MEDOC: A methodology for designing and evaluating large-scale real-time systems

by ERIC LE MER
ECA Automation
Saint-Cloud, France

ABSTRACT

We introduce a global design methodology for large-scale real-time systems; it is based on such concepts as data-flow analysis, sequential processes, communication links, abstract architecture. These concepts give us a guide for designing real-time systems; two tools are also introduced, having a specific action in the design process: OGIVE, a Petri net analyzer, makes the verification and the validation of the abstract structure; OSCAR, analytical queuing network, oriented, evaluates quantitatively some implementation choices of the real system. These tools are integrated very early in the design process to make us sure we shall not have dramatic regressions to do for, eventually, redesigning the system.
INTRODUCTION

The main problem in developing large-scale real-time systems is (1) to be sure the system will do what it has to do; (2) to be sure it will do it in a proper way; (3) to be sure the system is a real-time one. This is why we need some techniques and tools defining a methodological environment, as an aid for the designer, which is as automated as possible. Such an environment is introduced in this paper; the focus will be on validation and evaluation aspects. The paper attempts to show how the continuity of the concepts is preserved during the design process according to a top-down methodology by stepwise refinements.

We first introduce two tools: (1) OSCAR (Outil de Simulation pour la Conception d'une Architecture Repartie) and (2) OGIVE (Outil Graphique Interactif de VErification)

OSCAR

OSCAR is an analytical tool based on queuing theory concepts. We can use this tool in the following diagram:

1. Build the model
2. Create a new library
3. Run the model

Build the model

In this respect we have constructed a modelization language. The main features are (1) nets and (2) queues and chains. That means that in the nets we describe the global structure of the queuing network; we may also note the hierarchical ability of the designer to describe the internal structure of the network in more and more detail. For instance, in the first phase, it is possible to model something as a macro-queue—and in a second step to refine this queue into a net description. We shall see further how to handle these subnets, which can be considered macro-queues.

In the queues we describe the main parameters of the network, which are as follows:

1. Local classes, by which we establish a relation between the server and the job. (Note: The job signifies the entity circulating through the network. It can be a batch job, a transaction, a DBMS request, a message, or a similar item. This means the job is characterized according to its specific behavior with the server it requires. One can see the local class as the semantics of the job.)
2. Service discipline, i.e., FCFS, LCFS, PS, IS.
3. Service rate, which can be independent of or dependent on the length of the queue of the server.
4. Type of server, active or passive. In fact, because of analytical restrictions, we only model the memory; and we suppose that, for instance, each job needs the same number of partitions.
5. Service time according to the class of job the server is proceeding with.

The chains are topological descriptions of the job circulation; therefore we can say that the chain is the syntax of the job. OSCAR can deal with open or closed chains.

The global model becomes a tree whose sub-root nodes are always nets and whose leaf nodes are queues and chains. For execution, we consider only the leaves.

Create a new library

At each step of the modelization, the designer may build nets or refine some queues issued from the previous step. These nets are put in libraries, which constitute the global model.

By the command language of OSCAR, one can:

1. Select a library
2. Merge several libraries
3. Overlap several libraries

in order to build more or less complex models by means of elementary descriptions. After each command the user is given a description of the net that has been built and validates it by creating a new library.

Run the model

Once the designer has built a satisfactory model, he can run this same basic model after initialization and eventual loop declarations on data. We have two analytical algorithms: NCA multichain and MVA multichain (NCA stands for normalizing convolution algorithm and MVA for mean-value analysis). The appendix introduces our version of the NCA multichain, which is nearly a straightforward extension of the NCA monochain. As far as I know, I have never seen the convolution algorithm with multichain developed in the literature. (Note: In the NCA the normalizing constant is not computed.)

According to the structure of the network and to the desired results, OSCAR chooses the best algorithm with some internal criteria. However, as networks get very large, it is no
longer possible to get exact results, and we need some approximation techniques. We can share these techniques in two sets:

1. Algorithm approximations that allow us to reduce significantly (1) the run time and (2) the memory requirements.
2. Structure approximations that allow us to reduce the complexity of a model.

The basic concept is the aggregation technique. According to the nature of the complexity, we can aggregate (1) jobs, (2) chains, or (3) queues. (Note: It is under this technique [aggregation of chains] that we handle open and mixed networks.) The selection of any kind of approximation can be manual (the will of the user) or automatic (entry point of OSCAR). One can find the outputs for a 3-chain academic example in the appendix, in French.

