GENERATION OF TRANSITION CLASS MODELS FROM FORMAL QUEUEING NETWORK DESCRIPTIONS*

Johann Christoph Strelen
Rheinische Friedrich-Wilhelms-Universität Bonn
Römerstr. 164, 53117 Bonn, Germany
E-mail: strelen@cs.uni-bonn.de

KEYWORDS

Queueing Networks, Tools, Multi-Paradigm Models, Generation, Markov Chains, Monte Carlo Simulation.

ABSTRACT

A main feature of a novel tool is described, the algorthmic translation of formal queueing network descriptions into transition class models. Transition classes are a high-level modelling paradigm for the structured and compact definition of Markov chains. Transition class models can be solved with many known solution techniques for Markov chains, exact and approximate, and simulation, even hybrid analysis is possible. The queueing network description language and the translator are extensible with respect to new network features, for example new node types. The tool is extensible with respect to new modelling paradigms and new solvers. Models can be multi-paradigm.

INTRODUCTION

Queueing networks (QN) are a widely and successfully used paradigm for stochastic models of computer, communication, and manufacturing systems. In the literature, lots of books and articles exist about modelling with queueing networks and about the analysis of the models. Usually, a tool is used for the analysis of a QN model.

There exist many excellent tools for the analysis of models, for example QNA [19, 32], PEPSY [2], and DyQNtool$^+$ [15] for queueing networks, QPN [1] for queueing networks and coloured generalized stochastic Petri nets, SHARPE [21] for Markov reward models, GreatSPN [4, 5], SPNP [6] for generalized stochastic Petri nets (GSPN), UltraSAN [8] for stochastic activity networks, a class of stochastic Petri nets, DSPNexpress [20] for stochastic Petri nets with exponentially distributed or deterministic delays, SPN2MGM [13] for quasi-birth-death models, PEPA [11] and TIPP [17] for stochastic process algebras, to enumerate a few; for an overview, see for example [14].

The results of this paper are part of the development of a new tool which is planned to having some interesting features. Different modelling paradigms are supported, for example queueing networks and generalized stochastic Petri nets (GSPN). A model may be multi-paradigm, some parts of it may be given as QN, others as GSPN, for example.

Basically, all supported models define Markov chains. This has advantages and disadvantages. The class of models is large, many model features are supported. Primarily, the analysis provides state probabilities. Performance measures like throughput of jobs, utilization and availability of system components, response times, and performability indices are calculated using the state probabilities and the transition rates or the transition probabilities (Markov reward models). The holding times of states are independent, in continuous time Markov chains exponentially distributed, otherwise constant, but general distributions can be modelled with Coxian or phase-type (PH) distributions or Markovian arrival processes (MAP), autoregressive processes with Markov modulated Poisson processes (MMPP).

The same model can be solved with different solvers as appropriate. Many elaborated techniques for the solution of Markov chains are available [9, 22, 10]. For Markov chains with a very large state space, we developed an approximation technique which we call *disaggregation-aggregation (DA) iteration* [26, 24, 25, 27, 30]. For stiff Markov chains, we developed a method of type simultaneous vector iteration [22, 23], and Courtois' approximate technique for nearly completely decomposable systems [7] was applied.

In [28], we consider Monte-Carlo simulation for Markov chain models. Markov chains with very large state spaces can be analyzed, many simulation runs can be executed in parallel, and Courtois' method is adapted to simulation models with rare events. Hybrid solution is possible using two different solvers for parts of the same model.

The tool is extensible with respect to new model features, for example a new node type for queueing networks, extensible with respect to more modelling paradigms - the second paradigm which we are introducing is generalized stochastic Petri nets, and extensible with respect to new solvers.

It is planned that submodels can be bound together into models, and we hope that it will be possible to analyze qualitative properties of models as well.

How is all this obtained? The key idea is as follows: The high-level models are translated into lower-level models, in one or two steps (for the notion of models in different levels of abstraction see [16, "General Modelling Tool Frame-

*Extended Version of [29]
ESS2000, August 29, 2000

work"]). As output of the first translation step, we propose a modelling technique which we call *transition classes (TC)* [24, 25]. TC models define Markov chains, but they are more abstract, more readable, more structured, and smaller than state space descriptions and transition probability or generator matrices. TC models can be simulated directly, or the Markov chain can be generated, or other numerical representations which are suited for approximate techniques like DA iteration.

In our tool, the TC models are the common interface between high-level models and the solvers. If a translator for a new modelling paradigm is provided which generates TC models, these models can be solved with all the available solvers, and if a new solver for TC models is included, all models can be solved with it if the restrictions of the solver are observed. For the extensibility of a specific high-level class of models like queueing networks, the model description language and the translator are designed to be extensible.

Most of the mentioned features were tried in prototypes for the tool.

