On Relationships Between Stochastic Process Algebras with Æmilia and Queueing Network Models
On Relationships Between Stochastic Process Algebras with \textit{\AE}milia and Queueing Network Models

Simonetta Balsamo, Marta Simeoni
Dipartimento di Informatica
Universit\`a \textit{Ca’ Foscari} di Venezia
\{balsamo, simeoni\}@dsi.unive.it

Technical Report CS-2005-10

Abstract
Various models and methods have been proposed and applied for quantitative system performance evaluation, including Queueing Networks (QN) and Stochastic extensions of Process Algebras (SPA), that show different characteristics and advantages. These formalisms have been applied to model and analyze the performance of hardware and software systems at different levels of abstraction, and more recently at the Software Architectures level. In this paper we investigate some relationships between SPA and QN, and more specifically we compare the SPA specifications based on \textit{\AE}milia and a class of QN. We propose an approach to translate a QN model into an \textit{\AE}milia specification in order to take advantage of the model definition based on SPA that allows the combination of functional and performance analysis and compositional, graphical and hierarchical modeling of complex systems. This work extends and reverses a previous comparison of the two formalisms based on the mapping from \textit{\AE}milia specifications to QN. The integration of these two formalisms aims to combine their main advantages as tools for system performance analysis, in order to efficiently describe and analyze both functional and performance properties of system specifications within the same integrated framework.

1 Introduction

Stochastic models have been widely proposed and applied for system performance evaluation in the last decades, including Queueing Networks (QN), Stochastic Process Algebras (SPA) and Stochastic Petri Nets, and each class of models shows different characteristics and advantages. These formalisms have been considered to model and analyze the performance of hardware and software systems at different levels of abstraction, recently also at the Software Architectures level.
QN models have been extensively applied to represent and analyze resource sharing systems, such as production, communication and computer systems and they provide a powerful tool for system performance evaluation and prediction [24, 25, 26, 22, 31]. QN allow representing system resources as a collection of service centers that provide service to a collection of customers that represent the users. Some extensions of QN have been introduced to represent other interesting features of real systems such as synchronization and concurrency constraints [25, 22, 28, 32]. The analysis of QN consists of evaluating a set of global and local performance measures that can be defined at the level of the entire system and of the single service center. Under exponential and independence assumptions the QN can be modeled and analyzed by an associated Markov Chain. Some efficient solution algorithms have been defined to analyze QN models showing product form solution of the steady state probability, and approximate analytical methods can be applied to solve non-product form QN.

SPA is a compositional specification formalism for concurrent and distributed systems that allows combining qualitative and quantitative system analysis [18, 12]. SPA are a stochastic extension of Process Algebras with time and probabilistic elements associated to actions to represent temporal behavior. This combination of process algebras and stochastic process allows modeling and evaluating functional and quantitative temporal behavior. Performance analysis of SPA relies on the solution of the underlying Markov Chain that can be defined under exponential and independence assumptions on the actions. However, the state space explosion limits a direct solution of the MC. Some recent work propose syntactic characterization of the SPA terms or exploit results from QN to define restricted forms of interactions between SPA components in order to identify special classes of SPA that allow a product form solution of the steady state probability, e.g. [17, 19, 29].

In this paper we investigate some relationships between SPA and QN, and more specifically we compare the SPA specifications based on Æmilia and a class of QN. We propose an approach to translate a QN model into an Æmilia specification in order to take advantage of the model definition based on SPA that allows to combine functional and performance analysis and compositional, graphical and hierarchical modeling of complex systems. Æmilia [1, 12] is an Architectural Description Language based on the SPA EMPA [10] for performance modeling and analysis of concurrent and distributed systems. It provides a formal specification language for the compositional, graphical and hierarchical representation of software systems. We consider the class of multiclass QN with various types of service centers and different service disciplines, ranging from Processor Sharing, FCFS, and Infinite Servers to priority scheduling. We provide a translation of the various QN components into Æmilia specification by defining the corresponding Architectural Types that are eventually composed to obtain the specification of the overall QN.

This work extends and reverses a previous comparison of the two formalisms based on the mapping from Æmilia specifications to QN presented in [4, 5]. Here, we consider the other way round mapping, that is, we describe how to translate a queueing network model into an Æmilia specification. Studying these relationships between classes of quantitative performance models of systems should allow a better comprehension of the relative merit of each type of model and its expressiveness and potentiality. On one side the translation of classes of mod-
els from QN to SPA could help in identifying special constraints that allow a product form solution for Æmilia specification. On the other side these relations can be used to define an integrated approach for system performance analysis based on the two types of models. The integration of the two formalisms, QN and SPA, aims to combine their main advantages and tools for system performance analysis, in order to efficiently describe and analyze both functional and performance properties of system specifications within the same integrated framework. The combination and integration of different modeling techniques for performance analysis is an interesting research direction. The analysis of complex systems whose components can be specified and modeled by different formalisms, can take advantage of the possibility to relate and compose different types of performance models, such as QN and a SPA, such as Æmilia. For example, if a subsystem of a given system is defined by QN, it could be translated into SPA in order to compose it with other existing SPA submodel specifications, so allowing an integrated functional and performance analysis of the entire system. Finally note that there is a growing interest in the integration and relation of different types of performance and specification models in current research in the area of software performance modeling [7].

The paper is structured as follows: Section 2 introduces SPA and Æmilia specifications and Section 3 briefly recalls QN models. In Section 4 we describe the mapping from QN components to Æmilia specifications and Section 5 presents the translation an entire QN model. The last section presents concluding remarks.

2 Stochastic Process Algebras and Æmilia

In this section we briefly introduce Stochastic Process Algebras and the syntax, the semantics and the analysis support of Æmilia, an Architectural Description Language (ADL) based on the SPA EMPA [10].

Process Algebras [2] (such as CCS [27], CSP [20]) are a widely known modeling technique for the functional analysis of concurrent systems. These are described as collections of entities, or processes, executing atomic actions, which are used to describe concurrent behaviors, which synchronize in order to communicate. Processes can be composed by means of a set of operators, which include different forms of parallel composition. Process algebras provide a formal model of concurrent systems, which is abstract (the internal behavior of the system components can be disregarded) and compositional (systems can be modeled in terms of the interactions of their subsystems). They provide formal techniques for the verification of system properties, such as equivalences and pre-orders.

Process Algebras can describe systems at different levels of abstraction. Many notions of equivalence or pre-order are defined to study the relationship between different descriptions of the same system. Behavioral equivalences allow one to prove that two different system specifications are equivalent when "uninteresting" details are ignored, while pre-orders are suitable for proving that a low level specification is a satisfactory implementation of a more abstract one.

