Client - server programs analysis in the EPOCA environment

To cite this article: Susanna Donatelli et al 1996 Distrib. Syst. Engng. 3 149

View the article online for updates and enhancements.

Related content
- A software architectural design method for large-scale distributed information systems
  Hassan Gomaa, Daniel Menascé and Larry Kerschberg
- Scalability and performance experiments using synthetic distributed server systems
  C M Woodside and C Schramm
- Context management and its applications to distributed transactions
  Ajay D Kshemkalyani, George Samaras and Andrew Citron
Client–server programs analysis in the EPOCA environment*

Susanna Donatelli†§, Nicola Mazzocca‡∥ and Stefano Russo‡¶

† Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy
‡ Dipartimento di Informatica e Sistemistica, Università di Napoli ‘Federico II’, Via Claudio 21, 80125 Napoli, Italy

Received 22 May 1996

Abstract. Client–server processing is a popular paradigm for distributed computing. In the development of client–server programs, the designer has first to ensure that the implementation behaves correctly, in particular that it is deadlock free. Second, he has to guarantee that the program meets predefined performance requirements. This paper addresses the issues in the analysis of client–server programs in EPOCA. EPOCA is a computer-aided software engineering (CASE) support system that allows the automated construction and analysis of generalized stochastic Petri net (GSPN) models of concurrent applications. The paper describes, on the basis of a realistic case study, how client–server systems are modelled in EPOCA, and the kind of qualitative and quantitative analysis supported by its tools.

1. Introduction

Client–server processing is a popular paradigm for distributed computing, that allows computation to be spread among several cooperating nodes, and processing to reside close to the source of data being processed [2]. In this model, the processes in a distributed application are distinguished according to their active or passive role. Servers encapsulate resources: they are normally passive, and become active when they receive a request from a client. Clients are active processes: they need some service during their execution, and for this purpose they issue a request to the appropriate server.

In the development of client–server programs, several complex problems have to be addressed. First of all, developers must be aware of the presence of potential deadlock situations in the program execution. Even for small systems with few clients and servers, especially in synchronous message-passing environments, the potential for deadlock is very high. Such situations could hardly be revealed and discovered by traditional testing and debugging techniques. The employment of proper formal methods is thus needed to guarantee that the program is deadlock free. Second, performance issues have to be addressed. In particular, it is important to be able to predict the program performance, in order to support design and tuning decisions aimed at guaranteeing good response times at an acceptable level of utilization of resources (services, processing nodes, inter-node communication bandwidth, etc). Performance metrics of particular interest for client–server systems include: expected service time, server’s reactivity, resource utilization, CPU idle time, node’s communication/processing ratio, average degree of parallelism.

These two aspects—the program qualitative analysis and the performance prediction—can be coped with using an approach based on the construction of a formal model of the distributed program. Petri nets are known to be a suitable formalism for this purpose, and stochastic classes of Petri nets make possible the computation of the above mentioned performance metrics for client–server programs. However, Petri net based approaches need to be supported by appropriate computer-aided software engineering (CASE) tools to be practically usable. In fact, the proof of correctness and the performance prediction conducted on the program net model require that many technical problems be solved, ranging from the definition of the model abstraction level, to the model construction and analysis, and to the translation of the selected performance metrics in terms of the stochastic performance model.

The goal of this paper is to describe how such problems are addressed in the EPOCA system. EPOCA [3–5] is a CASE system that allows Petri net based analysis of concurrent programs: results and properties of a program are derived from results and properties of a generalized stochastic Petri net (GSPN) model of it.

EPOCA is the integration of two environments, namely the DISC programming system and the GreatSPN tool. The
DISC (distributed C) system [8] consists of a language, which is an extension of C for programming distributed systems, and of a development environment that provides compilation, monitoring and profiling facilities. The extensions to C include concurrent constructs based on the CSP (communicating sequential processes) model, which is particularly suited to programming loosely coupled distributed architectures. GreatSPN [9] is a tool for the graphical editing and for the analysis of GSPNs. It runs under Sunview (older versions), Openwin and Motif. GreatSPN supports qualitative analysis of GSPN models (state space based deadlock analysis and invariant properties), as well as quantitative analysis (steady state probabilities of states, mean number of tokens in places, and throughput of transitions).

