 + p_1 \\ \overline{X}_{\texttt{pipe,max}} / \overline{X}_{\texttt{conc,max}} &\cong 2 + 0.5 \cdot p_1 \\ \overline{X}_{\texttt{conc,max}} / \overline{X}_{\texttt{seq,max}} &\cong 2 \end{aligned}$$
Therefore, in this sub-scenario, the pipeline architecture outperforms — in terms of mean number of programs compiled per unit of time — the sequential architecture (resp. the concurrent architecture) of a factor that ranges between 4 and 5 (resp. between 2 and 2.5) depending on the frequency of the programs of class `1`. In addition, we see that the concurrent architecture outperforms the sequential architecture of a factor 2. We conclude that in this sub-scenario the pipeline architecture is the architecture of choice.
- In the second sub-scenario, there is one compilation phase, say lexical analysis, whose average duration is several orders of magnitude greater than the average duration of the other phases, i.e. $\mu_{\texttt{l}} \ll \mu_j$ for all $j \in \{\texttt{p}, \texttt{c}, \texttt{o}, \texttt{g}\}$. In this case $\overline{X}_{\texttt{seq,max}} \cong \mu_{\texttt{l}}$, $\overline{X}_{\texttt{pipe,max}} = \mu_{\texttt{l}}$, and $\overline{X}_{\texttt{conc,max}} \cong 2 \cdot \mu_{\texttt{l}}$. It follows that:
$$\begin{aligned} \overline{X}_{\texttt{pipe,max}} / \overline{X}_{\texttt{seq,max}} &\cong 1 \\ \overline{X}_{\texttt{conc,max}} / \overline{X}_{\texttt{pipe,max}} &\cong 2 \\ \overline{X}_{\texttt{conc,max}} / \overline{X}_{\texttt{seq,max}} &\cong 2 \end{aligned}$$
We conclude that in this sub-scenario, in which one of the five phases is a bottleneck, splitting the various phases among different components operating in parallel brings no advantage, and the architecture of choice is the concurrent one.
- In the third sub-scenario, the average durations of the five compilation phases range between a minimum value and a maximum value that are several orders of magnitude apart, i.e. $\mu_{\texttt{min}} \leq \mu_j \leq \mu_{\texttt{max}}$ for all $j \in \{\texttt{l}, \texttt{p}, \texttt{c}, \texttt{o}, \texttt{g}\}$ with $\mu_{\texttt{min}} \ll \mu_{\texttt{max}}$. In this case $(4 + p_1)^{-1} \cdot \mu_{\texttt{min}} \leq \overline{X}_{\texttt{seq,max}} \leq (4 + p_1)^{-1} \cdot \mu_{\texttt{max}}$, $\overline{X}_{\texttt{pipe,max}} = \mu_{\texttt{min}}$, and $2 \cdot (4 + p_1)^{-1} \cdot \mu_{\texttt{min}} \leq \overline{X}_{\texttt{conc,max}} \leq 2 \cdot (4 + p_1)^{-1} \cdot \mu_{\texttt{max}}$. It follows that:
$$\begin{aligned} (4 + p_1) \cdot (\mu_{\texttt{min}} / \mu_{\texttt{max}}) &\leq \overline{X}_{\texttt{pipe,max}} / \overline{X}_{\texttt{seq,max}} \leq 4 + p_1 \\ (4 + p_1) \cdot (\mu_{\texttt{min}} / \mu_{\texttt{max}}) \,/\, 2 &\leq \overline{X}_{\texttt{pipe,max}} / \overline{X}_{\texttt{conc,max}} \leq (4 + p_1) \,/\, 2 \\ 2 &\leq \overline{X}_{\texttt{conc,max}} / \overline{X}_{\texttt{seq,max}} \leq 2 \end{aligned}$$
which generalizes the results of the previous two sub-scenarios, showing that the concurrent architecture is always twice as faster as the sequential one,

and that the pipeline architecture is not always better than the other two, as $(4+p_1)\cdot(\mu_{\mathtt{min}}/\mu_{\mathtt{max}})$ and $(4+p_1)\cdot(\mu_{\mathtt{min}}/\mu_{\mathtt{max}})\,/\,2$ can be less than 1 because so is $\mu_{\mathtt{min}}/\mu_{\mathtt{max}}$.

7 Conclusion and Future Perspectives

In this paper we have presented a methodology for the prediction, the improvement, and the comparison of typical average performance indices of alternative architectural designs developed for a software system. The methodology relies on the SPA based ADL called Æmilia, which provides a textual and graphical environment in which architectural descriptions can be developed in an easy and controlled way, and on QNs, which are structured performance models equipped with fast solution algorithms for computing typical average performance measures and allow such performance measures to be interpreted back at the SA description level. The combined use of Æmilia and QNs is made possible by a suitable translation, which can be applied to a reasonably wide class of Æmilia specifications satisfying certain syntax restrictions and has a complexity linear in the number of software components declared in the Æmilia specifications. The methodology and the translation have been illustrated on a scenario-based comparison of a sequential SA, a pipeline SA, and a concurrent SA for a compiler system.