OGIVE is a graphic and interactive tool dealing with Petri nets. The main features of OGIVE are the same as those of OSCAR; however, instead of modeling in terms of queues, we model by means of simple Petri nets. The general structure of OGIVE is the following one (Figure 1): The user is guided by a menu, as shown on Figure 1, at each step of the modelization. There are two means to draw a Petri net:

1. A graphic method, menu-directed: places, transitions, and weights on the graph are drawn by showing the word on the menu and by pointing the pen on the screen. One can move, suppress, or mark on the drawing any kind of elementary entities.
2. An interactive method, menu-directed, by which we declare the nodes of the net, the next of each node, the weights and so forth.

Once the net is drawn, it is possible to keep it in a library; the state of the library can be obtained at the beginning of the session by BEGIN.

By the module MERGE one is able to merge several nets by two means—merging transitions and merging places—as soon as the nets are homogenous. The MERGE operation is manual; that is, the user must declare what places/ transitions he/she wants to merge. Then it is possible to proceed to the analysis of the Petri net. The usual structural properties of Petri nets can be obtained by the ANALYSIS module (safety, liveness, boundedness, marking graph, and similar properties); if the net is too large, it is possible to reduce the Petri net in order to eliminate some places or transitions. Of course, this operation must preserve the properties of the original net. This process must be interpreted as an abstraction process, because we reveal the skeleton of the model and thus abstract the implementation aspects.

Another way to analyze Petri nets is the use of the INVARIANT module. By this means we can get the place/transition invariants; elementary, minimal, or total invariants; or other invariants.

METHODOLOGICAL PROCESS

This section shows how these tools are integrated into the entire design phase and what they are supposed to do or to prove. Usually the design phase is shared in some two or three steps with, eventually, the possibility of regressions. MEDOC suggests two steps: in the first one, called functional design, we define and specify what the system is going to do; in the second one, called organic design, we describe how the functions will work.

Functional design

At that level we isolate the main functions of the system—for instance, initialization, display, and computation—and we specify the interfaces of these with the environment, especially the flow of the data. This is why we have a description that is data-flow-oriented, with some basic entities, which are (1) processes, (2) dynamic data, (3) static data, and (4) sources and sinks. We establish the links between processes, sources, and sinks through dynamic data. The representation is graphic, with circles, squares, queues, slashes, and arrows. It is possible to translate this specification into a high-level specification language, by which we control the internal completeness of the description. Throughout the process we must respect two essential rules:

1. The refinement of a process \( P \) from level \( i \) to level \( i + 1 \) must be done on one sheet of paper.
2. From level \( i \) to level \( i + 1 \) all the interfaces must be preserved.
For each level we define (1) the (new) static and dynamic data to be described and (2) the processes to be described and refined. A process must be refined if its specification is not detailed enough to be described or if it still contains some external entities belonging to the environment. In this case we must discard this specification element from the strict specification of the process: for instance, in the design of a display function we usually find at the top levels the display screen and the operator, which do not belong to the implemented display function.

At the end of this phase, when all the processes are to be described, we can start the organic phase. We have hidden an important aspect of this design phase: at each level, in parallel with the functional specification, we build a queuing model that we may call an abstract model, since we do not actually take care of the implementation on a computer and in a data processing environment. We are more interested in the evaluation of a process, or a set of processes, if we do not have any hardware constraints.

Since the specification of the whole system is a tree, with OSCAR and the hierarchical models and merge operations, we are able to evaluate the performance of each process, or specific part of a process, or a set of processes, by grouping and merging the corresponding models. If we have some hardware constraints, we can obtain very early an evaluation and a prediction of the performance of the system and a first idea of the best implementation of the software. If we do not have such constraints, we can evaluate at least some configurations, centralized or distributed. For a study case we can optimize the process communications, the storage of the data, or similar items. In such a case we estimate our own constraints for the future hardware configurations in terms of CPU speed, mean access time for a disk, maximum number of terminals, speed of the communication lines, and so forth. In doing so, we express only a tendency of the future behavior of the system; we do not attempt to reveal the exact truth.

Here we have a dummy architecture in which each process is run on a specific processor and the communications between processors and processes are implemented separately. Even if this architecture is not realistic today, the actual architecture is nothing other than a simulation of the internal behavior of the designed system. We now have to choose between the minimum and the maximum one.

---

ONE processor for ALL processes
ONE processor for EACH process
REAL SYSTEM

We have to consider two questions before making the choice:
(1) Will the designed system work properly? (2) How will it work regarding the real-time constraints?

