Quantitative analysis of software architectures

Simonetta Balsamo*, Marco Bernardo• and Vincenzo Grassi°

* Dipartimento di Informatica, Università Ca' Foscari di Venezia - Italy
• Centro per l'Applicazione delle Scienze e Tecnologie dell'Informazione, Università di Urbino - Italy
° Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata" - Italy

Abstract

Quantitative analysis of software systems is a critical issue in the development of applications for heterogeneous distributed and mobile systems. It has been recognised that performance analysis should be integrated in the software development life cycle since the early stages. We focus on quantitative analysis of software architectures (SA) and in particular on performance models and languages to represent, evaluate and predict performance characteristics in the software development life cycle. We consider SA to describe both the structure and the behavior of software systems at a high level of abstraction, and we present and compare some different approaches and models for quantitative analysis of SA. We consider both approaches based on the study and definition of languages and notations to formulate quantitative requirements early in the software life-cycle, and methods that identify and develop performance evaluation models that can offer an high degree of integration with functional analysis models. The proposed approaches are based on Stochastic Process Algebras, Queueing Network Models and their extensions, and Markovian models, respectively. They allow the definition and modelling of various characteristics of distributed and mobile software systems and the derivation of appropriate quantitative models to evaluate significant system performance indices. We present a discussion of the relative merit of the various approaches and open problems of ongoing research in the field. The presented approaches have been developed in the framework of the Italian National Research project SALADIN on Software Architecture and Languages to coordinate Distributed Mobile components.

1. Introduction

Quantitative analysis of software systems is a critical issue in the development of applications for heterogeneous distributed and mobile systems. In the last decade it has been recognised that quantitative analysis and specifically performance analysis should be integrated in the software development life cycle since the early stages. We focus on quantitative analysis of software architectures (SA) and in particular on performance models and languages to represent, evaluate and predict performance characteristics in the software development life cycle.

Recently the software architecture description of a software system has received much interest as a potentially good candidate to conduct early predictive performance analysis of the system under development [BCK98]. We consider SA to describe both the structure and the behavior of software systems at a high level of abstraction, and we present and compare some different approaches and models for quantitative analysis of SA. We consider both

approaches based on the study and definition of languages and notations to formulate quantitative requirements early in the software life-cycle, and methods that identify and develop performance evaluation models that can offer an high degree of integration with functional analysis models. The proposed approaches are based on different formalism including Stochastic Process Algebras, Queueing Network Models and their extensions, and Markovian models. They allow the definition and modelling of various characteristics of distributed and mobile software systems and the derivation of appropriate quantitative models to evaluate significant system performance indices.

The approach based on a Queueing Network model proposes a methodology to derive the performance model from a SA formal specification. The method is based on the analysis of the SA dynamic behaviour and it can be considered independent from the specific model. Then the obtained queueing network is parameterised and evaluated by a scenario based technique. The approach based on Stochastic Process Algebras offers a natural integration between the behavioural model and the performance evaluation model. It is based on a developed tool called *Æ*milia and a design performance evaluation method of SA specified using *Æ*milia. This allows the formalisation of architectural styles in an operational framework suited to already existing architectural description languages, so that it is possible to investigate properties that are common to all the instances of an architectural style. The approach based on a Markovian models is targeted at distributed applications deployed in wide area or physically mobile environments, where we consider different interaction styles for such applications, including the "location unaware" and "location aware" styles. The methodology derives from different specification formalisms a Markovian models and provides insights about whether interaction styles have a positive impact on some efficiency indexes such as the generated network traffic or the energy consumed by portable devices that support a given application.

In this paper we present, discuss and compare these various models and languages for quantitative performance analysis of software architectures, and we focus on the relative merit of the various approaches and open problems of ongoing research in the field.The presented approaches have been developed in the framework of the Italian National Research project SALADIN on Software Architecture and Languages to coordinate Distributed Mobile components [I02].

The paper is organized as follows. Section 2 introduces performance models and languages of software architectures, and presents three different approaches and methodologies respectively based on Queueing Network models, Stochastic Process Algebras and Markov reward/decision processes. Section 3 presents the advantages and drawbacks of the approaches and discusses the open problems and perspectives. Conclusions are given in Section 4.

2. Software architecture performance models and languages

Different approaches and methodologies have been recently proposed to integrate quantitative performance analysis and evaluation in the software development life cycle, starting from SPE, the Software Performance Engineering methodology introduced by Smith [S90, SW02]. Various specification languages and different performance models and methods have been proposed [WOSP00, WOSP02].

We focus on performance analysis of Software Architectures and we present three different approaches.

The first approach, presented in Section 2.1, is based on a Queueing Network model and proposes a methodology to derive the performance model from a SA formal specification. The method derives the basic or extended Queueing Network by analysing the SA dynamic behaviour, independently of the specific model. Then the parameterised Queueing Network is evaluated by considering different scenarios.
The second approach, presented in Section 2.2, is based on Stochastic Process Algebras that integrate the behavioural and the performance evaluation model. The performance evaluation method of design of SA is specified using the developed tool called Æmilia. The formalisation of architectural styles in an operational framework allows investigating properties of architectural styles.
The third approach, presented in Section 2.3, is based on Markov reward/decision processes and provides models for distributed applications in wide area or physically mobile environments, with different interaction styles. The methodology derives a Markov model from different specification formalisms and evaluates interaction styles efficiency considering the generated network traffic or the energy consumption.

2.1 Software performance modeling based on queueing networks

Queueing network models (QN) have been extensively applied in the last decades as a powerful tool for modeling and performance evaluation and prediction of computer systems [K76, L83, J90]. QNs have been used in the Software Performance Engineering methodology (SPE) by Smith to evaluate software systems on an execution platform [S90, SW02]. More recently various approaches have been proposed to derive QN models from software specification [WOSP00, WOSP02, BS01b, CM00].
We shall now present a software performance modeling approach based on QNs. The proposed software performance evaluation methodology applies at the SA level and derives a performance evaluation model, based on a QN model, from a SA formal specification [BIM98, ABI01]. Although different models for describing the dynamic behaviour of a SA [CIW99, BIM98, AABI00b] have been used, the approach can be considered independent from the specific model and can be summarised as follows. The dynamic behaviour of a SA is examined in order to define the QN model representing the SA behaviour. Then the parameters of the QN are defined by considering the SA or various competing SAs under different scenarios and the QN model is evaluated to derive performance indices. The goal of the proposed approach is to provide a quantitative system performance analysis of two or more SAs, even at their high level of abstraction. From our perspective of software architectures, QN models provide a powerful tool that can be defined, parameterized and evaluated at a low cost and with a level of abstraction that allows a faithful modeling.
Informally, a QN model is defined by the service centers the customers and network topology. Service center characteristics include the service time, the buffer space with its queueing scheduling and the number of servers. The buffer space of each service center can be finite to represent finite capacity system resources or population constraints. Customers are described by their number for closed models and by the arrival process to each service center for open models, the service demand to each service center and the types of customer. Network topology models how the service centers are interconnected and how the customers move between them. Different types of customer in the QN can model different behaviors of the customers, i.e., various types of external arrival process, different service demands and different types of network routing.

In the proposed approach we consider performance modelling at the SA abstraction level. We do not assume to have further information on the system under development from the subsequent steps, such as design, implementation and deployment, and we evaluate concurrent execution of SA to validate possible critical design choices at this level.
SAs represent high level system description in terms of subsystems (*components*) and the way they interact (*connectors*) and they provide two type of descriptions: a static one modelling the topology of the system, and a dynamic one modelling the way components can interact when the system is in execution. We consider the latter for software performance evaluation, since the components dynamic behavior and interaction among each other determines the values of the performance indices. The SA static description, on the contrary, provides information useful to define the structure of the QN performance model. Intuitively, there is a natural correspondence between a SA component and a QN service center, and its precise definition will depend on the way the component interacts at execution time with the rest of the system.
The SA dynamic behavior can be modelled by different formalisms, such as finite state machines, Petri Nets or Labelled Transition Systems (LTSs). The proposed approach does not refer to a specific architecture description language, rather it will directly refer to the system architecture dynamic model that we assume to be a LTS that are widely used in system specification and SA modelling [AG97]. We assume to obtain the LTS model out of a SA description in which components are modelled as communicating concurrent subsystems, and the system state is obtained by observing the concurrent behavior of all the subsystems.
By analysing the SA dynamic behavior it is possible to single out the real degree of concurrency and synchronisation among components and this information allows for the generation of meaningful (i.e. faithful with respect to the SA description) QN performance models. For example, QNs with infinite capacity queues naturally model SAs in the two following cases: sequential components with synchronous communication, and concurrent components with asynchronous communication through buffers.
In this case we can obtain simple product form networks that can be efficiently analyzed [K76, L83, J90]. More precisely, a software component or a set of components can be represented with a simple server or complex server. A complex server represents the service given by the associated components, where the associated service time is the summation of the single service time associated to each component.
On the other hand if we need to model software architectures with concurrent components and synchronous communication, since the class of QN models with infinite capacity queues is not sufficiently expressive, QN with blocking as performance models can be used [BDO01, AABI00a]. This allows to model more accurately interaction capabilities of SAs by capturing some features of the communication systems, because QNs with finite capacity queues and blocking represent some synchronization constraints. Specifically, synchronous communication of concurrent components can be modeled by service centers with finite capacity queues and an appropriate blocking protocol, as in QNM with *BAS* blocking [AABI00a].

2.1.1 Deriving Queueing Network models from Software Architecture behavior

The scheme of the software performance approach based on QNs is illustrated in Fig. 1. Starting with a SA description, from its behavioral model, given by the LTS, an algorithm derives a QN as performance model. The complete definition of the QN is also based on on some additional information about the state annotation and the type of communication among SA components. Then performance evaluation of the QN is carried on by considering

parameter instantiation determined by possible scenarios. The obtained performance results of the QN are interpreted at the SA level to derive feedback on the SA development process.

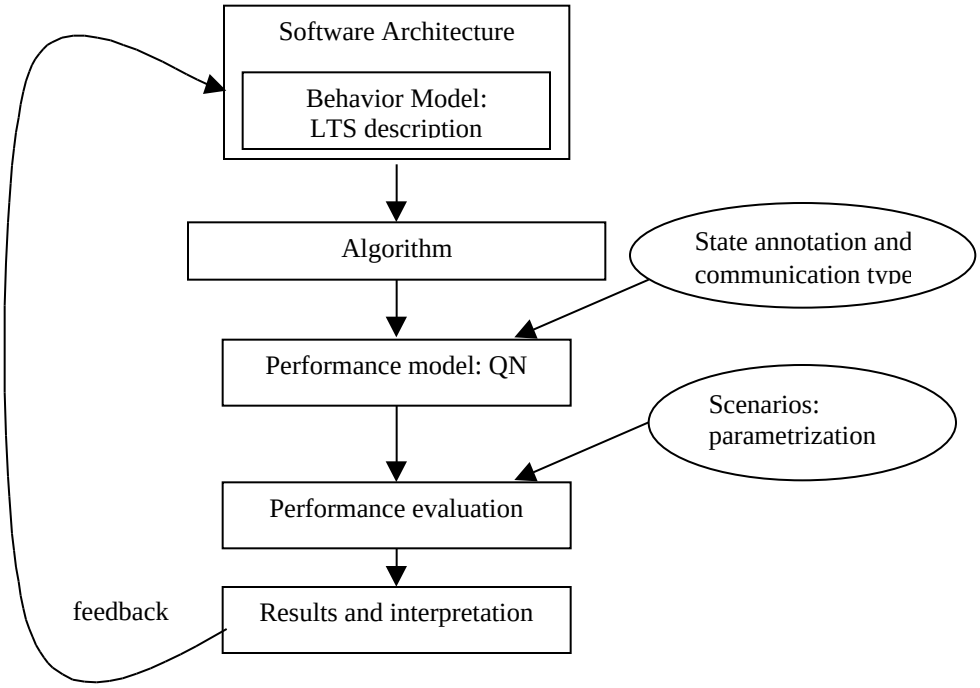


Figure 1. Scheme of the approach based on QNs

We shall now describe the three main phases: the algorithm to derive the QN from the SA specification, the QN performance evaluation and the feedback analysis.

Algorithm to derive the QN from the SA

The definition of a QN model can be split in three steps:

1. *definition*, i.e. definition of service centers, their number, class of customers and topology
2. *parameterization*, to define the alternatives of the study, e.g. by selecting the arrival processes and service rates
3. *evaluation*, to obtain a quantitative description of the system behavior described by the QN, i.e. computation of a set of figures of merit or performance indices, such as resource utilization, system throughput and customer response time. These indices can be *local* to a resource or *global* to the entire system.

The algorithm to derive a QN model from a SA is based on the analysis of the LTS that represents the dynamic behavior of the SA. Without loss of generality, we assume to deal with a LTS in which the transitions corresponding to non-deterministic behavior are marked in order to distinguish interactions corresponding to an actual non-deterministic choice in the computation and those corresponding to concurrent communications that are modeled in the LTS as non-deterministic computations. This can be easily obtained when building the LTS from the system description.

Informally, in order to define the QN from the LTS we represent the SA components and connectors by service centers of the QN, and the interactions among system components with the QN topology.

The algorithm is organized into two sequential steps. The first step (*PHASE 1*) examines all the LTS paths and the states along the path in order to single out all the pairs of interacting components. The second phase (*PHASE 2*) analyzes and compares these interaction pairs to derive the service centers and the network topology of the evaluation model.

Le P denotes the set of SA components and connectors whose dynamic behavior is described by the LTS. Let $G=(S,A,T)$ denote the labeled directed graph representing the LTS, where $S=\{S_1,\ldots,S_N\}$ is the set of states, $A=\{a_1, \ldots , a_N\}$ is the set of directed arcs, $a_i =(S_j,S_k)$, S_j,S_k S , $T=\{T_1, \ldots , T_M\}$ is the set of labels that represent the communications and T_i is the transition label of arc a_i.

