Deriving Performance Models of Software Architectures from Message Sequence Charts

F. Andolfi, F. Aquilani
Area Informatica
Universitá dell'Aquila
I-67010 L'Aquila, Italy
andolfi@univaq.it

S. Balsamo
Dip. di Informatica
Universitá di Venezia
30172 Mestre (VE)
balsamo@dsi.unive.it

P. Inverardi
Area Informatica
Universitá dell'Aquila
I-67010 L'Aquila, Italy
inverard@univaq.it

ABSTRACT

In this paper we present an approach to automatically derive a performance evaluation model, based on a Queueing Network Model, from a Software Architecture (SA) specification. The approach assumes the SA is described by means of Message Sequence Charts (MSCs) whose objects are components and interactions are modelled as message sequences, that is, how messages are sent and received between a number of objects. We analyse MSCs in terms of the trace languages that they generate trying to single out the real degree of parallelism among components and their dynamic dependencies. This information is then used to build a faithful Queueing Network Model (QNM) corresponding to the SA description that forms the basis to conduct performance analysis. The approach builds on our previous work on performance evaluation of a SA in which we assumed the SA description given in the formal ADL based on CHAM (CHemical Abstract Machine). From that description we could build a finite state model (FSM) of the global system behaviour from which we derive a performance model, a QNM. However, this approach has the drawback of high computational complexity due to possible state space explosion of the FSM obtained by the SA description of real world projects. The new approach is proposed to overcome this limitation by considering the MSCs as the SA description from which we derive the performance model. Moreover we observe that in many application domains MSCs or, using the UML terminology, Sequence Diagrams are commonly in use. Our present work is a step towards a solution to the performance model production which is both effective and compatible with many companies software development standards.

Keywords

Software Architecture, MSCs, Performance Evaluation, Queueing Network Model, performance indices

WOSP 2000, Ontario, Canada
© ACM 2000 1-58113-195-X/00/09 ...$5.00

1. INTRODUCTION

In this paper we present an approach for the automatic generation of a performance evaluation model, based on a Queueing Network Model [15], from a Software Architecture (SA) specification described through Message Sequence Charts (MSCs)[17, 2].

Given that our ultimate goal is to make quantitative analysis of complex systems early in the software life cycle possible, there are two main motivations for our work. First, we believe that the performance model creation should be automatized as much as possible. If this is the case the starting software artifax for the creation must be an accurate specification of the dynamic system behavior, in our case the software architecture dynamic behavior. Second, the dynamic model must be tractable and practical, where tractable means that it is possible to represent and manipulate it and practical means that we could expect system designer to produce it. In the following we will try to illustrate our approach in light of these two main motivations.

MSCs exist in several variations and have been widely used as a design tools in many event-based distributed systems. Recently, with the increased popularity of UML, some attempts at the use of Sequence Diagrams (SD), the UML [2, 14] counterparts of MSCs, as the basis for quantitative analysis of Software Architectures have been proposed [3]. From now on we will interchangeably use the terms MSC or SD. Our approach assumes the SA is described by means of Message Sequence Charts where the objects of the MSCs are components and their interactions are modelled as message sequences, that is, how messages are sent and received between a number of objects. We analyse MSCs in terms of the trace languages [5] they generate trying to single out the real degree of parallelism among components and their dynamic dependencies. This information is then used to build a faithful QNM model corresponding to the SA description that forms the basis to conduct performance analysis. The approach builds on our previous work on performance evaluation of a SA in which we defined a methodology for software performance evaluation at the SA level which assumed the SA description given in the formal ADL based on CHAM (CHemical Abstract Machine) [8, 6]. From that description we could build a finite state model (FSM) of the global system behaviour whose analysis lead to the definition of a performance model, i.e. a QNM. In the new approach we automatically build the QNM from the analysis of the MSCs

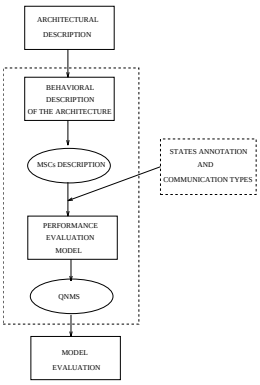

Figure 1: Phases of our approach.

instead of the FSM. A drawback of the previous approach is that in real world projects often even at the SA description level FSMs are too big to be represented (due to the state explosion problem). Our new approach tries to overcome this limitation by using MSCs. Moreover MSCs are commonly in use in many application domains and constitute a relevant part, if not the only one, of the design artifacts dealing with the dynamic behaviour of the system.

MSCs are always described in a limited number, which supposedly describe from the designer point of view, all the *relevant* system dynamic behaviors. Although they represent an incomplete description of the global system behavior they are often enough to meaningfully characterize it. Thus MSCs are a tool that can be effectively used by the system designers and are tractable.

Referring to the methodology for performance evaluation of a SA, as sketched in Figure 1, in this paper we propose to replace the performance model creation phase.

The paper is organized as follows: in Section 2, the MSCs formalism and its use to specify software architecture are introduced. The description of a simple system, the Compressing Proxy, which will be used throughout the paper to illustrate the performance model creation is also presented. Section 3, briefly summarizes the characteristics of Queueing Network Models (QNMs) and of their creation process. Section 4, presents the theory underlying the derivation of the QNM from the MSCs specification of a Software Architecture. The application of the theory to the Compressing Proxy system is carried out in Section 5. Section 6 adds some concluding remarks and discusses future work. The Appendix contains a description of the algorithm to generate the QNM model.

2. MSCS AND SA

In the general understanding, MSCs illustrate how objects interact with each other [13]. They focus on message sequences, that is, how messages are sent and received between a number of objects. A sequence diagram also reveals the interaction for a specific scenario i.e. a specific interaction between the objects that happens at some point in time during the system's execution. On the horizontal axis there are the objects involved in the sequence. Each one is represented by an object rectangle with the object name. The vertical dashed line represents the object's lifeline. Communications between objects are represented as horizontal arrows between object's lifelines. The end of arrows indicates the type of communication. In our approach the objects are substituted by architectural components and obviously the arrows indicate the time when communication occurs. In this paper we assume that the MSCs descriptions we deal with satisfy the following properties:

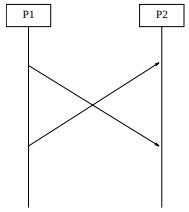

Figure 2: A MSCs not verifying the non degeneracy property.