In this paper, we deal with the description of the queueing networks, and with the translation of QN models into TC models. The algorithmic transformation of queueing networks into transition class models needs a formal description of the network. We propose a formal queueing network description language for that. It would be very tedious to collect all known features of queuing networks from the literature, all types of nodes, all scheduling algorithms, all types of customers, and so on. Moreover, it may happen and is likely that new things will be devised soon. Therefore we propose an extensible description language. The generator is some kind of translator of this language which is also extensible. The number of different kinds of nodes are very large due to the fact that this variety is some kind of cartesian product of the different arrival processes, different kinds and numbers of waiting rooms, different kinds and numbers of servers, different scheduling algorithms, different kinds of customers, and so on. To cope with this problem, the description of the networks not only provides types of complete nodes but also types of building parts (*mini nodes*) like arrival processes, waiting buffers, servers, and so on which can be combined into complex nodes. The expressive power of the queueing network description language and, in turn, of the generator, is shown to be quite strong.

In the first section, the most important features of TC models are depicted in short. The second section describes the ideas of this paper, the formal queueing network description language, and the algorithm for the generator which is a translator into transition class models. The expressive power of the method is indicated.

TRANSITION CLASSES

Irreducible aperiodic Markov chains, homogeneous in time, with finite state space $\mathcal{Z}^T$, $|\mathcal{Z}^T| = n \in I\!N$, are considered. Either they are aperiodic and discrete in time, $(\mathbf{Z}(t),\ t \in I\!N)$ (DTMC), or continuous in time, $(\mathbf{Z}(t),\ t \in I\!R)$ (CTMC). They stem from stochastic models of computer, communication, or manufacturing systems which consist of interacting subsystems, their *(system) components*. At every moment, each component is in one of its possible *component states*, and we denote the whole system state by *state tuples* $\mathbf{z} = (z_1, \ldots, z_K)$, where z_k indicates the state of the *elementary* component k. More complex system components are composed of some elementary components. Their states are given as tuples of natural numbers. Sometimes virtual system components are also useful, for example a "component" which indicates the number of jobs in an open queueing network.

Example. In a queueing network with blocking, Poisson arrivals, if any, and $\cdot/\mathrm{M}/1/\nu_k/\mathrm{FCFS}$ nodes, the system components are the nodes. Their states are the numbers z_k of jobs in them, $z_k \leq \nu_k,\ k = 1, \ldots, K$. A system state is given by the $K-$tuple $\mathbf{z} = (z_1, \ldots, z_K)$. □

Submodels can be understood as (complex) components; their states are given by some elements $z_k, z_{k'}, \ldots$ of the state tuple.

Transition classes (TC) structure the design of the Markov models, provide a concise description of the state space, and typically the transition probability or generator matrix is given by only a few rules, even for huge state spaces. Transition class models can be simulated immediately, it is possible to generate the Markov chain automatically, and transition classes are useful for approximate methods and aggregation/disaggregation techniques: the numbering of the states is not essential, things like row or column interchanges of the transition probability matrix are not needed.

In general, a transition class describes many state transitions, e.g. job arrivals in a specific node of a queueing network which may occur in different states.

In many interesting models, e.g. performance or reliability models, there is only a moderate number of transition classes due to similarity of transitions and small numbers of states which are reachable in one transition (sparsity). This is important for efficiency.

Transition classes are triplets $\tau = (\mathcal{U}, u, \alpha)$. A transition according to transition class τ from state $\mathbf{y}$ into state $\mathbf{z}$ may occur only if $\mathbf{y} \in \mathcal{U}$, where $\mathcal{U}$ denotes the *source states set* of this transition class. $\mathcal{U} \cap \mathcal{Z}^T$ is the set of feasible old states, from where the transitions can occur ($\mathcal{U}$ may additionally contain some infeasible state tuples for the sake of a simple description of $\mathcal{U}$, for example in programs). The *destination state function* $u : \mathcal{Z}^T \to \mathcal{Z}^T$ defines the new states, i.e. a transition of class τ leads from state $\mathbf{y} \in \mathcal{U}$ into state $u(\mathbf{y})$. u may depend on the old state $\mathbf{y}$. The function u is given by functions u_k, $u(\mathbf{y}) = \big(u_1(\mathbf{y}), \ldots, u_K(\mathbf{y})\big)$. Transitions of class τ occur with probability or rate $\alpha(\mathbf{y})$. α may depend on the old state $\mathbf{y}$.

Example. We consider a closed queueing network which consists of $K \in I\!N$ nodes. In each node $k \in [1..K]$ is a single exponential server, service rate μ_k. Buffer space is available for no more than ν_k jobs. When the service of a job in node k is finished the job tries to go to node l with probability $t_{k,l},\ l \in [1 : K]$. If there is no buffer space available, the job remains in node k and instantaneously receives another service (*repetitive-service with random destination policy*).