Transitions in the LTS describe explicit interactions between system elements or implicit interactions between some system components and the environment or external components. Hence we define a partition of T in two sets of internal and external transitions, denoted by *IT* and *ET*, respectively. When there are external interactions, i.e. $ET \neq$, we represent the environment with a new element called *ENV* in P. For systems with only internal interactions we usually derive a closed QN, otherwise we obtain an open network. The algorithm derives the QN that represents the real level of concurrency of the system. The goal is to identify the components that are strongly synchronized so that their behavior is sequential and the components that are independent and can be concurrently active.
First, we consider each component as an autonomous server, thus modeling the maximum level of concurrency. Successively we simplify the QN structure by analyzing the true level of concurrency and the communication types.

The algorithm can be sketched as follows. For a detailed description of the method and the algorithm see [ABI01].

1) *PHASE 1*
Analysis of the LTS.

1.1 Perform a LTS visit by considering all the paths in graph G to derive interaction sets formed by interaction pairs.
- Examine all the adjacent states of a visited state and the connecting transitions. If the transition corresponds to a communication the algorithm generates a pair, called *Interaction Pair (IP)*, denoted by (p_1, p_2) that signals a flow of data elements from p_1 top_2, where p_1 is the system component that acts as sender in the communication and p_2 is the component that acts as receiver.

1.2 For synchronous communication, where p_1 starts after the completion of p_2, both elements of the pair are system components, while asynchronous communication is modeled by introducing connecting elements with buffer, called *Passive Connecting Element (PaC)*, that are elements of the corresponding interaction pair.

1.3 Mark the interaction pairs corresponding to a system non-deterministic behavior, so to correctly distinguish, and model, concurrency and non-determinism, in the second phase.

1.4 Put all the interaction pairs derived at the *i*-th visit step in *Interaction Set* I_i, that represents all the communications among system elements that can happen at a given time.

2) *PHASE 2*

In order to derive the QN model of the SA described by the LTS, we examine the obtained *Interaction Sets* to associate elements of the QN.

2.1. Examine *Interaction sets* I_i, defined in *Phase 1* to generate the service centers and the topology of the QN.

- Besides the element *ENV P* representing the environment, distinguish the system components that are represented as internal or external elements of the QN. Hence identify a subset *A* of set *P*, that contains those SA elements to be represented as external environment in the QN.
- Elements in *A* can be seen as sources which model the production of system customers or elements from which the system communicates the results to the environment. If the element *ENV* has been introduced, then it belongs to set *A.* From *A* we can determine whether the QN is open or closed.
- The analysis of the interactions among system components, i.e. interaction sets I_i, allows identifying their *real* level of concurrence. We identifies components that are strongly synchronized and have a sequential behavior, even if they are modeled in the SA description as independent entities, and independent components that can be concurrently active. Each component is first considered as an autonomous server, thus modeling the maximum level of concurrency, then it can become part of a more structured QN element.

2.2 If an *interaction pair* (p_1, p_2) corresponds to a synchronous communication, then define a corresponding QN element that is a complex server with a unique service composed of p_1 followed by p_2 to represent the sequence of operations.

- If an *interaction pair* (p_1, p_2) corresponds to an asynchronous communication, then the component that receives data is modeled as a service center with a infinite buffer that implicitly models the communication channel. Moreover from this interaction pair we can build the customer transition in the QN.
- The *interaction pair* (p_1, p_2) with an external element, i.e. where p_1 or p_2 belong to *A*, corresponds either to a service center with exogenous arrivals (if $p_1 \quad A$) or from which there are departures (if $p_2 \quad A$).

2.3 To model synchronous communication among concurrent system components assign distinct service centers to the communicating components to model their independence, associate to the receiver component a service center with a zero capacity buffer and assume *BAS* blocking mechanism for the sender component.

- To model a non-deterministic computation introduce multi-customer service centers that at the end are transformed in simple-customer service centers whose service times depend on the service times of the original classes.
- To model one to many and many to one communications assign to the involved components distinct service centers, with a zero capacity buffer if the communication is synchronous.

2.4 When each interaction set I_i has been examined perform *merging* operations to reach the final configuration of the service centers. Several merging operations reduce the number of service centers of the QN to model only the necessary concurrency of the system

components. The result of these merging operations is the final set of interconnected service centers of the QN.

Modeling synchronous communication by QN with finite capacity

As defined at step 2.3 of the algorithm, we model synchronous communication of concurrent components by service centers with finite capacity queues and an appropriate blocking protocol. In QN with finite capacity queues, when a queue reaches its maximum capacity then the flow of customers into the service center is stopped, both from other service centers and from external sources in open networks, and the blocking phenomenon arises. Various blocking mechanisms have been defined and analyzed and we can model synchronous communication with the type called Blocking After Service (*BAS*) [BDO01, AABI00a]. Under this policy when a user, after having completed the service at a service center *i*, tries to enter a saturated node *j*, it is forced to wait at node *i*, till the destination node *j* can be entered. The server of source node *i* stops processing jobs (it is blocked) until destination node *j* releases a job. When more than one node is blocked by the same saturated node, a scheduling discipline must be considered to define the unblocking order of the blocked nodes. We refer to FBFU (First Blocked First Unblocked) discipline, where the first node blocked is the first unblocked one.

To model synchronous communication among concurrent system components, we assign distinct service centers to the communicating components in order to exploit component concurrency in the SA. We model such components as service centers, we associate to the receiver component a service center with a zero capacity buffer and impose a *BAS* blocking mechanism to the sender component in order to model synchronization. Specifically we model the components that can receive a synchronous communication with a finite capacity service center with one server and where the buffer capacity is set to one. That is, we model a component C_i with a single server service center i, where we allow no queueing by setting B_i=1 as the queue capacity, i.e. the maximum number of customers admitted at service center i, that is in the queue and in the server.

Example. For example consider the model of component C_1 that tries to communicate with component C_2 which is active on some operations; then component C_1 becomes blocked until component C_2 is free and can receive information from C_1. Synchronous communication between the two components C_1 and C_2 is modeled by associating a service center to each component, say S(C_1) and S(C_2), respectively, as illustrated in Fig.2.

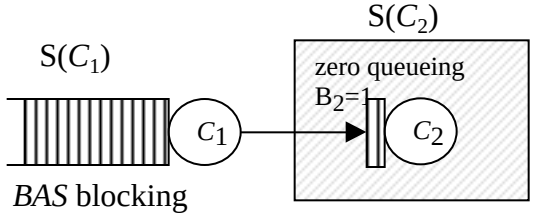


Figure 2. Model of synchronous communication between concurrent components

Then S(C_2) has single capacity queue and S(C_1) has *BAS* blocking mechanism. Hence we can describe more complex contexts where the components C_1 and C_2 are simultaneously

active, but also situations in which C_1 attempts to communicate with C_2, when the latter is still working. Indeed *BAS* mechanism allows blocking the component C_1 waiting for C_2 to complete its service. By setting in S(C_2) finite capacity B_2=1, then it can receive service requests from S(C_1) only when its server is not occupied. When S(C_2) is full, if S(C_1) at the completion of its service attempts to send a customer (a request) to S(C_2), then S(C_1) is blocked until a departure (service completion) occurs from S(C_2), according to *BAS* definition. This corresponds to the system behavior that we want to represent in the performance model.

Consequently we model one to many and many to one communications by assigning to the involved components distinct service centers, with a single capacity queue if the communication is synchronous. One to two and two to one communication models for synchronous communication is illustrated in Figure 3. Service center S(C_1) in Fig. 3a, S(C_2) and S(C_3) in Fig. 3.b can have in turn finite capacity respectively, if C_1, C_2 and C_3 are also destination components of a synchronous communication.

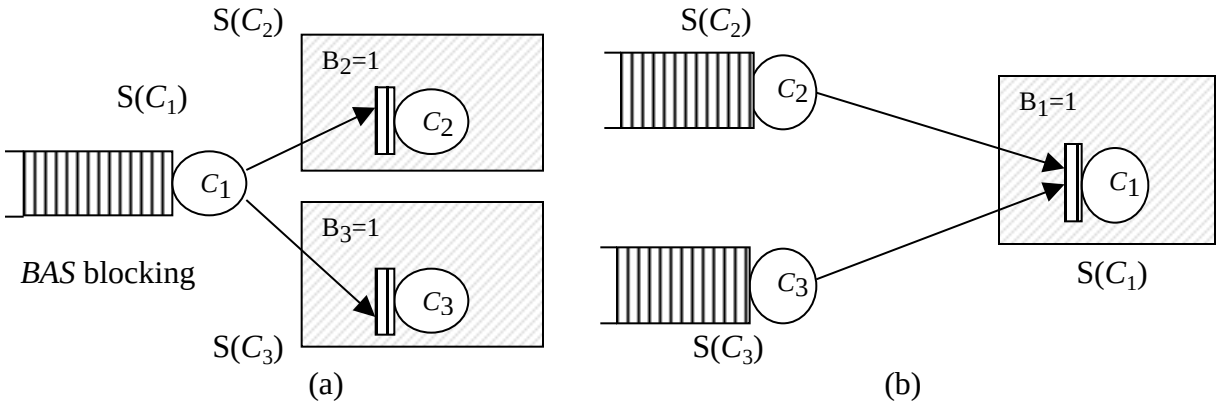


Figure 3. Models of one to two (a) and two to one (b) synchronous communication between concurrent components.

In two to one synchronous communication we have to assume a scheduling of the communication requests (the customers in the QN) arriving at C_1 from C_2 and C_3 in Fig. 3.b. This corresponds to the definition of the unblocking scheduling in *BAS* blocking definition. First Blocked First Unblocked scheduling maintains the order of communication request times.

A different case is the two to one communication where there is a synchronous and an asynchronous communication. If component C_2 in Fig. 3b is a connection element, the corresponding service center that models incoming asynchronous communication has infinite capacity queue, and the customers of the QN waiting in the infinite queue represent the communication arriving requests. When they are blocked in the service center by a finite capacity destination node, this represents the attempt to send a request to a busy destination component. Finally note that when both components C_2 and C_3 are connection elements, we have asynchronous communication to component C_1 that can be modeled by an infinite capacity service center S(C_1). A similar case is when component C_1 is a connection element.

Hence, with this modeling approach we can complete the definition of the QN model as concerns the communication among components. We introduce finite capacity service centers

with single queue capacity and *BAS* blocking only for those components that use synchronous communication.

2.1.2 Performance evaluation of the Queueing Network model

From the algorithm sketched in the previous section we derive a QN from a SA description based on LTS, but we do not obtain a completely specified model, because of the high level of abstraction of the SA model. In order to solve the performance model we have still to perform the parameterization step of the modeling process, i.e. we have to define the following parameters:

- the distributions characterizing the service times
- the customer arrival process for every service center
- the network routing probability.

The complete specification of the QN has to be done by the designer according to the system requirements and by considering the specific class of models. It is important to select the quantitative parameters so that the resulting QN belongs to a class that allows an efficient solution method.

We assume that the system reaches a steady-state behavior that is a stationary condition holds, and we analyze the steady-state system model. The analysis technique of the obtained QN depends on its characteristics. Under general assumptions it can be reduced to the analysis of the underlying stochastic Markov process, but under some constraints the QN analysis can be greatly simplified, such as for the class of *product-form* QNs that can be efficiently analyzed [K76, L83, J90]. Hence we usually try do derive models that belong to this class of QNs.

The analysis of QNs with finite capacity and blocking is in general more complex than basic QNs. Such models have a product form solution only under particular constraints also depending on the blocking type, and they can be in general analyzed by approximate analytical methods or simulation [L83, BDO01]

Parameter instantiations identify potential implementation scenarios and the evaluation process is carried out possibly by symbolic analysis. The system designer defines the implementation scenarios that may be determined also by the knowledge or assumptions on the final implementation platform.

This system evaluation and comparison of the performance results obtained by the analysis of the QN under various scenarios can provide useful insights on how to carry on the development process in order to satisfy a given performance constraint.

2.1.3 Performance analysis feedback

After the evaluation of the QN of the SA the proposed approach provides a set of results in terms of performance indices for each set of parameters that represent a possible scenario.

Hence an important and critical issue of the software performance methodology is the feedback analysis, that is the designer has to interpret these quantitative results as feedback at the SA design level, according to the framework shown in Figure 1. Such results may suggest confirming or modifying architectural choices at the design level according to given performance requirements or criteria.

Therefore it is important to provide a clear and direct correspondence between the SA design and the derived performance model. The high level of abstraction of the QN model allows defining a clear correspondence between SA components and model components, so making easier the feedback process. In our opinion this is in general an important feature of software quantitative performance that have to be considered in each evaluation approach.

In the following we illustrate the approach with a simple example.

2.1.4 Example

We now illustrate the presented approach of software performance model based on QN with finite capacity with a simple example of software architecture. Consider the SA for the design of a Compressing Proxy system as described in [CIW99], with four components as illustrated in Fig. 4, where components are denoted by square boxes and processes by ovals. Such system is introduced with the purpose of improving the performance of Unix-based World Wide Web browsers over slow networks by an HTTP server that compresses and uncompresses data to and from the network.