EPOCA is centred around a development methodology which consists of the following steps:

- design of the software architecture, in terms of processes and their interactions;
- application implementation in a C-based CSP-like language;
- construction of a qualitative GSPN representation of the program;
- program behaviour analysis and validation via analysis and execution of the GSPN model;
- introduction of program quantitative parameters in the qualitative model;
- computation of program performance metrics.

The methodology iterates through these steps, until an implementation is obtained, which is considered correct and meets desired performance requirements.

The performance features considered in this paper are that of the software, independently of any hardware restriction, that is to say with infinite resources: this gives an upper bound on the performance of the application, as well as a performance characterization of the program which is independent of any specific hardware. This characterization can be used, for example, to compute the communication costs for a communication graph, or the computation costs for a task graph model [1]. The software model can be integrated, in a second step, with a hardware model, to compute the performances of an application on a specific machine: this issue is out of the scope of the paper, but the interested reader can find in [6] a proposal for a methodology to integrate hardware and software models.

To support the program validation steps of this methodology, EPOCA provides these main functionalities:

- automatic translation of DISC programs into GSPN models;
- program qualitative analysis, i.e. analysis of time-independent properties;
- program quantitative characterization through the automatic definition and computation of program performance indices from net performance indices.

Of course, the overall EPOCA environment includes also the whole set of functionalities of DISC, for program development (makefile generator, compiler, linker tools, run time support, monitor, post-mortem analyser and window-based interface), and those of GreatSPN for net editing and analysis (graphical interface for the definition of GSPN models, their validation and the computation of the associated performance indices).

The paper describes, on the basis of a meaningful case study, how client–server systems are modelled and analysed in EPOCA. More specifically, we first describe the application of net analysis techniques to detect the presence of deadlocks, and discuss how to include the model of the program variables to enhance the analysis. Then, we show the quantitative measurements that can be obtained by analysing the performance model. The case study considered is an inter-departmental secretary office service.

The remainder of the paper is the following. In section 2 we introduce the client–server case study. Section 3 presents the first tentative implementation. Section 4 describes the iterative process of correcting the program using the qualitative analysis provided by EPOCA. Section 5 discusses the timing aspects of the model and shows a few performance figures of the program. Section 6 concludes the paper with a critical view on what can and cannot be done with EPOCA.

2. Case study

The University of Naples is administratively organized into departments, which are geographically distributed in the town. Each department has a local office, and there is a centralized administration office. An administration procedure may require data that are located on the local site, or in the central office, or both. Besides data, processing is distributed too: some functionalities required by a single procedure reside on the central site. At the central office, however, a procedure is assumed to be completely self-contained, that is to say it does not need to access any other site.

We assume that the system has the following client/server architecture. There are clients both in the departments and at the central office; the latter are meant to produce global summary reports, hence they are supposed to access only data local to the central site. There is a unique central server (server A), plus two local servers at each department. The local servers are of two types (named B and C for simplicity), depending on whether they access services in the central site or not. Server A and servers of type C use only local resources, while servers of type B use local resources and issue a request to the central site (remote site). In summary, there are four types of requests:

- **central**: requests from a central site client; they are served by server A at the central site (central actually means “local to the central site”);
- **local**: requests from a department client, that are satisfied locally; they are served by server C of the site where the request originates;
- **remote**: requests from a department client directly to the central server; they are served by the server A at the central site;
mixed: requests by a department client, requiring both local and remote processing; they are processed by server B of the site where the request originates, which in turn accesses server A at the central site.

This case study is representative of a very common architecture in client–server applications, named the ‘three-tiered model’ in [2]. In this model, a top tier contains a mainframe host that accesses the central corporate database. The second tier is a database server running a DBMS (usually, relational), with (a copy of) a part of the corporate data, and the third tier contains the clients originating the requests (PCs, workstations or terminals).

In this kind of client–server applications, the question of component placement is decided essentially by:

- the quantity of data relevant to any request,
- the number of clients,
- the number of transactions per unit time,
- the technical characteristics of the client and server platforms,
- the technical characteristics of the interconnection network.

All these aspects are taken into account in our analysis through a set of parameters, characterizing: the rates and relative frequencies of the four types of client requests, the data transmission costs (local and remote), and the processing costs.

Starting from these specifications, we describe in the following subsection the design of the software architecture of our client–server system. For this task, which represents the first step in the EPOCA methodology, we use the PARSE notation [10], that provides an intuitive graphical formalism to define processes and their structural interconnections.