Before detailing the process, we show it globally in Figure 2, keeping in mind that the structure of the system is a connected graph, regardless of the implementation details (see Figure 3). One can see that the process described in Figure 4 is a down-top process: i.e., we start from the basic descriptions and models of sequential algorithms (SA) and communications links (CL) and then by successive merge get the complete model. This is a direct consequence of the top-down design process.

The first question is answered with the aid of OGIVE. By this means we can validate the qualitative structure of the communication links, especially parallelism, synchronization, deadlocks, starvation, and so forth. Once this work is done, we input time in the model(s) when necessary. To do so, we...
have some macro-libraries at our disposal to describe some specific mechanisms, as, for instance, mutual exclusion, fork-join, send-receive, and remote call. These indications give guidelines for the coming organic phase and its future implementation in an operational environment. We can summarize the functional phase as shown in Figure 5.

![Figure 5—Validation and evaluation process](image_url)

**Organic phase**

At the end of the functional phase the leaves of the functional tree have to be described. The description includes two separate aspects: (1) sequential algorithms (SA) and (2) communication links (CL). At this level we must explain how the system works. The first step is to transform our functional description into an implementation description by means of the following entities: (1) processes (sequential), (2) channels, (3) clocks, and (4) nets.

Each process communicates with the other processes by exchanging messages via channels. One process cannot know which other process is going to consume the message it is sending. We suppose, first, that each process is implemented on a specific immaterial processor (we know, approximately, the required performances of this processor, issued from the functional model of the process it is supposed to run). So we get a theoretical distributed architecture in which we specify the communications, CL, by means of a meta-language PAMELA; from the global structure of the architecture, the skeleton, in terms of SA and CL. The dynamic data are transformed into messages going through the CL, and the static ones are distributed within the processes to which they are related. The internal structure of the data is also a tree; each tree is a type, which can be seen, more or less, as an OSCAR local class.

Then the sequential processes (SA) are described by means of PAMELA. Mainly, we specify the global structure of the process: i.e., the control structures (IF... THEN... ELSEIF; LOOP; EXIT;...), the procedure/action calls, and the declaration of static data. These parameterized macros are connected automatically by OGIVE. The designer specifies time on each transition to get a timed Petri net model; there are two ways for including time. One can see that the second way of firing a transition complicates very much the study of the net. This work is going on in collaboration with LAAS/CNRS at Toulouse, France.

The goal is the following: once we have obtained a timed Petri net model of a local mechanism, we run it under an operational workload; we compute some "cycle time" and some information queuing model oriented: service discipline, service time, service rate. Then we introduce the Petri net model in a global OSCAR model (hierarchical net) and transform it in a local queue with the parameters computed before it is integrated into the global queuing network. We only model some specific complex mechanisms that we want to validate qualitatively and quantitatively.

This process allows us to choose the best implementation of the entire system. It should be noted that often this approach is oriented by the fact that, at the beginning of the project, the hardware architecture and basic software are imposed by the results of the first investigations. In such cases the main work consists of evaluating the performances of these elements, especially those of the basic software, e.g., path length, interactions with the application, and exceptions.

**CONCLUSION**

We have introduced a global design methodology based on concepts such as data flow analysis, sequential processes, communication links, and abstract architecture. These concepts give us a guide for designing real-time systems. Two tools have been also introduced that perform a specific action in the design process: OGLIE, a Petri net analyzer, performs the verification and validation of the abstract structure; and OSCAR, which is analytical-queuing-network-oriented, evaluates quantitatively some implementation choices of the real system. These tools are integrated very early in the design process to insure that we will not have dramatic regressions to make in redesigning the system.

**REFERENCES**

APPENDIX—NCA multichain and an academic OSCAR example

We first introduce the following notations:

\( K = (K_1, \ldots, K_s) \) if \( s \) is the number of chains; state vector of \( Q(K) \) network of queues

\[ Q(k) = \text{Network issuing of } Q \text{ without queue } i \]

\( g(k) \) Normalizing constant

\( P_i(j,k) \) Steady-state probability of the event: \( j \) customers at queue \( i \) when there are \( k \) customers in the network

\( \lambda_i(k) \) Visit rate at queue \( i \) for class \( r \)

\( \tau_i(k) \) Mean-waiting time at queue \( i \)

\( S_i(k) \) Service time of queue \( i \) for class \( r \)

\( n_i(k) \) Queue length at queue \( i \) for a \( k \) population in the network

\( \Lambda_i(k) \) Throughput of queue \( i \) for class \( r \)

\( \mu_{i0} \) Service rate of queue \( i \) when \( j \) jobs are waiting in the queue.