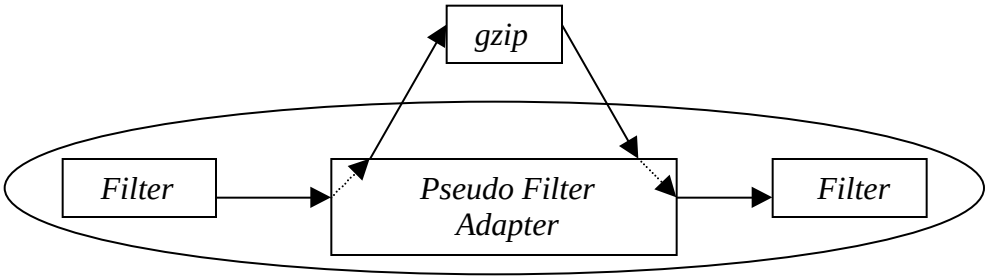


Figure 4. The Compressing Proxy SA

The filters communicate using a function-call-based stream interface. A filter *F* is said to read data whenever the previous filter in the series invokes the proper interface function in *F*. The interface also provides a function to close the stream. The *gzip* program is also a filter, but at the level of a UNIX process, so using the standard UNIX input/output interface. Communication with *gzip* occurs through UNIX pipes. A difference between UNIX filters, such as *gzip* and the HTTP filters is that the formers explicitly choose when to read, whereas the latter are forced to read when data are pushed at them**.** To assemble the Compressing Proxy from the existing HTTP server and *gzip* without modification, we must create an adapter. This acts as a pseudo HTTP filter, communicating with the upstream and downstream filters through a function-call interface, and with *gzip* using pipes connected to a separate *gzip* process that it creates.
From the LTS describing the SA by the algorithm sketched in Section 2.1.1, we can obtain a queueing network model that represents the Compressing Proxy system. We assume synchronous communication. By applying the algorithm we partition the four components into internal and external elements, defining sets *IT* and *ET,* and we introduce element *ENV*. From the LTS analysis we derive the *Interaction Pairs* and the *Interaction Sets*, as described in *PHASE I*. Then by examining the *Interaction Sets* we derive the service centers and the structure of the QN. By considering the modeling of the communication system and the merging operations we eventually obtain the QN model with finite capacity and *BAS* blocking illustrated in Fig. 5. The model is an open two node network with finite capacity queues B_{AD}=1 and B_{GZIP}=1. We have external arrivals and departures only from the *AD* service center (*Adapter*). We assume FIFO service discipline. The service center *AD* represents components *Adapter* and *Filter*, the service center *GZIP* represents component *gzip* of the software architecture. Specifically, from the analysis we obtain the tuple [*FILTER1, AD, GZIP, AD, FILTER2*] that characterizes the service and it states that a

customer requires services to the processing elements in that order. The first element of the tuple is an external element because the network is open.
Synchronous communication between *Adapter* and *gzip* is modeled by the finite capacity service center B_{GZIP}=1 and *BAS* blocking for the sender service center *AD*. Similarly, the synchronous communication between the two concurrent component *gzip* to *Adapter*, i.e. from the former to the latter, leads to the finite capacity service center B_{AD}=1 and *BAS* blocking for server *GZIP*. The network routing chain definition derives from the component interactions.

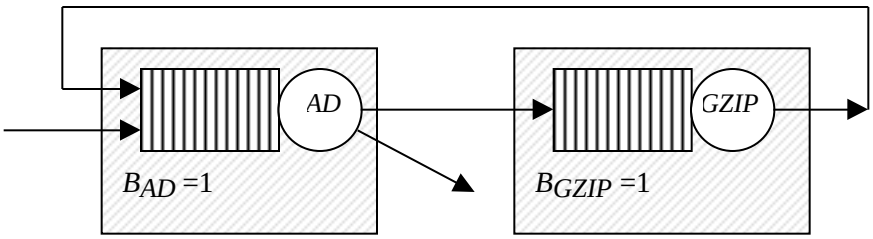


Figure 5. The queueing network model of the Compressing Proxy system.

A customer arrival at the network represents the arrival of data at the first filter in the SA. The queueing network model represents at the architectural level the interaction and potential concurrency of the Compressing Proxy SA.
Hence we obtain a simple two node queueing network model with finite capacity and *BAS* blocking that can be solved with exact analysis based on the underlying Markov process to derive the steady-state joint queue length distribution from which one can evaluate a set average performance indices. In particular, under the constraint of exponential service time distribution, a closed-form solution of the stationary probabilities can be derived [BDO01].
We can solve such queueing network model for different values of the network parameters, so comparing and predicting the performance of the Compressing Proxy system under various scenarios. This can provide useful insights on the design process development as concerns the meeting of quantitative performance requirements.

2.2 Stochastic Process Algebra based Architectural Description Languages

Stochastic process algebra (SPA) [HI96, HE98, BB00] is a compositional specification language of algebraic nature that integrates process algebra theory [M89, HO85, BW90] and stochastic processes. SPA allows for the functional verification and performance evaluation of formal models of concurrent and distributed systems through standard techniques like equivalence/preorder checking, model checking, reward based Markovian analysis, and simulation [CPS93, CGP99, BKH, S94, HO71, J90, CGH99, BB02]. The modeling of complex systems is carried out with SPA through some algebraic operators, which assemble the descriptions of individual subsystems into a single system description. Among such operators, we mention in particular the parallel composition operator, which regulates the cooperation within a family of subsystems through suitable sets of synchronizing activities.

Despite of its compositionality, SPA is not suitable to work with at the architectural level of design because, as it is, it does not result in a sufficiently easy and controlled way of

describing systems. As an example, if a system is made out of a certain number of components, with SPA the system is simply described as the parallel composition of a certain number of subterms, each representing the behavior of a single component, with suitable synchronization sets to represent the component interactions. It is desirable to be able to describe the same system at a higher level of abstraction, where the parallel composition operators and the related synchronization sets are transparent to the designer. It is more natural to separately define the behavior of each type of component, to indicate the actions through which each component type interacts with the others, to declare the instances of each component type that form the system, and to specify the way in which the interacting actions are attached to each other in order to make the component instances interact. This view brings the advantage that the system components and the component interactions are clearly elucidated, with the synchronization mechanism being hidden. Another strength is the capability of defining the behavior - possibly parametrized w.r.t. action rates and the interactions of a component type just once and subsequently reusing it as many times as there are instances of that component type in the system. Additionally, it is desirable that composite systems can be described in a hierarchical way, and that a graphical support is provided for the whole modeling process.

Besides this useful syntactical sugar, in the SPA framework checks are needed to detect possible mismatches when assembling components together and to identify the components that cause such mismatches. A typical example is deadlock freedom. If we put together some components that we know to be deadlock free, we would like that their combination is still deadlock free. In order to investigate that, we need suitable checks that allow deadlock to be quickly detected and some diagnostic information to be obtained for localizing the source of deadlock. As another example, in order to evaluate the performance of a system, its model must be completely specified from the performance viewpoint. In this case, a check at the syntax level is helpful to easily detect and pinpoint possible violations of the performance closure.

In this section we show how SPA can be enhanced to work with at the architectural level of design. Based on ideas contained in [AG97, BCD01, BF02a, BF02b], we illustrate how SPA can be turned into a fully fledged ADL for the modeling, functional verification, and performance evaluation of complex systems. Recalled that the transformation is largely independent of the specific SPA, we concentrate on $EMPA_{gr}$ [BB00] and we exhibit the resulting SPA based ADL called *Æmilia* [BDC02, BBS02]. The description of a system with *Æmilia* can be done in a compositional, hierarchical, graphical and controlled way. 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 or the invocation of the specification of a previously modeled system, while the interactions are described through actions occurring in the behavior. Second, 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. This process is supported by a graphical notation. Then, the whole behavior of the system is a family of $EMPA_{gr}$ terms transparently obtained by composing in parallel the behavior of the declared instances according to the specified attachments. From the whole behavior, integrated, functional and performance semantic models can be automatically derived, which can undergo to the analysis techniques mentioned

at the beginning of this section. In addition to that, Æmilia comes equipped with some architectural checks for ensuring deadlock freedom and performance closure.

2.2.1 Stochastic Process Algebra

SPA is characterized by three main ingredients: the actions modeling the system activities, the algebraic operators whereby composing the subsystem specifications, and the synchronization disciplines. In this section we illustrate such ingredients for the specific language $EMPA_{gr}$ [BB00].

An action is composed of a type a and an exponential rate λ: $< a, \lambda >$. The type indicates the kind of activity that is performed by the system at a certain point, while the rate indicates the reciprocal of the average duration of the activity assuming that the duration is an exponentially distributed random variable. A special action type, traditionally denoted by τ, designates a system activity whose functionality cannot be observed and serves for functional abstraction purposes. In order to increase the expressiveness, prioritized, weighted immediate actions of the form $<a, \infty_{l, w}>$ are present, which are useful to model activities whose timing is irrelevant from the performance viewpoint as well as activities whose duration follows a phase type distribution.

Several algebraic operators are then included. The zeroary operator $\underline{0}$ represents the term that cannot execute any action. The action prefix operator $< a, \lambda >.E$ denotes the term that can execute an action with type a and rate λ and then behaves as term E. The functional abstraction operator E / L, where L is a set of action types not including τ denotes the term that behaves as term E except that the type a of each executed action is turned into τ whenever $a \in L$. The functional relabeling operator $E[\phi]$, where ϕ is a function over action types preserving observability, denotes a term that behaves as term E except that the type a of each executed action becomes $\phi(a)$. The alternative composition operator $E_1 + E_2$ denotes a term that 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.
In the case of exponentially timed actions, the action choice is regulated by the race policy (the fastest one succeeds), so that each action of E_1 and E_2 has an execution probability proportional to its rate. In the case of immediate actions, they take precedence over exponentially timed actions and the choice among them is governed by the preselection policy: the lower priority immediate actions are discarded, then each of the remaining immediate actions is given an execution probability proportional to its weight. The parallel composition operator $E_1 \|_S E_2$, where S is a set of action types not including τ, denotes a term that asynchronously executes actions of E_1 or E_2 whose type does not belong to S, and synchronously executes - according to a synchronization discipline - equally typed actions of E_1 and E_2 whose type belongs to S. Finally, a constant A denotes a term that behaves according to the associated defining equation $A \triangleq E$, which allows for recursive behaviors. The action prefix operator and the alternative composition operator are called dynamic operators, whereas the functional abstraction operator, the functional 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 adopted synchronization discipline is the generative-reactive one, which is based on the systematic use of prioritized, weighted passive actions of the form $< a, *_{l, w} >$ whose duration is not specified. The idea is that the non-passive actions probabilistically determine the type of action to be executed at each step, while the passive actions of the determined type probabilistically react in order to identify the subterms taking part in the synchronization. In order for two equally typed actions to synchronize, in this approach one of them must be passive and the rate of the resulting action is given by the rate of the non-passive action multiplied by the local execution probability of the passive action.

On the basis of the syntactical ingredients above, the semantics is defined in the usual operational structured way by means of axioms and inference rules that formalize the meaning of each algebraic operator. The resulting semantic models are state transition graphs called the integrated semantics, where states are in correspondence with process terms and transitions are labeled with actions. From the integrated semantic model, two projected semantic models can be derived by discarding action rates or action types, respectively. The former model is called the functional semantics, as its transitions are not decorated with performance related information, thus representing only the functional behavior of the system. The latter model, instead, is called the Markovian semantics, as it expresses the Markov chain (MC) governing the stochastic behavior of the system. The Markovian semantics is a discrete time MC (DTMC) or a continuous time MC (CTMC) depending on whether only immediate transitions occur or not. If there are only immediate transitions, then each immediate transition is assumed to take one time unit and it is relabeled with the corresponding probability. Should exponentially timed and immediate transitions coexist (in different states), a CTMC is derived by suitably eliminating the states having outgoing immediate transitions.

On the integrated semantics a notion of equivalence is defined, which is called the Markovian bisimulation equivalence. This equivalence relates terms on the basis of their ability of simulating each other functional and performance behavior. The Markovian bisimulation equivalence is a congruence with respect to all the operators, which means that it allows for compositional reasoning. Additionally, by virtue of the relationship with ordinary lumping, two Markovian bisimulation equivalent terms are guaranteed to possess the same performance characteristics.

2.2.2 Æmilia: Textual and Graphical Notations

Æmilia is an ADL based on EMPA_{gr} A description in Æmilia represents an architectural type. As shown in Table 1, 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 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, immediate with the same priority level, or passive with the same priority level) and that, to comply with the generative-reactive synchronization discipline of $EMPA_{gr}$, very chain of attachments contains at most one interaction whose associated rate is exponential or immediate.

archi_type	‹ name and numeric parameters ›
archi_elem_types	‹ architectural element types: behaviors and interactions ›
archi_topology	
archi_elem_instances	‹ architectural element instances ›
archi_interactions	‹ architectural interactions ›
archi_attachments	‹ architectural attachments ›
end	

Table 1 - Structure of an Æmilia textual description

We now illustrate the textual notation of Æmilia by means of an example concerning a pipe-filter system. This system is composed of three identical filters and one pipe. Each filter acts as a service center of capacity two that is subject to failures and subsequent repairs, which is characterized by a service rate σ, a failure rate φ, and a repair rate ρ. For each item processed by the upstream filter, the pipe instantaneously forwards it to one of the two downstream filters according to the availability of free positions in their buffers. If both have free positions, the choice is resolved probabilistically based on $p_{routing}$. The Æmilia textual description is provided in Table 2. Wherever omitted, priority levels and weights are taken to be 1.

Such a description establishes that there are three instances F_0, F_1, and F_2 of *FilterT* as well as one instance P of *PipeT*, connected in such a way that the items flow from F_0 to P and from P to F_1 or F_2. It is worth observing that the system components are clearly elucidated and easily connected to each other, and that the numeric parameters allow for a good degree of specification reuse: e.g., the behavior of the filters is defined only once. Additionally, the *accept_item* input interaction of F_0 and the *serve_item* output interactions of F_1 and F_2 are declared as being architectural. Therefore, they can be used for hierarchical modeling, e.g. to describe a client-server system where the server structure is like the pipe-filter organization above.

archi_type *PipeFilter* (**exp_rate** $\sigma_0, \sigma_1, \sigma_2, \varphi_0, \varphi_1, \varphi_2, \rho_0, \rho_1, \rho_2$;
weight $p_{routing}$)

archi_elem_types

elem_type *FilterT* (**exp_rate** σ, φ, ρ)

behavior $Filter \triangleq \langle accept\_item, * \rangle . Filter' + \langle fail, \varphi \rangle . \langle repair, \rho \rangle . Filter$

$Filter' \triangleq \langle accept\_item, * \rangle . Filter'' + \langle serve\_item, \sigma \rangle . Filter + \langle fail, \varphi \rangle . \langle repair, \rho \rangle . Filter'$

$Filter'' \triangleq \langle serve\_item, \sigma \rangle . Filter' + \langle fail, \varphi \rangle . \langle repair, \rho \rangle . Filter''$

interactions **input** *accept_item*
output *serve_item*

elem_type *PipeT* (**weight** *p*)

behavior $Pipe \triangleq \langle accept\_item, * \rangle . (\langle forward\_item_1, \infty_{1,p} \rangle . Pipe + \langle forward\_item_2, \infty_{1,1-p} \rangle . Pipe)$

interactions **input** *accept_item*
output $forward\_item_1$, $forward\_item_2$

archi_topology

archi_elem_instances $F_0 : FilterT (\sigma_0, \varphi_0, \rho_0)$
$F_1 : FilterT (\sigma_1, \varphi_1, \rho_1)$
$F_2 : FilterT (\sigma_2, \varphi_2, \rho_2)$
$P : PipeT (p_{routing})$

archi_interactionsinput *accept_item*
output $F_1.serve\_item$, $F_2.serve\_item$

archi_attachments **from** $F_0.serve\_item$ **to** $P.accept\_item$
from $P.forward\_item$ **to** $F_1.accept\_item$
from $P.forward\_item$ **to** $F_2.accept\_item$

end

Table 2: Textual description of *PipeFilter*

Æmilia comes equipped with a graphical notation as well, in order to provide a visual help during the architectural design of complex systems. Such a graphical notation is based on flow graphs [M89]. In a flow graph representing an architectural description in Æmilia, 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. As an example, the architectural type *PipeFilter* can be pictorially represented through the flow graph of Fig. 6. From a methodological viewpoint, when modeling an architectural type with Æmilia, it is convenient to start with the flow graph representation of the architectural type and then to textually specify the behavior of each AET.