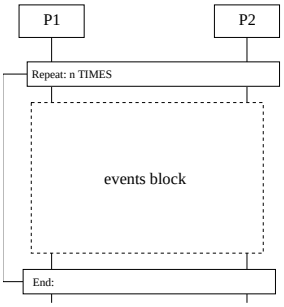

Figure 3: MSC with a repeat cycle.

1. They contain state information about each component;
2. Each system component is contained in a MSC;
3. MSC can contain repeat cycles (Fig.3);
4. They verify the non degeneracy property [5] (Fig.2), roughly this means that arrows can only occur horizontally;
5. The MSCs must be representative of the major system's behaviours;
6. The MSCs refer to the same initial configuration.

2.1 The Compressing Proxy Architecture

In this section we present the design of the Compressing Proxy system. Our description is derived from that given in [10]. To improve the performance of UNIX-based World Wide Web browsers over slow networks, one could create an HTTP (Hyper Text Transfer Protocol) server that compresses and uncompresses data that it sends across the network. This is the purpose of the Compressing Proxy, which

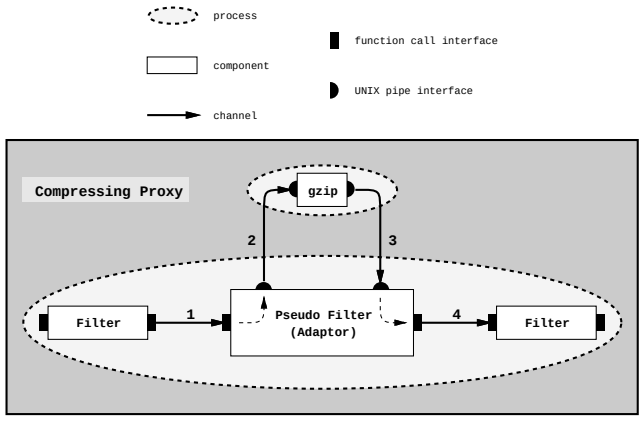

Figure 4: The Compressing Proxy.

weds the **gzip** compression/decompression program to the standard HTTP server available from CERN. A CERN HTTP server consists of *filters* strung together in series. The filters communicate using a function-call-based stream interface. Functions are provided in the interface to allow an upstream filter to "push" data into a downstream filter. Thus, 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. Therefore, it uses the standard UNIX input/output interface, and communication with **gzip** occurs through UNIX pipes. An important difference between UNIX filters, such as **gzip**, and the CERN HTTP filters is that the UNIX filters explicitly choose when to read, whereas the CERN HTTP filters are forced to read when data are pushed at them. To assemble the Compressing Proxy from the existing CERN HTTP server and **gzip** without modification, we must insert **gzip** into the HTTP filter stream at the appropriate point. But since **gzip** does not have the proper interface, we must create an adaptor, as shown in Figure 4. This adaptor acts as a pseudo CERN HTTP filter, communicating normally 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.

3. THE QUEUEING NETWORK MODEL

Queueing Network Models (QNMs) have been extensively applied to represent and analyse the performance of resource sharing systems, such as production, communication and computer systems. A QNM is represented as a *network of queue* which is quantitatively evaluated through analytical or simulation methods to obtain a set of measures, the *performance indices*. A network of queue is a collection of *service centers*, which represent system resources, and a set of *customers*, which represent users sharing the resources. A QNM can be represented as a directed labelled graph where nodes are service centers, arcs represent transitions of customers along nodes and labels on arcs define the routing probability. Different types or classes of customers with various characteristics of customer service and routing behaviour can be modelled by multiclass queueing networks. In single class QNM all the customers have the same behaviour. QNM can be open, closed or mixed. In an open network model there is at least one arc entering and at least one arc going out the network. In an open network each customers arrives from outside, asks for service at a service center, possibly queueing for it, and then it either leaves the system or asks for service to some other service center. In a closed network there is no external arrival and exit arc and there is a constant number of customers circulating in the system. Multiclass networks can be either open, closed or mixed. A mixed network is open with respect to some customer types and closed with respect to others. The queueing capacity of each service center is formed by a set of servers and a queue to keep waiting customers. The queue capacity can be finite to represent system resources with limited waiting rooms or population constraints. QNMs with finite capacity queues are more appropriate models for systems with finite buffer and blocking, but their analysis is in general more complex than infinite capacity QNMs and it is often carried on with approximated techniques [18]. This class of QNMs is referred to as Queueing networks with Blocking [7, 15, 18]. When a user tries to enter a saturated node, i.e. a node whose queue has attained its maximum capacity, then there is a *block*. In the literature various models have been proposed to represent different types of blocking mechanisms. We just recall the Blocking After Service (BAS) mechanism that we shall apply to model system synchronization. Under BAS policy when a customer, after completion of service at node i, tries to enter a saturated node j, is blocked and has to wait at node i, so blocking the service. The service remains blocked until a buffer position becomes available in node j, i.e. as soon as a customer leaves node j. When more than one node are blocked by a saturated node, we have to define an unblocking discipline. We refer to FBFU (First Blocked First Unblocked) discipline, where the first unblocked node is the first one that has been blocked. For simplicity, hereafter we only deal with open models. Informally a QNM can be described by three basic elements:

- **service centers** also called nodes
- **customers**
- **node inter-connection structure** (or topology)

The modelling process with a QNM can be split in three steps: (1) *definition*, which normally encompasses the definition of service centers, their number, class of customers and topology; (2) *parameterisation*, 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 behaviour described by the QNM.

4. FROM THE MSC*s* TO THE QNM

In this section we present the algorithm to derive the QNM performance evaluation model out of the MSCs specifications of a software architecture. This approach is justified by the fact that the automata describing the whole system is often too large. The main idea is to begin from a high level description of components interactions to extract qualitative system's information. To this aim, in our opinion, the MSCs formalism is a good choice.

