

Heterogeneous Queueing Networks with Blocking: An Architectural Approach based on SPA

Simonetta Balsamo¹

Marco Bernardo²

¹Dipartimento di Informatica

Università "Ca' Foscari" di Venezia - Italy

²Centro per l'Applicazione delle Scienze e Tecnologie dell'Informazione

Università di Urbino - Italy

Abstract

Queueing networks (QNs) with finite capacity queues and blocking can be applied for performance analysis and evaluation of computer and communication systems with finite capacity resources and population constraints. Heterogeneous QNs with blocking model systems where components have different blocking policies or mechanisms. Performance analysis and functional verification, e.g. deadlock freedom, of heterogeneous QNs are in general difficult. We propose a framework for combined functional and performance analysis of heterogeneous QNs with finite capacity and blocking based on an architectural description language (ADL) relying on stochastic process algebra (SPA). First, we define a library of parametrized models of finite capacity queue service centers supporting certain blocking mechanisms, as they are the basic building blocks of the QNs with blocking. This is done with *Æmilia*, an ADL for assembling in a controlled way the SPA descriptions of the behavior of the components of the service centers. Then, we construct *Æmilia* models of heterogeneous QNs with blocking by simply composing the basic QN building blocks described with *Æmilia*. Functional properties, such as deadlock freedom, can be automatically verified on the *Æmilia* models of the heterogeneous QNs with blocking. Likewise, the Markov chains underlying the heterogeneous QNs with blocking can be automatically derived from their *Æmilia* models. Moreover, we can easily define new and different blocking policies in a heterogeneous QN by providing the related *Æmilia* descriptions that can be compositionally integrated in the QN model and analyzed by the proposed approach.

Keywords: analytical modeling, queueing models, blocking, deadlock freedom, stochastic process algebra, stochastic modeling.

1 Introduction

Queueing networks (QNs) have been widely applied as a modelling tool for performance evaluation and prediction of discrete flow systems, such as computer and communication systems [11, 13]. QNs with finite capacity queues and blocking have been introduced as more realistic models of systems with finite resources and population constraints [15, 2]. When a customer arrives at a finite capacity queue that is full, it cannot enter the queue and it is blocked. Various blocking types or mechanisms can be defined specifying the behavior of such blocked customers in the network, such as for example Repetitive Service (RS), Blocking After Service (BAS), and Blocking Before Service (BBS). QNs with blocking are in general difficult to solve. Their stationary state queue length distributions show a product form solution only under special constraints. Hence, most of the methods applied to analyze such models are approximation, simulation, and numerical techniques. Moreover, deadlock may occur in QNs with finite capacity queues and blocking. Deadlock avoidance and prevention techniques must be applied, depending on the blocking mechanism used to model the QN. Heterogeneous QNs with blocking model systems where components have different blocking policies or mechanisms. Such models allow complex systems to be represented, where different protocols are applied to manage blocking due to finite capacity resources. The analysis of heterogeneous QNs is in general more difficult than homogeneous QNs, both to evaluate performance indices and to verify functional properties

such as deadlock freedom. In particular, the definition of the state space of the Markov process underlying the heterogeneous QNs is complex because blocking mechanisms in QNs are usually informally specified for each service center, but they often affect the behavior and the state of other service centers.

In this paper we address the compositional specification of arbitrarily complex heterogeneous models of QNs with finite capacity and blocking, in a way that allows functional and performance analysis to be automatically carried out. We aim at defining a framework for the analysis of blocking models that can be easily extended to include some interesting features, such as heterogeneous blocking mechanisms, the definition of new blocking policies and protocols, and multiple classes of customers.

Our framework is based on stochastic process algebra (SPA) [8, 10, 9, 6]. This is an algebraic formalism introduced in the past decade that, thanks to its compositionality and abstraction capabilities, has turned out to be well suited for the functional verification and the performance evaluation of concurrent and distributed systems. In particular, the framework is based on an enhancement of a specific, tool supported [5] SPA called EMPA_{gr} [6], which has been chosen due to its expressive power that allows for the modeling of exponential and zero durations, probabilities, and priorities. In order to take advantage of the compositionality provided by SPA, our approach to the study of heterogeneous QNs with blocking comprises two steps. First, we define a library of parametrized models of finite capacity queue service centers supporting certain blocking mechanisms, as they are the basic building blocks of the QNs with blocking. This is done with $\text{\AE}milia$ [1], an architectural description language (ADL) for assembling in a controlled way the EMPA_{gr} descriptions of the behavior of the components of the service centers. Then, we construct $\text{\AE}milia$ models of heterogeneous QNs with blocking by simply composing the basic QN building blocks described with $\text{\AE}milia$. Functional properties, such as deadlock freedom, can be automatically verified on the $\text{\AE}milia$ models of the heterogeneous QNs with blocking. Likewise, the Markov chains underlying the heterogeneous QNs with blocking can be automatically derived from their $\text{\AE}milia$ models. Moreover, we can easily define new and different blocking policies in a heterogeneous QN by providing the related $\text{\AE}milia$ descriptions that can be compositionally integrated in the QN model and analyzed by the proposed approach.

The paper is organized as follows. Sect. 2 recalls the features of the specification language $\text{\AE}milia$ and introduces the first description of a queueing system formed by a single service center with finite capacity queue. Sect. 3 presents the modeling with $\text{\AE}milia$ of some basic queueing systems with finite capacity queues and various blocking types: BAS, RS, and BBS. Sect. 4 deals with modeling of heterogeneous QNs with various blocking types, combining the basic queueing system models in the proposed framework for functional and performance analysis, and illustrates an example. Finally, Sect. 5 reports some concluding remarks.

2 $\text{\AE}milia$: A SPA Based ADL

In this section we recall – through a running example based on finite capacity queueing systems – the basics of $\text{\AE}milia$ [1], the specification language for the compositional, graphical, and hierarchical modeling of complex systems that we adopt in our framework for the study of functional and performance properties of heterogeneous QNs with blocking. In essence, $\text{\AE}milia$ is the result of the integration of two formalisms: PADL [3, 4] and EMPA_{gr} [6]. The former is a process algebra based ADL equipped with some architectural checks for the detection of architectural mismatches. The latter is an expressive SPA supporting the functional verification and the performance evaluation of concurrent and distributed systems.

With $\text{\AE}milia$ the description of a complex system can be done compositionally. First, we have to define the behavior of the types of components in the system and their interactions with the other components. The functional and performance aspects of the behavior are described through a family of EMPA_{gr} terms, while the interactions are described through actions occurring in such terms. Then, we have to declare the instances of each type of component present in the system and the way in which their interactions are attached to each other in order to allow the instances to communicate. The whole behavior of the system is a family of EMPA_{gr} terms automatically obtained by composing in parallel the behavior of the declared instances according to the specified attachments. From the whole behavior, suitable state transition based models can be automatically derived, on which functional verification and performance evaluation can be carried out.

2.1 Syntax and Translation Semantics

\mathcal{A} emilia is based on the stochastic process algebra EMPA_{gr} , which we briefly recall below.