State transitions occur when a job leaves node k for node i, $i \neq k$. In node k, there must be at least one job, and in node i buffer space must be available. Therefore, the source states set is $\mathcal{U} = \{\mathbf{z} \in \mathcal{Z}^T \mid z_k > 0,\ z_i < \nu_i\}$. Only the states of the nodes k and i change, $u_k(\mathbf{z}) = z_k - 1,\ u_i(\mathbf{z}) = z_i + 1$, and the transition rate is $\alpha = \mu_k t_{k,i}$. □

A transition class model consists of $\mathcal{TC}$, the set of transition classes, and a single feasible state $\mathbf{y}$. In our example, there is a TC for each pair of connected nodes.

Submodels can be used for abstraction and modularization. Let $z_k, z_{k'}, ...$ denote the elements of the state tuple which indicate the states of a submodel S. Some transition classes $\tau_1, \tau_2, ...$ belong to S: First all TCs $\tau = (\mathcal{U}, u, \alpha)$ where z_k or $z_{k'}$ or ... restrict the source state set $\mathcal{U}$, for example $\mathcal{U} = \{\mathbf{z} \in \mathcal{Z}^T \mid z_k \neq 0\}$, and secondly, all TCs $\tau = (\mathcal{U}, u, \alpha)$ which change one of the elements $z_k, z_{k'}, ...$, for example $u_k(\mathbf{z}) \neq z_k$. Clearly, a TC may belong to more than just a single submodel: some submodels may change their states with the same state transition.

The following theorem states that the transition class paradigm is quite general.

Theorem 1. For each irreducible Markov chain with finite state space, there is an equivalent transition class model.

Sketch of proof. Consider an irreducible Markov chain with finite state space $\mathcal{Z}$ and generator matrix $[q_{y,z}]_{y\in\mathcal{Z},\, y\in\mathcal{Z}}$ or transition probability matrix $[p_{y,z}]_{y\in\mathcal{Z},\, y\in\mathcal{Z}}$. We build an equivalent transition class model as follows. For each pair $(y, z) \in \mathcal{Z}^2$ of states with $q_{y,z} > 0$ or $p_{y,z} > 0$, respectively, a transition class $TC = (\mathcal{U}, u, \alpha)$ is established where

$$\begin{aligned} \mathcal{U} &= \{y\}, \\ u(y) &= z, \\ \alpha &= q_{y,z} \ \text{ or } \ \alpha = p_{y,z}, \ \text{ respectively.} \end{aligned}$$

Together with a state $y \in \mathcal{Z}$, this is a transition class model which is obviously equivalent to the considered Markov chain. □

In [25, 24, 27], transition classes are presented for different interarrival and service time distributions, for different service disciplines, for fork-join synchronization.

Due to similarities between stochastic process algebras [3, 18, 12] and transition classes, we conjecture that transition class models offer the possibility to analyze qualitative properties, namely safety properites, progress properties, and lifeness. This will be investigated later.

THE GENERATOR

A formal description must be given for a queueing network which is to be transformed by an algorithm into a transition class model. Now we are describing the elements of such a *formal queueing network description*.

Queueing networks consist of *nodes*. *Jobs (customers)* may be created in a source according to an arrival process or are existing at the beginning, visit the nodes, wait there in waiting rooms (queues), are selected for service according to a strategy, get service which may be interrupted, and go to another node or leave the network. Jobs may split or join. Jobs may belong to different *job classes*.

If a job may go from node k to node k', there is an *edge* (k, k') of the network.

A node is modelled with a composed system component which consists of elementary components; we will say, *the node is the owner of these elementary components*. As a special case, a node may have only one or even no elementary component at all. For example, a node may be modelled with an elementary component which indicates the number of waiting jobs.

A node may be the *owner of transition classes* which perform state transitions concerning this node. For example, a node may have a transition class for arrivals from outside of the network, and a transition class for the completion of a service after which the job goes to another node or leaves the network.

Any pair (k, k') of different nodes may have a common *interface*. For example, if there is an edge from node k to node k', these nodes have an interface. A standard feature of this interface concerns jobs which are finished in node k and are sent to node k'. Such an action is caused by a transition class of node k, and the state of both nodes is altered. The state change within node k is done directly, but in node k' it is accomplished using an interface operation.

Interface features other than the sending of jobs are optional, that is they are not used at all interfaces.

Nodes

A node is an instance according to a *node type*. A node type has a name and is a prototype of nodes with certain features. Instances have these features but may differ with respect to some attributes which must be specified in the declaration of the nodes. A node type is defined by patterns for transition classes, for components, and for interface features, and some rules define in which way the actual attribute values influence the generation of concrete items. When a node is translated, according to these patterns transition classes, components, and interface features are generated, using the actual attribute values.