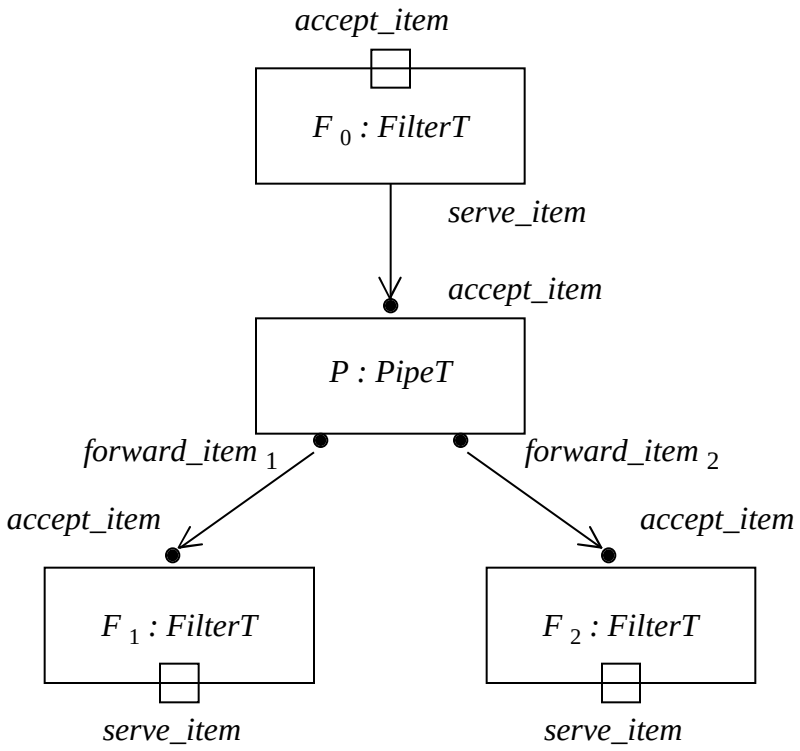


Figure 6 - Flow graph of *PipeFilter*

2.2.3 Translation Semantics

The semantics of an Æmilia specification is given by translation into $EMPA_{gr}$. While only the dynamic operators of $EMPA_{gr}$ can be used in the syntax of an Æmilia specification, the more complicated static operators of $EMPA_{gr}$ are transparently used in the semantics of an Æmilia specification. The translation into $EMPA_{gr}$ is accomplished 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 functional abstraction 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 pipe-filter system of Table 2 we have

$$[\![\mathit{FilterT}]\!] = [\![F_0]\!] = [\![F_1]\!] = [\![F_2]\!] = \mathit{Filter} / \{\mathit{fail}, \mathit{repair}\}$$
$$[\![\mathit{PipeT}]\!] = [\![P]\!] = \mathit{Pipe}$$

thus abstracting from the internal activities *fail* and *repair*.

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. Recalled that the parallel composition operator is left associative, for the pipe-filter system we have

$$[\![\mathit{PipeFilter}]\!] = [\![F_0]\!] [\mathit{serve\_item} \rightarrow a] \parallel_{\emptyset}$$
$$[\![F_1]\!] [\mathit{accept\_item} \rightarrow a_1] \parallel_{\emptyset}$$

$$[[F_2]]\,[accept\_item \rightarrow a_2]\ \|_{\{a, a_1, a_2\}}$$
$$[[P]]\,[accept\_item \rightarrow a,$$
$$forward\_item \rightarrow a_1,$$
$$forward\_item \rightarrow a_2]$$

The use of the functional relabeling operator is necessary to make the AEIs interact. As an example, F_0 and P must interact via *serve_item* and *accept_item*, which are different from each other. Since the parallel composition operator allows only equally typed actions to synchronize, in $[[PipeFilter]]$ each *serve_item* action executed by $[[F_0]]$ and each *accept_item* action executed by [[P]] is relabeled to an action with the same type a. In order to avoid interference, it is important that a be a fresh action type, i.e. an action type occurring neither in $[[F_0]]$ nor in $[[P]]$. Then a synchronization on a is forced between the relabeled versions of $[[F_0]]$ and $[[P]]$ by means of operator $\|_{\{a, a_1, a_2\}}$. It is worth reminding that the transformation of *PipeFilter* into [[*PipeFilter*]], which can be analyzed through the techniques mentioned at the beginning of Sect 2.2.1, is completely transparent to the designer.

2.2.4 Architectural Checks

Æmilia is equipped with some architectural checks that the designer can use to verify the well formedness of an architectural description and, in case a mismatch is detected, to identify the components that cause the mismatch. Most of such checks are based on the weak bisimulation equivalence [M89], which captures the ability of the functional semantic models of two terms to simulate each other behaviors up to internal actions.
The first two checks take care of verifying whether the deadlock free AEIs of an architectural type fit together well, i.e. do not lead to system blocks when assembled together. The first check (compatibility) is concerned with architectural types whose topology is acyclic. For an acyclic architectural type, if we take an AEI K and we consider all the AEIs $C_1,\ldots, C_n$ attached to it, we can observe that they form a star topology whose center is K, as the absence of cycles prevents any two AEIs among $C_1,\ldots, C_n$ from communicating via an AEI different from K. It can easily be recognized that an acyclic architectural type is just a composition of star topologies. An efficient compatibility check based on the weak bisimulation equivalence (together with a simple constraint on action priorities) ensures the absence of deadlock within a star topology whose center K is deadlock free, and this check scales to the whole acyclic architectural type. The basic condition to check is that every C_i is compatible with K, i.e. the functional semantics of their parallel composition is weakly bisimulation equivalent to the functional semantics of K itself. Intuitively, this means that attaching C_i to K does not alter the behavior of K, i.e. K is designed in such a way that it suitably coordinates with C_i.

Since the compatibility check is not sufficient for cyclic architectural types, the second check (interoperability) deals with cycles. A suitable interoperability check based on the weak

bisimulation equivalence (together with a simple constraint on action priorities) ensures the absence of deadlock within a cycle $C_1,\ldots, C_n$ of AEIs in the case that at least one of such AEIs is deadlock free. The basic condition to check is that at least one deadlock free C_i interoperates with the other AEIs in the cycle, i.e. the functional semantics of the parallel composition of the AEIs in the cycle projected on the interactions with C_i only is weakly bisimulation equivalent to the functional semantics of C_i. Intuitively, this means that inserting C_i into the cycle does not alter the behavior of C_i , i.e. that the behavior of the cycle assumed by C_i matches the actual behavior of the cycle. In the case in which no deadlock free AEI is found in the cycle that interoperates with the other AEIs, a loop shrinking procedure can be used to single out the AEIs in the cycle responsible for the deadlock.

On the performance side, there is a third check to detect architectural mismatches resulting in performance under specification. This check (performance closure) ensures that the performance semantic model underlying an architectural type exists in the form of a CTMC or DTMC. In order for an architectural type to be performance closed, the basic condition to check is that no AET behavior contains a passive action whose type is not an interaction, and that every set of attached local interactions contains one interaction whose associated rate is exponential or immediate.

2.2.5 Families of Architectures and Hierarchical Modeling

An Æmilia description represents a family of architectures called an architectural type. An architectural type is an intermediate abstraction between a single architecture and an architectural style. An important goal of the software architecture discipline is the creation of an established and shared understanding of the common forms of software design. Starting from the user requirements, the designer should be able to identify a suitable organizational style, in order to capitalize on codified principles and experience to specify, analyze, plan, and monitor the construction of a system with high levels of efficiency and confidence. An architectural style defines a family of systems having a common vocabulary of components as well as a common topology and set of constraints on the interactions among the components. As examples of architectural styles we mention main program-subroutines, pipe-filter, client-server, and the layered organization. Since an architectural style encompasses an entire family of software systems, it is desirable to formalize the concept of architectural style both to have a precise definition of the system family and to study the architectural properties common to all the systems of the family. This is not a trivial task because there are at least two degrees of freedom: variability of the component topology and variability of the component internal behavior.
An architectural type is an approximation of an architectural style, where the component topology and the component internal behavior can vary from instance to instance of the architectural type in a controlled way, which preserves the architectural checks. More precisely, all the instances of an architectural type must have the same observable functional behavior and conforming topologies, while the internal behavior and the performance characteristics can freely vary. An instance of an architectural type can be obtained by invoking the architectural type and passing actual AETs preserving the observable functional behavior of the formal AETs, an actual topology (actual AEIs, actual architectural interactions, and actual attachments) that conforms to the formal topology, actual names for the architectural interactions, and actual values for the numeric parameters.

The simplest form of architectural invocation is the one in which the actual parameters coincide with the formal ones, in which case the actual parameters are omitted for the sake of conciseness. The possibility of defining the behavior of an AET through an architectural invocation as well as declaring architectural interactions can be exploited to model a system architecture in a hierarchical way. As an example, consider the pipe-filter organization of Table 3 and suppose that it is the architecture of the server of a client-server system. The flow graph description of the resulting client-server system is depicted in Fig. 7, while its textual description is reported in Table 3. The client description is parametrized w.r.t. the request generation rate λ, while the communication link description is parametrized w.r.t. the communication speed δ. As can be observed, the behavior of the server is defined through an invocation of the previously defined architectural type *PipeFilter*, where the actual names *accept_request*, *generate_outcome*, and *generate_outcome* substitute for the formal architectural interactions F_0.*accept_item*, F_1.*serve_item*, and F_2.*serve_item*, respectively.

A more complex form of architectural invocation is the one in which actual AETs are passed that are different from the corresponding formal AETs. In this case, we have to make sure that the actual AETs preserves the functional behavior determined by the formal ones. To this purpose, Æmilia is endowed with an efficient behavioral conformity check based on the weak bisimulation equivalence (together with a simple constraint on action rates) to verify whether an architectural type invocation conforms to an architectural type definition, in the sense that the architectural type invocation and the architectural type definition have the same observable functional semantics up to some relabeling. The basic condition to check is that the functional semantics of each actual AET is weakly bisimulation equivalent to the functional semantics of the corresponding formal AET up to some relabeling.
This behavioral conformity check ensures that all the correct instances of an architectural type possess the same compatibility, interoperability, and performance closure properties. In other words, the outcome of the application of the compatibility, interoperability, and performance closure checks to the definition of an architectural type scales to all the behaviorally conforming invocations of the architectural type.
The most complete form of architectural invocation is the one in which both actual AETs and an actual topology are passed that are different from the corresponding formal AETs and formal topology, respectively. In this case, we have to additionally make sure that the actual topology conforms to the formal topology. There are three kinds of admitted topological extensions, all of which preserve the compatibility, interoperability, and performance closure properties under some general conditions. The first kind of topological extension is given by the and/or one. The idea behind the and/or extensions is to permit a variable number of coordinated/alternative AEIs to be attached to a set of already existing AEIs. The second kind of topological extension is given by the exogenous one. The idea behind the exogenous extensions is that, since the architectural interactions of an architectural type are the frontier of the whole architectural type, it is reasonable to extend the architectural type at some of its architectural interactions with instances of the already defined AETs, in a way that follows the prescribed topology. The third kind of topological extension is given by the endogenous one. The idea behind the endogenous extensions is that of replacing a set of AEIs with a set of new instances of the already defined AETs, in a way that follows the prescribed topology. In this case, we consider the frontier of the architectural type with respect to one of the replaced AEIs to be the set of interactions previously attached to the local interactions of the replaced AEI.