As an extension of NCA monochain we compute

\[ p_{\infty}(j,k) = \Pi_n(j) g^{(n)}(k) / g(k) \] (1)

where \( g^{(n)}(k) \) is the normalizing constant for network \( Q^{[n]} \) and

\[ \Pi_n(j) = \binom{N(1)}{j_1} \times \cdots \times \binom{N(n)}{j_n} / j_1! j_2! \cdots j_n! \]

with

\[ j_1 + \cdots + j_n = j \]

and

\[ g(k) = \sum_{\ell \in \mathbb{N}} \Pi_n(k) \]

with

\[ F(k) = \{ (k_1, \ldots, k_n) / \sum_{i=1}^n k_i = k \} \]

then, from equation (1) we get

\[ p_n(\vec{0}, k) = g^{[n]}(k) / g(k) \]

by induction we get the following recurrent equation:

\[ p_n(j,k) = \sum_{s=1}^n p_n(j-1,k-p_{\infty}(s-k)) W_{N,k} \Lambda_i(k) / \mu_{i0}(j) \]

\[ \text{if } j \geq 1 \]

\[ \text{if in direction } \vec{e}_s \text{ the coordinate of } k \text{ is null} \]

\[ \text{then } p_n(j-1,k-p_{\infty}(s-k)) = 0 \]

using the relation on the throughput

\[ \Lambda_i(k) = g(k-p_{\infty}(s-k)) / g(k) \]

(3)

We get

\[ p_{\infty}(j,k) = \sum_{s=1}^n p_n(j-1,k-p_{\infty}(s-k)) g(k-p_{\infty}(s-k)) W_{N,k} / g(k) \mu_{i0}(j) \]

If we set

\[ p_{\infty}(j,k) = p_n(j,k) g(k) \]

we have

\[ p_{\infty}(j,k) = \sum_{i=0}^k p_n(j-1,k-p_{\infty}(s-k)) g(k-p_{\infty}(s-k)) W_{N,k} / \mu_{i0}(j) \]

(4)

So we have the algorithm NCA multichain:

For a queue \( N, \) knowing \( g^{[n]}(s) ; p(0,\vec{1}) \ldots p(n,\vec{1}) ; g(1) \ldots g(1+k) \) the same as \( g^{[n+1]}(s) \) and by the same way it is possible to compute \( p', g, \) \( p \) for queue \( N+1. \)

We solve the network by iterating on the queues after the initialization as follows:

\[ g^{[2]}(t) = \sum_{i=1}^n \Pi_i(t) \]

\[ = \binom{N(1)}{t_1} \cdots \binom{N(1)}{t_n} / t_1! \cdots t_n! \]

The example is the following (See Figure 7):

There are three chains containing respectively 2, 4, 1 jobs.
The description could be as follows (the underlined words are keywords):

Reseau cpuio (0, 3, 3)
File disk 1, fifo, active
  nbc = 2
  temps serv = t1, 1 ; t2, 2
  Taux serv = 1
File disk 2
  copy disk 1
File cpu, ps, active
( ... )
Chaine Ch1/ fist chain /
  type = fermée
  charge = 2
  cpu, disk 1, 1; disk 1, cpu, 1 ;
Chaine Ch2
( ... )
Fin reseau

The outputs from MVA analysis are shown in Figure 8.

Specification of bounded channel in L. R. with test on full channel and its translation in Petri net

L. R. is the high-level kernel of PAMELA for specifying the real-time concepts, especially the communication links and fairness; consider the net in Figure 9, which expresses communication by means of a bounded channel with test on full channel (no determinism).

A producer of X sends a message ≠ 1 when he wants to put an information in X. If it is allowed to (≠ ≠) it sends the message on Xx.

When the producer wants to know the state of X it sends a message ≠ ≠ 2 on Xx and waits for the answer on REP.

The consumer sends message ≠ ≠ 3 on DEM when it wants an information and waits for the information on Xs.