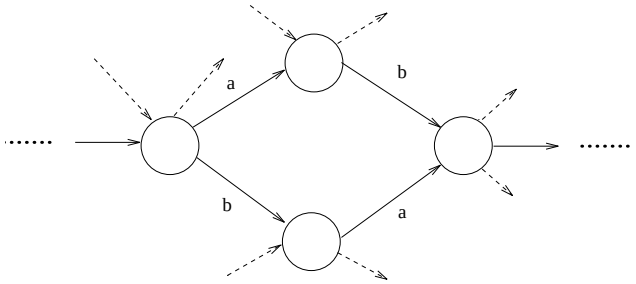

Figure 5: Example of a complete portion of automata

In order to derive the QNM we must have information such as communication among components, communication types, concurrency and ldeterminism among components. All these information are dynamic so we can take them out only of a behavioural model of the architecture. MSCs allows for modelling dynamic behaviour, the common assumption here, see for example [5], is that it is always possible to synthesize a global finite state system model out of a set of MSCs. A MSC then represents a valid path on this automata. In the following we will refer to this intended global model as the associated automata. In this paper we will deliberately not address the issue of which intended model underlies a MSCs description, several approaches exist. For the sake of simplicity, in our context it is enough to know that a suitable complete model can be synthesized.

The idea, which follows the approach in [5], is to derive from each MSC a *regular expression* describing the events sequence performed in the MSC. Then we consider regular expressions analysis. The analysis consists of comparing regular expressions in order to find out a prefix. When we have found this prefix we analyse the remaining parts of the regular expressions; if these have particular shape (as listed below) we can deduce the needed information. This information is collected into sets, called *interaction sets*. To make this kind of reasoning as simpler as possible we make the following non restrictive assumptions on the system's model:

- the primitive basic communication between components is one to one;
- the associated automata is complete, i.e. from every state it is possible to do all possible actions. With this assumption we can get the parallelism between components. (Fig.5);

When all interaction sets are built we derive the QNM; i.e. we generate inter-connection service centers with finite and infinite queues, modelling respectively, synchronous and asynchronous behaviours.

4.1 MSCs Encoding

Following the idea in [5], in order to process algorithmically the MSCs, we need an encoding. Let $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$ be the MSCs set describing the architectural behaviour; given a MSC $M_i \in \mathcal{M}$ on components $P_1, P_2, \ldots, P_m$, we introduce the following definitions:

DEFINITION 1. *An* event e_j *on a generic MSC* M_i *is a communication among two components, denoted by an arrow.*

DEFINITION 2. *Given two events* e_j, e_k *we define a* visual ordering $<_i$: $e_j <_i e_k$ *if and only if the event* e_j *precedes the event* e_k *in the MSC* M_i *temporal sequence.*

DEFINITION 3. *Given the event* e_j *the associated* label *is:* $S(P_1, P_2)^c$ *where* P_1 *and* P_2 *are the sending and receiving component respectively, in the communication described by the event* e_j, $c \in \{s, a\}$ *is the communication type, where* s *stands for synchronous and* a *for asynchronous communication.*

DEFINITION 4. *Given events* $e_1, \ldots, e_j$ *the associated* trace *is the regular expression* $l_1 \ldots l_j$ *where:*

- l_k *corresponds to the label of the event* e_k *or to a regular expression* $\{b_1 \ldots b_q\}^n$ *where* $b_1 \ldots b_q$ *is the trace associated to the events block as showed in Figure 3;*
- *for all* $s, t \in \{1, \ldots, j\}$ *if* $s < t$ *then* $e_s <_i e_t$.

Following the above definitions we can encode the MSCs. First we generate the labels associated to the events; then we go on building the traces according to the visual ordering given by the events sequences in the considered MSC. During the labels generation we produce the *Based Interaction Sets* representing all the communication in the MSCs. In particular if, in the i-th step, we generate a label $S(P_1, P_2)^c$ then we put the $(P_1, P_2)^c$ (i.e. the *Interaction Pair*), in a set called Based Interaction Set $I_i = \{(P1, P2)^c\}$.

4.2 Traces Analysis

In this section we describe the traces analysis in order to generate the *Structured Interaction Sets* composed of two Interaction Pairs corresponding to concurrent components, *one* to *two*, *two* to *one* and alternative (non deterministic) communications. The trace analysis is based on a comparison of pairs of traces searching a matching among their prefixes. In our approach, in fact, a trace singles out an automata computation, so a common prefix corresponds to the time in which two computations begin to have distinct behaviours. This traces interpretation explains the hypothesis 6 made in Section 2 on the MSCs initial configuration. Intuitively this idea is justified by the following arguments:

Let T_1, T_2 *be traces and* X *be the common prefix. If the states reached by execution of prefix* X *are not in conflict (conflict is a relation on states components that establishes when, given a component, two states represent two mutually exclusive component behaviours), we affirm that there exists a global state in the complete automata reachable by the trace* X *and from which it is possible to perform the remaining part of traces* T_1 *and* T_2.

In the following we illustrate, for each type of pair of traces, the corresponding generated Structured Interaction Set .

Trace types:

- A Trace pair models a *concurrent behaviour* if it has a labels exchange as:

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_i,P_j)^{c_3}S(P_s,P_l)^{c_4}\dots$$

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_s,P_l)^{c_4}S(P_i,P_j)^{c_3}\dots$$

where $i \neq s,t$, $s \neq t$,$j \neq i$ and $j \neq s$. In this case we build the Structured Interaction Pair:

$$I_i = \{(P_i,P_j)^{c_3},(P_s,P_l)^{c_4}\}.$$

- A Trace pair models a *one to two communication* if it has a labels exchange as:

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_i,P_j)^{c_3}S(P_i,P_s)^{c_4}\dots$$

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_i,P_s)^{c_4}S(P_i,P_j)^{c_3}\dots$$

where $j \neq s$ and $i \neq j,s$. In this case we build the Structured Interaction Pair:

$$I_i = \{(P_i,P_j)^{c_3},(P_i,P_s)^{c_4}\}.$$

- A Trace pair models a *two to one communication* if it has a labels exchange as:

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_j,P_i)^{c_3}S(P_s,P_i)^{c_4}\dots$$

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_s,P_i)^{c_4}S(P_j,P_i)^{c_3}\dots$$

where $j \neq s$ and $i \neq j,s$. In this case we build the Structured Interaction Pair:

$$I_i = \{(P_j,P_i)^{c_3},(P_s,P_i)^{c_4}\}.$$

- A Trace pair models an *alternative communication* if it has a labels exchange as:

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_i,P_j)^{c_3}S(P_s,P_l)^{c_4}\dots$$

$$S(P_x,P_y)^{c_1}\dots S(P_k,P_z)^{c_2}S(P_i,P_q)^{c_5}S(P_r,P_u)^{c_6}\dots$$

where $r \neq s,u \neq t$, $i \neq j,q$ and $j \neq q$. In this case we build the Structured Interaction Pair:

$$I_i = \{(P_i,P_j)^{c_3},(P_i,P_q)^{c_5}\}^{ND}.$$

In the following we summarize the Structured Interaction Sets generation algorithm:

For All Traces S_i **do**
  For All Traces S_j and $i \neq j$ **do**
    <Find all common prefix p >
    <Generate Interaction Set for p
    from the above rules>

4.3 QNM Generation

Once we have analysed all traces pairs, the I_i computed sets are examined in order to associate them to elements of the QNM. Additional information is necessary to determine the QN topology. In particular we need to identify internal and external components of the system. External components can be seen as sources which model the production of system customers or processing elements from which the system communicates the results to the environment. From these components we determine whether the resulting QNM is open or closed. Without loss of generality we assume that components are partitioned into two set *Ex* and *In*. Let us informally introduce how the generation QNM algorithm works. The idea is to analyse the interaction among system components (i.e. I_i sets) to understand their *real* level of concurrency and consequently generate the corresponding QNM. This is very important in order to obtain QNMs that faithfully represent the given system. The goal is to understand which components are strongly synchronized so that they result in a sequential behaviour and which are independent from each others and can be concurrently active. At the beginning each component is considered as an autonomous server, along the computation it can become part of a more structured element. Intuitively, an Interaction Set $I_i = \{(P_1,P_2)^s\}$ is modelled as a complex server representing a unique service composed of P_1 following P_2 and that expresses the sequentiality of operations. An Interaction Set $I_i = \{(P_1,P_2)^a\}$ is modelled by a service center with an infinite buffer that implicitly models the communication channel. Moreover the algorithm builds the transition $P_1 \longrightarrow P_2$ in the topology of the QNM. The Interaction sets $I_i = \{(P_1,P_2)^c\}$ having an external element define the service centers in which there are exogenous arrival (the external element is P_1) and the service centers from which there are departure (the external element is P_2). The algorithm models a non-deterministic computation, $\{(P_1,P_2)^s,(P_1,P_3)^s\}^{ND}$, by introducing a multi-customers service center that at the end of the generation process is transformed in a simple-customer service center whose service time depends on the service times of the original classes. To model synchronous communication among concurrent system components, the algorithm assigns distinct service centers to the communicating components in order to model their independence. It also associates to the receiver component a service center with a zero capacity buffer and imposes a BAS blocking mechanism to the sender component in order to model synchronization. One to two, $\{(P_1,P_2)^{c_1},(P_1,P_3)^{c_2}\}$, and two to one, $\{(P_2,P_1)^{c_1},(P_3,P_1)^{c_2}\}$, communications are modelled assigning to the involved components distinct service centers, with a zero capacity buffer if the communication is synchronous. When all the sets I_i have been examined the algorithm proceeds by performing *merging* operations to reach the final configuration of the service centers. There are several merging operations as listed in the last part of the algorithm. The result of these merging operations is a set of interconnected service centers. The interesting reader can find in the Appendix a complete description of the algorithm, as well as the definition and graphical notation of the QNM components elements, with explicit reference to software architectures.

4.4 Performance Analysis

After the generation of the QNM we can analyze the performance model of SA [6, 8]. Due to the high level of abstraction od SA, this can be done either symbolically through analytical techniques or by simulation. One can devise the performance of the given architecture under further hypothesis by suitably instantiating the symbolic parameters or choosing their values. This information identifies potential implementation scenarios. Analyzing the behaviour of the systems in these scenarios can provide useful insigths on how to carry on the development process in order to satisfy the chosen performance criteria.

5. DERIVING THE QNM FOR THE COMPRESSING PROXY

In this section we derive the QNMs associated with the Compressing Proxy. First we describe the architectural behaviour by the following MSCs, realized by the tool [17], the star (*) symbol stands for a *don't care* state label, i.e. it matches with any state labels:

We now apply the methodology presented in Section 4, to carry on the modelling process.

5.1 Encoding of Compressing Proxy MSCs

From the definitions in Section 4.1 we derive the traces. Since we assume that all communications are synchronous, we can omit the notation for the communication type in the label.

MSC1BIS-Trace:

$\{S(CF_u, Ad)S(Ad, Gzip)^{N_1}S(Gzip, Ad)^{N_2}S(Ad, CF_d)\}^N$

MSC8BIS-Trace:

$S(CF_u, Ad)S(Ad, Gzip)^{N_1}S(Gzip, Ad)^{N_2}S(CF_u, Ad)S(Ad, CF_d)$

MSC9BIS-Trace:

$S(CF_u, Ad)S(Ad, Gzip)^{N_1}S(CF_u, Ad)S(Gzip, Ad)^{N_2}S(Ad, CF_d)$

MSC10BIS-Trace:

$S(CF_u, Ad)S(Ad, Gzip)^{N_1}S(Gzip, Ad)^{N_2}$

$S(Ad, Gzip)S(Ad, CF_d)S(CF_u, Ad)$

MSC11BIS-Trace:

$S(CF_u, Ad)S(Ad, Gzip)^{N_1}S(Gzip, Ad)S(Ad, CF_d)$

$S(Ad, Gzip)^{N_2}S(CF_u, Ad)$

5.2 Traces Analysis

From the labels we generate the Based Interaction Sets:

$I_1 = \{(CF_u, Ad)^s\}$

$I_2 = \{(Ad, Gzip)^s\}$

$I_3 = \{(Gzip, Ad)^s\}$

$I_4 = \{(Ad, CF_d)^s\}$

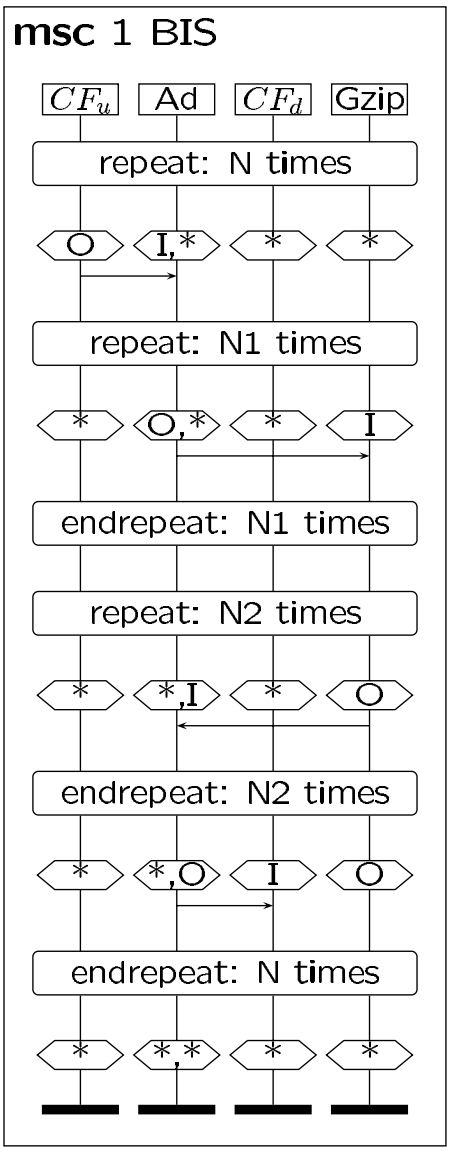

Figure 6: The strictly sequential behaviour between CF_u, Ad, Gzip and CF_d.

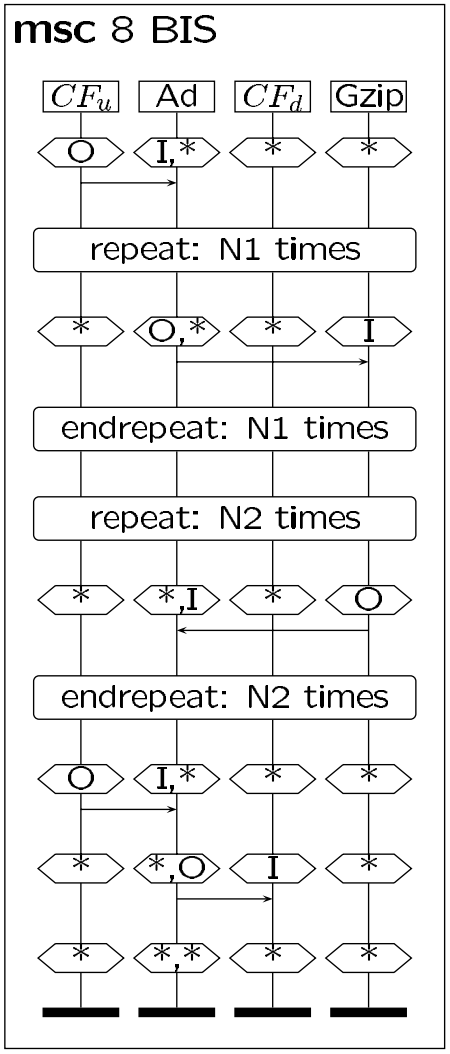

Figure 7: Before Adaptor sends to CF_d it receives a new stream from CF_u.

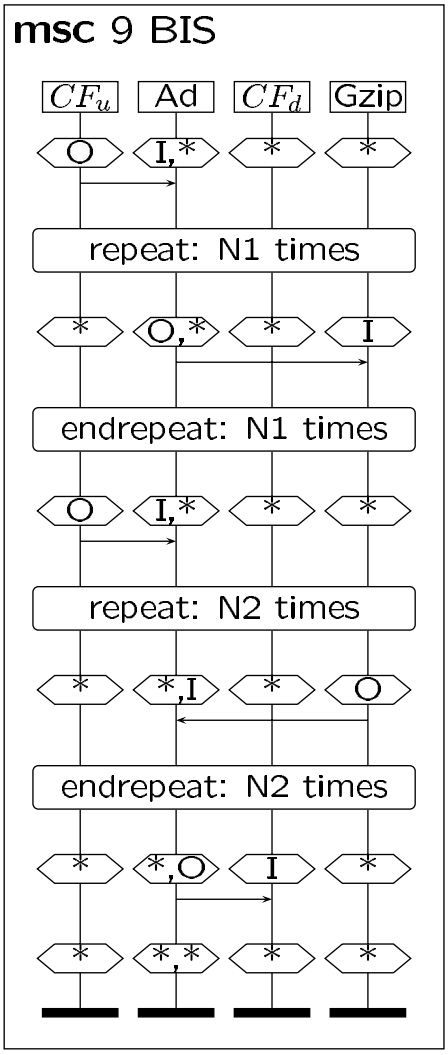

Figure 8: Iterates the communication CF_u to Ad and Ad Gzip, then re-iterates CF_u to Ad before Gzip ends.