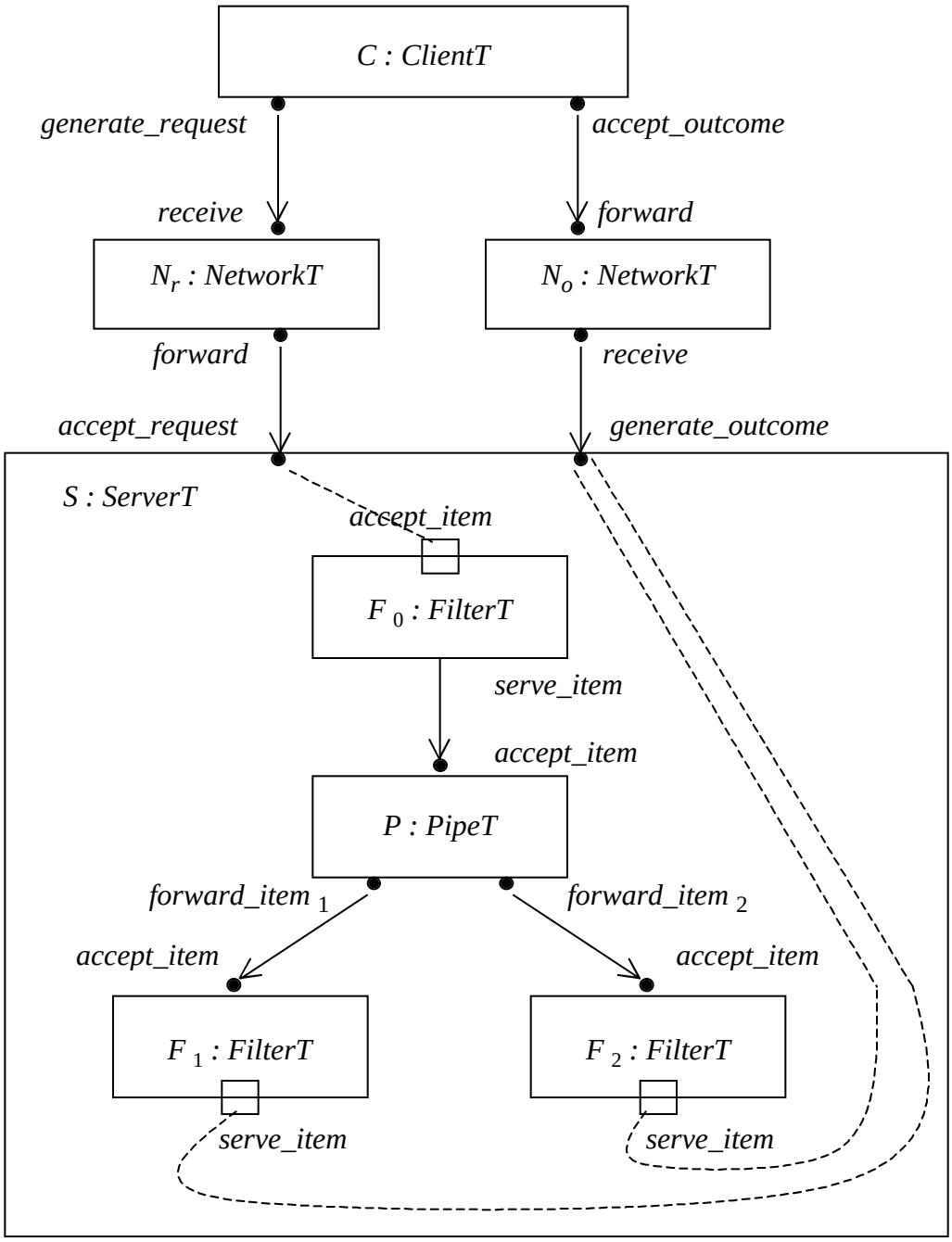


Figure 7. Flow graph of *ClientServer*

archi_type *ClientServer* (**exp_rate** $\lambda, \delta_r, \delta_o, \sigma_0, \sigma_1, \sigma_2, \varphi_0, \varphi_1, \varphi_2, \rho_0, \rho_1, \rho_2$; **weight** $p_{routing}$)

archi_elem_types

elem_type *ClientT* (**exp_rate** λ)

behavior *Client* $\triangleq$ ‹*generate_request*, λ› . ‹*accept_outcome*, * › . *Client*

interactions **input** *accept_outcome*
output *generate_request*

elem_type *NetworkT* (**exp_rate** δ)

behavior *Network* $\triangleq$ ‹*receive*, * › . ‹*forward*, δ› . *Network* ‹*accept_outcome*, * › . *Client*

interactions **input** *receive*
output *forward*

elem_type *ServerT* (**exp_rate** $\sigma_0, \sigma_1, \sigma_2, \varphi_0, \varphi_1, \varphi_2, \rho_0, \rho_1, \rho_2$; **weight** $p_{routing}$)

behavior *Server* $\triangleq$ *PipeFilter (; /* actual AETs */ ; /* actual AEIs */ ; /* actual arch. iteractions */ ; /* actual attachments */ accept_request, generate_outcome, $\sigma_0, \sigma_1, \sigma_2, \varphi_0, \varphi_1, \varphi_2, \rho_0, \rho_1, \rho_2, p_{routing}$) generate_outcome ;*

interactions **input** *accept_request*
output *generate_outcome*

archi_topology

archi_elem_instances *C* : *ClientT* (λ)
N_r : *NetworkT* (δ_r)
N_o : *NetworkT* (δ_o)
S : ServerT ($\sigma_0, \sigma_1, \sigma_2, \varphi_0, \varphi_1, \varphi_2, \rho_0, \rho_1, \rho_2, p_{routing}$)

archi_interactions

archi_attachments **from** *C.generate_request* **to** N_r*receive*
from N_r.*forward* **to** *S.accept_request*
from *S.generate_outcome* **to** N_o*receive*
from N_o.*forward* **to** *C.accept_outcome*

end

Table 3: Textual description of *ClientServer*

2.2.6 An Analysis Example

As an example of performance evaluation, we compute with TwoTowers [BCS01] the system throughput (mean number of requests served per unit of time) and the utilization of each of the three processing elements in the server of the architectural type *ClientServer*. We assume that such an architectural type is invoked with a perfect pipe-filter based server (processing elements never fail) and the following actual numeric parameters: 5.0 for the request generation rate, 1.0 for the network propagation time, 0.5 for the routing probability, and 3.0, 3.5, 4.5 for the service rates of the three processing elements, respectively. Before carrying out the performance evaluation, we investigate the well formedness of the architectural description. Starting from *PipeFilter*, we observe that it passes the architectural compatibility check, hence it is deadlock free. Then we note that the instance of *PipeFilter* with perfect filters behaviorally conforms to the definition of *PipeFilter*, hence it is deadlock free too. The server component of *ClientServer* is thus deadlock free. Afterwards, we see that *ClientServer* passes the architectural interoperability check, which means that the whole architectural description under consideration is deadlock free. Likewise, we easily discover that *ClientServer* passes the performance closure check, hence its underlying performance model exists (in the form of a CTMC).The performance measures of interest are expressed through rewards according to the technique of [BB02]. For the system throughput, actions F_1.*serve_item* and F_2.*serve_item* are given rewards 3.5 and 4.5, respectively. In this way, all the states enabling action F_1.*serve_item* or F_2.*serve_item* are given a reward equal to the rate of that action, while all the other states are given reward zero. In the case of the utilization of the upstream processing element, instead, action F_0.*serve_item* is given reward 1. By so doing, the utilization is correctly computed as the sum of the steady state probabilities of all and only the states enabling action F_0.*serve_item* i.e. the states in which the processing element is used. The utilization of the other two processing elements is specified similarly. The results obtained by solving the underlying CTMC show that the system throughput is 0.35877 and the utilization of the three processing elements are 0.11959, 0.0512528, and 0.0398633, respectively.

clients	states	transitions	throughput	utilization$_0$	utilization$_1$	utilization$_2$
10	1067	2430	0.635987	0.224099	0.088165	0.072757
20	2667	6150	0.699606	0.242998	0.096040	0.080769
30	4267	9870	0.744251	0.265766	0.101318	0.086586
40	5867	13590	0.794978	0.292759	0.107302	0.093205
50	7467	17310	0.858178	0.319252	0.114846	0.101381

Table 4: Performance analysis results

We now observe that the configuration of the client-server system we are considering is extremely simple. In particular, it would be interesting to see how the performance figures above vary when increasing the number of clients. To model this scenario, one may think of modifying the architectural type Client Server through extensible or connections that allow an arbitrary number of instances of *ClientT* to be attached to *ServerT*. However, this would result in an exponential growth of the state space as the number of clients increases, which would make the obtainment of the performance results quite time consuming. Fortunately, we can overcome this drawback by resorting to the Markovian bisimulation equivalence. It can easily

be shown that the parallel composition of an arbitrary number n of *Client* terms is Markovian bisimulation equivalent to the following family of sequential terms:

$$Client_n \triangleq \langle generate\_request, n \lambda \rangle . Client_{n-1}$$

$$Client_i \triangleq \langle generate\_request, i \lambda \rangle . Client_{i-1} + \langle accept\_outcome, * \rangle . Client_{i+1} \qquad 0 < i < n$$

$$Client_0 \triangleq \langle accept\_outcome, * \rangle . Client_1$$

Since the Markovian bisimulation equivalence is a congruence, i.e. it is substitutive with respect to the algebraic operators, the semantics - hence the properties - of the whole architectural description does not change if we use $Client_n$ instead of the parallel composition of n terms *Client*. The results of the analysis for n varying from 10 to 50 are shown in Table 4, where the second (third) column indicates the number of states (transitions) of the integrated semantic model underlying the architectural description, the fourth column indicates the system throughput, and the three trailing columns indicate the utilization of the three processing elements. As can be noted, the state space grows linearly, thus causing the analysis process to scale.

2.3 Performance modeling for mobile software architectures

Location-awareness has been suggested as an innovative approach (in opposition to the traditional *location-transparent* approach to the design of distributed applications) to be applied since the early design phases of software applications for wide area environments, mainly to cope with the problems that such applications must face, caused by the heterogeneity and dynamicity of the resources available at different sites in wide area environments, where computing nodes may have very different processing capabilities, and communication links may be characterized by very different bandwidth and intermittent connectivity. Explicitly considering components location straightforwardly leads to consider the possibility of changing dynamically the components location as a new dimension in the design and implementation of distributed applications. In fact, design paradigms based on the idea of *code mobility* have been recently introduced, where components of an application may (autonomously or upon request) move to different locations, during the application execution [FPV98]. Besides, software technologies are readily available (e.g. Java-based), that provide tools to implement these paradigms. The potential advantages of code mobility include, among others, increased flexibility and customizability, and better performance. In particular, with respect to performance, an argument in favor of these new paradigms is the consideration that in a large scale distributed environment it may be convenient to move components close to their current partners, with the goal of transforming non local interactions into local ones.

It should be noted that code mobility, as it is intended in this framework, represents a different concept with respect to the well known concept of process migration, even if the adopted mechanisms to implement them may be similar. Process migration is a (distributed) OS issue, realized transparently to the application (usually to get load balancing), and hence does not represent a tool in the hands of the application designer; on the contrary, code mobility is intended to bring the ability of changing location under the control of the designer, so representing a new tool he/she can exploit to accomplish quality (e.g., performance) requirements.

Looking at code mobility in a software architecture perspective [BCK98], we see that it presents features that are of architectural concern. Indeed, the main consequence of code mobility is a modification of the application architectural configuration, i.e., which are and where are located the components a given component interacts with at a given time. This, together with the fact that different but functionally equivalent software architectures can be defined using mobile code paradigms, calls for a careful analysis of the impact of code mobility since the early phases of the software design, when the software architecture of an application is defined [BCK98]. Indeed, it is widely recognized that, in general, the adoption of a particular architecture can have a large impact on quality attributes of the final application such as modifiability, reusability, reliability, and performance [SW02].

Hence, it is important to provide the software designer with tools that support him/her in making the right architectural choices concerning the adoption of some code mobility paradigm in the definition of the application software architecture, with respect to its impact on quality attributes like performance. In this respect, the definition of a validation methodology for performance attributes of mobile software architectures should take into consideration the following issues:

- definition of a suitable notation for the modeling of code mobility based software architectures (mobile software architectures, for short);
- definition of a suitable target model for the analysis of some performance attribute of mobile software architectures;
- definition of an efficient translation methodology from the description of a mobile software architecture, expressed in some notation, to the target performance model.

In the following, we present some different solutions that have been explored for the above issues, putting in evidence their respective merits and limits.

To illustrate the proposed modeling approaches, we will use a simple application example based on a traveling agency scenario, where a travel agency periodically contacts K flying companies to get information about the cost of a ticket for some itinerary, exchanging a sequence of N messages with each company, to collect the required information. Using a traditional client-server approach, this means that the agency should explicitly establish N RPCs with each company to complete the task. On the other hand, adopting for example a MA style, the agency could deliver an agent that travels along all the K companies exchanging locally N messages with each company, and then reporting back to the agency the collected information.

2.3.1 Modeling of mobile software architectures

A thorough discussion of issues concerning mobile software architectures can be found in [FPV98], where, in particular, some different mobility styles are identified, based on the distinction between whether they require the creation of an independent copy of a component at a new location, or a location change of a component that preserves its identity. In the former case a further distinction is made about whether the copy is created at the location of the component that starts the interaction (*code on demand* (COD)), or at the location of the component that accepts the interaction (*remote evaluation* (REV)), while in the latter case a single paradigm is identified (*mobile agent* (MA)). Note that COD and REV only require code shipping from one site to another, while MA additionally requires shipping of the entire component computational state (i.e. code and execution stack). Hence, notations for mobile software architectures should allow to model these different styles.

Modeling of software architectures is, in general, the subject of a rich body of literature that can be classified according to two directions. Several papers are devoted to the presentation of different kinds of formal architecture description languages (ADLs), equipped with precise syntax and semantics (see [MT00], and references therein). However, their integration in the design practice with other development artifacts presents some difficulty, and this has suggested alternative approaches, to support architectural concerns within a more widely accepted modeling language like the Unified Modeling Language (UML), thanks also to the possibility of exploiting the rich set of development tools of the UML environment [MRR02, SGW01]. We have proposed contributions concerning mobile software architectures, along both these directions. A common characteristics of the proposed modeling notations is the attempt to stress, even at the "syntactic" level, the concept of code mobility as an architectural issue, with one of its main impacts concerning the "quality" of the interactions a component has with its partners. Another common characteristic is the idea of introducing in the notation the possibility of expressing uncertainty about the adoption of some code mobility style. Indeed, in the early stage of software development, where architectural issues are taken into consideration, the designer could not feel confident enough to decide about the best mobile architectural style. Expressing this uncertainty allows us to devise an "uncertainty-driven" analysis methodology, aimed at providing insights about the most advantageous mobility style.

Formal notation for mobility modeling

Several formalisms have been suggested to provide a firm base to reasoning about properties of code mobility-based applications [DFPV98, MIL99, NPD01, PRM01, WF98]. However, many of these formalisms cannot be considered as ADLs, since they lack some of the essential features of an ADL, that is the explicit modeling of both components and interactions (i.e. connectors) as first class entities. One of the exceptions is the COMMUNITY language proposed in [WF98]. The approach presented in [CG02] has been defined as an extension of the COMMUNITY language, even if it is not tied, in principle, to a particular language. Besides some slight syntactic change, the main modification is the introduction of a new connector type, as described below.

In COMMUNITY an architecture description consists of the following four sections (we refer to [WF98] for a detailed definition): *Components-type*, that defines the type of the architecture components; *Connectors-type*, that defines the type of the architecture connectors; *Components*, that defines the actual instances of the architecture components; *Connections*, that defines the actual instances of the architecture connectors.