Stochastic Process Algebra (SPA) are extensions of Process Algebras aiming at the integration of qualitative-functional and quantitative-temporal aspects into a sin-
gle specification technique [18]. Temporal information is added to actions by means of continuous random variables, representing activity durations. Such information enriches the Labeled Transition System (LTS) semantic model, hence making it possible the evaluation of functional properties (e.g. liveness, deadlock), temporal indices (e.g. throughput, waiting times) and combined aspects (e.g. probability of timeout, duration of action sequences) of the modeled systems.

The quantitative analysis of the modeled system can be performed by constructing, out of the enriched LTS, the underlying stochastic process that, under exponential assumptions on the action durations, yields a Markov Chain (MC). A serious limitation of the SPA analysis is the state space explosion of the MC, which often makes the performance analysis unfeasible. Some recent work propose syntactic characterization of the SPA terms or exploit results from QN to define restricted forms of interactions between SPA components in order to identify special classes of SPA that allow a product form solution of the steady state probability, e.g. [17, 19, 29].

2.1 Æmilia

Æmilia [1, 12] is an Architectural Description Language based on the SPA EMPA gr.
More specifically, it is the result of the integration of the two earlier formalisms PADL [11] and EMPA gr [10]. The former is a process-algebra-based ADL, equipped with some architectural checks for the detection of deadlock-related architectural mismatches within families of software systems called architectural types. The SPA EMPA gr allows for both the functional verification and the performance evaluation of concurrent and distributed systems. We briefly recall how this combination of the two formalisms gives rise to the syntax, the semantics, and the analysis support for Æmilia.

A description in Æmilia defines an architectural type, which represents a family of software systems sharing certain constraints on the component observable behavior as well as on the architectural topology. 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.

A sequential EMPA gr defining equation specifies a possibly recursive behavior in the following way:

\[
\text{behavior_id}(\text{formal_param_list}; \text{local_variable_list}) = \text{seq_term}
\]

where a sequential EMPA gr term is written according to the following syntax:

\[
\begin{align*}
\text{seq_term} &::= \text{stop} \\
& | \quad \langle \text{action_type}, \text{action_rate} \rangle . \text{seq_term}_1 \\
& | \quad \text{choice} \{ \text{seq_term}_2 \text{list} \} \\
\text{seq_term}_1 &::= \text{seq_term} \\
& | \quad \text{behavior_id}(\text{actual_parameter_list}) \\
\text{seq_term}_2 &::= \text{seq_term} \\
& | \quad \text{cond}(\text{boolean_guard}) \rightarrow \text{seq_term}
\end{align*}
\]
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>`. `seq_term_1` can execute an action having a certain type and a certain rate and then behaves as specified by `seq_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 { seq_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).

![Diagram of legal attachments]

Figure 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 [27].

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 EMPAGR, which establishes that two non passive actions cannot synchronize, every set of attached interactions must contain at most one interaction whose associated rate is exponential or infinite.

Finally, the behavior variation section of an AET provides some flexibility by allowing some parts of the system behavior to be changed according to some specific modeling- or analysis-related needs. Behavioral variations may be concerned with making some system activities unobservable, preventing some activities from occurring, or renaming some activities.

On the analysis side, Æmilia inherits from EMPAGR standard techniques to assess functional properties as well as performance measures. Among such techniques we
mention model checking [13], equivalence checking [14], Markovian analysis [30] based on rewards [21] as described in [10], and discrete event simulation [25], all of which are available in the Æmilia-based software tool TwoTowers, starting from release 3.0 [9]. 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 [1].

3 Queueing Networks

Queueing Network (QN) models have been widely applied as system performance models to represent and analyze resource sharing systems, such as production, communication and computer systems [24, 25, 26, 22, 31]. A QN model is a collection of interacting service centers representing system resources and a set of customers representing the users sharing the resources. Its informal representation is a direct graph whose nodes are service centers and edges represent the behavior of customers’ service requests. The analysis of QN consists of evaluating a set of global and local performance measures that can be defined at the level of the entire system and of the single service center.

The popularity of QN models for system performance evaluation is due to the relative high accuracy in performance results and the efficiency in model analysis and evaluation. Informally, the creation of a QN model can be split into three steps: definition, that includes the definition of service centers, their number, class of customers and topology; parameterization, to define model parameters, e.g., arrival processes, service rate and number of customers; evaluation, to obtain a quantitative description of the modeled system, by computing a set of figures of merit or performance indices such as resource utilization, system throughput and customer response time. These indices can be local to a resource or global to the whole system.

Under exponential and independence assumptions the QN can be modeled and analyzed by an underlying Markov Chain, whose direct solution is limited by its state space explosion. In this setting the class of product-form networks plays an important role, since they can be analyzed by efficient solution algorithms to evaluate average performance indices and steady state probability. Specifically, algorithms such as convolution and Mean Value Analysis have a computational complexity polynomial in the number of QN components. These algorithms, on which most approximated analytical methods are based, have been widely applied for performance modeling and analysis [25, 26, 22]. The most famous result concerning product form QN is known as BCMP theorem [8] and defines a class of QN with product form solution for open, closed or mixed models with multiple classes of customers and various service disciplines and service time distributions. Different types of customer in the QN can model different behaviors. This allows representing various types of external arrival process, different service demands and different types of network routing. A chain gathers the customers of the same type, and consists of a set of classes that represent different phases of processing in the system for a given type of customer. Classes are partitioned on 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. Each chain can be open or closed depending on whether external arrivals and departures are allowed. A mixed QN has both open and closed chains. Multiclass models can be used for a more precise representation of system behavior and to obtain more detailed performance indices.

Product form solution of QN holds under special assumptions, and is related to some properties of the QN defined on the underlying Markov Chain. Important properties are quasi-reversibility and partial balance. Informally, quasi-reversibility of a service center states that the current state, the past departures and the future arrivals are mutually independent [24, 23]. Examples of quasi-reversible queues are:

I Multiclass service center with First Come First Served (FCFS) queueing discipline and exponential service time distribution, identical for each customer class.

II Multiclass service center with Processor Sharing (PS) scheduling and arbitrary phase type service time distribution, i.e., formed by a network of exponential stages [24, 23].

III Multiclass service center with infinite number of servers, that is with IS scheduling and arbitrary phase type service time distribution.

IV Multiclass service center with Last Come First Served with preemption (LCFS-Pr) scheduling and arbitrary phase type service time distribution.