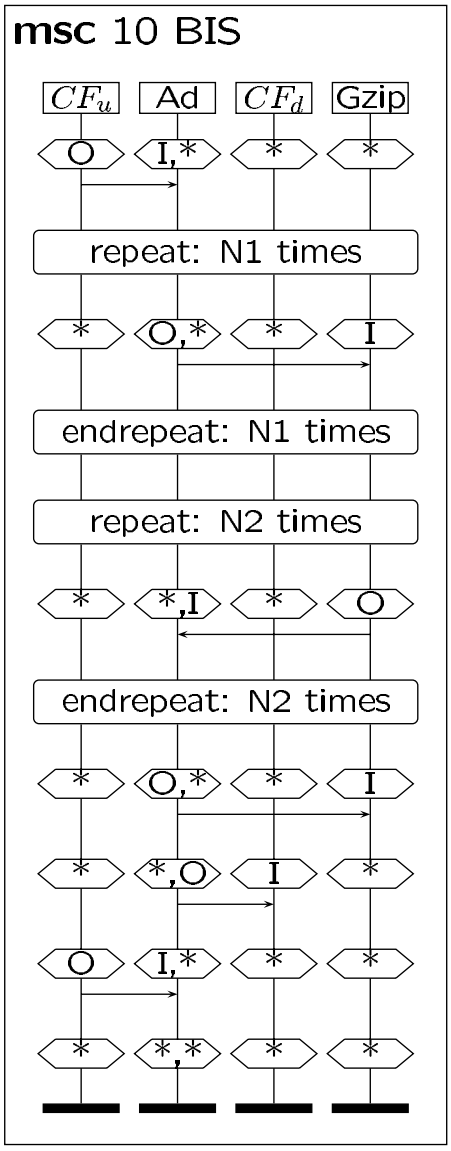

Figure 9: CF_u sends to Ad, Ad sends to Gzip, Gzip must end and Ad sends to CF_d all it did, then Ad is ready to get new input.

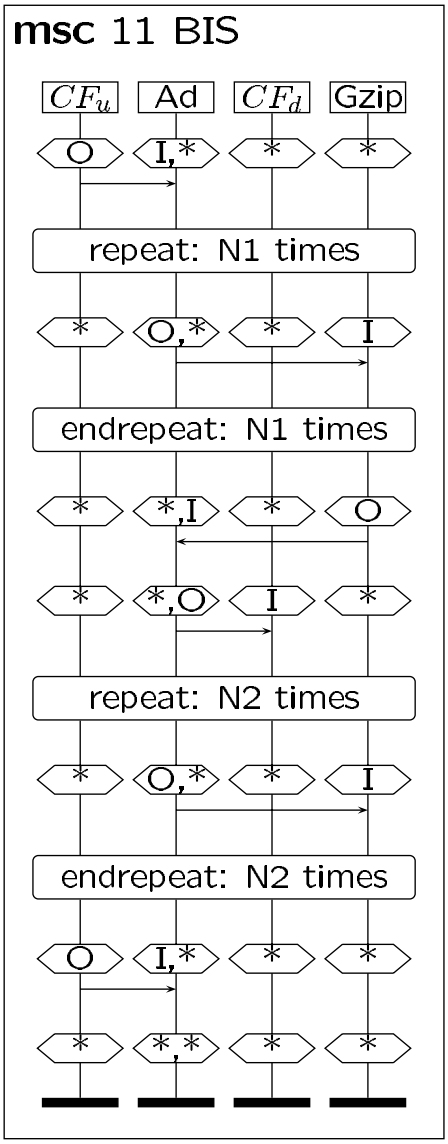

Figure 10: Gzip sends back to Ad before completion of all work, then Ad sends all to CF_d and can input again.

while from the analysis of traces we can extrapolate the following Structured Interaction Sets:

- From a instantiation of MSC8BIS-Trace and MSC9BIS-Trace we can derive $I_5 = \{(Gzip, Ad); (CF_u, Ad)\}$, in fact the partial states $msc8(*, [*, I], *, O)$ and $msc9(O, [I, *], *, *)$ are not in conflict, so the associated Interaction Pairs are in the same Interaction Set:

$$S(CF_u, Ad)S(Ad, Gzip) \begin{array}{l} S(Gzip, Ad)S(CF_u, Ad) \\ S(CF_u, Ad)S(Gzip, Ad) \end{array}$$

- From a instantiation of MSC10BIS-Trace and MSC11BIS-Trace we can derive $I_6 = \{(Ad, Gzip); (Ad, CF_d)\}$, in fact the partial states $msc10(*, [O, *], *, I)$ and $msc11(*, [*, O], I, *)$ are not in conflict, so the associated Interaction Pairs are in the same Interaction Set:

$$S(CF_u, Ad)S(Ad, Gzip)S(Gzip, Ad) \begin{array}{l} S(Ad, Gzip)S(Ad, CF_d) \\ S(Ad, CF_d)S(Ad, Gzip) \end{array}$$

- From a instantiation of MSC8BIS-Trace and MSC1BIS-Trace we can derive $I_7 = \{(CF_u, Ad); (Ad, CF_d)\}$, in fact the partial states $msc8(O, [I, *], *, *)$ and $msc1(*, [*, O], I, *)$ are not in conflict, so the associated Interaction Pairs are in the same Interaction Set:

$$S(CF_u, Ad)S(Ad, Gzip)S(Gzip, Ad) \begin{array}{l} S(CF_u, Ad)S(Ad, CF_d) \\ S(Ad, CF_d)S(CF_u, Ad) \end{array}$$

5.3 QNM Generation

By applying the algorithm described in the Appendix with assumptions $Ex = \{CF_u, CF_d\}$ and $In = \{Ad, Gzip\}$ we obtain the QNM representing the Compressing Proxy dynamic behavior. The resulting model is shown in Figure 11. The model is an open QNM with two service centers with finite capacity. We have exogenous arrivals and departures from the system only from the Ad service center. The tuple

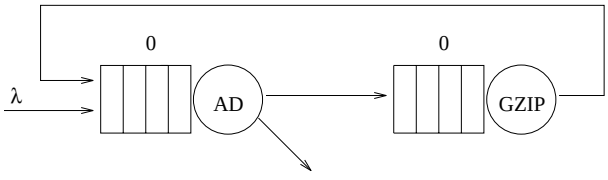

Figure 11: Compressing Proxy QNM.

[CF_u, Ad, Gzip, Ad, CF_d] characterizes the service and it states that a customer requires services to the processing elements in the order indicated in the tuple. The first element of the tuple is an element in A reflecting that the network is open. We assume a $FCFS$ discipline.