A component type consists of an **init** statement that assigns initial values to the component local variables, and a set of *named guarded* actions that can modify only local variables, where each action consists of the simultaneous atomic execution of assignment statements. A special local variable λ indicates the component location. An action is chosen (non deterministically) for execution if its guard is true. The execution model of an application described by this language can be operationally viewed as a non-deterministic fair interleaved selection of actions in the **do** sections of all the instantiated components, that starts from the initial state described by the union of all the **init** sections. At each execution step, only actions whose guards hold true are considered for execution.

A connector type defines a pattern of interaction among components. In COMMUNITY, some connector types are introduced. One of them is the *Communicator* connector type, that models synchronous message sending between two components. Its prototype is as follows:

connector *Communicator*(c1, c2: **program**; a1, a2 : **action**; x1, x2: **any_type**; I: **bool**)

In this definition c1 and c2 are the names of the connected components, a1 and a2 are the names of actions performed by c1 and c2, respectively, "synchronized" by the connector, while x1 and x2 are c1 and c2 variables, respectively, used to send and receive the exchanged value; I is a condition that controls the connector activation. The semantics of this connector is such that action a1, that starts communication, is executed only when both its guard and I hold true. A communication is completed when the guard of action a2 holds true. In practice, the guards of action a1 and a2 model the willingness of the two partners to participate in the communication, while I can be used to model some condition that actually enables communication but that is non-local to any component (e.g., modeling the concept of two components' co-location). The *Communicator* formal semantics, specified in [WF98], guarantees that the "receive" action a2 is executed only after the communication takes place, i.e. after the transfer of the value of x1 to x2 has been completed.

Based on this connector type, an additional one has been defined in [CG02], called *Mob_Comm*, that can also model code mobility. One of the main reasons that motivates the idea of [CG02] of modeling code mobility by a connector is that it reinforces the idea that mobility should be considered as an interaction related design option, that have an impact on the "quality" of the interactions. The prototype of the new connector is as follows:

connector *Mob_Comm*(c1, c2: **program**; a1, a2 : **action**; x1, x2: **any_type**; I: **bool**; M: **mob_condition**)

All the common parameters play the same role in both connectors. In addition, the parameter M, which differentiates *Mob_Comm* from *Communicator*, plays a special role, since it allows to express the mobility of the connected components and also, as outlined above, possible uncertainty about it. It carries a "mob_condition" type because, besides the boolean values **true** and **false** of a standard condition, it can also be instantiated with a special value "**?**", with the following semantics:

- when M is instantiated with a **false** value (in general, an expression yielding this value), *Mob_Comm* has exactly the same semantics as *Communicator*, i.e. synchronous message sending from c1 to c2 component.
- when M is instantiated with a **true** value, then the connector semantics is modified as follows with respect to *Communicator*: the activation rule is the same as *Communicator* (i.e. controlled by I condition), but when the connector is activated the location of c1 changes to that of c2, before transferring the value of x1 to x2 according to the *Communicator* semantics.
- when M is instantiated with "**?**", the connector semantics corresponds to the non deterministic execution of both the above options; this is helpful to generate a model where the mobility policy must not necessarily be chosen in advance, but can result (as the optimal one) from the model solution.

It should be noted that *Mob_Comm* appears suited to model a *mobile agent* style, whereas it is less clear whether it can model other kinds of mobile code paradigms as well.
Using this formalism, the architecture of the considered example can be modeled as follows, in the case of a single message exchange (i.e., N=1) with each flying company, adopting a MA style.

System *Travel Agency*
Components-type

program TRAVELAG(λ)
var sent, received: **bool**; query: ...; resp: **list of** ...;
init sent = **true and** received = **true and** f_comp = 0
do prep_req: [sent **and** received ---> query := Newquery() || sent := **false** || received := **false**]
[] send_req(i): [sent = **false** ---> sent := **true**]
[] get_resp(i): [**true** ---> received := **true** || Process(resp)]
end

program COLLECTOR(λ)
var sent, done, newreq: **bool**; ; query: ...; response: ...; q : **list of** …; flying_id: **int**;
init done = **false and** flying_id = 0 **and** newreq = **false and** q = **nil and** sent = **false**
do wait_req: [**true** ---> flying_id:= (flying_id +1)**mod**K || newreq := **true**]
[] reply_to_req: [done ---> q := **nil** || done := **false**]
[] req_to_dev: [**not** sent ---> sent := **true**]
[] wait_from_dev: [**true** ---> q := cons(response, q) || flying_id:= (flying_id +1)**mod**K || sent := **false**]
[] completed: [flying_id = 0 **and** newreq ---> done := **true** || newreq := **false**]
end

program FLYINGCOMP(λ)
var id: **int**; rep : **bool**; ans: ...; query: ...;
init rep = **false and** id=...(unique id ≥0 and <K)
do wait_from_coll: [**true** ---> ans := f(query) || rep := **true**]
[] reply_to_coll: [rep ---> rep := **false**]
end

Connector-type

connector *Mob_Comm*(c1,c2: **program**; a1, a2: **action**; x1, x2: **any_type**; I: **bool**; M: **mob_condition**)

Components

ag(L_K): TRAVELAG ; c(L_K): COLLECTOR; $f_i(L_i)_{i=0..K-1}$: FLYINGCOMP;

Connections

ag_to_coll: *Mob_Comm*(ag, c, ag.query, c.query, ag.send_req, c.wait_req, **true**, **false**)
coll_to_ag: *Mob_Comm*(c, ag, c.q, ag.resp, c.reply_to_req, ag.get_resp, **true**, **?**)
$coll\_to\_f_{i,\ i=0..K-1}$: *Mob_Comm*(c, f_i, c.query, f_i.query, c.req_to_d, f_i.wait_from_coll, c. flying_id = f_i.id, **?**)
$fly\_to\_coll_{i,\ i=0..K-1}$: *Mob_Comm*($f_i$, c, f_i.ans, c.response, f_i.reply_to_coll, c.wait_from_dev, **true**, **false**)

In this example, L_i (i=0, …K-1) is the location of each FLYINGCOMP f_i component, while L_K is the location of the TRAVELAG ag component (and the initial location of the COLLECTOR c). From this textual description, we can note, looking at the *Connections* section, the existence of two connectors (ag_to_coll and coll_to_ag) used, respectively, to communicate the initial query from ag to c, and the overall collected information from c to ag. Other 2K connectors ($coll\_to\_f_i$ and $fly\_to\_coll_i$) are used for the message exchange between c and the f_i's. Moreover, we can also note that the M condition is set to **false** in ag_to_coll and $fly\_to\_coll_i$, while it is set to "**?**" in coll_to_ag and $fly\_to\_coll_i$. Because of the defined semantics for the connector *Mob_Comm*, we are expressing in this way that ag and the f_i's components do not change their location, while there is uncertainty about the convenience of designing c as a mobile agent that moves to the locations of its partners when interacts with them.

Semi-formal (UML) notation for mobility modeling

The advantage of a formal architecture description language mainly consists in the possibility of a rigorous and non-ambiguous modeling activity. However, the use of formal notations does not have yet gained widespread acceptance in the practice of software development. On the contrary, a semi-formal notation like UML [BOO99], has quickly become a de-facto standard in the industrial software development process. UML is based on object oriented methodology, and consists of two parts: a *notation*, used to describe a set of diagrams (also called the syntax of the language), and a *metamodel* (also called the semantics of the language) that specifies the abstract integrated semantics of UML modeling concepts. The notation encompasses several types of diagrams, that provide specific views of the system (e.g. Class and Object Diagrams to describe the static software structure, Sequence and Collaboration Diagrams to describe interaction dynamic behavior, or Deployment Diagrams to describe the run-time software components allocation). Moreover, UML also provide extension mechanisms that allow to extend in a controlled way the language, to adapt it to the needs of particular application domains. These mechanisms include *stereotypes*, that extend the UML vocabulary introducing new kinds of building blocks, *tagged values*, that extend the properties of a UML building block, and *constraints*, that extend the semantics of a UML building block.

Standard UML can be used as notation for the modeling of mobile architectures, since UML already provides some mechanisms for this goal. They are mainly based on the use of a tagged value `location` within a component to express its location, and of the `copy` and `become` stereotypes to express the location change of a component. The former stereotype can be used to specify the creation of an independent component copy at a new location (like in the COD and REV styles), and the latter to specify a location change of a component that preserves its identity (like in the MA style). In [BOO99] it is shown how to use these mechanisms within a Collaboration Diagram to model the location change of a mobile component interleaved with interactions among components.

Example. The travel agency application can be modeled by the Collaboration Diagram illustrated in Figure 8, based on standard UML, in case of MA style, and assuming only two flying companies (i.e., K=2).

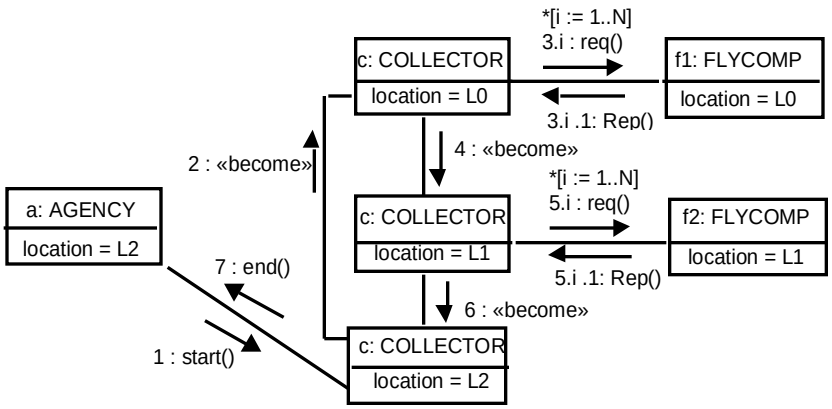
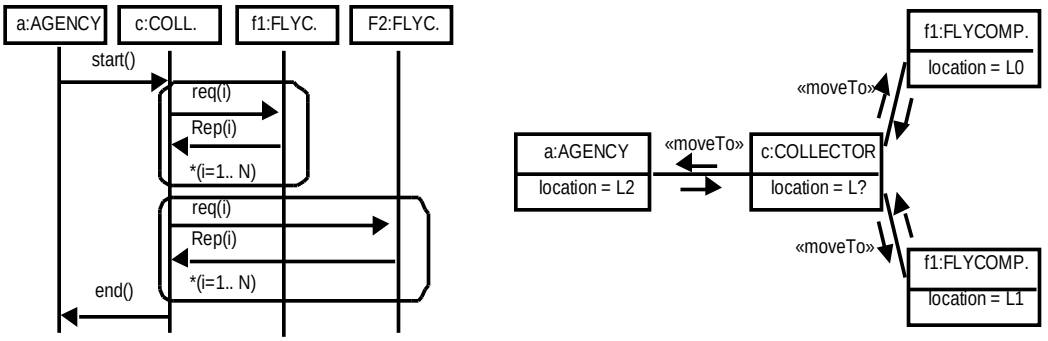


Figure 8: Collaboration Diagram example

However, this modeling approach presents some drawbacks, since it mixes together two different views, one concerning the architectural style (e.g. the fact that a component behaves according to some mobility style), and the other one concerning the actual sequence of messages exchanged between components during a particular interaction. Moreover, this approach may lead to a proliferation of objects in the diagrams that actually represent the same object at different locations. Both these drawbacks can lead to quite obscure models of the application behavior. To overcome the drawbacks of standard UML as notation for mobile architectures, a different approach have been suggested in [GM01] to UML modeling of mobile software architectures, based on the use of both Collaboration and Sequence Diagrams (CD and SD), with a clear separation of concerns between them, with the goal, as remarked at the beginning of this section, of both expressing mobility as an architectural issue, and including in the model the uncertainty about the adoption of a mobile style. In particular, the CD is used to model only architectural concerns regarding the interaction *structure* (i.e. who interacts with whom) and *style*, without showing the actual sequence of exchanged messages, while the SD describes the actual sequence of interactions between components, which is basically independent of the adopted style and obeys only to the intrinsic logic of the application.

Standard UML notation is used in the SD to describe the interaction logic. With regard to architectural issues, the interaction structure is modeled by the links that connect components in CD, with arrows specifying unidirectional or bidirectional interactions. For the interaction style, the main goal of the proposal in [GM01] is to distinguish a style where component location is statically assigned, from a style where components do change location to adapt to environment change. To this purpose, we use the standard `location` tagged value to specify the component location, while we extend the UML semantics by introducing a new stereotype `moveTo` that applies to messages in the CD. Where present, `moveTo` indicates that the source component moves to the location of its target before starting a sequence of consecutive interactions with it. If no other information is present, this style applies to each sequence of interactions shown in the associated SD, between the source and target components of the `moveTo` message; otherwise a condition can be added to restrict this style to a subset of interactions between two components. It should be noted that this approach appears suitable to model only mobile architectures where the architecture style is of MA type.

Example. According to the adopted modeling framework, the travel agency example application can be modeled as shown in Fig. 9. Figure 9.a shows a SD that describes in detail the “logic” of the interaction, i.e. the sequence of messages exchanged among the components. In this diagram no information is present about the adopted style, that is whether or not some component changes location during the interactions. This information is provided by the CD in Fig. 9.b, that models a style where component mobility is considered. More precisely, the diagram shows that only `c` can change location, and according to the `moveTo` semantics described above, it moves to the location of `a`, `f1` or `f2` before interacting with them.
Note that in Fig. 9.b the location of `c` is left unspecified (L?), since it can dynamically change. In general, it is possible to give it a specified value in the diagram that would show the "initial" location of the mobile object in an initial deployment configuration.



(a) (b)

Figure 9**.** Travel agency example: (a) interaction logic, (b) architectural style

To model uncertainty about the architectural style (i.e. location and possible mobility of components), an additional stereotype `moveTo?` has been proposed in [GM01], that extends the semantics of the `moveTo` stereotype described above. When a message between two components in a "style" CD is labeled with `moveTo?`, this means that the source component "could" move to the location of its target at the beginning of a sequence of interactions with it. In a sense, this means that based on the information the designer has at that stage, he/she considers acceptable both static and a mobile architecture. Hence, the general suggested UML support to model a (possibly) mobile architectural style consists of a CD where some messages are unlabeled, some can be labeled with the (possibly constrained) `moveTo` stereotype, and some with the `moveTo?` stereotype. The former two cases correspond to a situation where the designer feels confident enough to decide about the best architectural style, while the latter to a situation where the designer lacks such a confidence. In the following, we call "mobile UML" the proposed UML notation for mobile software architectures.