Other examples of quasi-reversible queues are defined in terms of the particular class of symmetric queueing discipline [23] which include LCFS-Pr and PS. However, this is only a sufficient condition for product form solution.

Various extensions of the class of BCMP product form QN have been derived, under various conditions and additional constraints on the QN [3]. Examples of product form QN have been proved for cases of state dependent routing, finite capacity queues with some blocking type, different service disciplines, batch arrivals and batch services also related to discrete time QN, and the special class of G-networks with positive and negative customers used to represent special system behaviors [16].

Moreover extensions of classical QN models, namely Extended Queueing Network (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 and population constraints and simultaneous resource possession. EQN in general do not show product form solution and can be analyzed by approximate solution techniques [25, 26, 22, 6]. Another example of QN extension is the Layered Queuing Network (LQN), which allows the modeling of client-server communication patterns in concurrent, and/or distributed software systems [28, 32]. LQN models can be solved by analytic approximation methods based on methods for EQN with simultaneous resource possession and by Mean Value Analysis.

In this paper we consider multiclass QN with various type service centers, including, but not limited to product form QN. We consider QN where service centers have unlimited buffer capacity and various types of service disciplines, e.g., FCFS, PS, IS, priority with and without preemption.
4 Specifying QN components with Æmilia: a case study

The main goal of this paper is to provide a component-wise translation of QN models into Æmilia specifications. We start by illustrating in this section how the different components of a QN model can be represented with Æmilia. To this aim, we introduce a simple QN model of a computer system as running example, and translate its main components into a corresponding and equivalent Æmilia specification. In this context a QN component and its corresponding Æmilia specification are equivalent in terms of isomorphism of their underlying Markov processes. The equivalence between the QN components considered in this section and the corresponding Æmilia specifications is intuitively clear, although we do not provide a formal proof.

Consider a simple interactive computer system depicted in Figure 2, composed by a set of terminals, one server and a set of input/output devices, connected through a local area network.

![Figure 2: A computer system example](image)

We assume two types of users that access the system as normal users or super users, respectively. The latter have a higher priority in accessing the server and the input/output devices. We assume moreover that users can freely change their role.

The system can be modeled with the open multiclass QN model depicted in Figure 3 where new users accessing the system are represented with a source of arrivals, and the activities of terminals, server and input/output devices are represented by service centers. The two types of users (normal and super users) are represented by a unique chain with two different types of customers whose routing is illustrated in Figure 3 by solid and dashed lines, respectively. We assume a finite number of terminals that external users can access. An external user that finds all the terminals busy at arrival times is lost. A user that enters the system requires services to the computer systems components modeled by the service centers of the QN and eventually leaves the system.
so allowing a possible entry of incoming users.

Figure 3: QN model of the computer system example

We shall now translate the above system components into the corresponding Æmilia architectural types (AT). More precisely, we consider some suitable QN modeling alternatives for each service center of the QN as well as for the arrival process and customer departures, and propose their mapping into Æmilia. Then, in section 5 we will define the Æmilia specification of the whole QN by exploiting the hierarchical composition of Æmilia architectural types and by gluing together the various components.

4.1 Modeling the arrivals

We model the arrival of new users to the system of Figure 3 by a multiclass arrival process, that is, as an exponential generator of customers belonging to the two considered classes of users. More generally, let us consider a QN with \( n \) types of customers and with Poisson arrivals. Let \( \lambda \) be the overall arrival rate to the QN and \( r_p_i, 1 \leq i \leq n \), be the probability that an external arriving customer joins class \( i \). Then, the arrival process of the QN can be translated into the architectural type (AT) shown in Figure 4.

The architectural type Multiclass Arrival has parameters \( \lambda \) and \( r_p_i \), that is the arrival rate and the routing probabilities, respectively. Each customer is first generated with exponential interarrival time \( \lambda \) and then assigned to one of the \( n \) classes (depending on the result of the choice operation) and correspondingly delivered by one of the output interactions \( \text{deliver}_1, \ldots, \text{deliver}_n \).

For the case of the model in Figure 3, this Æmilia AT applies for two classes of users which can then enter in one of the six service centers, that is \( n = 12 \). Then, the resulting AT has parameters \( \lambda \) and the routing probabilities \( r_p_1, \ldots, r_p_{12} \) where only \( r_p_1 \) and \( r_p_7 \) are greater than zero. It consists of twelve choice alternatives and twelve output interactions \( \text{deliver}_1, \ldots, \text{deliver}_{12} \), which will be later synchronized with the service center representing the system terminals.

4.2 Modeling the Service Centers

The QN model of Figure 3 is composed by three types of service centers representing the activities of terminals, server and input/output devices, respectively. In this sec-
ARCHI_TYPE Multiclass_Arrival(rate lambda,
    weight rpl,..., rpn)

ARCHI_ELEM_TYPES

ELEM_TYPE Arrival_Type(rate lambda,
    weight rpl,..., rpn)

BEHAVIOR
Arrivals(void; void) =
    <arrive, exp(lambda)>.Arrivals'()
Arrivals'(void;void) =
    choice {
        <choose_1,inf(1,rp1)>.<deliver_1,inf>.Arrivals(),
        ...
        <choose_n,inf(1,rpn)>.<deliver_n,inf>.Arrivals()
    }

INPUT_INTERACTIONS
UNI arrive
OUTPUT_INTERACTIONS
UNI deliver_1, ..., deliver_n;

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES
A : Arrival_Type(lambda, rpl, ..., rpn);

ARCHI_INTERACTIONS
A.arrive,
A.deliver_1, ..., A.deliver_n

ARCHI_ATTACHMENTS
void

END

---

Figure 4: Multiclass arrival process

---

4.2.1 Modeling the terminals

The first service center we consider represents terminals as resources required by the users to ask for server and input/output services. In order to access the system, incoming users ask for an available terminal and, once connected, send its requests to the server. The set of terminals can be modeled by an infinite server (IS) service center, because we assume to have one customer for each terminal. Each connected user sends a request to the system and waits for a response before proceeding in sending further requests, so there is no queue at the terminals’ node. This IS queue can be also modeled as a load dependent single server service center with service rate proportional to the number of customers in the node. Therefore we define the Æmilia AT for a multi-class infinite server service center for \( n \) classes of customers, with parameters the set of \( \mu_i \), \( 1 \leq i \leq n \), as depicted in Figure 5. Each arriving customer of class \( i \) is represented by an input interaction \( arrive_i \) and is immediately served with the
ARCHI_TYPE SC_InfiniteServers(rate μ_1, ..., μ_n)