For example, the node type MM1_loss describes nodes with a waiting buffer (queue), a Poisson arrival process, FCFS scheduling, an exponential server, and jobs are lost if they try to go to a neighbour node in which no buffer space is available. A transition class is provided for arrivals, and for each outgoing edge there is a transition class which finishes a service and tries to send the finished job to its neighbour node but the job is lost if the target queue is full. Parameters are λ, the rate of the Poisson process, μ, the rate of the server, ν, the capacity of the buffer, and some transition probabilities for the random selection of target nodes for ready served jobs. A node of the MM1_loss type is the owner of one elementary component in which the state of the queue is stored. This state variable has the identifier Length.

If Node1 is an instance of this node type MM1_loss, the attributes and state variables can be accessed with dot notation, for example Node1.lambda is the service rate of

the node Node1, and Node1.Length is at every moment the number of jobs in its queue.

Formal queueing network descriptions consist in node declarations, one feasible state, and, optional, interface definitions.

Each node has a name. A node declaration begins with the node's name, the node type follows, and attribute value definitions, in general.

We present a nearly complete formal queueing network description, only the declarations of two more nodes, Inner and Sink, are omitted. By default, on all edges, the interface feature of type send is adopted, hence the interface operation I_send and the inquiry may_I_send can be used. Later on, we will describe another interface feature type, and give an interface definition as an example.

```
(* A node, name Source, type MM1_loss *)
Source: MM1_loss
(   mu = 1          (* Service rate *)
    nu = 3          (* Buffer capacity *)
    lambda = 0.8    (* Arrival rate *)
    t[Inner] = 0.5  (* Transfer probability
                       to node Inner *)
    t[Sink] = 0.5   (* Transfer probability
                       to node Sink *)
)

Initialize:             (* One possible state *)
(   Source.Length= 0    (* Number of jobs in the queue *)
    Inner.Length= 0     (* Number of jobs in the queue *)
    Sink.Length= 0      (* Number of jobs in the queue *)
)
```

Interfaces

A transition class of a node may change the state of its own components directly, but if components of other nodes are also to be altered, this must be accomplished using *interface operations*. The state changes of a transition class $\tau = (\mathcal{U}, u, \alpha)$ are defined by the destination state function u. Hence, the destination state functions apply interface operations.

Interface features belong to *interface features types*. For example the standard feature which handles the sending of jobs from a node to another node has the type send.

For any other interface feature type, say x, there is a set $\mathcal{I}^{\mathsf{x}}$ which containes the interfaces which adopt this interface feature. That means, the interface feature of typ x is adopted at the interface between nodes k and k' iff $(k, k') \in \mathcal{I}^{\mathsf{x}}$. Example: Consider two nodes Q2 and Q3 with blocking-after-service. If in node Q2 a job is served completely, it tries to go to node Q3. But, if there no buffer space is left, the job remains in node Q2, and blocks the server there. Immediately if a ready served job leaves node Q3, the blocked job of node Q2 goes to node Q3. This action is executed by node Q3 using the interface operation I_take. In the formal queueing network description, the interface feature of type take is adopted by a declaration: (Q3,Q2): take

Each interface feature type defines interface operations and interface inquiries. An interface operation usually causes an action in a node, namely a state change. For an interface feature type, say act, the operation is called I_act. If this interface feature is adopted at (k, k'), this operation may be used in node k when the state of node k' is to be altered. An *interface inquiry* asks at a node if the according interface operation can be executed or not. The answer is 0 for "no" or a number $n > 0$ for "yes", and the reason is given, namely the states of some elementary system components are indicated which allow or forbid the action. For an interface feature type, say act, the operation is called may_I_act. If this interface feature is adopted at (k, k'), this operation may be used in node k. For example, if node k wants to send jobs to node k', node k asks at node k' if space is left there using the inquiry may_I_send. If the answer is 3, up to 3 jobs may be sent using the operation I_send.

The source state set $\mathcal{U}$ of a transition class $(\mathcal{U}, u, \alpha)$ is the set of states in which the transition can take place. Using the generator, this source state set is calculated by the owner node of the transition class. To this end, the node considers the states of its own elementary components and uses interface inquiries for the others.

In the above example, consider the following situation: In another node Q1, a job is blocked which tries to go to node Q2. In node Q2, a job is blocked which tries to go to node Q3. In node Q3, a service finishes. Hence node Q3 executes I_take in order to fetch a job from node Q2. There, a buffer is freed, and, in turn, Q2 executes I_take in order to fetch a job from node Q1. This indicates that an interface operation may trigger others. The same holds for interface inquiries.

Extensibility

The generator must be extensible because obviously not all thinkable types of networks can be foreseen. New node types and new interface feature types can be added, or nodes can be put together out of *mini nodes*, see next subsection.