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

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

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

In the syntax above, “ $\underline{0}$ ” is the term that cannot execute any action. Term $\langle a, \tilde{\lambda} \rangle . E$ can execute action $\langle a, \tilde{\lambda} \rangle$ and then behaves as term E . Term E/L behaves as term E with each executed action $\langle a, \tilde{\lambda} \rangle$ turned into $\langle \tau, \tilde{\lambda} \rangle$ whenever $a \in L$. Term $E[\varphi]$ behaves as term E with each executed action $\langle a, \tilde{\lambda} \rangle$ turned into $\langle \varphi(a), \tilde{\lambda} \rangle$. Term $E_1 + E_2$ behaves as either term E_1 or term E_2 depending on whether an action of E_1 or an action of E_2 is executed. If the choice involves exponentially timed actions, the race policy applies and each involved action is selected with a probability proportional to its rate. If the choice involves immediate actions, they take precedence over exponentially timed ones and the generative preselection policy applies: each involved immediate action with the highest priority level is selected with a probability proportional to its weight. If the choice involves passive actions, the reactive preselection policy applies: for every action type, each involved passive action of that type with the highest priority level is selected with a probability proportional to its weight (the choice among involved passive actions of different types is nondeterministic). Term $E_1 \parallel_S E_2$ asynchronously executes actions of E_1 or E_2 not belonging to S and synchronously executes equal actions of E_1 and E_2 belonging to S provided that one of them is passive. In case of synchronization, the resulting action has the same type as the two original actions, while its rate is given by the rate of the original nonpassive action multiplied by the reactive execution probability of the original passive action (or is passive in case of synchronization of two passive actions). The action prefix operator and the alternative composition operator are called dynamic operators, whereas the hiding operator, the relabeling operator, and the parallel composition operator are called static operators. A term is called sequential if it is composed of dynamic operators only. The semantics for EMPA_{gr} is operationally defined through inference rules that map every term $E \in \mathcal{G}$ to a state transition graph $\mathcal{I}[E]$, whose states are represented by process terms and whose transitions are labeled with actions. After pruning the lower priority transitions, from the integrated semantic model $\mathcal{I}[E]$ it is possible to derive a purely functional semantic model $\mathcal{F}[E]$ by removing action rates from the transitions, and a purely performance semantic model $\mathcal{M}[E]$ by essentially removing action types from the transitions. $\mathcal{M}[E]$, which is defined only if $\mathcal{I}[E]$ has no passive transitions, is a continuous time or a discrete time Markov chain depending on whether $\mathcal{I}[E]$ has exponentially timed transitions or not. A notion of equivalence, called strong Markovian bisimulation equivalence and strictly related to the ordinary lumpability, is defined on the integrated semantic models of EMPA_{gr} terms, that can be used to compositionally reduce the state space of such models.

A description in \mathcal{A} emilia represents an architectural type. The description of an architectural type starts with the name of the architectural type and its numeric parameters, which often are values for exponential rates and weights. Each architectural type is defined as a function of its architectural element types (AETs) and its architectural topology. An AET is defined as a function of its behavior, specified either as a family of sequential EMPA_{gr} terms 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. The architectural topology is specified through the declaration of a set of architectural element instances (AEIs) representing the system components, a set of architectural (as opposed to local) interactions given by some interactions of the AEIs that act as interfaces for the whole architectural type, and a set of directed architectural attachments among the interactions of the AEIs. Every interaction is declared to be

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

an input interaction or an output interaction and the attachments must respect such a classification: every attachment must involve an output interaction and an input interaction of two different AEIs. An AEI can have different types of interactions (input/output, local/architectural); it must have at least one local interaction. Every local interaction must be involved in at least one attachment, while every architectural interaction must not be involved in any attachment. In order to allow several AEIs to synchronize, every local interaction can be involved in several attachments provided that no autosynchronization arises, i.e. no chain of attachments is created that starts from a local interaction of an AEI and terminates on a local interaction of the same AEI. On the performance side, we require that, for the sake of modeling consistency, all the occurrences of an action type in the behavior of an AET have the same kind of rate (exponential or immediate with the same priority level or passive with the same priority level) and that, to comply with the synchronization discipline of EMPA_{gr}, every chain of attachments contains at most one interaction whose associated rate is exponential or immediate.

We show in Table 1 an *Æmilia* textual description for an architectural type representing a queueing system. The FIFO queue has capacity n and receives jobs to be processed from m_{in} different arrival sources. The arrival of a job from source j is represented through a passive action with type $arrive_j$, while the extraction of the job at the beginning of the queue performed by the server is represented through a passive action with type $extract$. Both kinds of actions are passive because the time at which they happen is determined by the components with which the queue interacts. The job processing is carried out by a single server at a rate μ , which then forwards the processed jobs to m_{out} different destinations with probabilities $p_1, \dots, p_{m_{\text{out}}}$, respectively. The extraction of the job at the beginning of the queue is represented through an immediate action with type $extract$, while the departure of a processed job with destination j is represented through an immediate action with type $leave_j$. Both kinds of actions are immediate as their duration is taken to be irrelevant from the performance viewpoint. The selection of the destination is realized a priori through a choice among suitably weighed immediate actions with type $choose_j$. The service of a job is represented through an exponentially timed action with type $serve$. In order to make the queue and the server communicate, their $extract$ interactions are attached to each other. Instead, the input interactions of the queue and the output interactions of the server are declared as being architectural interactions, so that they can be used for hierarchical modeling purposes when embedding the description of the queueing system in the description of a larger system. The same queueing system is depicted in Fig. 1 through the *Æmilia* graphical notation, which is based on flow graphs [14]. In a flow graph, the boxes denote the AEIs, the black circles denote the local interactions, the white squares denote the architectural interactions, and the directed edges denote the attachments.

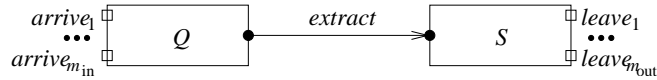


Figure 1: Flow graph of *QueueingSystem*

We observe that the *Æmilia* textual description of the queueing system is fully parameterized w.r.t. the queue capacity, the number of arrival sources and destinations, the service rate, and the routing probabilities. It is worth pointing out that further degrees of parameterization could be added to the *Æmilia* description above. As an example, if the service rate and the routing probabilities depended on the number of jobs in the queue, then we should introduce a vector of n service rates $[\mu_i]_{1 \leq i \leq n}$ and a vector of n sets of routing probabilities $[\{p_{i,1}, \dots, p_{i,m_{\text{out}}}\}]_{1 \leq i \leq n}$. Every subterm of the form $\langle extract, * \rangle. Queue_{i-1}$ in state $Queue_i$ for $i = 1, \dots, n$ should be replaced with

$$\langle extract_i, * \rangle. Queue_{i-1}$$

and the server behavior should be modified as follows

$$Server \triangleq \sum_{i=1}^n \langle extract_i, \infty \rangle. \sum_{j=1}^{m_{\text{out}}} \langle choose_j, \infty_{1,p_{i,j}} \rangle. \langle serve, \mu_i \rangle. \langle leave_j, \infty \rangle. Server$$

with every $extract_i$ passive interaction of the queue attached to the corresponding $extract_i$ immediate interaction of the server. As another example of possible degree of parameterization, if there were s servers instead of a single one, then we could exploit the fact that $QueueT$ and $ServerT$ are separately defined as

archi_type	<i>QueueingSystem</i> (int $n, m_{\text{in}}, m_{\text{out}}$; exp_rate μ ; weight $p_1, \dots, p_{m_{\text{out}}}$)
archi_elem_types	
elem_type	<i>QueueT</i> (int n, m_{in})
behavior	$Queue_0 \triangleq \sum_{j=1}^{m_{\text{in}}} \langle arrive_j, * \rangle. Queue_1$ $Queue_i \triangleq \sum_{j=1}^{m_{\text{in}}} \langle arrive_j, * \rangle. Queue_{i+1} +$ $\quad \langle extract, * \rangle. Queue_{i-1} \quad 1 \leq i \leq n-1$ $Queue_n \triangleq \langle extract, * \rangle. Queue_{n-1}$
interactions	input $arrive_1, \dots, arrive_{m_{\text{in}}}$ output $extract$
elem_type	<i>ServerT</i> (int m_{out} ; exp_rate μ ; weight $p_1, \dots, p_{m_{\text{out}}}$)
behavior	$Server \triangleq \langle extract, \infty \rangle.$ $\quad \sum_{j=1}^{m_{\text{out}}} \langle choose_j, \infty_{1,p_j} \rangle. \langle serve, \mu \rangle. \langle leave_j, \infty \rangle. Server$
interactions	input $extract$ output $leave_1, \dots, leave_{m_{\text{out}}}$
archi_topology	
archi_elem_instances	$Q : QueueT(n, m_{\text{in}})$ $S : ServerT(m_{\text{out}}; \mu; p_1, \dots, p_{m_{\text{out}}})$
archi_interactions	input $Q.arrive_1, \dots, Q.arrive_{m_{\text{in}}}$ output $S.leave_1, \dots, S.leave_{m_{\text{out}}}$
archi_attachments	from $Q.extract$ to $S.extract$
end	