6. CONCLUSIONS

In this paper we have presented an approach for the automatic construction of the performance model of software architecture specified in MSCs. The method allows the automatic derivation of a QNM that can then be evaluated. Recently, software architectures have been devised as the appropriate design level for carrying out quantitative analysis [4, 9, 11]. Software architecture represents the first system abstraction in which both the static decomposition of the system in subsystems and the dynamic interactions among subsystems is defined. We believe that, in order to be successful, this kind of analysis has to be automated as much as possible. Our approach is based on the assumption that, in building the evaluation model, as much work as possible has to be automatically derived by the Software Architecture description. This requires to exploit the Software Architecture descriptions. As a matter of fact, MSCs or SDs are commonly used as design tools to model dynamics of complex event-driven software systems. Although necessarily incomplete, the information provided by MSCs can in many practical cases be the only one realistically available. Finite State Models of complex systems are theoretically achievable but practically impossible in many real cases. It is therefore interesting to research in the direction of implicitly synthesizing global information from partial finite representations of the system global behaviour like MSCs. The approach presented in this paper is a first attempt to bypass the state explosion limitation, we are at present experimenting this approach on the motivating case study [12], whose complexity is obviously much bigger than the simple Compressing Proxy system. In that case study, carried out in a joint project with an important Italian telephone company, we were asked to provide the SA description of (part of) one of their product system. The ultimate goal was to make predictive analysis and conduct evaluation of the architectural choices possible, where their primary interest was in predicting system performance behaviour. We used UML as the main description notation and we complemented UML descriptions with the process algebra description language FSP [1, 16], in order to be able to model the global dynamic system behavior. This modelling technique was useless to analyse the system global behavior due to state explosion, thus we resorted to (enriched) sequence diagrams. This was the starting point for the study we presented in this paper.

7. ACKNOWLEDGMENTS

All the authors have been supported by Progetto MURST di rilevante interesse nazionale Saladin, http:saladin.dm.univaq.it

8. REFERENCES

[1] *Finite State Processes (FSP).* on line at: http: www-dsc.doc.ic.ac.uk/jnm/book/ltsa/Appendix-A.html.

[2] *Rational Corporation. Uml Resource Center. UML documentation.* version 1.3. on line at: http://www.rational.com/uml/index.jtmpl.

[3] *Rational Perfomance Architect.* on line at: http://www.rational.com/uml/index.jtmpl.

[4] Ieee proceedings of the first international workshop on software and performance. Wosp98, October 1998.

[5] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In Proc. 22^{nd} Int. Conf. on Softw. Eng. (ICSE2000), Limerick (Ireland), June 4^{th}-11^{th} 2000.

[6] F. Aquilani, S. Balsamo, and P.Inverardi. An approach to performance evaluation of software architectures. *Internal Report Universita' dell'Aquila, submitted for publication*, Marzo 2000.

[7] S. Balsamo. Properties and analysis of queueing network models with finite capacities. *Performance Evaluation of Computer and Communication Systems (L.Donatiello and R. Nelson Eds.)*, (729):21–52, 1994.

[8] S. Balsamo, P. Inverardi, and C. Mangano. Performance evaluation of software architectures. In *ACM Proc. WOSP*, 1998.

[9] L. Bass, P. Clements, and R. Kazman. Analyzing development qualities at the architectural level. *SEI Series in Software Engineering*, 1998.

[10] D. Compare, P. Inverardi, and A. L. Wolf. Uncovering architectural mismatch in component behavior. *Science of Computer Programming*, 33(2):101–131, 1999.

[11] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, October 1999.

[12] P. Inverardi, L. Marcucci, and H. Muccini. Formal and semi-formal descriptions of a cross-connector software architecture. *Technical Report Universita' dell'Aquila, April 2000.*

[13] ITU-TS. *ITU-TS recommendation Z.120. Message Sequence Charts*. ITU-TS, Geneva, 1994.

[14] I. Jacobson, G. Booch, and Rambaugh. *The Unified Software Development Process*. Addison Wesley publisher.

[15] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, 1994.

[16] J. Magee and J. Kramer. *Concurrency: State Models and Java programs*. Wiley Publisher, 1999.

[17] T. Mannisto, T.Systa, and J.Tuomi. Sced report and user manual. *Report A-1994-5, Department of Computer Science, University of Tampere*, February 1994.

[18] H. G. Perros. *Queueing Network with Blocking: Exact and Approximate Solutions*. Oxford University Press, 1994.

APPENDIX

In the following we introduce the algorithm to generate QNM. We assume that for each component in the MSCs set a server exists.