For the addition of a new node type, its name is made known within the generator. Its data structure is designed. In general, some elementary system components are planned for waiting buffer states, server states, scheduling information etc., and patterns for transition classes are defined.

Each time when a node of the new type is established, an instance of the data structure is provided for the attribute values. The elementary system components are added to the state tuple, and the transition class model is augmented by transition classes.

The three elements of each transition class $\tau = (\mathcal{U}, u, \alpha)$ are defined with functions. These functions must be provided for the new node type. They use the states, attributes, and interfaces.

The functions for the calculation of the source state set $\mathcal{U}$ decide for a given state if the transition is possible. With respect to the elementary system components of the node itself, this decision is made directly, with respect to other components via interface inquiries which are sent to other nodes.

Similarly, the destination state function u uses the elementary system components of the node and interface operations which are sent to other nodes, for the change of their components.

The probabilities or rates α are calculated from the attribute values, using the node's state and interface inquiries which are used to obtain information about the state of other nodes.

On the other hand, a node of the new node type may receive interface inquiries or interface operation requests. The reaction are answers about the own state or changes of the own state, respectively. Hence, the existing interface inquiries and interface operations must be provided in the new node type.

As an example, let us consider the new node type MM1_PR_loss. The service discipline is preemptive-resume, service times are exponential, the blocking discipline is of type loss. For each priority exists a waiting buffer, jobs of priority p belong to job class p. In each waiting buffer, Poisson arrivals occur.

The attributes are as follows: The number of priorities, the capacity of each waiting buffer, the service rate, the rate of each arrival process, the number of exits to other nodes and their waiting buffers, the branching probabilities to these exits and to the exit out of the whole network.

For each queue, there is an elementary system component the state of which counts the number of jobs in it.

For each waiting buffer, there is a transition class which models the arrival process. For each exit and each priority, there is a transition class which models the transition of a job to the target queue and the losses if the target queue is full. For each priority, a transition class models the transition of jobs to the exit out of the network.

The node may receive an interface inquiry may_I_send which asks if in the queue for priority-p-jobs space is left. The answer is 0 or the amount of available buffer space, and the number of the elementary system component which models the queue. Similarly, the node may receive an interface operation I_send which demands to change the state according to the receipt of a job for a specified queue. If buffer space is left in the queue, the elementary system component which models the queue is augmented, if not, nothing happens.

If a new interface feature type is introduced into the generator, all existing models without this new type remain correct but it happens that an existing node type must be modified. This modification consists in the addition of the new interface operation and inquiry. Usually, a new interface feature type is needed togehter with a new node type.

As an example, consider the new interface feature type take and a node of the existing type MM1_loss. If in a node with blocking-after-service discipline a blocked job is waiting to go to the next node with the existing type MM1_loss, this node must fetch the job using the new interface operation I_take as soon as buffer space becomes free. Obviously, the new interface operation must be made available in the old node type.

A new interface type is realized as follows. Its name, say cause, is made known within the generator. In each model, the set $\mathcal{I}^{\mathsf{cause}}$ of interfaces which adopt the new interface feature is generated, initialized to be empty. If a declaration (k, k'): cause is encountered in a network description, the node pair is added to the set $\mathcal{I}^{\mathsf{cause}}$. For all nodes which may be the receiver of may_I_cause or I_cause, this new interface inquiry and operation is implemented.

Expressive Power

Here quite a general class of queueing networks is defined and it is shown that these networks can be described with the QN description language and, in turn, can be translated into transition class models. The only purpose of this class of queueing networks is to point out the expressive power of the QN description language and the tool, we do not recommend to use the generator for these general queueing networks which are defined only for theoretical purposes.

In such a queueing network with K nodes, each node k may be in one of a finite number of states, say $\mathcal{S}_k = [1..N_k],\ N_k \in I\!N$. Hence, the state space of the whole model is a subset of the cartesian product of the node state spaces, $\mathcal{Z}^T \subseteq \mathcal{S}_1 \times \ldots \times \mathcal{S}_K$.

The state transitions of the network may change the states of one or more nodes, may depend on the states of many nodes, may refer to the transitions of jobs from nodes to others, to splitting or joining of jobs, to the modification of job priorities, may change internal states of nodes like service time phases, served queues of multiqueue nodes, and so on. We allow all state changes which can be expressed in a Markov chain over the state space $\mathcal{Z}^T$. That is, the queueing networks class is as general as it can be expressed with such a Markov chain.

The following theorem states the universality of the generator.

Theorem 2. Any queueing network of the just defined general class of queueing networks can be described with the QN description language and can be translated into transition class models using the generator.

Sketch of proof. For each pair of (k, k') of different nodes, an interface of type change is provided with the following properties. If node k calls may_I_change(k'), the answer is the local state of node k'. Thus a node may come to know the other nodes states. Together with its own state, afterwards the node knows the whole networks state.