2.1. Design with PARSE

In PARSE, the design of a parallel software system is expressed at the highest level by means of the process graph notation [10]. This is a graphical formalism which enables the developer to describe the architecture of the parallel system in a hierarchical way, in terms of the components (called process objects) and of the structural interconnections between them.

Basically, the process graph notation classifies process objects in three general classes: data server, function server and control process. Data servers and function servers are passive objects, which process requests received by control processes, which in turn act as active clients. Differently from function servers, data servers can have a state that persists between two requests. Also control processes have a persistent externally visible state; they are responsible for initiating and coordinating processing in a parallel system. Finally, processes interfacing external equipment or devices are named external interfaces. They perform the interaction (input/output) with the external world, without any processing. It is important to point out that in the process graph notation, a parallel design is expressed in a hierarchical way, since every process object can decompose into a set of lower-level parallel processes. Decomposable objects are called classes, while executable processes are named primitive objects.

The semantics of the communication mechanisms between processes are expressed by communication paths, which are categorized in synchronous, asynchronous, bidirectional (client–server) and broadcast (one-to-many).

In a hierarchical design, a non-primitive object has internal paths between lower-level objects, but of course its external interface must remain unchanged.

Our case study is described at the topmost level by the process graph depicted in figure 1. This provides a CentralSite class interconnected to \( n \) replicas of the department class via synchronous paths.

The CentralSite class is further decomposed as shown in the process graph of figure 2. This is composed by the ServerA process, the local clients, a communication manager (CommMgr) which manages the interactions with the departments, and a Dispatcher process which multiplexes/demultiplexes internal and external requests/replies to the server. Incoming messages are handled non-deterministically by Dispatcher and CommMgr, as described graphically by the box-like port. The messages are passed over synchronous paths, apart from the interaction between the clients and the dispatchers, which are of the request/reply type.

The DepartmentSite class is further decomposed too (figure 3). It consists of the ServerB and ServerC processes, the department clients, and a DeptDispatcher process which multiplexes/demultiplexes internal and external requests/replies to the two local servers and the remote one.
3. Program implementation

From the PARSE design, we have derived the DISC implementation included in appendix A: the DISC constructs for parallelism are basically those of CSP [7], with a synchronous input operator $\text{?}\text{?}$, and a synchronous output operator $!!$. This first version of the application includes: a central site, a department site, and a single client at each site. According to the specification, the client of the central site requires only local services, while the department client may ask for either a completely local, a completely remote or a mixed request. The message type encodes the type of request, according to the following definitions:

0 local request
1 mixed request
2 remote request

Messages generated at central sites are all of type 0.

As a consequence the central site dispatcher is able to distinguish whether a message should be forwarded to the client of the central site or to the communication manager, specifically

\[
\text{msg.type } = 0 \quad \Rightarrow \quad \text{send on rep channel}
\]

\[
\text{msg.type } = 1 \quad \Rightarrow \quad \text{send on fromDis channel}
\]

The execution reveals a deadlock situation (observed through monitoring). Due to the relative speed of the processes the same deadlock is always observed: is there any other deadlock? We can answer this question using the support for the analysis provided by EPOCA.

4. Qualitative analysis: the iterative process

In this section we describe the iterative process that, starting from the program given in appendix A, allows the programmer to develop in successive steps, and with the support for qualitative analysis provided by EPOCA, an implementation that is judged correct (in the sense that it does not deadlock).

4.1. The initial model

Starting from the program given in appendix A, EPOCA builds, by automatic translation, the net shown in figure 4: for greater clarity we have manually modified and moved the tag of places and transitions.

EPOCA supports a static analysis approach, and therefore only activities inherent to the processes' generation and communication are modelled explicitly, as explained in [11]. The GSPN of figure 4 shows the net model of the application in the case of a single remote site, and of a single client at the remote and at the central site. The net depicts, from left to right, the processes Client, Dispatcher, ServerA, CommMgr, DeptDispatcher, ServerB, ServerC, and Dclient. These processes are generated by the PAR command in processes root, CentralSite, and Department, represented in the net by transitions PAR_root, PAR_Ctr, and PAR_Depr; the corresponding places in the GSPN model are tagged with names starting with Cl, Dsp, A, CM, DDsp, B, C, DCl.