2.3.2 Target models for performance analysis of mobile software architectures

In general, suitable target models are stochastic models that describe the system dynamics, whose evaluation provides information about the performance one can expect by adopting a mobile architectural style. Moreover, if also the uncertainty about the adoption of a mobile style has been expressed in the architectural model, the selected target model should express in some way also this uncertainty, and the insights gained by the model solution should allow the designer to remove the uncertainty. In terms of the modeling notations described in the previous section, this means substituting the "**?**" value of the M parameter in the *Mob_Comm* connector with a **true** or **false** value, or substituting the `moveTo?` messages in the preliminary "style" CD with (possibly constrained) `moveTo` messages, or with no such message at all, if the obtained insights provide evidence that a static architectural style is more advantageous.
Having this goal, two different target models have been selected, namely, a Markov Reward/Decision Process (MRP/MDP) [P94] or a queueing network (QN) model [J90].

Markov Reward/Decision Process as target model
This kind of model has been adopted when the performance attributes of interest are mainly interaction-related measures (e.g., generated network traffic), without considering possible contention with other applications. We recall here that a MRP models a state transition

system, where the next state is selected according to a transition probability that only depends on the current state. Moreover, each time a state is visited or a transition occurs, a reward is accumulated, that depends on the involved state or transition. Typical measures that can be derived from such a model are the reward accumulated in a given time interval, or the reward accumulation rate in the long period. A MDP extends the MRP model by associating to each state a set of alternative *decisions*, where both the rewards and the transitions associated to that state are decision dependent. A *policy* for a MDP consists in the selection, for each state, of one of the associated decisions, that will be taken each time that state is visited. Hence, different policies lead to different system behaviors and to different accumulated rewards. In other words, a MDP defines a family of MRPs, one for each different policy that can be determined. Algorithms exist to determine the optimal policy with respect to some optimality criterion (e.g. minimization of the accumulated reward) [P94].

Such models embed features suitable in case of mobile software architectures:

- the MRP/MDP probabilistic structure (state transitions and transition probabilities) can be used to model the uncertainty about the actual application execution pattern;
- the MRP/MDP reward structure can be used to model the performance-related "cost" incurred during the application execution;
- the MDP decisional structure (alternative decisions associated to some states) allows us to include in the model the uncertainty about a mobility style, with the choice between alternative mobility options, such as "moving" or "not moving" some components, modeled as the implementation of a given *mobility policy*, i.e. a choice between alternative decisions in the process states.

Execution Graphs and Extended Queueing Networks as target models

These kinds of models have been adopted in [GM02] in the framework of classic SPE techniques [S90, SW02], when the performance attributes of interest are delay-based measures like throughput or response time and we are possibly interested in considering contention with other applications on the use of system resources. In particular, the considered target models, namely, Execution Graphs (EG) [S90] and Extended Queueing Network Models (EQNM) [J90], have been improved in [GM02] by adding some new feature, to make them suitable for the modeling of mobile software architectures.

An EG is a graph whose nodes represent software workload components and whose edges represent transfers of control. Each node is weighted by a demand vector that represents the resource usage of the node (i.e., the demand for each resource). In the SPE framework this kind of model can be used to capture the essential aspects of software behavior in a model separate from the executing platform model, and is suitable, by itself, for the analysis of performance attributes without consideration to the impact of contention. In case of mobile software architectures, the EG formalism is enough to express the cost of components mobility, since it is sufficient to introduce, in opportune points of the EG that models the application behavior, nodes that model the resource consumption caused by component mobility. However, to express in this model also the uncertainty about the possible adoption of code mobility, we have extended the EG formalism. The new formalism, called *mob?*-EG, extends the EG formalism because of the presence of a new kind of node, called *mob?*, characterized by two different outcomes, "*yes*" and "*no*", that can be non-deterministically selected, followed by two possible EGs. The EG corresponding to branch "*yes*" models the application behavior when a component mobility style is adopted, while the EG of the branch "*no*" models the application behavior in the static case. *mob?* nodes have a "null cost", since

they simply model a (non-deterministic) alternative in the path followed by the application. It should be noted that each path in the *mob?*–EG corresponds to a different mobility strategy, concerning when and where components move.

EQNMs are well-known QN models, typically used to analyze delay-based performance attributes, taking into account resource contention. In the SPE framework the overall EQNM, modeling an entire system (software application and hardware platform) is obtained by appropriately merging the EG model of the software application with a QN model of the hardware platform. Once the EQNM is completely specified, it can be analyzed by use of classical solution techniques (simulation, analytical technique, and hybrid simulation) to obtain performance indices such as the mean network response time or the utilization index.
Analogously to EGs, also EQNMs provide tools sufficient to model the cost of component mobility that can be modeled as the visit to appropriate service centers. However, an extension is again necessary if we want to model also uncertainty about components mobility. To this purpose, the extension introduced in [GM02] is based on the definition of new service centers, called *r?(outing)*, that model the possibility, after the visit to a service center (and therefore the completion of a software block) to choose, in a non-deterministic way, which is the routing to follow: the one modeling the static strategy or the one modeling the mobile strategy.
In such a way, a job visiting center *r?* generates two different mutually exclusive paths: one path models the job routing when the component changes its location, the other one models the routing of a static component. Note that, as node *mob?* in the EG, nodes *r?* are characterized by a null service time, since they only represent a routing selection point. The obtained model is called *mob?*-EQNM and is characterized by different routing chains starting from nodes *r?*. It should be noted that these different routing chains are mutually exclusive; in other words a *mob?*-EQNM actually models a family of EQNMs, one for each different path through the *r?* nodes, corresponding to a different mobility policy.

2.3.3 From mobile architecture models to performance models

In this section we outline methodologies that have been proposed for the translation from the mobile architecture models described in section 2.3.1 to the performance models described in section 2.3.2. In particular, translation methodologies have been proposed from the formal language based on an extension of the COMMUNITY language to MRP/MDP; from mobile UML to MRP/MDP; from mobile UML to *mob?*-EQNM. A common assumption for all these methodologies is that the starting models of a mobile software architecture are augmented with appropriate annotations expressing the “cost” of each interaction with respect to a given performance measure. For example, if we are interested in the generated network traffic, these annotations should include at least the size of each exchanged message.

From “extended COMMUNITY” to MRP/MDP
The process of deriving and solving the MRP/MDP model from the “extended COMMUNITY” mobile software architecture description consists of three steps: derivation of a labeled graph from the architecture model; construction of a MRP/MDP based on the obtained labeled graph; MDP simplification and solution [CG02].

The first step consists in constructing a labeled graph LG, i.e. a graph where each node represents a given application state (e.g. components locations and internal state), and each arc represents the execution of a given action in the system (e.g. the transfer of a value during

an interaction). Each arc label contains information about the cost of the corresponding action. This cost must fairly be related to the performance index we want to evaluate. For example, if the performance index we are interested in is the total network traffic over wireless links, then the cost of an action that transfers a value x from a component c1 to a component c2 could be *sizeof*(x) if c1 and c2 communicate through a wireless link, and 0 otherwise. The derivation of LG from the architecture description may be automatically performed, e.g. using a transition system that expresses the operational semantics of the adopted language. Because of the introduction of the *Mob_Comm* connector, with the condition M that can take the "**?**" value, some pairs of outgoing arcs from a node could model the two non-deterministic alternatives concerning the mobility of some component. We call this node as a *decisional* node.

In the second step, LG is used as a "skeleton" around which a suitable MRP/MDP can be built. In particular, *states* and *transitions* of MDP correspond to nodes and arcs of LG, respectively, while the *reward* to be associated to each MDP transition is given by the cost label of the corresponding LTS arc; with regard to MDP *decisions*, only the MDP states obtained from decisional nodes have two alternative decisions, that correspond to selecting the mobility or no-mobility option (with the corresponding cost). To complete the MDP construction, we only need to attach suitable probabilities to the process transitions. This is the only step that cannot be performed automatically, and requires human intervention.

Finally, in the third step we first try to alleviate the computational problems caused by the state explosion. To this purpose, a simplification procedure have been defined in [CG02] that drops all the non-decisional states whose outgoing arcs have cost equal to zero, and modify transitions among the remaining nodes so that the new process is equivalent to the original one (in the sense that the policy that optimizes the original MDP also optimizes the reduced one). This reduction can be performed automatically. After this simplification, the MDP can be solved, finding a *policy* that selects a decision in each state, so that the total accumulated reward is optimized. In the adopted perspective, if the obtained optimal policy selects in some states the mobility option, this can be considered as an indication that code mobility may represent an effective style for the considered application.

From "mobile UML" to MRP/MDP

In the translation methodology adopted in [GM01], a MRP/MDP state corresponds to a possible configuration of the components location, while a state transition models the occurrence of an interaction between components or a location change, and the associated reward is the cost of that interaction. In case of MDP, the decisions associated to states model the alternative choices of mobility or no mobility as architectural style, for those components that are the source of a `moveTo?` message. The translation method from the mobile UML model to the MDP consists in the definition of some elementary generation rules, that define the contribute of the interactions between any pair of components to the overall state transitions (frequencies and rewards, possibly decision dependent). These rules exploit information extracted from the UML diagrams (SD and CD) used to model the mobile software architecture. These rules provide the basis for the definition of a MDP generation algorithm.

Once the MDP has been generated, it can be solved to determine the optimal policy, that is the selection of a decision in each state that optimizes the reward accumulated in the corresponding MRP. Of course, the optimal policy depends on the values given to the system

parameters (e.g., the size of the messages and of the possibly mobile component). Different values for these parameters model different scenarios.

***From "mobile UML" to mob?*-EG *and mob?*-EQNM**

This translation methodology has been proposed in [GM02], based on SPE techniques. As explained before, to cope with mobility, and uncertainty about it, EG and EQNM formalisms have been extended by defining the *mob?*-EG and *mob?*-EQNM formalisms. The proposed SPE-based methodology to derive performance models expressed in these new formalisms from the "mobile UML" diagrams that describe a mobile software architecture starts with the addition of appropriate performance annotations to the UML diagrams (in our case SDs and CDs). Then, the SDs and CDs are processed to get a *mob?*-EG, that describes the software dynamics, and the mutually exclusive paths corresponding to different components mobility strategies.

Example. The structure (without labels showing performance related information) of the *mob?*-EG derived from the SD and CD of figure 9 is illustrated in Figure 10.

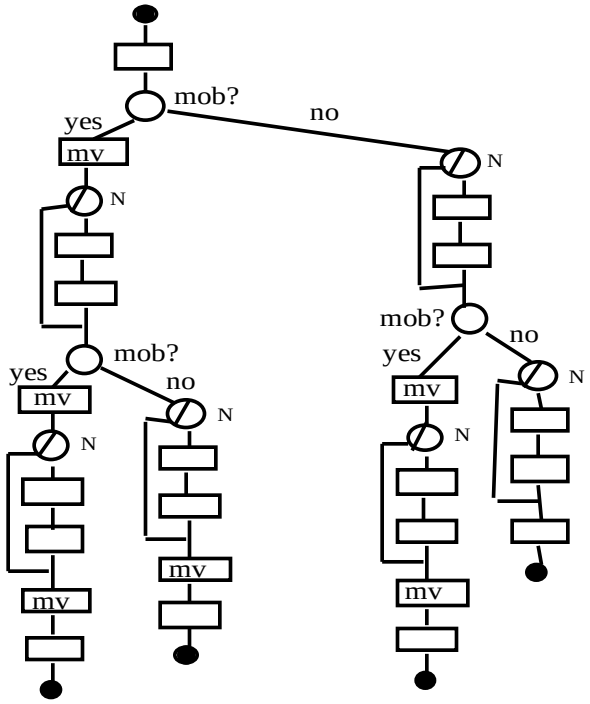


Figure 10. Example of *mob?*-EG.

As explained before, *mob?*-EG can be considered by itself as the target model for a first kind of performance evaluation corresponding to the special case of a stand-alone application where the application under study is the unique in the execution environment (therefore there is no resource contention). In this case performance evaluation can be carried out by standard graph analysis techniques [S90] to associate an overall "cost" to each path in the *mob?*-EG as a function of the cost of each node that belongs to that path. Note that each path in the *mob?*–EG corresponds to a different mobility strategy, concerning when and where components move. Hence these results provide an optimal bound on the expected performance for each strategy, and can help the designer in selecting a subset of the possible

mobility strategies that deserve further investigation in a more realistic setting of competition with other applications.

Once the *mob?*-EG has been obtained (and possibly processed for the above described stand-alone analysis), the last step of the proposed methodology consists of the merging of the *mob?*-EG with a QN modeling the executing platform. The merging leads to the complete specification of a *mob?*-EQNM by defining job classes and routing, using information from the blocks and parameters of the *mob?*-EG.

Example. Figure 11 illustrates an example of *mob?*-EQNM derived from the *mob?*-EG shown in Figure 10, exploiting also information about the execution platform (e.g., obtained from a UML Deployment Diagram). The figure evidences the mutually exclusive routing chains, after traversing the *r?* nodes.

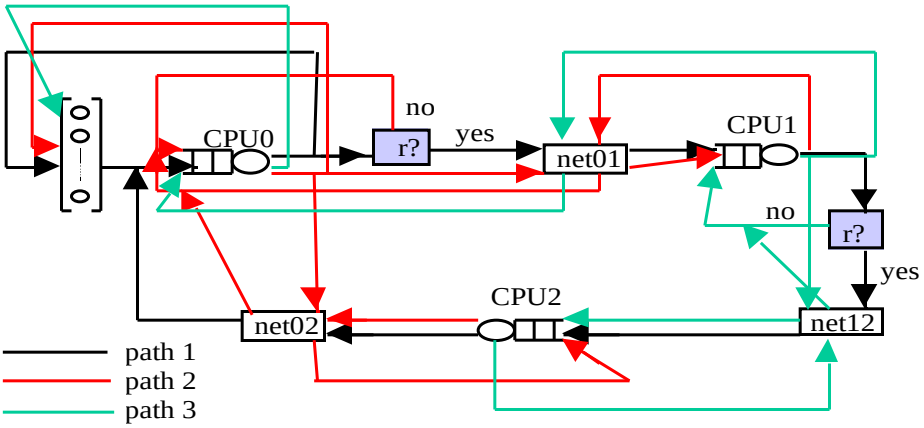


Figure 11. Example of *mob?*-EQNM.