If a node k calls I_change(k', z), $z \in \mathcal{S}_{k'}$, the new state of node k' becomes z. Thus a node may change the states of any other node, and the node may change its own state immediately. Hence a node may change the state of the whole model.

For each transition of the Markov chain, a transition class TC is provided, as it is indicated in the proof of theorem 1. For each of these transition classes, a node is selected as owner. This is possible since the overall state is available in every node, and since every node is able to change the state.

The type of every node is defined by all the transition classes which the node owns. □

Mini Nodes

The introduction of a new node type into the generator is not very simple, quite a lot of programming work must be done. On the other hand, there is a very large variety of node types applied in queueing networks, differing with respect to the arrival processes, the waiting rooms, the service discipline, the service time distribution, the number of servers, the blocking discipline, and so on. By means of a smaller number of building parts which we call *mini nodes*, many different node types can be realized. Mini nodes are similar to nodes but less complex.

A mini node is an instance according to a type, may have attributes, may be the owner of elementary system components and of transition classes. Two mini nodes may be connected via an edge and may have a common interface.

Arrival processes are a class of mini nodes. An arrival process has an edge to another mini node, a queue. Arrival processes are active nodes, they have at least one transition class. An arrival can occur if in the connected queue buffer space is left. The node knows this via an interface inquiry may_I_send and sends the new job to the queue with the interface operation I_send. A Poisson arrival process, type M_arrival, has an attribute, namely the arrival rate, but no state, hence no elementary system component. A Markovian arrival process needs an elementary component for the internal states, has some attributes and transition classes for internal transitions without arrivals and a single transition class for arrivals; details are given in [24]. Markov modulated Poisson processes are similar to Markovian arrival processes. They are used for dependent arrivals. Mini nodes for arrivals can also provide bulk arrivals.

Waiting buffers (queues) are passive mini nodes, they do not have transition classes. Their states are modified with interface operations: Jobs are put into with I_send by other nodes or by arrival processes. Jobs which are ready served are fetched by a server with the operation I_take. The well-known simple queue of a MM1-node, for example, has a single elementary component which counts the number of jobs in the queue. The buffer capacity is the only attribute. We call the type of this queue simple_queue.

More complex are servers. They differ with respect to the service time distribution, the service discipline, the blocking discipline, they may have multiple service facilities.

The most simple server is of type M_FCFS_loss_server. It is state-less, the service time is exponential, the service rate is an attribute. The server works if in the waiting room is at least one job; the server knows this by means of the interface inquiry may_I_take. The transition classes model the completion of a service and the transition of the served job to a specific succeeding node (waiting room). The state transition in the queue where the job was during the service is caused by the interface operation I_take, and in the target queue by I_send. This last operation has no consequence if the queue is full - this means the job is lost.

Other service time distributions are approximated with phase-type distributions, see [24]. Here, an elementary component is used for the states of the inner Markov chain, and some transition classes perform the state transitions between these states without ending the service time.

Autocorrelated service times can be modelled with Markov modulated Poisson processes. The details are very similar to Markovian arrival processes.

In polling systems and priority nodes, the server is connected with some queues, and in general, the state of an elementary component indicates which of them is served at every moment. In priority nodes, every queue is dedicated to a priority class. A transition class for the end of a service moves the job out of the node and decides which is the next queue for service. The state is changed accordingly. The service rate, the number of queues, and transition probabilities are attributes.

Let us consider three special mini node types. Mini nodes of the first type only change job classes. Either, an incoming job receives a given class, or there is a function which calculates the new class from the old. The job leaves immediately to the next node, that means the node is immediate. Moreover, these nodes are stateless, they do not have components, and they are passive, they do not have transition classes. Predecessor nodes use the operation I_send for sending the jobs, and these nodes forward the jobs with I_send as well.

A mini node Fork of type fork_loss splits an incoming job into n jobs and sends each of them to a queue. If in one of the receiving queues not enough buffer space is available, the incoming job is lost and no splitted jobs are generated. This information is obtained with the inteface inquiry may_I_send. All transitions are effectuated by the operation I_send. These nodes are immediate, stateless, and passive. n is a attribute.

A mini node Join of type join_loss consists of n queues. Every time when in a queue a job arrives, the other queues are inspected if jobs are there. If no queue is empty, one job is taken from each, and a single job is sent to a successor node by means of the operation I_send. This job may be lost if the target node is full. These mini nodes are immediate and passive.