To provide an intuition on what is, and what is not, explicitly represented by the GSPN, let us consider in some detail the model of process serverA. Process serverA executes an infinite loop of three actions: reception from channel request, some sequential activity (a 'printf' command), and a transmission on channel reply. Both request and reply are formal parameters that represent a synchronous communication with process Dispatcher on two different actual channels, named toSer and fromSer in the process code. The subnet that represents serverA in the GSPN model of figure 4 is composed by places A1 to A7, with their input and output transitions.

First action. The subnet composed by places A2, A3, A4, A5 and transitions t2a, t2b, T2a, T2b models the 'request ?? msg' line of code: since in the Dispatcher code there are two communications on the corresponding toSer channel, then the subnet that represents the communication has been duplicated: transition t2b models a synchronization with the Dispatcher process while it is executing the first branch of the alternate (ALT) command (subnet t4, Dsp3, T4, Dsp6, Dsp13, t9), and T2b the delay necessary to exchange information on the channel, while transitions t2a and T2a represent a synchronization and a communication with the same Dispatcher process when it is executing the third branch of the alternate (subnet t6, Dsp5, T6, Dsp8, Dsp15, t13). Observe that the msg variable is not modelled in the GSPN, according to the classical principles of static analysis.

Second action. The 'printf' command is modelled as a delay expressed by transition T3.

Third action. The line of code 'reply!! msg' represents a synchronization with the Dispatcher process on the second branch of the alternate. The synchronization and communication between the two processes is represented by transitions t5 and T5 in the Dispatcher subnet, so that the subnet that models the third action is composed by A6, t5, Dsp4, T5, A7. The choice in EPOCA is to draw the common part of the communication subnet graphically as a subnet of the process that executes the input command.

It is important to note that, since the program variables are not explicitly represented, then tests are modelled...
as non-deterministic choices: for example the ‘if’ in the second branch of the alternate of the Dispatcher process is modelled by subnet $D_{sp7}, t_7, t_8$, a free choice conflict out of place $D_{sp7}$.

It is well known that the main problem with static analysis is that of non-faults: the model used for the analysis, in this case a GSPN, can reveal a deadlock that can never happen in a real execution of the program, since only the program control flow, but not data and their modification, are explicitly represented in the model. Conversely, all program deadlocks are discovered by a static analysis.

The analysis of the net indicates the following problems: the initial state is transient, the net is not live and 27 of the 520 states are deadlocks.

The initial state is transient because the par activities in the Root, CentralSite, and Department processes are, obviously, executed only once, to generate all the processes of the application. This is not really a problem: it is GreatSPN itself that, starting from the initial transient state, is able to find an initial state that can be reached infinitely often (if there is one).

The net is obviously not live (indeed none of the transitions can be fired infinitely often starting from any reachable state) since there are deadlocks.

Of the 27 deadlocks, many are non-faults. For example the following deadlock state, taken from the list produced by GreatSPN, is a non-fault.

>From #115 (0 timed trans) [ C14, Dsp2, A2, CM7, Dsp16, B2, C2, DC14 ]

[Client, Dispatcher, ServerA, CommMgr, DeptDispatcher, ServerB, ServerC, Dclient]

State 115 is a deadlock in the model, since 0 transitions can be fired, but this state corresponds to an impossible situation in the program. Interpreting the state of the net as program state, 115 represents a situation in which ServerA has served a request from the local client, but the dispatcher has forwarded the answer from ServerA to the communication manager, instead of directly to the local client. This situation cannot happen in the program, since the Dispatcher process code after receiving a message from ServerA (a message on channel fromSer) checks the type of the message to decide to whom the message should be forwarded: to CommMgr, using channel fromDis, if the type is not zero, msg.type!=0, or to the local client, using channel reply, if the type is zero. The problem here is that the model of a communication does not include the information carried by the message. For reading convenience we replicate the code here:

```plaintext
process Dispatcher(request, reply, toSer, fromSer, toDis, fromDis)
:: { while(TRUE) {
    alt { request ?? msg => {toSer !! msg;}
        /* request from local client */
        fromSer ?? msg => {
            /* reply from server */
            if(msg.type != 0) {
                /* was a remote request */
                fromDis !! msg; } /* send reply to ComMgr */
            else { reply !! msg; } /* reply directly to local client */
            toDis ?? msg => {
                /* remote request */
                toSer !! msg; }
            /* submit to server */
            on fail terminate(fail); }
    } /* endwhile */
} endprocess
```

It is clear that to check whether a deadlock in the model corresponds to a program deadlock is an expensive, brain consuming, and difficult to automate operation. This
example suggests that modelling all variables that can influence the pattern of communications we may indeed be able to considerably reduce, and eventually completely eliminate, non-faults. In this case the control variable is msg.type, the field ‘type’ of variable ‘msg’.