$CP = \emptyset$
CONC $= \emptyset$
for all Interaction set I_i **do**
  if $|I_i| = 1$ **then**
    "sia $I_i = \{(p_1, p_2)^c\}$"
    if $p_1 \in EX$ **then**
      "build $CdS_c(p_2)$"
      "link ENV a p_2"
    else if $p_2 \in EX$ **then**
      "link p_1 a ENV "
    else if $c = s$ **then**
      "build **SC**(p_1, p_2)"
      $CP = CP \cup \{(p_1, p_2)\}$
    else
      "build $CdS_{inf}(p_2)$"
      $CP = CP \cup \{(p_1, p_2)\}$
  else
  "sia $I_i = \{(p_1, p_2)^{c_1}; (p_3, p_4)^{c_2}\}$"
  if $(\forall_{i,j} p_i \neq p_j, i = 1,2 \wedge j = 3,4) \wedge (c_1 = s) \wedge (c_2 = s)$ **then**
    if $(p_2 \in IN)$ **then** "build $CdS_1(p_2)$"
    if $(p_4 \in IN)$ **then** "build $CdS_1(p_4)$"
    CONC $=$ **CONC** $\cup \{\{p_i, p_j\} | i = 1,2 \wedge j = 3,4$
        $\wedge p_i, p_j \notin Ex\}$
  else if $(\forall_{i,j} p_i \neq p_j, i \neq j) \wedge (c_1 = a) \wedge (c_2 = s)$ **then**
    if $(p_4 \in IN)$ **then** "build $CdS_1(p_4)$"
    CONC $=$ **CONC** $\cup \{\{p_i, p_j\} | i = 1,2 \wedge j = 3,4$
        $\wedge p_i, p_j \notin Ex\}$
  else if $(\forall_{i,j} p_i \neq p_j, i \neq j) \wedge (c_1 = a) \wedge (c_2 = a)$ **then**
    CONC $=$ **CONC** $\cup \{\{p_i, p_j\} | i = 1,2 \wedge j = 3,4$
        $\wedge p_i, p_j \notin Ex\}$
  else if $(p_1 = p_3) \wedge (p_1 \in IN) \wedge (p_2 \neq p_4) \wedge$
        $(c_1 = s) \wedge (c_2 = s) \wedge (I_i \in ND)$ **then**
    if $(p_2 \in IN) \wedge (p_4 \in IN)$ **then** "build **SM**$[(p_1, p_2); (p_1, p_4)]$"
    else if $(\exists i \in \{2,4\}\ t/c\ p_i \in IN)$ **then**
      "build **SC**(p_1, p_i)"
      $CP = CP \cup \{(p_1, p_i)\}$
  else if $(p_1 = p_3) \wedge (p_2 \neq p_4) \wedge (c_1 = s) \wedge (c_2 = s)$ **then**
    if $(p_1 \in IN)$ **then** "build $CdS_1^o(p_1)$"
    if $(p_2 \in IN)$ **then** "build $CdS_1^o(p_2)$"
    if $(p_4 \in IN)$ **then** "build $CdS_1^o(p_4)$"
    CONC $=$ **CONC** $\cup \{\{p_2, p_4\} | p_2, p_4 \notin Ex\}$
  else if $(p_1 = p_3) \wedge (p_2 \neq p_4) \wedge (c_1 = s) \wedge (c_2 = a)$ **then**
    "build $CdS_1^o(p_1)$"
    if $(p_2 \in IN)$ **then** "build $CdS_1^o(p_2)$"
    CONC $=$ **CONC** $\cup \{\{p_2, p_4\} | p_2, p_4 \notin Ex\}$
  else if $(p_1 = p_3) \wedge (p_2 \neq p_4) \wedge (c_1 = a) \wedge (c_2 = a)$ **then**
    "build $CdS_1^o(p_1)$"
    CONC $=$ **CONC** $\cup \{\{p_2, p_4\} | p_2, p_4 \notin Ex\}$
  else if $(p_1 \neq p_3) \wedge (p_2 = p_4) \wedge (c_1 = s) \wedge (c_2 = s)$ **then**
    "build $CdS_1^o(p_2)$"
    CONC $=$ **CONC** $\cup \{\{p_1, p_3\} | p_1, p_3 \notin Ex\}$
  else if $(p_1 \neq p_3) \wedge (p_2 = p_4) \wedge (c_1 = a) \wedge (c_2 = s)$ **then**
    "build $CdS_1^o(p_2)$ *e* $CdS_{inf}(p_1 C p_2)$"
    "link p_1 a $p_1 C p_2$ c $p_1 C p_2$ a p_2"
    CONC $=$ **CONC** $\cup \{\{p_1, p_3\} | p_1, p_3 \notin Ex\}$
  else if $(p_1 \neq p_3) \wedge (p_2 = p_4) \wedge (c_1 = a) \wedge (c_2 = a)$ **then**
    CONC $=$ **CONC** $\cup \{\{p_1, p_3\} | p_2, p_3 \notin Ex\}$
  else if $(p_1 = p_3) \wedge (p_1 \in IN) \wedge (p_2 \neq p_4) \wedge$

```
            (c_1 = a) ∧ (c_2 = a) ∧ (I_i ∈ ND) then
        CONC = CONC ∪ {{p_2, p_4} | p_2, p_3 ∉ Ex}
for all CdS_1(p) do
in_use= FALSE
    for all SM[(p_1, p_2); (p_1, p_3)] do
      if (p = p_2) then
        if "CdS_1(p) non compulsory" then
          if (in_use = FALSE) then
            if ({p_1, p_2} ∈ CONC}) then
              "link p_1 a p_2 "
              "build SC(p_1, p_3)"
              CP = CP ∪ {(p_1, p_2)}
              "delete SM[(p_1, p_2); (p_1, p_3)]"
          else "CdS in uso"
            "link p_1 a p_2 "
            "build SC(p_1, p_3)"
            "delete SM[(p_1, p_2); (p_1, p_3)]"
        else "CdS compulsory"
          "link p_1 a p_2 "
          "build SC(p_1, p_3)"
          "delete SM[(p_1, p_2); (p_1, p_3)]"
      if (p = p_3) then
        if "CdS_1(p) non compulsory" then
          if (in_use = FALSE) then
            if ({p_1, p_3} ∈ CONC}) then
              "link p_1 a p_3 "
              "build SC(p_1, p_2)"
              CP = CP ∪ {(p_1, p_3)}
              "delete SM[(p_1, p_2); (p_1, p_3)]"
          else "CdS in uso"
            "link p_1 a p_3 "
            "build SC(p_1, p_2)"
            "delete SM[(p_1, p_2); (p_1, p_3)]"
        else "CdS compulsory"
          "link p_1 a p_3 "
          "build SC(p_1, p_2)"
          "delete SM[(p_1, p_2); (p_1, p_3)]"
    for all SC((p_1, p_2)) do
      if (p = p_2) then
        if "CdS_1(p) non compulsory" then
          if (in_use = FALSE) then
            if ({p_1, p_2} ∈ CONC}) then
              "link p_1 a p_2 "
              "delete SC(p_1, p_2)"
              in_use=TRUE
              CP = CP ∪ {(p_1, p_2)}
          else "CdS in uso"
            "link p_1 a p_2 "
            "delete SC(p_1, p_2)"
        else "CdS compulsory"
          "link p_1 a p_2 "
          "delete SC(p_1, p_2)"
    if (in_use = FALSE) ∧ (CdS(p) non compulsory) then
      "delete CdS_1(p_1)
    for all CdS_1(p_1) do
      for all CdS_inf(p_2) do
        if (p_1 = p_2) then
          if (in_use = TRUE) then
            "build CdS_inf(Cp_2) c CdS_1(p_2)"
            "delete CdS_inf(p_2)"
"Merge and link all nodes"
```