Table 1: Textual description of *QueueingSystem*

they represent two conceptually distinct entities. In order to take into account the presence of s servers, in *QueueT* we should replace every subterm of the form $\langle extract, * \rangle. Queue_{i-1}$ for $i = 1, \dots, n$ with

$$\sum_{k=1}^s \langle extract_k, * \rangle. Queue_{i-1}$$

then we should simply reuse *ServerT* by declaring s instances of it

$$S_1, \dots, S_s : ServerT(m_{\text{out}}; \mu; p_1, \dots, p_{m_{\text{out}}})$$

with the following attachments

$$\begin{aligned} & \mathbf{from} \mathit{Q.extract}_1 \mathbf{to} \mathit{S}_1.extract \\ & \vdots \\ & \mathbf{from} \mathit{Q.extract}_s \mathbf{to} \mathit{S}_s.extract \end{aligned}$$

As a further degree of parameterization, there may be several different classes of jobs. The resulting multiclass queueing system can be described by suitably modifying *QueueT* and *ServerT*. As far as *QueueT* is concerned, there are two possibilities. If there is a single queue for all the classes, then all the *extract* action types must be suitably indexed, so that the server knows the class to which each selected job belongs. If instead there are as many queues as there are classes of jobs, each with its capacity constraint, then the *extract* action types related to different classes are automatically kept distinct. As far as *ServerT* is concerned, it must be equipped with a service rate parameter as well as a vector of routing probabilities for each class of job. Additionally, in the presence of distinct queues for the different classes of jobs, *ServerT* must include a scheduling policy to select the next job to be served, which can be FIFO, round robin, or priority based.

The semantics of an *Æmilia* specification is given by translation into EMPA_{gr} in two steps. In the first step, the semantics of all the instances of each AET is defined to be the behavior of the AET projected onto its interactions. Such a projected behavior is obtained from the family of sequential EMPA_{gr} terms representing the behavior of the AET by applying a hiding operator on all the actions that are not interactions. In this way, we abstract from all the internal details of the behavior of the instances of the AET. For the queueing system of Table 1 we have

$$\begin{aligned} \llbracket \text{QueueingT} \rrbracket &= \llbracket Q \rrbracket = \text{Queue}_0 \\ \llbracket \text{ServerT} \rrbracket &= \llbracket S \rrbracket = \text{Server} / \{ \text{choose}_1, \dots, \text{choose}_{m_{\text{out}}}, \text{serve} \} \end{aligned}$$

In the second step, the semantics of an architectural type is obtained by composing in parallel the semantics of its AEIs according to the specified attachments, possibly after relabeling to the same type the interactions whose types are involved in the same chain of attachments. For the queueing system of Table 1 we have

$$\llbracket \text{QueueingSystem} \rrbracket = \llbracket Q \rrbracket \parallel_{\{\text{extract}\}} \llbracket S \rrbracket$$

Every *Æmilia* description \mathcal{A} is thus transparently translated into an EMPA_{gr} term $\llbracket \mathcal{A} \rrbracket$, which has an integrated semantic model $\mathcal{I}[\llbracket \mathcal{A} \rrbracket]$, a functional semantic model $\mathcal{F}[\llbracket \mathcal{A} \rrbracket]$, and a performance semantic model $\mathcal{M}[\llbracket \mathcal{A} \rrbracket]$ (if $\mathcal{I}[\llbracket \mathcal{A} \rrbracket]$ has no passive transitions). Such models can be used to predict functional and performance properties of the system via functional verification (model checking, equivalence/preorder checking) and Markovian/simulation analysis with the TwoTowers tool [5].

2.2 Architectural Type Definition and Invocation: Hierarchical Modeling

An *Æmilia* description represents a family of system architectures called an architectural type. All the members of the family must have the same observable functional behavior and topology, while the internal behavior and the performance characteristics can vary. Given an architectural type \mathcal{A} defined with formal AETs $\mathcal{C}_1, \dots, \mathcal{C}_m$, formal architectural interactions a_1, \dots, a_l , and formal numeric parameters v_1, \dots, v_h , an instance of \mathcal{A} can be obtained through an invocation of the form

$$\mathcal{A}(\mathcal{C}'_1, \dots, \mathcal{C}'_m; a'_1, \dots, a'_l; v'_1, \dots, v'_h)$$

where $\mathcal{C}'_1, \dots, \mathcal{C}'_m$ are actual AETs preserving ² the observable functional behavior of the formal AETs, a'_1, \dots, a'_l are actual names for the architectural interactions, and v'_1, \dots, v'_h are actual values for the numeric parameters.

The architectural type invocation mechanism, together with the possibility of defining architectural interactions, can be exploited to model system architectures in a hierarchical way. This is simply achieved by letting the behavior of an AET to be defined not only through a family of sequential EMPA_{gr} terms, but also through an invocation of a previously defined architectural type. If the behavior of an AET is defined through an architectural type invocation, the interactions of that AET are given by the actual names for the architectural interactions specified in the invocation. The semantics of that AET and its instances is then given by the semantics of the architectural type invocation, with all the local interactions of the invoked architectural type being hidden as they are internal activities from the AET viewpoint.

3 Modeling Queueing Systems with Blocking

In this section we address the modeling with *Æmilia* of finite capacity queueing systems that are part of QNs where blocking phenomena can occur. The purpose is twofold. On one hand, we want to assess the effectiveness of a SPA based ADL like *Æmilia* to formalize several different blocking policies. On the other hand, we want to develop a library of fully parameterized descriptions of queueing system nodes that can be easily combined to model arbitrarily complex QNs with blocking. The blocking mechanisms that we consider are Blocking After Service (BAS), Repetitive Service (RS), and Blocking Before Service (BBS) [2]. Other blocking mechanisms, like the one typical of kanban networks, can be modeled similarly. For the sake of simplicity, we consider queueing systems with a single class of jobs and a single server whose service rate and routing probabilities are not state dependent. Queueing systems not satisfying this constraint can be dealt with as shown in Sect. 2.1.

²According to the weak bisimulation equivalence [14].

3.1 Blocking After Service

Assume that node i is regulated by the blocking after service mechanism (BAS). According to this policy, upon completion of the service of a job at node i whose destination is node j , the job cannot leave node i until node j has free positions in its queue. Node i is thus blocked as long as the served job cannot join its destination queue. This blocking mechanism is mainly used to model production systems and disk I/O subsystems. It is also referred to as classical, transfer, manufacturing and production blocking in the literature.