There are two classical ways of translating variables in Petri nets: to add a place for each possible value, or to encode into the marking of a single place the value of the variable. Since a condition for a GSPN to be solvable is to have a finite number of places and a finite number of states, we consider only variables with values that can be mapped into a finite set of values.

The extension of EPOCA to automatically translate model variables is currently under development: only the case of non-negative integer variables is considered, basically to account for variables that are used as counters or flags. In this implementation it will be the responsibility of the user to select the variables that have to be automatically modelled by the tool. Each variable is modelled by a place, the number of tokens in the place being the value of the variable, and subnetrs are added to realize simple assignment statements and simple tests over the variable value.

For the time being we have to model variables in a manual way, modifying through the EPOCA net interface the automatically produced net: an error prone work indeed, so we may try to simplify their model by reasoning on their meaning.

According to the interpretation of the message type field given before, we can observe that it is used as follows

- by the department dispatcher to decide
  - how to treat a request from a client (remote, local, or mixed)
  - how to treat a reply from the central site (to discriminate between answers to all-remote queries, that should go to the client, and mixed queries, that should go to server A)
- by the central site dispatcher to decide
  - to whom to forward a message received from server A (to discriminate whether this is an answer to a remote query or to a local one).

Considering that there are at most three significant values for the ‘msg.type’ variable at the department site, and two at the central site, each variable has been modelled with as many places as there are significant values. We have therefore added three places in the model of the department site, to represent the possible values (remote, local and mixed) of the message tested in DeptDispatcher (modelled by places remote, local, and mixed, and two places (named locA and fD) in the model of the central site, that represents the possible message types (local and non-local) checked by Dispatcher.

Five places have therefore been added to the net shown in figure 4, with all connections to transitions that either modify and/or test the variables’ values. The analysis of the reachability graph of the modified net reveals the following four deadlocks, and all of them correspond to a deadlock in the program.

>From #119 (0 timed trans)
[ remote, Cl4, Dsp8, A6, CM2, DDsp2, B2, C2, DC14, fD]
>From #132 (0 timed trans)
[ remote, Cl4, Dsp6, A6, CM2, DDsp2, B2, C2, DC14, locA]
>From #167 (0 timed trans)
[ mixed, Cl4, Dsp8, A6, CM2, DDsp2, B6, C2, DC14, fD]
>From #170 (0 timed trans)
[ mixed, Cl4, Dsp6, A6, CM2, DDsp2, B6, C2, DC14, locA]

States # 119 and # 132 depict the same situation at the department site (there is an outstanding request caused by a request with message type ‘remote’, corresponding to the value 2 in the DISC implementation), while at the central site we have, for # 119,

- a request of the local Client has been accepted (variable msg.type = locA);
- ServerA has served the request and it is trying to send the reply data back to Dispatcher;
- Dispatcher has accepted a request from the remote site and it is blocked waiting to send it to ServerA.

The deadlock is obviously caused by the fact that Dispatcher can accept a request for ServerA even if ServerA has not finished processing the previous request, while ServerA cannot accept any more requests until it has sent the answer of the previous one back to Dispatcher.

State #132 is in a dual situation: a request from a department has been accepted (msg.type = fD), ServerA has served the request and it is trying to send the reply data back to Dispatcher, but Dispatcher has accepted a request from the local Client and it is blocked waiting to send it to ServerA.

The deadlock does not depend on the fact that the request from the peripheral site has been generated by a message of type 1 or 2, and indeed we have the two additional deadlock states #167 and #170, that differ from the previous ones only for having the place mixed marked, instead of remote.

It is interesting to observe that the four deadlock states correspond to two ‘dual’ situations, and actually only one of the four was detected by monitoring the execution in DISC, so despite the fact that this is a very simple case of deadlock, it already represents a case in which the static analysis conveys more information than monitoring.

4.2. Solving the problem

To solve the deadlock, the programmer has introduced a variable busy to ‘protect’ ServerA: no more requests for ServerA can be accepted by Dispatcher if ServerA is busy.