As an example, we combine some mini nodes to a fork-join system in which the jobs are duplicated:

```
(* Fork node *)
Fork: fork_loss
(   Multiplier = 2    (* Number of queues *)
    t[Q1] = 1         (* Transfer probability to node Q1 *)
    t[Q2] = 1         (* Transfer probability to node Q2 *)
)
(* Queue of the first subnode *)
Q1: simple_queue
(   nu = 3            (* Buffer capacity *)
    t[Server1] = 1    (* Transfer probability to the
                           server of the first subnode *)
)
Q2:   (* Queue of the second subnode, similar *)
```

```
(* Server of the first subnode *)
Server1: M_FCFS_loss_server
(   mu = 0.1        (* Service rate *)
    t[Join,1] = 1   (* Transfer probability to the join node,
                       local queue 1 *)
)
Server2:  (* Server of the second subnode, similar *)
(* Join node *)
Join: join_loss
(   Multiplier = 2  (* Number of local queues *)
    t[Fork] = 1     (* Transfer probability to the fork node *)
)
(* In a closed model, initially some jobs must be present *)
Initialize:
(   Q1.Length= 2    (* Number of jobs in the queue *)
    Q2.Length= 2    (* Number of jobs in the queue *)
)
```

In the last part of the example, the states of the queues Q1 and Q2 are initialized.

The reader may note that with the concept of extensibility new possibilities can be introduced, for example synchronization or simultaneous use of resources.

Submodels

Submodels are defined with descriptions of (sub-)queueing networks. These descriptions contain names of nodes which are not part of the submodel, i. e. *external names*. For example, a job may be sent from a submodel to an external node. External and internal nodes have common interfaces, and there may be interface feature declarations for them. On the other hand, internal node names can be exported, they can be external names in other submodels.

The development of a preprocessor which binds together submodels into complete models seems to be straightforward - we did not yet do this.

Mobile Components

Mobile components are planned. They are jobs with attributes which visit nodes like usual jobs. In the QN description language and in the generator, they are quite similar to nodes: Mobile components have names, attributes, they may possess elementary components and transition classes, they may have interfaces with other mobile components and with nodes. The nodes to which mobile components go are slightly different to other nodes: Their waiting rooms are not counters but real queues.

G. Wegener [31] developed a prototype of the generator which is able to generate queueing network models with nodes of three types: MM1_loss, MM1_PR_loss, and nodes with HOL priority scheduling. This generator is integrated into an experimental tool for the evaluation of Markov chain models which are given as transition class models.

CONCLUSION

Some ideas, we hope useful and novel, for a tool for Markov chain models were presented. Future research concerns the further development of the tool which supports the whole technique including modularization, mobile components, new solvers, methods for the analysis of qualitative properties of transition class models like safety properties, progress properties, and liveness, and a similar generator for stochastic Petri nets is being developed. An interesting question is whether the transition classes can be used to automatically detect some special structure, for example a matrix-analytic structure of the Markov chain or a product-form solution.

Acknowledgement

The author would like to thank Guido Wegener for valuable discussions about the generation of transition classes from a formal queueing network description and for the development of the prototype generator, and unknown referees for helpful remarks and interesting suggestions.

References

[1] F. Bause, P. Buchholz, and P. Kemper. QPN-tool for the specification and analysis of hierachically combined queueing Petri nets. In H. Beilner and F. Bause, editors, *Quantitative Evaluation of Computer and Communication Systems*, volume 977 of *LNCS*, pages 224–238. Springer, Berlin, Heidelberg, New York, 1995.

[2] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains, Modelling and Performance Evaluation with Computer Science Applications*. John Wiley and Sons, New York, 1998.

[3] P. Buchholz. Markovian process algebra: Composition and equivalence. In *Proc. 2nd Workshop on Process Algebras and Performance Modelling*, pages 11–30, Erlangen, 1994.

[4] G. Chiola. GreatSPN User Manual; Version 1.3. Technical report, Dipartimento di Informatica Università di Torino, 1987.

[5] G. Chiola. GreatSPN 1.5 Software Architecture. In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation*, pages 121–136. Elsevier, Amsterdam, 1992.

[6] G. Ciardo, J. Muppala, and K. S. Trivedi. SPNP: Stochastic Petri net package. In *Proc. PNPM'89*, pages 142–151. IEEE Computer Soc. Press, 1989.

[7] P. J. Courtois. *Decomposability, queueing and computer system applications*. Academic Press, London, 1977.

[8] J. Couvillion, R. Freire, R. Johnson, W. D. Obal, M. A. Qureshi, M. Rai, W. H. Sanders, and J. E. Twedt. Performability modelling with UltraSAN. *IEEE Software*, 8:69–80, 1991.

[9] W. J. Stewart (ed.). *Numerical Solution of Markov Chains*. Marcel Dekker, New York, Basel, Hong Kong, 1991.

[10] W. J. Stewart (ed.). *Computation with Markov Chains*. Kluwer, Boston, 1995.

[11] S. Gilmore and J. Hilston. The PEPA workbench: A tool to support a process algebra based approach to performance modelling. In G. Haring and G. Kotsis, editors, *Computer Performance Evaluation, Modelling Techniques and Tools*, volume 794 of *LNCS*, pages 121–146. Springer, Berlin, Heidelberg, New York, 1994.