If we look at Table 1, we soon realize that *QueueT* can be used to model a queue having m_{in} upstream BAS nodes and that *ServerT* can be used to model the server of a BAS node. If the $leave_j$ architectural interaction of the BAS server node is attached to the $arrive_j$ architectural interaction of the queue of a downstream node, whenever the job currently being served has queue j as its destination, then it cannot leave the server as long as the queue is in state $Queue_n$.

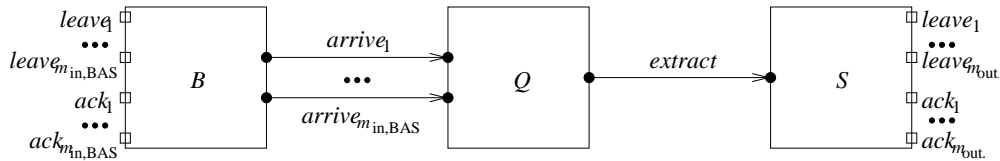


Figure 2: Flow graph of $QueueingSystem_{\text{BAS}}$

If a queue has multiple upstream BAS nodes, several of them can be simultaneously blocked when the queue is full. In this case, it is necessary to define the order in which the blocked jobs (one for each blocked upstream BAS node) will be unblocked as space becomes available in the queue. A commonly adopted policy is first-blocked-first-unblocked: the first job that joins the blocking queue when a departure occurs is the one that was blocked first. In order to take FBFU into account, in Table 2/Fig. 2 we show a BAS based variation of the queueing system in Table 1/Fig. 1. If we compare the two descriptions, we see in the new one the presence of an additional element B that acts as a FIFO scheduler for the upstream BAS nodes. When a job finishes receiving service at one of the upstream BAS nodes, the node communicates to B that the job is ready to leave and waits for an acknowledgement (see $ServerT_{\text{BAS}}$). If there is space in the queue, then the job joins the queue when it is its turn, and an acknowledgement is sent back to the server that can thus proceed with the next job in its queue (see $FBFUBufferT$). Note that the definition of *QueueT* is unchanged, even though its *arrive* interactions are no longer architectural.

3.2 Repetitive Service Blocking

Assume that node i is regulated by the repetitive service blocking mechanism (RS). Whenever a job completes its service at node i and attempts to enter destination node j , if node j is full at that time then the job remains at node i , where it will receive a new and independent service. The RS mechanism has two variants. In the random destination case (RS-RD), the destination node j is randomly chosen by the job every time it starts being served at node i . In the fixed destination case (RS-FD), the destination node j is chosen by the job the first time it starts being served at node i . This blocking mechanism is mainly used to model telecommunication systems. It is also referred to as rejection, retransmission and repeat blocking in the literature.

As shown in Fig. 3, a queueing system whose server complies with RS-RD or RS-FD can be modeled through an *Æmilia* description similar to that of Table 2. Let us modify *QueueT* and $ServerT_{\text{BAS}}$ defined in Table 2 in order to reflect RS. We start by considering the queue of a node that is one of the destinations of some nodes regulated by RS. This queue must inform all its RS upstream nodes about the presence/unavailability of free positions in the queue. In general, if the first $m_{\text{in,BAS}}$ arrival sources of the queue are governed by BAS and the subsequent $m_{\text{in,RS}}$ arrival sources of the queue are governed by RS, then *QueueT* must be modified as follows:

archi_type	$QueueingSystem_{BAS}(\mathbf{int} n, m_{in,BAS}, m_{out}; \mathbf{exp_rate} \mu; \mathbf{weight} p_1, \dots, p_{m_{out}})$
archi_elem_types	
elem_type	$FBFUBufferT(\mathbf{int} m_{in,BAS})$
behavior	$FBFUBuffer_{\varepsilon} \triangleq \sum_{j=1}^{m_{in,BAS}} \langle leave_j, * \rangle . FBFUBuffer_j$ $FBFUBuffer_{j \circ \sigma} \triangleq \sum_{\substack{h \notin \sigma \\ 1 \leq h \leq m_{in,BAS}}} \langle leave_h, * \rangle . FBFUBuffer_{j \circ \sigma \circ h} + \langle arrive_j, \infty \rangle . \langle ack_j, \infty \rangle . FBFUBuffer_{\sigma} \quad 0 \leq \sigma < m_{in}$
interactions	input $leave_1, \dots, leave_{m_{in,BAS}}$ output $arrive_1, \dots, arrive_{m_{in,BAS}}, ack_1, \dots, ack_{m_{in,BAS}}$
elem_type	$QueueT(\mathbf{int} n, m_{in,BAS})$
behavior	$Queue_0 \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle . Queue_1$ $Queue_i \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle . Queue_{i+1} + \langle extract, * \rangle . Queue_{i-1} \quad 1 \leq i \leq n-1$ $Queue_n \triangleq \langle extract, * \rangle . Queue_{n-1}$
interactions	input $arrive_1, \dots, arrive_{m_{in,BAS}}$ output $extract$
elem_type	$ServerT_{BAS}(\mathbf{int} m_{out}; \mathbf{exp_rate} \mu; \mathbf{weight} p_1, \dots, p_{m_{out}})$
behavior	$Server_{BAS} \triangleq \langle extract, \infty \rangle . Server'_{BAS}$ $Server'_{BAS} \triangleq \sum_{j=1}^{m_{out}} \langle choose_j, \infty_{1,p_j} \rangle . \langle serve, \mu \rangle . \langle leave_j, \infty \rangle . \langle ack_j, * \rangle . Server_{BAS}$
interactions	input $extract, ack_1, \dots, ack_{m_{out}}$ output $leave_1, \dots, leave_{m_{out}}$
archi_topology	
archi_elem_instances	$B : FBFUBufferT(m_{in,BAS})$ $Q : QueueT(n, m_{in,BAS})$ $S : ServerT_{BAS}(m_{out}; \mu; p_1, \dots, p_{m_{out}})$
archi_interactions	input $B.leave_1, \dots, B.leave_{m_{in,BAS}}, S.ack_1, \dots, S.ack_{m_{out}}$ output $B.ack_1, \dots, B.ack_{m_{in,BAS}}, S.leave_1, \dots, S.leave_{m_{out}}$
archi_attachments	from $B.arrive_1$ to $Q.arrive_1$ \dots from $B.arrive_{m_{in,BAS}}$ to $Q.arrive_{m_{in,BAS}}$ from $Q.extract$ to $S.extract$
end	