ARCHI_ELEM_TYPES

ELEM_TYPE Server_Type(rate μ_1, ..., μ_n)

BEHAVIOR

Server(integer cn_1 :=0, ... cn_n :=0; void) =
   choice {
      <arrive_1, _> . Server(cn_1 +1,...,cn_n),
      ...
      <arrive_n, _> . Server(cn_1,...,cn_n +1),
      cond(cn_1 > 0) ->
      <serve, exp(μ_1 * cn_1)> ,
      <leave_1,inf>.Server(cn_1 -1,...,cn_n)
      ...
      cond(cn_n > 0) ->
      <serve, exp(μ_n * cn_n)> ,
      <leave_n,inf>.Server(cn_1, ...,cn_n -1)
   }

INPUT_INTERACTIONS
UNI arrive_1,...,arrive_n

OUTPUT_INTERACTIONS
UNI leave_1,...,leave_n

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES
S : Server_Type(μ_1, ..., μ_n)

ARCHI_INTERACTIONS
S.arrive_1,..., S.arrive_n
S.leave_1,..., S.leave_n

ARCHI_ATTACHMENTS
void

END

Figure 5: Infinite Server service center

load dependent exponential service rate corresponding to its class and then delivered by one of the output interactions leave_1,...,leave_n.

For the example of Figure 3 we have only two classes of customers, that is \( n = 2 \). Hence, we have two input interactions arrive_1, arrive_2, two service rates \( μ_1,μ_2 \), and two output interactions leave_1 and leave_2. The input and output interactions will be later synchronized with the arrival process and the service center representing the system server, respectively.

4.2.2 Modeling the Server

The server of the example in Figure 3 can be modeled by a service center with an appropriate service discipline. Two examples of suitable scheduling disciplines are the processor sharing (PS) and first come first served (FCFS). Moreover, the service rate of the service center can be load dependent. We can model the PS scheduling by allowing all the waiting customers to simultaneously receive service with equal share
of the service rate. This is simply represented by assigning to each customer a load dependent service rate inversely proportional to the total number of customers in the node. The Æmilia AT of a PS service center with \( n \) classes of customers and service rate \( \mu_i, 1 \leq i \leq n \) is shown in Figure 6. The class \( i \) input interaction \( \text{arrive}_i \), leads to the choice of the appropriate service rate and the customer exit the node through the output interaction \( \text{leave}_i \).

Figure 6: Processor Sharing service center

For the special case of the example, we have \( n = 2 \), that is, there are just two input interactions \( \text{arrive}_1, \text{arrive}_2 \), two corresponding service rates \( \mu_1, \mu_2 \), and two output interactions \( \text{leave}_1 \) and \( \text{leave}_2 \).

Note that the general infinite server and processor sharing service centers differ only for the applied service rate: the availability of infinite servers is simulated by assigning the same service rate to each customer, while the equal sharing of the server time is simulated by assigning a slice of service rate inversely proportional to the number of current customers.

In order to represent a service center as a multiclass state-dependent service rate
with an unbounded FCFS buffer, we consider the simple case of the service rate proportional to the number of customers currently in the service center, namely in the queue and server. The Æmilia term modeling a single server multiclass service center with state-dependent service rate $\mu(n)$ when $n$ is the number of customers in the node, is defined in Figure 7.

The AT is composed by two AETs representing the buffer and the server, respectively. The Buffer Type AET in this definition represents a multiclass unlimited queue with a FCFS discipline, as well as any abstract discipline independent of the service time, e.g., LIFO, RAND, or disciplines with priority independent of service time, and without preemption. Note that in case simultaneous requests from different classes of users the choice operator realizes a nondeterministic selection of the request to be served. Moreover, note that the classes do not distinguish the priority levels: they represent classes of users and not priority classes.

The Server Type AET evaluates the internal state of the service center, i.e., the number of customers in the node, and produce services with proportional rates. The evaluation of the state of the queue is realized with an explicit communication between the buffer and server terms.

The AT of Figure 7 adapted to the running example leads to consider just two classes of customers for both the buffer and the server terms.

### 4.2.3 Modeling the Input/output devices

Each of the input/output devices of the system in Figure 3 is modeled as a multiclass service center. We consider two different queueing disciplines: FCFS and abstract priorities without preemption.

A multiclass service center with unbounded FCFS buffer is represented with Æmilia AT as depicted in Figure 8.

In the second case, that is a multiclass service center where the queueing discipline is based on abstract priorities without preemption, the corresponding Æmilia AT is depicted in Figure 9.

Note that we consider a distinct priority assigned to each class of users. More precisely, each class of users identifies also the corresponding priority level. The Server Type AET selects the customer to be served according to its class and priority. For the example of Figure 3 the SC_LoadDependent AT applies for two classes of users and, correspondingly, two classes of priorities.

### 4.3 Modeling the exit process

In the example of Figure 3 users leave the system by ending their terminal session. This process is modeled in the QN model with a sink process, which can be represented with Æmilia as shown in Figure 10.

Note that the Sink AET consists just of a passive action, which is an input OR interaction, meaning that various exiting requests can be represented by a single sink process. Hence, the AT defined in Figure 10 represents the general Æmilia exit process that can be used by any QN model specification by suitably connecting it to the other components.
ARCHI_TYPE SC_LoadDependent(rate \mu_1,\ldots, \mu_n)

ARCHI_ELEM_TYPES

ELEM_TYPE Buffer_Type()

BEHAVIOR
Buffer(integer h_1 := 0,\ldots, h_n := 0; void) =
choice {
  <get_1,>_ . Buffer(h_1+1,\ldots, h_n),
  \ldots
  <get_n,>_ . Buffer(h_1,\ldots, h_n+1),
  \text{cond}(h_1 > 0) \rightarrow
  <put_1,>_ . Buffer(h_1 -1,\ldots, h_n),
  \ldots
  \text{cond}(h_n > 0) \rightarrow
  <put_n,>_ . Buffer(h_1,\ldots, h_n-1),
  \langle \text{buffer state}(h_1 + \ldots + h_n), \text{inf} \rangle .
  Buffer(h_1,\ldots, h_n)
}

INPUT_INTERACTIONS
UNI get_1,\ldots, get_n;
OUTPUT_INTERACTIONS
UNI put_1,\ldots, put_n;
buffer_state

ELEM_TYPE Server_Type(rate \mu_1,\ldots, \mu_n)

BEHAVIOR
Server(void; void) =
choice {
  <select_1, \text{inf}(1,1)> . Server'_1(),
  \ldots
  <select_n, \text{inf}(1,1)> . Server'_n()
}