As far as future work is concerned, first we would like to provide an automated support for our methodology. This will be accomplished by implementing the translation of Æmilia specifications into QN models as well as the solution of QN models in the architectural assistant module of the Æmilia-based software tool TwoTowers 3.0 [12]. Second, we would like to integrate our methodology within the software development cycle, both upstream and downstream. On the one hand, we would like to develop a translation from notations used in the software engineering practice, like e.g. UML, to our framework, in order to hide as much as possible all the formal details with which the typical designer may not be familiar. On the other hand, we would like to be able to generate code that is guaranteed to possess the performance requirements proved at the SA level. In this respect, a critical issue to address is taking into account the impact on the software performance of the hardware architecture and operating system on which the software system will be deployed.

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, *"Modelling with Generalized Stochastic Petri Nets"*, John Wiley & Sons, 1995.
2. A. Aldini and M. Bernardo, *"A General Deadlock Detection Approach for Software Architectures"*, to appear in Proc. of the *12th Int. Formal Methods Europe Symp. (FME 2003)*, LNCS, Pisa (Italy), 2003.
3. F. Aquilani, S. Balsamo, and P. Inverardi, *"Performance Analysis at the Software Architectural Design Level"*, in Performance Evaluation 45:205-221, 2001.

4. F. Baccelli, W.A. Massey, and D. Towsley, *"Acyclic Fork-Join Queueing Networks"*, in Journal of the ACM 22:248-260, 1989.
5. C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, *"Automated Performance and Dependability Evaluation Using Model Checking"*, in *Performance Evaluation of Complex Systems: Techniques and Tools*, LNCS 2459:261-289, 2002.
6. S. Balsamo, *"Product Form Queueing Networks"*, in *Performance Evaluation: Origins and Directions*, LNCS 1769:377-401, 2000.
7. S. Balsamo, M. Bernardo, and M. Simeoni, *"Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis"*, in Proc. of the *3rd Int. Workshop on Software and Performance (WOSP 2002)*, ACM Press, pp. 190-202, Roma (Italy), 2002.
8. S. Balsamo, V. De Nitto Personè, and R. Onvural, *"Analysis of Queueing Networks with Blocking"*, Kluwer, 2001.
9. S. Balsamo, L. Donatiello, and N. van Dijk, *"Bounded Performance Analysis of Parallel Processing Systems"*, in IEEE Trans. on Parallel and Distributed Systems 9:1041-1056, 1998.
10. F. Baskett, K.M. Chandy, R.R. Muntz, and G. Palacios, *"Open, Closed, and Mixed Networks of Queues with Different Classes of Customers"*, in Journal of the ACM 22:248-260, 1975.
11. H. Beilner, J. Mäter, and C. Wysocki, *"The Hierarchical Evaluation Tool HIT"*, in Proc. of the *7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 1994)*, LNCS 794, Wien (Austria), 1994.
12. M. Bernardo, *"TwoTowers 3.0: Enhancing Usability"*, to appear in Proc. of the *11th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2003)*, IEEE-CS Press, Orlando (FL), October 2003 (`http://www.sti.uniurb.it/bernardo/twotowers/`).
13. M. Bernardo, *"Symbolic Semantic Rules for Producing Compact STGLA from Value Passing Process Descriptions"*, to appear in ACM Trans. on Computational Logic, 2003.
14. M. Bernardo and M. Bravetti, *"Performance Measure Sensitive Congruences for Markovian Process Algebras"*, in Theoretical Computer Science 290:117-160, 2003.
15. M. Bernardo, P. Ciancarini, and L. Donatiello, *"Architecting Families of Software Systems with Process Algebras"*, in ACM Trans. on Software Engineering and Methodology 11:386-426, 2002.
16. M. Bernardo, L. Donatiello, and P. Ciancarini, *"Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language"*, in *Performance Evaluation of Complex Systems: Techniques and Tools*, LNCS 2459:236-260, 2002.
17. M. Bernardo and F. Franzè, *"Exogenous and Endogenous Extensions of Architectural Types"*, in Proc. of the *5th Int. Conf. on Coordination Models and Languages (COORDINATION 2002)*, LNCS 2315:40-55, York (UK), 2002.
18. M. Bernardo and F. Franzè, *"Architectural Types Revisited: Extensible And/Or Connections"*, in Proc. of the *5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306:113-128, Grenoble (France), 2002.
19. J.P. Buzen, *"Computational Algorithms for Closed Queueing Networks with Exponential Servers"*, in Comm. of the ACM 16:527-531, 1973.
20. K.M. Chandy and C.H. Sauer, *"Computational Algorithms for Product Form Queueing Networks"*, in Comm. of the ACM 23:573-583, 1980.