Table 2: Æmilia description of $QueueingSystem_{BAS}$

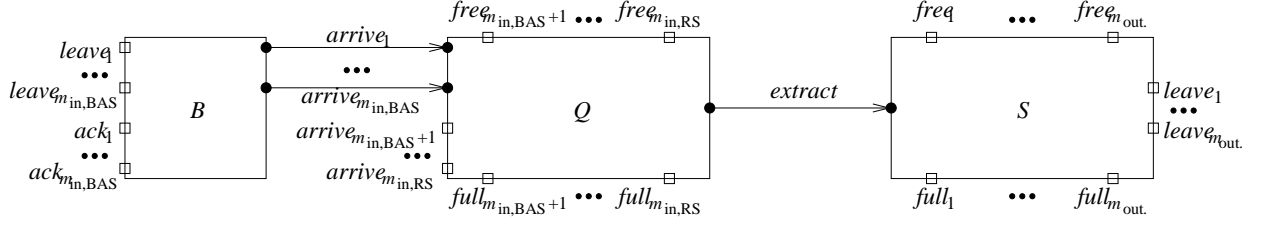


Figure 3: Flow graph of $QueueingSystem_{RS-RD}$ and $QueueingSystem_{RS-FD}$

elem_type	$QueueT(\mathbf{int} n, m_{in,BAS}, m_{in,RS})$	
behavior	$Queue_0 \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle . Queue_1 +$ $\sum_{j=m_{in,BAS}+1}^{m_{in,BAS}+m_{in,RS}} \langle free_j, \infty \rangle . \langle arrive_j, * \rangle . Queue_1$ $Queue_i \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle . Queue_{i+1} +$ $\sum_{j=m_{in,BAS}+1}^{m_{in,BAS}+m_{in,RS}} \langle free_j, \infty \rangle . \langle arrive_j, * \rangle . Queue_{i+1}$ $\langle extract, * \rangle . Queue_{i-1} \quad 1 \leq i \leq n-1$ $Queue_n \triangleq \langle extract, * \rangle . Queue_{n-1} +$ $\sum_{j=m_{in,BAS}+1}^{m_{in,BAS}+m_{in,RS}} \langle full_j, \infty \rangle . Queue_n$	
interactions	input	$arrive_1, \dots, arrive_{m_{in,BAS}+m_{in,RS}}$
	output	$extract, free_{m_{in,BAS}+1}, \dots, free_{m_{in,BAS}+m_{in,RS}}, full_{m_{in,BAS}+1}, \dots, full_{m_{in,BAS}+m_{in,RS}}$

where the newly added *free* and *full* interactions are declared to be architectural. If we compare the new definition of $QueueT$ with the one in Table 2, we see that the only difference is that the queue now signals to the upstream RS servers whether it has free positions or it is full.

For the sake of simplicity, let us assume that the job of a RS node that finds its destination queue full is inserted at the beginning of the queue of the RS node, i.e. the job is immediately reprocessed. In order to model a RS-RD server, $ServerT_{BAS}$ must be changed into the following $ServerT_{RS-RD}$:

elem_type	$ServerT_{RS-RD}(\mathbf{int} m_{out}; \mathbf{exp_rate} \mu; \mathbf{weight} p_1, \dots, p_{m_{out}})$	
behavior	$Server_{RS-RD} \triangleq \langle extract, \infty \rangle . Server'_{RS-RD}$ $Server'_{RS-RD} \triangleq \sum_{j=1}^{m_{out}} \langle choose_j, \infty_{1,p_j} \rangle . \langle serve, \mu \rangle .$ $(\langle free_j, * \rangle . \langle leave_j, \infty \rangle . Server_{RS-RD} +$ $\langle full_j, * \rangle . Server'_{RS-RD})$	
interactions	input	$extract, free_1, \dots, free_{m_{out}}, full_1, \dots, full_{m_{out}}$
	output	$leave_1, \dots, leave_{m_{out}}$

where the newly added *free* and *full* interactions are declared to be architectural. When the server finishes serving a job, it asks the destination queue of the job whether it has free positions. By the definition of $QueueT$ above, the server receives either the free signal or the full signal. In the former case, the job is able to leave the server thereby joining the destination queue (an attachment will be defined between $leave_j$ of the server and $arrive_{j'}$ of the destination queue). In the latter case, the service of the job is repeated.

In the case of a RS-FD server, $ServerT_{BAS}$ must be changed into the following $ServerT_{RS-FD}$:

elem_type	$ServerT_{RS-FD}(\text{int } m_{\text{out}}; \text{exp_rate } \mu; \text{weight } p_1, \dots, p_{m_{\text{out}}})$
behavior	$Server_{RS-FD} \triangleq \langle \text{extract}, \infty \rangle. \sum_{j=1}^{m_{\text{out}}} \langle \text{choose}_j, \infty_{1,p_j} \rangle. Server'_{RS-FD,j}$ $Server'_{RS-FD,j} \triangleq \langle \text{serve}, \mu \rangle. (\langle \text{free}_j, * \rangle. \langle \text{leave}_j, \infty \rangle. Server_{RS-FD} + \langle \text{full}_j, * \rangle. Server'_{RS-RD,j})$
interactions	input $\text{extract}, \text{free}_1, \dots, \text{free}_{m_{\text{out}}}, \text{full}_1, \dots, \text{full}_{m_{\text{out}}}$ output $\text{leave}_1, \dots, \text{leave}_{m_{\text{out}}}$

In this case, the destination queue is chosen once and for all as soon as the job is extracted by the server.

3.3 Blocking Before Service

Assume that node i is regulated by the blocking before service mechanism (BBS). According to this policy, a job at node i declares its destination node j before it starts receiving service. Upon entering the server at node i , the service of the job does not start until node j is found to have free positions. The service of the job is then interrupted every time node j becomes full prior to the completion of the job. The service is resumed as soon as a departure occurs at node j , in which case the amount of service previously received by the job is assumed to be lost, i.e. a new service of the same job starts when node i becomes unblocked. This blocking mechanism is used to model production, telecommunication, and computer systems. It is also referred to as service or immediate blocking in the literature.

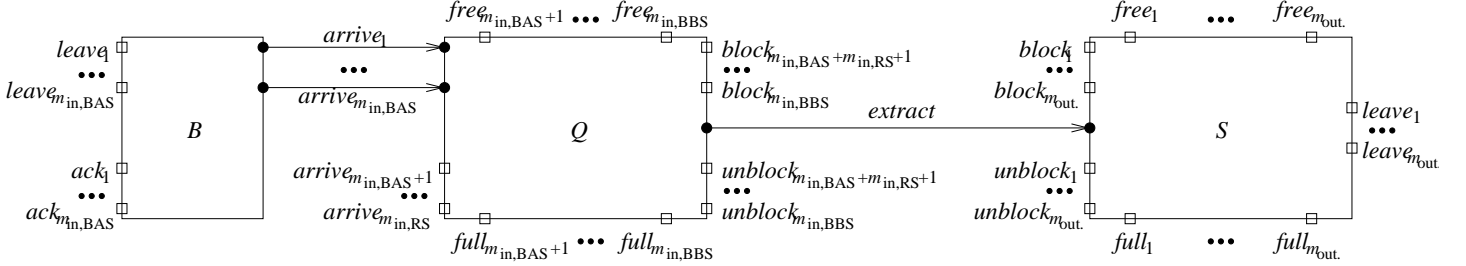


Figure 4: Flow graph of $QueueingSystem_{BBS}$

As shown in Fig. 4, a queueing system whose server complies with BBS can be modeled through an \mathcal{A} emilia description similar to that outlined in Sect. 3.2. Let us modify $QueueT$ and $ServerT_{RS-RD}$ defined in Sect. 3.2 in order to reflect BBS. As far as $QueueT$ is concerned, it now has to support the presence of upstream BBS nodes. In essence, the new version of $QueueT$ must permit the upstream BBS nodes to check for the presence of free positions also at the beginning of the job service, block the upstream BBS nodes as soon as it becomes full, and unblock the upstream BBS nodes as soon as it starts again to accept jobs. In general, if the first $m_{\text{in,BAS}}$ arrival sources of the queue are governed by BAS, the subsequent $m_{\text{in,RS}}$ arrival sources of the queue are governed by RS, and the final $m_{\text{in,BBS}}$ arrival sources of the queue are governed by BBS, then $QueueT$ must be modified as follows:

elem_type
behavior

$$\begin{aligned}
& QueueT(\mathbf{int} \ n, m_{in,BAS}, m_{in,RS}, m_{in,BBS}) \\
Queue_0 & \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle. Queue_1 + \\
& \sum_{j=m_{in,BAS}+1}^{m_{in,RS}} \langle free_j, \infty \rangle. \langle arrive_j, * \rangle. Queue_1 + \\
& \sum_{j=m_{in,BAS}+m_{in,RS}+m_{in,BBS}}^{m_{in,BAS}+m_{in,RS}+1} (\langle arrive_j, * \rangle. Queue_1 + \\
& \langle free_j, \infty \rangle. Queue_0) \\
Queue_i & \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle. Queue_{i+1} + \\
& \sum_{j=m_{in,BAS}+1}^{m_{in,RS}} \langle free_j, \infty \rangle. \langle arrive_j, * \rangle. Queue_{i+1} + \\
& \sum_{j=m_{in,BAS}+m_{in,RS}+m_{in,BBS}}^{m_{in,BAS}+m_{in,RS}+1} (\langle arrive_j, * \rangle. Queue_{i+1} + \\
& \langle free_j, \infty \rangle. Queue_i) + \\
& \langle extract, * \rangle. Queue_{i-1} \qquad 1 \leq i \leq n-2 \\
Queue_{n-1} & \triangleq \sum_{j=1}^{m_{in,BAS}} \langle arrive_j, * \rangle. \bigcirc_{h=m_{in,BAS}+m_{in,RS}+1}^{m_{in,BAS}+m_{in,RS}+m_{in,BBS}} \langle block_h, \infty \rangle. Queue_n + \\
& \sum_{j=m_{in,BAS}+1}^{m_{in,BAS}+m_{in,RS}} \langle free_j, \infty \rangle. \langle arrive_j, * \rangle. \\
& \bigcirc_{h=m_{in,BAS}+m_{in,RS}+1}^{m_{in,BAS}+m_{in,RS}+m_{in,BBS}} \langle block_h, \infty \rangle. Queue_n + \\
& \sum_{j=m_{in,BAS}+m_{in,RS}+1}^{m_{in,BAS}+m_{in,RS}+m_{in,BBS}} (\langle arrive_j, * \rangle. \langle arrive_j, * \rangle. \\
& \bigcirc_{h=m_{in,BAS}+m_{in,RS}+1}^{m_{in,BAS}+m_{in,RS}+m_{in,BBS}} \langle block_h, \infty \rangle. Queue_n + \\
& \langle free_j, \infty \rangle. Queue_{n-1}) + \\
& \langle extract, * \rangle. Queue_{n-2} \\
Queue_n & \triangleq \langle extract, * \rangle. \bigcirc_{h=m_{in,BAS}+m_{in,RS}+1}^{m_{in,BAS}+m_{in,RS}+m_{in,BBS}} \langle unblock_h, \infty \rangle. Queue_{n-1} + \\
& \sum_{j=m_{in,BAS}+1}^{m_{in,BAS}+m_{in,RS}+m_{in,BBS}} \langle full_j, \infty \rangle. Queue_n
\end{aligned}$$

interactions **input** $arrive_1, \dots, arrive_{m_{in,BAS}+m_{in,RS}+m_{in,BBS}}$
output $extract, free_{m_{in,BAS}+1}, \dots, free_{m_{in,BAS}+m_{in,RS}+m_{in,BBS}},$
 $full_{m_{in,BAS}+1}, \dots, full_{m_{in,BAS}+m_{in,RS}+m_{in,BBS}},$
 $block_{m_{in,BAS}+m_{in,RS}+1}, \dots, block_{m_{in,BAS}+m_{in,RS}+m_{in,BBS}},$
 $unblock_{m_{in,BAS}+m_{in,RS}+1}, \dots, unblock_{m_{in,BAS}+m_{in,RS}+m_{in,BBS}}$

where the newly added *block* and *unblock* interactions are declared to be architectural. If we compare the new definition of *QueueT* with the one in Sect. 3.2, first we see that in every state $Queue_i$ for $0 \leq i \leq n-1$ (in state $Queue_n$) the queue can now be sensed to have free positions (to be full) by each of the upstream BBS nodes as well. Second, we note that in state $Queue_{n-1}$ ($Queue_n$) the queue now blocks (unblocks) all the upstream BBS nodes as soon as it receives (forwards) a job.

In order to model a BBS server, $ServerT_{RS-RD}$ of Sect. 3.2 must be changed into the following $ServerT_{BBS}$:

elem_type	$ServerT_{BBS}(\mathbf{int} \ m_{out}; \mathbf{exp_rate} \ \mu; \mathbf{weight} \ p_1, \dots, p_{m_{out}})$
behavior	$Server_{BBS} \triangleq \langle extract, \infty \rangle. \sum_{j=1}^{m_{out}} \langle choose_j, \infty_{1,p_j} \rangle. Server'_{BBS,j} +$ $\sum_{h=1}^{m_{out}} \langle block_h, * \rangle. Server_{BBS} +$ $\sum_{h=1}^{m_{out}} \langle unblock_h, * \rangle. Server_{BBS}$ $Server'_{BBS,j} \triangleq \langle free_j, * \rangle. Server''_{BBS,j} +$ $\langle full_j, * \rangle. Server'''_{BBS,j} +$ $\sum_{h=1}^{m_{out}} \langle block_h, * \rangle. Server'_{BBS,j} +$ $\sum_{h=1}^{m_{out}} \langle unblock_h, * \rangle. Server'_{BBS,j} \quad 1 \leq j \leq m_{out}$ $Server''_{BBS,j} \triangleq \langle serve, \mu \rangle. Server'''_{BBS,j} +$ $\langle block_j, * \rangle. Server'''_{BBS,j} +$ $\sum_{\substack{h \neq j \\ 1 \leq h \leq m_{out}}} \langle block_h, * \rangle. Server''_{BBS,j} +$ $\sum_{\substack{h \neq j \\ 1 \leq h \leq m_{out}}} \langle unblock_h, * \rangle. Server''_{BBS,j} \quad 1 \leq j \leq m_{out}$ $Server'''_{BBS,j} \triangleq \langle leave_j, \infty \rangle. Server_{BBS} +$ $\sum_{h=1}^{m_{out}} \langle block_h, * \rangle. Server'''_{BBS,j} +$ $\sum_{h=1}^{m_{out}} \langle unblock_h, * \rangle. Server'''_{BBS,j} \quad 1 \leq j \leq m_{out}$ $Server''''_{BBS,j} \triangleq \langle unblock_j, * \rangle. Server''_{BBS,j} +$ $\sum_{\substack{h \neq j \\ 1 \leq h \leq m_{out}}} \langle block_h, * \rangle. Server''''_{BBS,j} +$ $\sum_{\substack{h \neq j \\ 1 \leq h \leq m_{out}}} \langle unblock_h, * \rangle. Server''''_{BBS,j} \quad 1 \leq j \leq m_{out}$
interactions	input $extract, free_1, \dots, free_{m_{out}}, full_1, \dots, full_{m_{out}}$ $block_1, \dots, block_{m_{out}}, unblock_1, \dots, unblock_{m_{out}}$ output $leave_1, \dots, leave_{m_{out}}$

where the newly added *block* and *unblock* interactions are declared to be architectural. In state $Server_{BBS}$, the server extracts the job at the beginning of the queue and the job selects its destination j . In state $Server'_{BBS,j}$, the server is informed about whether the destination queue j has free positions or not. By the definition of $QueueT$ above, exactly one of the two conditions can be true. In state $Server''_{BBS,j}$, which is reached at the beginning/restart of the service of the job, either the job is processed till completion or its service is interrupted by the destination queue j because it has become full. In state $Server'''_{BBS,j}$, which is reached after the completion of the service of the job, the job tries to leave the server in order to join its destination queue j . Finally, in state $Server''''_{BBS,j}$, which is reached after an interruption of the job service, the server waits for a departure at the saturated destination queue j . We observe that in every state the server simply ignores (through the two trailing summations) the *block/unblock* signals received by the downstream queues that are not the destination of the job being served.