Server'_i(void; local integer cn) =
\langle \text{read buffer state}(cn), _ \rangle .
\langle \text{serve}_i, \text{exp}(\mu_i) \times (cn+1)\rangle,
\langle \text{leave}_i, \text{inf} \rangle . Server()

INPUT_INTERACTIONS
UNI select_1,\ldots, select_n, read_buffer_state
OUTPUT_INTERACTIONS
UNI leave_1,\ldots, leave_n

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES
B: Buffer_Type();
S: Server_Type(\mu_1,\ldots, \mu_n)

ARCHI_INTERACTIONS
B.get_1,\ldots, B.get_n,
S.leave_1,\ldots, S.leave_n

ARCHI_ATTACHMENTS
\text{FOR ALL } i \in 1..n \text{ FROM } B.put_i \text{ TO } S.select_i;
\text{FROM } B.buffer_state \text{ TO } S.read_buffer_state

END

Figure 7: Service center with load dependent service rate
ARCHI_TYPE SC_MMI(rate mu_1, ..., mu_n)

ARCHI_ELEM_TYPES

ELEM_TYPE Buffer_Type()
BEHAVIOR
Buffer(integer h_1,...,h_n := 0; void) =
choice {
    <get_1, _> . Buffer(h_1+1, ..., h_n),
    ...
    <get_n, _> . Buffer(h_1, ..., h_n+1),
    cond(h_1 > 0) ->
    <put_1, _> . Buffer(h_1 -1,...,h_n),
    ...
    cond(h_n > 0) ->
    <put_n, _> . Buffer(h_1,...,h_n-1),
}
INPUT_INTERACTIONS
UNI get_1,..., get_n;
OUTPUT_INTERACTIONS
UNI put_1,..., put_n;

ELEM_TYPE Server_Type(rate mu_1,..., mu_n)
BEHAVIOR
Server(void; void) =
choice {
    <select_1, inf(1,1)> . Server'_1(),
    ...
    <select_n, inf(1,1)> . Server'_n()
}
Server'_i(void; local integer cn) =
<serve_i, exp(mu_i)>.<leave_i, inf>.Server()
INPUT_INTERACTIONS
UNI select_1, ..., select_n,
OUTPUT_INTERACTIONS
UNI leave_1, ..., leave_n

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES
    B : Buffer_Type();
    S: Server_Type(mu_1,..., mu_n)

ARCHI_INTERACTIONS
    B.get_1, ..., B.get_n,
    S.leave_1, ..., S.leave_n

ARCHI_ATTACHMENTS
    FOR_ALL i IN 1..n FROM B.put_i TO S.select_i;
END

Figure 8: Service center with FCFS
ARCHI_TYPE SC_AbstractPriorities(rate \( \mu_1, \ldots, \mu_n \))

ARCHI_ELEM_TYPES

ELEM_TYPE Buffer_Type()

BEHAVIOR
Buffer(integer \( h_1, \ldots, h_n := 0 \); void) =
choice {
  \(<\text{get}_1, _> . \text{Buffer}(h_1 + 1, \ldots, h_n), \ldots\>
  \,<\text{get}_n, _> . \text{Buffer}(h_1, \ldots, h_n +1),\>
  \,\text{cond}(h_1 > 0) \to \,<\text{put}_1,_> . \text{Buffer}(h_1 -1,\ldots, h_n), \\
  \,\text{\ldots}\>
  \,<\text{put}_n,_> . \text{Buffer}(h_1,\ldots, h_n -1), 
}

INPUT_INTERACTIONS
\text{UNI get}_1, \ldots, \text{get}_n;

OUTPUT_INTERACTIONS
\text{UNI put}_1, \ldots, \text{put}_n;

ELEM_TYPE Server_Type(rate \( \mu_1, \ldots, \mu_n \))

BEHAVIOR
Server(void; void) =
choice {
  \,<\text{select}_1, \text{inf}(1,1)> . \text{Server'}_1(), \\
  \,<\text{select}_n, \text{inf}(n,1)> . \text{Server'}_n()
}
\text{Server'}_i(void; void) =
<\text{serve}_i, \text{exp}(\mu_i)>.<\text{leave}_i, \text{inf}>.\text{Server()}

INPUT_INTERACTIONS
\text{UNI select}_1, \ldots, \text{select}_n

OUTPUT_INTERACTIONS
\text{UNI leave}_1, \ldots, \text{leave}_n

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES
B : Buffer_Type();
S : Server_Type(\( \mu_1, \ldots, \mu_n \))

ARCHI_INTERACTIONS
B.get_1, \ldots, B.get_n,
S.leave_1, \ldots, S.leave_n

ARCHI_ATTACHMENTS
FOR_ALL i IN 1..n
  FROM B.put_i TO S.select_i;
END

Figure 9: Service center with priorities
4.4 The routing component

A further component needed to complete the QN model of Figure 3 is the routing matrix of the whole network. We recall that each chain of a QN model requires its own routing matrix, whose cardinality depends on the number of service centers of the QN and the number of classes of the chain. The routing matrix Æmilia AT is defined in Figure 11 for an open chain QN with \( M \) classes of customers where node 0 denotes the external arrivals and departures. In the QN definition we assume that the classes of customers are enumerated.

The Æmilia AT shown in Figure 11 completely defines a general routing matrix for \( M \) classes of users. The number of parameters and, consequently, of choice alternatives of an actual Æmilia term depends on the specific case. For the example represented in Figure 3, the routing matrix has \( M = 12 \) cardinality, since there are 6 service centers, each with two classes of customers belonging to the same chain. The enumeration of the 12 classes is shown in Figure 3.

4.5 Other QN components

We define now the Æmilia specification of other QN components, not directly related to the running example. In particular, we consider a state dependent arrival process and a service center with single server queueing discipline having abstract priorities and preemption.