21. W.-M. Chow, E.A. MacNair, and C.H. Sauer, *"Analysis of Manufactoring Systems by the Research Queueing Package"*, in IBM Journal of Research and Development 29:330-342, 1985.
22. G. Clark, S. Gilmore, J. Hillston, and M. Ribaudo, *"Exploiting Modal Logic to Express Performance Measures"*, in Proc. of the *11th Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation (TOOLS 2000)*, LNCS 1786:247-261, Schaumburg (IL), 2000.
23. E.M. Clarke, O. Grumberg, and D.A. Peled, *"Model Checking"*, MIT Press, 1999.
24. W.R. Cleaveland and O. Sokolsky, *"Equivalence and Preorder Checking for Finite-State Systems"*, in *Handbook of Process Algebra*, Elsevier, pp. 391-424, 2001.
25. A.E. Conway and N.D. Georganas, *"RECAL - A New Efficient Algorithm for the Exact Analysis of Multiple-Chain Closed Queueing Networks"*, Journal of the ACM 33:786-791, 1986.
26. R.G. Franks, A. Hubbard, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia, and C.M. Woodside, *"A Toolset for Performance Engineering and Software Design of Client-Server Systems"*, in Performance Evaluation 24:117-135, 1995.
27. R.G. Franks and C.M. Woodside, *"Performance of Multi-level Client-server Systems with Parallel Service Operations"*, in Proc. of the *1st Int. Workshop on Software and Performance (WOSP 1998)*, ACM Press, pp. 120-130, Santa Fe (NM), 1998.
28. E. Gelenbe, *"Queueing Networks with Negative and Positive Customers"*, in Journal of Applied Probability 28:656-663, 1991.
29. N. Götz, U. Herzog, and M. Rettelbach, *"Multiprocessor and Distributed System Design: The Integration of Functional Specification and Performance Analysis Using Stochastic Process Algebras"*, in Proc. of the *16th Int. Symp. on Computer Performance Modelling, Measurement and Evaluation (PERFORMANCE 1993)*, LNCS 729:121-146, Roma (Italy), 1993.
30. H. Hermanns, *"Interactive Markov Chains"*, LNCS 2428, 2002.
31. J. Hillston, *"A Compositional Approach to Performance Modelling"*, Cambridge University Press, 1996.
32. R.A. Howard, *"Dynamic Probabilistic Systems"*, John Wiley & Sons, 1971.
33. K. Kant, *"Introduction to Computer System Performance Evaluation"*, McGraw-Hill, 1992.
34. L. Kleinrock, *"Queueing Systems"*, Wiley, 1975.
35. S.S. Lavenberg, *"Computer Performance Modeling Handbook"*, Academic Press, 1983.
36. S.S. Lavenberg and C.H. Sauer, *"Approximate Analysis of Queueing Networks"*, in [35], pp. 173-221.
37. E.D. Lazowska, J. Zahorjan, G. Scott Graham, and K.C. Sevcik, *"Quantitative System Performance: Computer System Analysis Using Queueing Network Models"*, Prentice Hall, 1984.
38. R. Milner, *"Communication and Concurrency"*, Prentice Hall, 1989.
39. M.K. Molloy, *"Performance Analysis using Stochastic Petri Nets"*, in IEEE Trans. on Computers 31:913-917, 1982.
40. R. Nelson and A. Tantawi, *"Approximate Analysis of Fork-Join Synchronization in Parallel Queues"*, in IEEE Trans. on Computers 37:739-743, 1988.
41. M.F. Neuts, *"Matrix-Geometric Solutions in Stochastic Models – An Algorithmic Approach"*, John Hopkins University Press, 1981.
42. H.G. Perros, *"Queueing Networks with Blocking"*, Oxford University Press, 1994.

43. D.E. Perry and A.L. Wolf, *"Foundations for the Study of Software Architecture"*, in ACM SIGSOFT Software Engineering Notes 17:40-52, 1992.
44. M. Reiser and S.S. Lavenberg, *"Mean-Value Analysis of Closed Multichain Queueing Networks"*, in Journal of the ACM 27:313-322, 1980.
45. J.A. Rolia and K.C. Sevcik, *"The Method of Layers"*, in IEEE Trans. on Software Engineering 21:682-688, 1995.
46. M. Shaw and D. Garlan, *"Software Architecture: Perspectives on an Emerging Discipline"*, Prentice Hall, 1996.
47. C. Smith, *"Performance Engineering of Software Systems"*, Addison-Wesley, 1990.
48. W.J. Stewart, *"Introduction to the Numerical Solution of Markov Chains"*, Princeton University Press, 1994.
49. Simulog Corp., *"The QNAP2 Reference Manual"*, 1989.
50. K.S. Trivedi, *"Probability and Statistics with Reliability, Queueing, and Computer Science Applications"*, John Wiley & Sons, 2001.
51. P.D. Welch, *"The Statistical Analysis of Simulation Results"*, in [35], pp. 267-329.
52. C.M. Woodside, J.E. Neilson, D.C. Petriu, and S. Majumdar, *"The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-like Distributed Software"*, in IEEE Trans. on Computers 44:20-34, 1995.