The solution technique suggested in [GM02] for *mob?*-EQNM is based on solving the *mob?*-EQNM through well assessed techniques [J90, S90], separately considering each different EQNM belonging to the family modeled by the *mob?*-EQNM. When the number of different EQNMs is high, this solution approach could result in a high computation complexity. This problem can be alleviated by exploiting results from the stand-alone analysis of the *mob?*-EG. However, more efficient solution methods deserve further investigation. Starting from the obtained results it is possible to choose the mobility strategy, which is *optimal* according to the selected criterion, for example the one that minimizes the response time.

3. Discussion and open problems

In the previous section we have presented some approaches for quantitative software performance evaluation to validate the architectural design choices against system performance requirements. The method presented in Section 2.1 that derives a QN model from a SA dynamic model based on a LTS allows to carry on model evaluation either symbolically, by analytical methods or by simulation, taking advantage of the several techniques and efficient algorithms for QN model solution. The method applies at the architecture level so providing the designer with a framework to support subsequent design

decisions at this level of abstraction. Open research concerns the investigation of more conventional design formalisms, like UML as ADL. Some preliminary results of a method based on Message Sequence Charts instead of a complete LTS to model the dynamic behavior of SA [AABI00b] to model the system dynamic suggest that we can avoid the state explosion problem and use standard design artifact like MSC. Other approaches can be found in [WOSP00, WOSP02, BS01a, CM00, CM02].
Further research has to define a complete integration of such a performance evaluation methodology in architectural design tools. To this aim the derivation of the performance model has to maintain a clear correspondence between SA components and QN components, thus making it easier a direct interpretation and feedback of the performance analysis results at the architectural level. This should allow carrying on the system development steps by providing suitable refinements of the system performance requirements.

Concerning the developed translation methodologies from mobile architecture models to performance models presented in Section 2.3, the approaches are twofold. They are either targeted to the same family of performance models (MRP/MDP), starting from different mobile architecture notations (extended COMMUNITY or mobile UML), or start from the same mobile architecture notation (mobile UML) to get different performance models (MRP/MDP or QN models). The two kinds of considered target models have been used to validate different kinds of performance attributes, hence in the following discussion we focus on the merits and drawbacks of the two adopted architectural notations.
The merit of formal languages, like “extended COMMUNITY”, comes primarily from their lack of ambiguity in architecture modeling, and their precise compositional features. However, their use as starting point for a translation into a performance model (usually, as in our case, of Markov type) easily lead to face the state explosion problem. This drawback is generally exacerbated by the need of associating meaningful transition rates and rewards to all the states of the generated model, which is generally not an easy task. For this latter problem, one way to alleviate it could be based on bridging the gap between formal notations and other notations (like UML) more widely used by software designers, to facilitate the extraction of the required detailed information from artifacts produced by the designers.
On the other hand, the use of semi-formal notations like UML from which to derive performance models is not immune from problems as well, because of the potential ambiguities introduced in the architecture description, so that the general problem of deriving meaningful performance models from UML artifacts deserves further investigation by itself. Also UML modeling of mobile software architectures still appears not completely satisfactory, because of the lack of widely accepted models for all the mobile code styles. The proposed "mobile UML" notation is just a first step, specifically aimed at modeling the mobile agent style.

Performance analysis based on SPA discussed in Section 2.2 has the advantage to offer a natural integration between the behavioural model and the performance model. Moreover this approach allows the definition of architectural styles in an operational framework, that makes it easier properties investigation. However, this method has the drawback that performance analysis and model evaluation is limited by the state space dimension that grows exponentially with the system components. Since the analysis is based on the construction and solution of the underlying Markov chain, the performance model does not maintain any structure of the system, thus hampering the interpretation of the performance analysis results at the architectural description level.

In order to overcome these disadvantages, we can consider different performance models associated to the architectural description language Æmilia based on SPA so that we can apply more efficient solution techniques and take advantage of an higher abstraction level that should ease the interpretation of performance analysis. Therefore a possible research direction is exploiting the mapping of the architectural description language to other types of models with these characteristics. A first contribution in this direction is given in [BBS02] that proposes a mapping from Æmilia to open and closed QNs with phase-type arrival and service distributions and FIFO queues. This work can be further extended to more detailed and representative performance models, and the problem of how to feedback the performance results at the architectural description level has to be more deeply investigated.
Another possible perspective for future research is to investigate the extension of SPA based languages such as Æmilia to include mobile operator to represent mobile systems, and to compare this approach with from different mobile architecture notations languages such as COMMUNITY, as described in Section 2.3.

4. Conclusions

We have discussed quantitative analysis of software architectures based on different performance models and languages to represent, evaluate and predict performance characteristics in the software development life cycle. The presented approaches range from the languages and notations definition for quantitative requirements formulation early in the software life-cycle, to methods that identify and develop performance evaluation models to be integrated with functional analysis models. The various approaches based on different kinds of performance models, i.e., SPA, QN and MRP/MDP, allow appropriate modeling of various characteristics of distributed and mobile software systems to derive and evaluate system performance indices. There are several open problems to be considered, such as the incompleteness of information needed to derive performance model from system specification, the state space explosion of the model, and the necessity to completely automate the translation process from specification to performance model definition and evaluation. Future research has to lead to the development of integrated tools of performance analysis and prediction with architectural design tools that also include a clear and efficient feedback process.

References

[AG97] R. Allen, D. Garlan, "A Formal Basis for Architectural Connection", ACM Trans. on Software Engineering and Methodology, 6:213-249, 1997.

[AABI00a] F. Andolfi, F. Aquilani, S. Balsamo, P.Inverardi "On Using Queueing Network Models with Finite Capacity Queues for Software Architectures Performance Prediction". Proc. QNETs 2000, Third Int. Conf. On Queueing Networks with Finite Capacity Queues, Ilkley, UK, July, 2000.

[AABI00b] F. Andolfi, F. Aquilani, S. Balsamo, P. Inverardi. "Deriving Performance Models of Software Architectures from Message Sequence Charts", in [WOSP00].

[ABI01] F. Aquilani; S. Balsamo; Inverardi P. "Performance Analysis at the Software Architectural Design Level". Performance Evaluation, 45, Issue 2-3, 2001, 147-178.

[BW90] J.C.M. Baeten, W.P. Weijland, *Process Algebra*. Cambridge University Press, 1990.

[BKH99] C. Baier, J.-P. Katoen, H. Hermanns, "Approximate Symbolic Model Checking of Continuous Time Markov Chains", in Proc. of the 10th Int. Conf. on Concurrency Theory (CONCUR 1999), LNCS 1664:146-162, Eindhoven (The Netherlands), 1999.

[BDNO 01] S. Balsamo, V. De Nitto Personè, R. Onvural. *Analysis of Queueing Networks with Blocking.* Kluwer Ed., 2001.

[BIM98] S. Balsamo, P. Inverardi, C. Mangano, "Performance Evaluation of Software Architectures", in: ACM Proc. WOSP, Santa Fe, New Mexico 1998.

[BS01a] S. Balsamo, M. Simeoni "On Transforming UML models into performance models" WTUML: Workshop on Transformations in UML, ETAPS 2001.

[BS01b] S.Balsamo, M. Simeoni "Deriving Performance Models from Software Architecture Specifications" Proc. ESM 2001,SCS, European Simulation Multiconference 2001, Prague, 6-9 June 2001.

[BBS02] S. Balsamo, M. Bernardo, M. Simeoni, "Combining Stochastic Process Algebras and Queueing Networks for Software Architecture Analysis", in [WOSP02].

[BCK98] L. Bass, P. Clements, R. Kazman, *Software Architectures in Practice*, Addison-Wesley, New York, NY, 1998.

[BB02] M. Bernardo, M. Bravetti, "Performance Measure Sensitive Congruences for Markovian Process Algebras", to appear in Theoretical Computer Science, 2002.

[BCD01] M. Bernardo, P. Ciancarini, L. Donatiello "Detecting Architectural Mismatches in Process Algebraic Descriptions of Software Systems", Proc. of WICSA 2001, IEEE-CS Press, 77-86, August 2001

[BCD02] M. Bernardo, P. Ciancarini, L. Donatiello, "Architecting Families of Software Systems with Process Algebras", to appear in ACM TOSEM, 2002.

[BCS01] M. Bernardo, W.R. Cleaveland, W.S. Stewart, "TwoTowers 1.0 User Manual", http://www.sti.uniurb.it/bernardo/twotowers/+, 2001.

[BDC02] M. Bernardo, L. Donatiello, P. Ciancarini, "Stochastic Process Algebra: From an Algebraic Formalism to an Architectural Description Language", to appear in 'Performance Evaluation of Complex Systems: Techniques and Tools', LNCS, 2002.

[BF02a] M. Bernardo, F. Franzè, "Architectural Types Revisited: Extensible And/Or Connections", in Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002), LNCS 2306:113-128, Grenoble (France), 2002.

[BF02b] M. Bernardo, F. Franzè, "Exogenous and Endogenous Extensions of Architectural Types", in Proc. of the 5th Int. Conf. on Coordination Models and Languages (COORDINATION 2002), LNCS 2315:40-55, York (UK), 2002.

[BB00] 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.

[BOO99] G. Booch, J. Rumbaugh, and I.Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, New York, 1999.

[CGH99] G. Clark, S. Gilmore, J. Hillston, "Specifying Performance Measures for PEPA", in Proc. of the 5th Int. AMAST Workshop on Real-Time and Probabilistic Systems (ARTS 1999), LNCS 1601:211-227, Bamberg (Germany), 1999.

[CGP99] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking.* MIT Press, 1999.

[CPS93] W.R. Cleaveland, J. Parrow, B. Steffen, "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems", ACM Trans. on Programming Languages and Systems 15:36-72, 1993.

[CIW99] D. Compare, P. Inverardi and A. L. Wolf. "Uncovering Architectural Mismatch in Component Behavior, Science of Computer Programming", 33 (2), 1999, 101-131.

[CG02]V. Cortellessa, V., Grassi , "A Performance based Methodology to Early Evaluate the Effectiveness of Mobile Software Architectures", Journal of Logic and Algebraic Programming, Elsevier, 2002.

[CM00] V. Cortellessa, R. Mirandola, "Deriving a Queueing Network based Performance Model from UML

Diagrams", in [WOSP00].

[CM02] V. Cortellessa, R. Mirandola, "PRIMA-UML: a Performance Validation Incremental Methodology on Early UML Diagrams", to appear in Science of Computer Programming, Elsevier Sciences.

[DFPV98] R. De Nicola, G. Ferarri, R. Pugliese, B. Venneri “KLAIM: a kernel language for agents interaction and mobility”, IEEE Trans. on Software Engineering, vol. 24, no. 5, May 1998, 315-330.

[FPV98] A. Fuggetta, G.P. Picco, G. Vigna “Understanding code mobility” IEEE Trans. on Software Engineering, vol. 24, no. 5, May 1998, 342-361.

[GM01] V. Grassi, R. Mirandola "UML Modelling and Performance Analysis of Mobile Software Architecture", in Proc. UML Conference (UML 2001), LNCS2185, Springer Verlag, October 2001.

[GM02] V. Grassi, R. Mirandola, “PRIMAmob-UML: a Methodology for Performance analysis of Mobile Software Architecture”, in [WOSP02].

[HE98] H. Hermanns, "Interactive Markov Chains", Ph.D. Thesis, University of Erlangen (Germany), 1998.

[HI96] J. Hillston *A Compositional Approach to Performance Modelling.* Cambridge University Press, 1996.

[HO85] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[HO71] R.A. Howard, *Dynamic Probabilistic Systems.* John Wiley & Sons, 1971

[I02] P. Inverardi, "The SALADIN project: Summary Report", Software Engineering Notes, 2002, to appear.

[J90] R. Jain. *Art of Computer Systems Performance Analysis.* Wiley, New York, 1990.

[K76] L. Kleinrock. *Queueing Systems*. John Wiley and Sons, New York, 1976.

[L83] S.S. Lavenberg. Computer Performance Modeling Handbook. Prentice Hall, 1983.

[MT00] N. Medvidovic, R.N. Taylor “A classification and comparison framework for software architecture description languages” IEEE Trans. on Software Engineering, vol. 26, no. 1, Jan. 2000, 70-93.

[MRR02] N. Medvidovic, D.S. Rosenblum, D.F. Redmiles “Modeling software architectures in the Unified Modeling Language” ACM Trans. on Software Engineering, vol. 11, no. 1, Jan. 2002, 2-57.

[M89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[M99] R. Milner. *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999.

[NPD01] C. Nottegar, C. Priami, P. Degano “Performance evaluation of mobile processes via abstract machines” IEEE Trans. on Software Engineering, vol. 27, no. 10, Oct. 2001, 867-889.

[PRM01] G.P. Picco, G.-C. Roman, P.J. McCann “Reasoning about code mobility in mobile UNITY” ACM Transactions on Software Engineering and Methodology, vol. 10, no. 3, July 2001, 338-395.

[P94] M.L. Puterman, *Markov Decison Processes*, J. Wiley and Sons, 1994.

[SGW01] B. Selic, G. Gullekson, P.G. Ward, *Real-Time Object Oriented modeling*, John Wiley, 2001.

[S90] C. Smith. *Performance Engineering of Software Systems.* Addison-Wesley, Reading, MA, 1990.

[SW02] C. Smith, L. Williams. *Performance solutions: A Practical Guide to Creating Responsive, Scalable Software.* Addison Wesley, 2002.

[S94] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[WF98] M. Wermelinger, J.L. Fiadeiro, “Connectors for mobile programs”, IEEE Trans. on Software Engineering, vol. 24, no. 5, May 1998, 331-341.

[WOSP00] Proc. ACM WOSP 2000 Second Int. Workshop on Software and Performance (WOSP 2000), Ottawa, Canada, Sept. 17-20, 2000.

[WOSP02] Proc. ACM WOSP 2002 Third Int. Workshop on Software and Performance (WOSP 2002), Rome, Italy, July, 24-26, 2002.