Arrivals are state-dependent when the arrival rate is proportional to the number of customers of the target service center, that is, the number of customers currently present in the buffer and server. Hence, a state-dependent arrival architectural type
ARCHI_TYPE Router(weight rp01, ..., rp0M, 
   rp11,...,rpMM, rp00,...,rpM0)

ARCHI_ELEM_TYPES

ELEM_TYPE Router_Type(weight rp01, ..., rp0M, 
   rp11,...,rpMM, rp00,...,rpM0)

BEHAVIOR

Router(void; void) =
   choice {
      <in_0, _>.Router_0{},
      <in_1, _>.Router_1{},
      ...
      <in_M, _>.Router_M{}
   }

Router_i(void; void) =
   choice {
      <out_i0, inf(1, rpi0)>.Router(),
      <out_i1, inf(1, rpi1)>.Router(),
      ...
      <out_iM, inf(1, rpiM)>.Router()
   }

INPUT_INTERACTIONS

UNI in_0, ... in_M

OUTPUT_INTERACTIONS

UNI out_00, ... out_MM

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES

R : Router_Type(rp01, ..., rp0M, 
   rp11,...,rpMM, rp00,...,rpM0)

ARCHI_INTERACTIONS

R.in_0, ... R.in_M
R.out_00, ... R.out_MM

ARCHI_ATTACHMENTS

void

END

Figure 11: General router
ARCHI_TYPE State_dependent_Arrivals(rate lambda, 
weight rp1, ..., rpn)

ARCHI_ELEM_TYPES

ELEM_TYPE Arrival_Type(rate lambda, 
weight rp1, ..., rpn)

BEHAVIOR

Arrivals(local integer bnum, snum; void) =
<read_buffer_state?(bnum), _> .
<read_server_state?(snum), _> .
<arrive, exp(lambda * (bnum+snum)).Arrivals'()>

Arrivals'(void;void) =
choice {
<choose_1,inf(1,rp1)>.
<deliver_1,inf>.Arrivals(),
...
<choose_n,inf(1,rpn)>.
<deliver_n,inf>.Arrivals()}

INPUT_INTERACTIONS

UNI read_buffer_state, 
read_server_state

OUTPUT_INTERACTIONS

UNI deliver_1, ..., deliver_n;

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES

A : Arrival_Type(lambda, rpl, ..., rpn);

ARCHI_INTERACTIONS

A.arrive, A.read_buffer_state, 
A.read_server_state, 
A.deliver_1, ..., A.deliver_n

ARCHI_ATTACHMENTS

void

END

Figure 12: State-dependent arrival process

needs to receive information concerning the state of the connected service center. The Æmilia representation of a state-dependent arrival process is depicted in Figure 12. The \( n \) classes of users have intraclass routing probabilities \( rp_1, \ldots, rpn \). Each customer is first generated with interarrival time proportional to \( \lambda \) times the total number of customers in the target service center, and then assigned to one of the \( n \) classes and delivered by one of the output interactions \( deliver_1, \ldots, deliver_n \).

The Æmilia AT \( SCAbstractPriorities \) depicted in Figure 9 represents a service center with abstract priorities without preemption. In order to represent the service preemption due to the arrival of a request with higher priority we propose the Æmilia AT defined in Figure 13. The AT is provided with an unbounded FCFS buffer, and each class of users has an associated priority level: the lower is the level and the higher is the associated priority. More precisely, the \( i \)-th class of users has priority \( i \) and the service of a request of level \( i \) can be interrupted by any request having higher priority, that is
by any request of level \(1, \ldots, i - 1\). This is accomplished by modeling the service of requests of class \(i\) with a choice operation between requests of class \(i\) as well as of classes \(1, \ldots, i - 1\).

Finally, we consider the case of having more than one server in a service center, say \(m > 1\) servers, all with the same service rate. Figure 14 shows the specification of a \(M/M/m\) service center. All the cases of single server service centers we have considered so far can be easily extended by considering more than one server.

The AT in Figure 14 is composed by the Buffer\_Type AET representing an unbounded FCFS buffer and by the Server\_Type AET, which first drives (in a nondeterministic way) the \(n\) classes of requests to one of the \(m\) servers through the term Server, and then serves each request with the corresponding Server\_ij term.

5 Translating QN models into \(\textit{Æ}milia\) specifications

In this section we formalize the translation of QN models into \(\textit{Æ}milia\) specifications. The type of QN we consider are open and closed multiclass networks with arbitrary topology, exponential arrival and service distribution and unbounded buffers. An easy extension of the framework could allow for general arrival and service distributions, approximated with phase-type ones whenever necessary. In fact, we recall that phase-type distributions can be modeled in \(\textit{Æ}milia\) by a suitable interplay of exponentially timed actions and immediate actions (see [5] for a complete description).

We denote with \(\textit{Æ}milia\text{Spec}\) the space of \(\textit{Æ}milia\) specifications, that is, the space of all possible \(\textit{Æ}milia\) architectural types. The idea is to translate QN components such as arrival processes, service centers and routing matrices, into their corresponding \(\textit{Æ}milia\) architectural types by means of three functions that can be easily turned into effective algorithms and programs.

We start by defining how the input QN models need to be specified, starting from the arrival processes and service centers.

The types of arrival processes we consider are the following ones:

- **SA**: simple arrivals with exponential interarrival time (see Figure 4);
- **SDA**: state-dependent arrivals, with exponential interarrival time (see Figure 12).

They can be fully described by just their type and parameters. Such information is in fact sufficient to generate the syntax of their corresponding \(\textit{Æ}milia\) AT.

**Definition 5.1** An arrival process is defined by a tuple \(a = (\text{type, rate, rp, nc})\) where \(\text{type} \in \{SA, SDA\}\), \(\text{rate}\) is the interarrival time, \(\text{rp}\) are the routing probabilities, and \(\text{nc}\) is the number of user classes.

We denote with \(\mathcal{A}\) the set of available arrival processes.

For example, the arrival process of the QN model depicted in Figure 3 is described by the tuple \(\{(SA, \lambda, \{rp_1, rp_2\}, 2)\}\), namely it is a multiclass simple arrival with
ARCHI_TYPE SC_Preemption(rate mu_1, ..., mu_n)

ARCHI_ELEM_TYPES

ELEM_TYPE Buffer_Type()
BEHAVIOR
Buffer(integer h_1 := 0; ..., h_n := 0; void) =
    choice {
        <get_i, >.Buffer(h_1 + 1; ..., h_n),
        ...
        <get_n, >.Buffer(h_1, ..., h_n + 1),
        cond(h_1 > 0) ->
            <put_i, >.Buffer(h_1 - 1; ..., h_n),
        ...
        cond(h_n > 0) ->
            <put_n, >.Buffer(h_1; ..., h_n - 1),
    }
INPUT_INTERACTIONS
OR get_1; ..., get_n;
OUTPUT_INTERACTIONS
UNI put_1; ..., put_n;