The specification of this net Rx and of the corresponding actions is the following:

DEF(&Xs,&rep PORTE mes)rx(&xe, 1dem PORTE mes)
RESEAU
  STRUC
  (g&) merge (&xe,&dem)
  (&xs,&rep) g (merge)
FSTRUC
DEF (&intm PORTE mes) merge (&ei,&e2 PORTE mes)
Canal taille non bornee
FDEF
DEF (&s1,&s2 PORTE mes) g (&intg) PORTE mes
PROCESS
DEF f VAR TYPE file
  depos,prise CAR BOOLEEN
  z VAR TYPE mes
FDEF
DEF requete ACTION
  SI p(f) ALORS depos = VRAI SINON
  METTRE(&s2, ≠ ≠ ok) FSI
  test ACTION
  SI p(f) ALORS METTRE(&s2,VRAI) SINON
  METTRE(&s2,FAUX) FSI
  prise ACTION
  SI v(f) ALORS prise = VRAI SINON (z) get (f);
  METTRE(&s1,z) ;
  SI depos ALORS METTRE
  (&s2, ≠ ≠ ok);
  depos = FAUX
  PSI
FSI
depos ACTION
SI prise ALORS METTRE(&s2,z); prise = = FAUX
SIMON (f) put (z)
FSI
FDEF
EXEC
prise = = FAUX ; depos = = FAUX ;
BOUCLE
PRENDRE(&intg,z);
CAS z VAUT ≠ ≠ 1 FAIRE requete
VAUT ≠ ≠ 2 FAIRE test
VAUT ≠ ≠ 3 FAIRE prise
AUTREMENT depos
FCAS
FBBOUCLE
FEXEC
FDEF% g%g%
FDEF% rx%

Then we translate it in a Petri net (see Figure 10) to evaluate the length of the channel "rep" (communication between g and the producer), the channel "XS" (communication between g and the consumer), and the channel "int" (between merge and g).

One can see that this net, modeling Rx, works well (no deadlocks, no starvation) if

length ("rep") = 1
length ("XS") = 1
length ("int") = 3

These results are obtained by OGIVE, using, for instance, the INVARIANT module.
3 STATIONS 3 CHAINES TAUX CONSTANTS

NOMBRE DE FILES 3

NOMBRE DE CHAINES ET REPARTITION DES JOBS
3 2 4 1

DEMANDES DE SERVICE

FILE 1
CHAINE 1 .1000E - 01
CHAINE 2 .1500E - 01
CHAINE 3 .2000E - 01

FILE 2
CHAINE 1 .3000E - 01
CHAINE 2 .0000E + 00
CHAINE 3 .3000E - 01

FILE 3
CHAINE 1 .0000E + 00
CHAINE 2 .3500E - 01
CHAINE 3 .3500E - 01

MVA MULTICHAINES

<table>
<thead>
<tr>
<th>NO RESSOURCE</th>
<th>TAUX UTILISATION</th>
<th>LONGUEUR FILE</th>
<th>TEMPS DE REPONSE</th>
<th>DEBIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESSOURCE 1</td>
<td>.7389E + 00</td>
<td>.2018E + 01</td>
<td>.3638E - 01</td>
<td>.5548E + 02</td>
</tr>
<tr>
<td>(CHAINE 1)</td>
<td>.2499E + 00</td>
<td>.6468E + 00</td>
<td>.2588E - 01</td>
<td>.2499E + 02</td>
</tr>
<tr>
<td>(CHAINE 2)</td>
<td>.3625E + 00</td>
<td>.1039E + 01</td>
<td>.4297E - 01</td>
<td>.2417E + 02</td>
</tr>
<tr>
<td>(CHAINE 3)</td>
<td>.1265E + 00</td>
<td>.3330E + 00</td>
<td>.5266E - 01</td>
<td>.2499E + 02</td>
</tr>
</tbody>
</table>

| RESSOURCE 2  | .8445E + 00      | .1579E + 01   | .5607E - 01      | .2815E + 02 |
| (CHAINE 1)   | .7497E + 00      | .1353E + 01   | .5415E - 01      | .2499E + 02 |
| (CHAINE 2)   | .0000E + 00      | .0000E + 00   | .0000E + 00      | .0000E + 00 |
| (CHAINE 3)   | .9485E - 01      | .2253E + 00   | .7126E - 01      | .3162E + 01 |

| RESSOURCE 3  | .9566E + 00      | .3403E + 01   | .1245E + 00      | .2733E + 02 |
| (CHAINE 1)   | .0000E + 00      | .0000E + 00   | .0000E + 00      | .0000E + 00 |
| (CHAINE 2)   | .8459E + 00      | .2961E + 01   | .1225E + 00      | .2417E + 02 |
| (CHAINE 3)   | .1107E + 00      | .4417E + 00   | .1397E + 00      | .3162E + 01 |

Figure 8—Outputs from MVA analysis

Figure 9—Bounded channel

Figure 10—Bounded channel model

The process merge is fair, i.e., it selects each of its entries such that it gets an infinite number of times $X_e$ and DEM.