[12] N. Götz, H. Hermanns, U. Herzog, V. Mertsiotakis, and M. Rettelbach. *Quantitative Methods in Parallel Systems, chapter Stochastic Process Algebras – Constructive Specification Techniques Integrating Functional, Performance and Dependability Aspects*. Springer, Berlin, 1995.

[13] B. R. Haverkort. Matrix-geometric solution of infinite stochastic Petri nets. In *Proc. of the 1st International Computer Performance and Dependability Symposium*, pages 72–81. IEEE Computer Society Press, 1995.

[14] B. R. Haverkort and I. G. Niemegeers. Performability modelling tools and techniques. *Performance Evaluation*, 25:17–40, 1996.

[15] B. R. Haverkort, I. G. Niemegeers, and P. Veldhuyzen van Zanten. DyQNtool - a performability modelling tool based on the dynamic queueing network concept. In G. Balbo and G. Serazzi, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, pages 181–195. North-Holland, Amsterdam, 1992.

[16] Boudewijn R. Haverkort. *Performance of Computer Communication Systems*. John Wiley and Sons, Chichester, 1998.

[17] H. Hermanns and V. Mertsiotakis. A stochastic process algebra-based modelling tool. In *Proc. 10th Workshop for Performance Engineering of Computer and Communication Systems*, Berlin, Heidelberg, New York, 1995. Springer.

[18] J. Hilston. Compositional Markovian modelling using a process algebra. In W. J. Stewart, editor, *Computations with Markov Chains*, pages 177–196, Boston, 1995. Kluwer Academic Publishers.

[19] P. J. Kühn. Multiqueue systems with nonexhaustive cyclic service. *Bell Syst. Tech. J.*, 58:671–699, 1979.

[20] Christoph Lindemann. *Performance Modelling with Deterministic and Stochastic Petri Nets*. John Wiley and Sons, Chichester, 1998.

[21] R. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach using the SHARPE Software Package*. Kluwer Academic Publishers, Boston, 1996.

[22] W. J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[23] J. Ch. Strelen. A fast simultaneous iteration technique for the analysis of Markov chains. Interner Bericht II/89/1, Institut für Informatik, Universität Bonn, 1989.

[24] J. Ch. Strelen. Approximate analysis of queueing networks with Markovian arrival processes and phase type service times. In K. Irmscher, Ch. Mittasch, and K. Richter, editors, *Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen*, pages 55–70, Berlin, Offenbach, 1997. VDE-Verlag.

[25] J. Ch. Strelen. Approximate disaggregation-aggregation solutions for general queueing networks. In A.R. Kaylan and A. Lehmann, editors, *Proc. of the ESM 97*, pages 773–778. Society for Computer Simulation, 1997.

[26] J. Ch. Strelen. Approximate product form solutions for Markov chains. *Performance Evaluation*, pages 87–110, 1997.

[27] J. Ch. Strelen. Loss queueing networks with bursty arrival processes and phase type service times: Approximate analysis. In D. Kouvatsos, editor, *Fifth IFIP Workshop on Performance Modelling and Evaluation of ATM Networks - Participants Proceedings*, pages 87/1–87/10. University of Bradford, 1997.

[28] J. Ch. Strelen. Monte-Carlo simulation of Markov chains using a high-level modelling technique. In A. Bargiela and E. Kerkhoffs, editors, *Simulation Technology: Science and Art - Proceedings of the ESS 98*, pages 213–217. Society for Computer Simulation, 1998.

[29] J. Ch. Strelen. Generation of transition class models from formal queueing network descriptions. Appears in the proceedings of ESS2000, 2000.

[30] J. Ch. Strelen, B. Bärk, J. Becker, and V. Jonas. Analysis of queueing networks with blocking using a new aggregation technique. *Annals of Operations Research*, 79:121–142, 1998.

[31] G. Wegener. Bemerkungen zu MCTools. Universität Bonn, 1999.

[32] W. Whitt. The queueing network analyzer. *The Bell System Technical Journal*, 62(9), 1983.

Johann Christoph Strelen was born in Wiesbaden, Germany, in 1941. He received the Dipl.-Math. and Dr. rer. nat. degrees in mathematics and the Habilitation degree in Computer Science from the Technische Hochschule Darmstadt, Germany, in 1968, 1973, and 1981, respectively. There he was affiliated to the Computing Center (1968-1973), and assistant at the Computer Science Department (1974-1982). In 1973 he was a post doctoral fellow at the IBM Scientific Center, Grenoble, France. Since 1982 he has been a Professor of Computer Science at the Rheinische Friedrich–Wilhelms–Universität Bonn, Germany. His reseach interests include performance evaluation, distributed systems, and simulation. Dr. Strelen is a member of the Gesellschaft für Informatik (GI).