ELEM_TYPE Server_Type(rate mu_1; ..., mu_n)
BEHAVIOR
Server(void; void) =
    choice {
        <select_1, inf(1,1)>.Server'_1(),
        ...
        <select_n, inf(n,1)>.Server'_n()
    }
Server'_i(void; void) =
    choice {
        <serve_i, exp(mu_i)>.<leave_i, inf>.Server(),
        <select_1, inf(1,1)>.<enqueue_i, inf(i,1)>.Server'_1(),
        ...
        <select_i-1, inf(i-1,1)>.<enqueue_i, inf(i,1)>.Server'_i-1(),
    }
INPUT_INTERACTIONS
UNI select_1; ..., select_n
OUTPUT_INTERACTIONS
UNI leave_1; ..., leave_n
enqueue_i

ARCHI_TOPOLOGY

ARCHI_ELEM_INSTANCES
B: Buffer_Type();
S: Server_Type(mu_1; ..., mu_n)

ARCHI_INTERACTIONS
B.get_1; ..., B.get_n,
S.leave_1; ..., S.leave_n

ARCHI_ATTACHMENTS
FOR ALL i IN 1..n
    FROM B.put_i TO S.select_i;
    FROM S.enqueue_i TO B.get_i;
END

Figure 13: Abstract priorities with preemption
ARCI_TYPE SC_MMm(rate \mu_1, \ldots, \mu_n)

ARCI_ELEM_TYPES

ELEM_TYPE Buffer_Type()
BEHAVIOR
Buffer(integer h_1 := 0, \ldots, h_n := 0; void) =
choice {
  \langle \text{get}_1, \_ \rangle . Buffer(h_1+1, \ldots, h_n),
  \ldots
  \langle \text{get}_n, \_ \rangle . Buffer(h_1, \ldots, h_n+1),
  \text{cond}(h_1 > 0) ->
  \langle \text{put}_1, \_ \rangle . Buffer(h_1 -1, \ldots, h_n),
  \ldots
  \text{cond}(h_n > 0) ->
  \langle \text{put}_n, \_ \rangle . Buffer(h_1, \ldots, h_n-1),
}

INPUT_INTERACTIONS
UNI \text{get}_1, \ldots, \text{get}_n;
OUTPUT_INTERACTIONS
UNI \text{put}_1, \ldots, \text{put}_n;

ELEM_TYPE Server_Type(rate \mu_1, \ldots, \mu_n)
BEHAVIOR
Server(void; void) =
choice {
  \langle \text{select}_1, \inf(1,1) \rangle . Server_1(),
  \ldots
  \langle \text{select}_n, \inf(1,1) \rangle . Server_m(),
  \ldots
  \langle \text{serve}, \exp(\mu_j) \rangle . \langle \text{leave}_i, \inf \rangle . Server()}

INPUT_INTERACTIONS
UNI \text{select}_1, \ldots, \text{select}_n,
OUTPUT_INTERACTIONS
UNI \text{leave}_1, \ldots, \text{leave}_n

ARCI_TOPOLOGY

ARCI_ELEM_INSTANCES
B : Buffer_Type();
\text{FOR ALL} j \text{ IN} 1...m
S[j] : Server_Type(\mu_1, \ldots, \mu_n)

ARCI_INTERACTIONS
B.\text{get}_1, \ldots, B.\text{get}_n,
S.\text{leave}_1, \ldots, S.\text{leave}_n

ARCI_ATTACHMENTS
\text{FOR ALL} i \text{ IN} 1...n
\text{FOR ALL} j \text{ IN} 1...m
FROM B.\text{put}_i \text{ TO} S[j].\text{select}_i;

BEHAV_VARIATIONS

BEHAV_RENAMINGS
\text{FOR ALL} j \text{ IN} 1...m
\text{RENAME} S[j].\text{serve} \text{ AS} serve

END

Figure 14: M/M/m service center

23
interarrival time $\lambda$ and having two classes of users with routing probabilities $rp_1$ and $rp_2$.

Then, we consider the following types of service centers, each one dealing with $n$ classes of users and with servers with exponential service rate:

- **$MM1$**: single server with an unbounded FCFS buffer (see Figure 8);
- **$MMm$**: $m$ servers with an unbounded FCFS buffer (see Figure 14);
- **$LD$**: single server with load-dependent service rate and an unbounded FIFO buffer (see Figure 7);
- **$AP$**: single server with abstract priorities and an unbounded FCFS buffer (see Figure 9);
- **$APP$**: single server with abstract priorities and preemption, provided with an unbounded FCFS buffer (see Figure 13);
- **$IS$**: infinite servers (see Figure 5);
- **$PS$**: processor sharing (see Figure 6).

Besides the above listed service centers other cases can be easily added to the available Æmilia specifications, such as for example $LD$, $AP$, $APP$, $AD$ with multiple servers.

**Definition 5.2** A service center $cs$ is defined by a tuple $cs = (type, rate, ns, nc)$ where $type \in \{MM1, MMm, PS, IS, AP, AAP, LD\}$, $rate$ is the set of service rates, $nc$ is the number of servers, and $nc$ is the number of user classes at the service center.

We denote with $C$ the set of all the available service centers.

For example, the service center modeling the terminals in the QN depicted in Figure 3 is described by $\{(IS, \{\mu_1, \mu_2\}, -, 2)\}$, namely it is an infinite servers service center with two classes of users and associated service rate $\mu_1$ and $\mu_2$, respectively.

Finally, a complete QN model can be represented with a tuple summarizing all the information necessary to generate a corresponding Æmilia specification.

**Definition 5.3** A queueing network model is specified by a tuple $Q = (\Lambda, C, R, P)$ where $\Lambda \subseteq A$ is a set of arrival processes; $C \subseteq C$ is a set of service centers; $R$ is the number of user classes; and $P$ is a set of routing matrices, one for each chain of the QN model.

For example, the QN model depicted in Figure 3 is described as follows:

- $\Lambda = \{(SA, \lambda, \{rp_1, rp_2\}, 2)\}$;
- $C = \{(IS, \{\mu_1, \mu_2\}, 2), (PS, \{\mu_1, \mu_2\}, -, 2)\}$,
  $\quad (MM1, \{\mu_1, \mu_2\}, 1, 2), (MM1, \{\mu_1, \mu_2\}, 1, 2)$,
  $\quad (MM1, \{\mu_1, \mu_2\}, 1, 2), (MM1, \{\mu_1, \mu_2\}, 1, 2)\};$
- $R = 2$;
• $P$ consists of a unique $12 \times 12$ routing matrix.

where terminals and server are modeled by infinite servers and processor sharing service centers, respectively, and I/O devices are modeled by four single server multiclass service centers.