4 Heterogeneous Queueing Networks with Blocking

A QN with blocking can be formally described with $\mathcal{A}emilia$ through the library of parametrized $\mathcal{A}emilia$ descriptions of finite capacity queueing systems with blocking provided in Sect. 3. This is easily accomplished by selecting from the library those $\mathcal{A}emilia$ descriptions corresponding to the nodes of the QN with blocking, defining their actual numeric parameters, and attaching their architectural interactions to each other in a

way that reproduces the topology of the QN with blocking. This compositional process, besides facilitating the formalization of the functional and performance aspects of the QN with blocking, automatically produces state transition based models on which functional verification and performance evaluation can be carried out. As an example, from the overall *Æmilia* description of the QN with blocking we can build the Markov chain underlying the QN itself. As observed in [2], retrieving such a Markov chain is not an easy task because the notion of state is complicated by the mutual influences of the nodes. With our approach, instead, the underlying Markov chain can be automatically obtained via the operational semantics of the EMPA_{gr} term into which the overall *Æmilia* description is translated.

We now show that our approach is also useful to deal with another interesting problem in the study of QNs with blocking: the detection of deadlock due to the combination of cycles and finite capacity queues. We distinguish between two cases. In the first case, where every node is governed by the same blocking mechanism, we consider homogeneous QNs with blocking. For this kind of QNs with blocking, it is relatively easy to investigate the presence of deadlock [2, 12]. Let M be the number of nodes, N be the number of jobs, and B_i be the capacity of node $i = 1, \dots, M$ (which is given by the queue capacity plus the server space if it can be used to hold blocked jobs). Then a homogeneous QN with blocking governed by BAS, RS-FD, or BBS is deadlock free if, denoted by C the set of cycle in the network topology, we have

$$N < \min_{c \in C} \sum_{i \in c} B_i$$

i.e. the total number of jobs in the network is less than the sum of the capacities of the nodes along the smallest cycle in the network. Whenever a homogeneous QN with blocking is instead governed by RS-RD, then the network is deadlock free if

$$N < \sum_{i=1}^M B_i$$

i.e. the total number of jobs in the network is less than the sum of the capacities of the nodes in the network, which is the loosest constraint that can be imposed on a QN with blocking.

In the second case, where distinct nodes can be governed by different blocking mechanisms, we consider heterogeneous QNs with blocking. For this kind of QNs with blocking it is not easy at all to establish sufficient conditions for deadlock freedom. We show below that in this case one can profitably resort to a SPA based ADL like *Æmilia* to detect the presence of deadlock. The advantage is threefold. First, the model of the heterogeneous QN with blocking can be easily obtained by simply combining the parametric *Æmilia* descriptions of the single nodes taken from the library defined in Sect. 3. Second, the absence of deadlock can be automatically verified on the *Æmilia* description of the heterogeneous QN with blocking. Third, the Markov chain underlying the heterogeneous QN with blocking can be automatically constructed from the *Æmilia* description of the network.

Let us consider the heterogeneous QN with blocking in Fig. 5. The number associated with each queue is the related capacity, the number associated with each server is the related service rate, and all the other numbers are routing probabilities; the label of each server represents the blocking mechanism adopted by the server. In this heterogeneous QN with blocking we have three RS-RD nodes and two BAS nodes connected as shown in the figure. RS-RD QNs with blocking and BAS QNs with blocking have different sufficient conditions for deadlock freedom. Thus, in the case in which RS-RD nodes and BAS nodes are mixed together, we do not know which of the two conditions is still valid.

To solve the problem above, we search the library of Sect. 3 for the *Æmilia* descriptions of the queueing systems forming the heterogeneous QN with blocking. They are: ³

$$\begin{array}{ll} \text{QueueingSystem}_{\text{RS-RD}}(4, 1, 1, 0, 2; 1.5; 0.4, 0.6) & \text{for node 1} \\ \text{QueueingSystem}_{\text{RS-RD}}(4, 1, 1, 0, 2; 2.1; 0.3, 0.7) & \text{for node 2} \\ \text{QueueingSystem}_{\text{BAS}}(1, 0, 1, 0, 1; 3.6; 1.0) & \text{for node 3} \\ \text{QueueingSystem}_{\text{RS-RD}}(2, 0, 2, 0, 2; 4.2; 0.8, 0.2) & \text{for node 4} \\ \text{QueueingSystem}_{\text{BAS}}(3, 0, 1, 0, 1; 5.7; 1.0) & \text{for node 5} \end{array}$$

where we recall that:

- the first group of parameters comprises the queue capacity, the number of upstream BAS nodes, the number of upstream RS nodes, the number of upstream BBS nodes, and the number of downstream nodes;

³For the sake of readability, in these architectural type invocations we explicitly provide only the actual numeric parameters

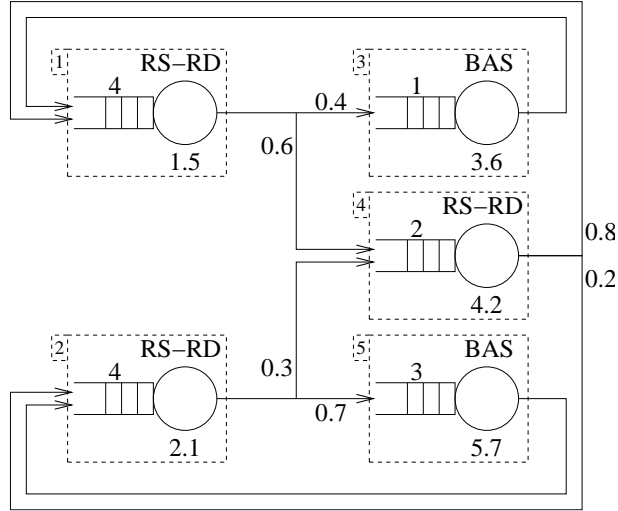


Figure 5: A heterogeneous QN with blocking

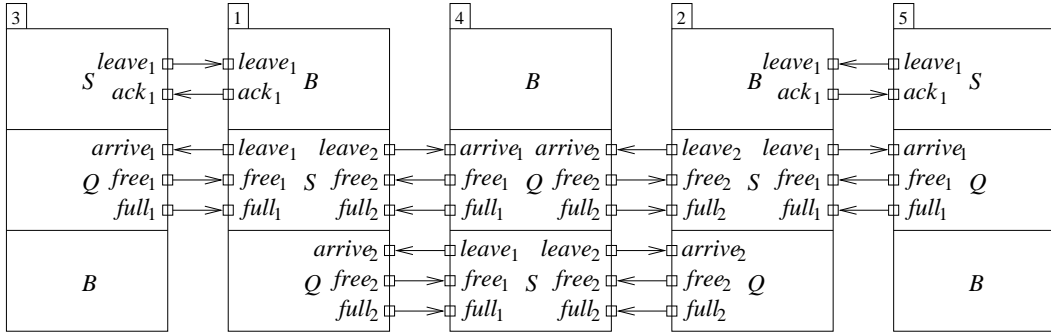


Figure 6: Flow graph of the heterogeneous QN with blocking

- the second group of parameters is given by the service rate;
- the third group of parameters comprises the routing probabilities to the downstream nodes.

Then, we compose the \mathcal{A} emilia descriptions above according to the topology in Fig. 5. The result is shown in Fig. 6. When dealing with architectural type invocations, the local interactions and the attachments involving them are no longer exposed. Therefore, only the architectural interactions of the five queueing systems and the attachments among them declared to reflect the topology in Fig. 5 are shown in Fig. 6.