In order to translate a QN model $Q = (\Lambda, C, R, P)$ into a corresponding and equivalent Æmilia specification we define the following functions:

1. $arrivals : A \rightarrow \text{ÆmiliaSpec}$.
   It maps an arrival process into its corresponding Æmilia specification. More precisely, it produces a pair $(aAT, aAET)$ where $aAT$ is the complete architectural type of the required arrival process and $aAET$ is an architectural element type whose behavior is just the invocation of $aAT$.

   For example, the arrival process of the QN model in Figure 3 is translated by the $arrival$ function into the $aAT$ represented in Figure 4 tailored w.r.t. the specified parameters, and the following $aAET$:

   ```
   ELEM_TYPE SA_Type(rate lambda, weight rp1, rp2)
   BEHAVIOR
     Arrival(void;void) = SimpleArrival(rate lambda, weight rp1, rp2)
   INPUT_INTERACTIONS
     deliver_1, deliver_2
   OUTPUT_INTERACTIONS
   which can be directly used in the Æmilia specification of any QN model using such arrival process.

   The next two functions translating service centers and routing matrices, respectively, are defined in a similar way.

2. $service\_centers : C \rightarrow \text{ÆmiliaSpec}$.
   It maps a service center into a pair $(scAT, scAET)$ representing the corresponding Æmilia AT and AET, respectively.

3. $routing : P \rightarrow \text{ÆmiliaSpec}$.
   Let $P$ be the space of all possible routing matrices. The function $routing$ maps a routing matrix into a pair $(rAT, rAET)$ representing the corresponding Æmilia AT and AET, respectively.

   Given now a QN model $Q = (\Lambda, C, R, P)$, the procedure to translate it into a corresponding Æmilia specification $spec$ is composed of the following main steps.

   (1) Let $spec$ be an empty AT. Initialize it with the formal parameters of all the QN model components;

   (2) Add to $spec$ the following AETs:

      (2.1) for each distinct $a \in \Lambda$ add to $spec$ the AET deriving from $arrival(a)$;
(2.2) for each distinct $sc \in C$ add to spec the AET deriving from $service\_centers(cs)$;
(2.3) for each matrix $p \in P$ add to spec the AET deriving from $routing(p)$;
(2.4) add an AET with behavior defined through invocation of the exit process
AT, if necessary;

(3) Add to spec one architectural element instance of the specified type for each
$sc \in C, a \in \Lambda$ and $p \in P$. Add an instance of the exit process AET, if necessary.

(4) Specify the architectural attachments with the routing components.

We shall now apply this procedure to translate the whole QN model of Figure 3 and
w.r.t. the modeling choices made in this section. The resulting complete Æmilia AT of
the QN is defined as follows:

ARCHI_TYPE ComputerSystem(rate lambda, mu-term1, mu-term2,
mu-server1, mu-server2,
mu-P11, mu-P12,
mu-P21, mu-P22,
mu-D11, mu-D12,
mu-D21, mu-D22,
weight rp1,... rp12,
rp1, ..., rp1212,
rp00, ..., rp120)

ARCHI_ELEM_TYPES
ELEM_TYPE SA_Type(rate lambda, weight rp1,... rp12)
BEHAVIOR
Arrival(void;void) = SimpleArrival(rate lambda,
weight rp1, ... rp12)
INPUT_INTERACTIONS
OUTPUT_INTERACTIONS
deliver_1, ... deliver_12
ELEM_TYPE IS_Type(rate mu_1, mu_2)
BEHAVIOR
SC(void;void) = SC_InfiniteServers(rate mu_1,mu_2)
INPUT_INTERACTIONS
arrive_i, arrive_2
OUTPUT_INTERACTIONS
leave_1, leave_2
ELEM_TYPE PS_Type(rate mu-server)
BEHAVIOR
SC(void;void) = SC_ProcessorSharing(rate mu_1, mu_2)
INPUT_INTERACTIONS
arrive_1, arrive_2
OUTPUT_INTERACTIONS
leave_1, leave_2
ELEM_TYPE MM1_Type(rate mu_1, mu_2)
BEHAVIOR
SC(void;void) = SC_MM1(rate mu_1, mu_2)
INPUT_INTERACTIONS
get_1, get_2
OUTPUT_INTERACTIONS
leave_1, leave_2
ELEM_TYPE Sink_Type()
BEHAVIOR
Sink(void;void) = Sink()
INPUT_INTERACTIONS
OR exit_customer
Note that the specification contains one AET for the arrival process, one AET for each service center of the QN and two more AETs for the exit process and the routing component, respectively. The architectural element instances section contains an instance of the arrival, sink and router components and as many nodes as the service centers of the QN model. The architectural attachments connect the arrival process to the router and the router to the exit process. Moreover, all nodes representing service centers are connected to the router and vice versa. A graphical representation of the complete QN is depicted in Figure 15, where, for the sake of readability, the connections from the various components to the router and vice versa are simply sketched.

Once the QN model has been translated into a corresponding Æmilia specification it is possible to take advantage of the functional verification and quantitative analysis supported by Æmilia and implemented in the tool TwoTowers [9]. More precisely, we recall that TwoTowers supports model checking, equivalence checking, Markovian analysis based on rewards, and discrete event simulation. Besides, it is also possi-
6 Conclusions

In this work we have considered and compared two types of models for quantitative performance analysis, a class of QN and Æmilia based SPA. We have proposed an approach to translate a QN model into an Æmilia specification in order to take advantage of the model definition based on SPA that allows to combine functional and performance analysis and compositional, graphical and hierarchical modeling of complex systems. This approach extends a previous comparison between Æmilia specification and QN, in the direction of an integration of these two formalisms. Such integration aims to combine the main advantages of the two methods as tools for system performance analysis, in order to efficiently describe and analyze both functional and performance properties of system specifications within the same integrated framework. The combination and integration of different modeling techniques for performance analysis is an interesting research direction, also in the area of software performance modeling. The proposed approach applies to multiclass QN with various types of service centers and different service disciplines, including PS, IS, FCFS, and different priority scheduling. Each QN component is mapped into an Æmilia specification by defining the corresponding Architectural Type and then all components are eventually glued together to specify the entire QN.

The definition of these relationships and translation between performance models, beside the possible advantage of model integration, should enhance the comprehen-
sion of the relative merit of each type of model and its expressiveness and potentiality. Specifically, this mapping of classes of models from QN to SPA could help in identifying special constraints that allow a product form solution for Æmilia specification.

Concerning future work, we would like to provide an automated support to the illustrated transformation technique, by implementing both the functions translating the various QN components and the final procedure gluing them together to compose the considered QN model.

Acknowledgment

We thank Marco Bernardo for his useful comments and suggestions.

References