Finally, we check with the TwoTowers tool the presence of deadlock in the EMPA_{gr} term representing the semantics of the \mathcal{A} emilia description of the heterogeneous QN with blocking. Before doing that, we have to inject the jobs into the \mathcal{A} emilia description. This does not require the explicit modeling of jobs flowing along the network, but is simply achieved by declaring for each queue and server element its initial state. If a queue of capacity n has initially $0 \leq n' \leq n$ jobs, then its initial state is declared to be $\text{Queue}_{n,n'}$. Similarly, if a server has already (not yet) extracted the job at the beginning of the queue, then its initial state is declared to be $\text{Server}'_{\text{BAS}}$ ($\text{Server}_{\text{BAS}}$), $\text{Server}'_{\text{RS-RD}}$ ($\text{Server}_{\text{RS-RD}}$), $\text{Server}'_{\text{RS-FD},j}$ ($\text{Server}_{\text{RS-FD}}$), or $\text{Server}'_{\text{BBS},j}$ ($\text{Server}_{\text{BBS}}$) depending on the blocking mechanism adopted by the server. The total capacity of the heterogeneous QN with blocking under study has a total capacity B given by

$$B = (4 + 1) + (4 + 1) + (1 + 1) + (2 + 1) + (3 + 1) = 19$$

The results of the analysis conducted with TwoTowers are shown in Table 3 for N varying between 14 (all the queues are full) and $14 + 5$ (all the queues are full and all the servers are busy). For each of the

N	$\mathcal{I}[_]$ states	$\mathcal{I}[_]$ transitions	$\mathcal{M}[_]$ states	$\mathcal{M}[_]$ transitions	absorbing states	satisfies formula
14 + 0	19672	32044	1324	9047	no	yes
14 + 1	13116	21448	944	6343	no	yes
14 + 2	7544	12484	592	3862	no	yes
14 + 3	3516	5886	312	1940	no	yes
14 + 4	1282	2333	128	736	no	yes
14 + 5	296	652	32	160	no	no

Table 3: Analysis results

considered configurations, Table 3 reports the size of the integrated semantic model and the size of the continuous time Markov chain obtained after removing the immediate transitions from the integrated semantic model. The sixth column shows the presence of absorbing states in the integrated/performance semantic models. Detecting the presence of absorbing states is however not enough to guarantee deadlock freedom in the heterogeneous QN with blocking. As can be noted, the outcome is no even in the case in which the network is fully saturated, with all the queues being full and all the servers being busy. The reason is that in the three RS-RD servers there is always some activity going on. In order to be more accurate, we have to check that from every state it is possible to reach another state from which a job can leave a node and enter its destination node. This can easily be done in three steps. First, in the EMPA_{gr} term representing the semantics of the \mathcal{A} emilia description of the heterogeneous QN with blocking, we hide all the actions not related to a job transfer and we relabel all the remaining actions to the same type *transfer_job*. This results in a modified EMPA_{gr} term whose state transition graph exposes only the *transfer_job* actions. Second, we define the following CTL formula [7]

$$A G \langle\langle \text{transfer_job} \rangle\rangle tt$$

which expresses the fact that every state (G for globally) along every computation (A for all) can perform a *transfer_job* action possibly preceded by the execution of arbitrarily many invisible actions ($\langle\langle \text{transfer_job} \rangle\rangle tt$). Third, we verify via model checking [7] whether the modified EMPA_{gr} term constructed in the first step satisfies the CTL formula above. The result is reported in the last column of Table 3, from which we can conclude that the heterogeneous QN with blocking does not deadlock as long as $N < 19$. In other words, we have discovered that in this case the sufficient condition for RS-RD QNs with blocking still applies.

5 Conclusion

In this paper we have proposed a framework for functional and performance analysis of heterogeneous QNs with finite capacity and blocking. The framework exploits the compositional and hierarchical modeling capabilities of \mathcal{A} emilia, an ADL based on the SPA EMPA_{gr} . Heterogeneous QNs with blocking can be used to model complex systems with different blocking mechanisms and they are in general difficult to analyze. In the proposed framework, we have defined a library of parametrized \mathcal{A} emilia models of service centers with finite capacity queues and various blocking mechanisms. Heterogeneous QNs can easily be derived by composing these basic QN building blocks described with \mathcal{A} emilia. Then functional properties, such as deadlock freedom, can be automatically verified on the \mathcal{A} emilia descriptions of the heterogeneous QNs with blocking, as illustrated by the example in Sect. 4. Likewise, the Markov chain underlying the heterogeneous QNs with blocking can be automatically derived from their \mathcal{A} emilia models. New blocking policies can easily and compositionally be integrated in the proposed model definition and analysis, by providing their \mathcal{A} emilia descriptions.

Further research will be devoted to the definition of more efficient techniques for deadlock detection in \mathcal{A} emilia models of heterogeneous QNs with blocking. Since \mathcal{A} emilia is already equipped with local checks for the detection of architectural mismatches resulting in deadlock when combining deadlock free components [1], the idea is to suitably modify such local checks in order to capture the specific interpretation of the concept of deadlock – no jobs moving from a node to another – in the field of heterogeneous QNs with blocking.

References

- [1] S. Balsamo, M. Bernardo, M. Simeoni, “*Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis*”, to appear in Proc. of the *3rd Int. Workshop on Software and Performance (WOSP 2002)*, Rome (Italy), 2002
- [2] S. Balsamo, V. De Nitto Personè, R. Onvural, “*Analysis of Queueing Networks with Blocking*”, Kluwer, 2001
- [3] M. Bernardo, P. Ciancarini, L. Donatiello, “*On the Formalization of Architectural Types with Process Algebras*”, in Proc. of the *8th ACM Int. Symp. on the Foundations of Software Engineering (FSE-8)*, ACM Press, pp. 140-148, San Diego (CA), 2000
- [4] M. Bernardo, P. Ciancarini, L. Donatiello, “*Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems*”, in Proc. of the *2nd Working IEEE/IFIP Conf. on Software Architecture (WICSA 2001)*, IEEE-CS Press, pp. 77-86, Amsterdam (The Netherlands), 2001
- [5] M. Bernardo, W.R. Cleaveland, W.S. Stewart, “*TwoTowers 1.0 User Manual*”, <http://www.sti.uniurb.it/bernardo/twotowers/>, 2001
- [6] M. Bravetti, M. Bernardo, “*Compositional Asymmetric Cooperations for Process Algebras with Probabilities, Priorities, and Time*”, in Proc. of the *1st Int. Workshop on Models for Time Critical Systems (MTCS 2000)*, Electronic Notes in Theoretical Computer Science 39(3), State College (PA), 2000
- [7] E.M. Clarke, O. Grumberg, D.A. Peled, “*Model Checking*”, MIT Press, 1999
- [8] N. Götz, U. Herzog, 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, Rome (Italy), 1993
- [9] H. Hermanns, “*Interactive Markov Chains*”, Ph.D. Thesis, University of Erlangen (Germany), 1998
- [10] J. Hillston, “*A Compositional Approach to Performance Modelling*”, Cambridge University Press, 1996
- [11] L. Kleinrock, “*Queueing Systems*”, John Wiley & Sons, 1975
- [12] S. Kundu, I. Akyildiz, “*Deadlock Free Buffer Allocation in Closed Queueing Networks*”, in *Queueing Systems Journal* 4:47-56, 1989
- [13] S.S. Lavenberg editor, “*Computer Performance Modeling Handbook*”, Academic Press, 1983
- [14] R. Milner, “*Communication and Concurrency*”, Prentice Hall, 1989
- [15] H.G. Perros, “*Queueing Networks with Blocking*”, Oxford University Press, 1994