The net produced by EPOCA on the new program was enhanced with the model of the msg.type and of the busy variable, and the analysis for the case of a single department with a single client revealed no deadlock, therefore we can assume that the program will not deadlock in any possible execution.
This new program actually solves the problem only in the limited case of a single department with just one client. Indeed the presence of a single client, that waits for a request to be satisfied before asking for a new one, ensures that there might not be more outstanding requests for the other resources of the program: \texttt{ServerB, ServerC}, and the communication channel between the central and department sites. There can be more than one request for \texttt{ServerA}, but \texttt{ServerA} is protected by the \texttt{busy} variable.

The program, modified in a straightforward manner to deal with more than one client, resulted in a deadlocked execution. The analysis of the net suggested that all program resources (hardware or software) have to be protected to ensure that no request for resource \texttt{X} can be accepted if \texttt{X} is busy. Therefore the program has been modified introducing a \texttt{busy} variable for each resource (\texttt{ServerB, ServerC} and the communication channel).

For efficiency reasons the programmer has triplicated the \texttt{req} channel between \texttt{DClient} and \texttt{DeptDispatcher}. Indeed with a single \texttt{req} channel, when, for example, \texttt{ServerB} is busy, \texttt{Dispatcher} does not accept any more request for it, but with a single request channel this implies not to accept any request, even for \texttt{ServerC} and \texttt{ServerA}, although they might be idle at that time.

The code has been changed accordingly, and we report in appendix B the modified code of the \texttt{DeptDispatcher} and the \texttt{DClient} processes.

The \texttt{GSPN} of the modified program, automatically produced by EPOCA, has been augmented with the model of the following variables: \texttt{busyA} at the central site, \texttt{busyA, busyB, busyC} and \texttt{busychan} at the department site; in this case we have followed the approach of one place per variable, and a token in a place means that the variable value is 1 (the true value in C-like languages).

The \texttt{msg.type} variable is modelled as before in the central site, while the model at the department site has changed: indeed since requests for different services go on different channels (\texttt{reqA, reqB, reqC}), the only information needed is whether the request is of mixed or completely remote type.

The analysis of the net model obtained shows no deadlock, and the state space is ergodic, therefore the net is stochastically ‘well behaved’ and we can safely compute the performance indices.

4.3. A program without deadlock is enough?

Although the detection of deadlocks is the primary goal of a static analysis tool, other aspects of the program behaviour can be investigated.

For example we have also built the net for the case of more than one client per department, and by the analysis of the net we discovered a programming error that does not lead to deadlock, but certainly produces a non-desired behaviour.

The \texttt{GreatSPN} tool can list all effective conflict sets of a net, that is to say the list of all subsets of transitions that are enabled in conflict\(^\dagger\) in at least one state. The

\^\dagger\ Two transitions are enabled in conflict when the firing of one disables the other.

\texttt{lists} for our net contained also pairs of transitions, one per client, corresponding to reception of a message on the \texttt{reply} channel upon request of different resources: for example the transition modelling the first client waiting for a reply after a \texttt{ServerC} request is in conflict with the transition modelling the second client waiting for a reply after a \texttt{ServerA} request.

This suggests, and it has been verified that it is indeed so, that in the program each client can ‘steal’ from the other a reply from a server, even from a server of a different type than the one the client is waiting for. Obviously this is an erroneous behaviour, and it has been corrected by triplicating also the \texttt{reply} channel.

5. The performance model

After some iterations between improved implementations and analysis, the program is considered correct, in the sense that it has no deadlock. It is now time for studying, with the support of EPOCA, its quantitative behaviour.

The same nets used to study the absence of deadlock in the implementation, a qualitative property, are now used to study the performance of the implementation.

The goal of the quantitative analysis is to study both the hardware and software choices, eventually causing a redesigning of the application to take into consideration performance issues.

In particular for our application we are interested in determining the parameters of an adequate hardware configuration and data allocation. For example: the minimum speed of the communication necessary to guarantee the required performance, the processing speed of the servers (power computation and device speed), or the time to satisfy a remote versus a local request to servers.

Moreover it may be interesting to study the level of parallelism with different clients or different implementations of the interactions among processes.

Starting from the nets automatically produced by EPOCA, the quantitative analysis requires two main steps: definition of the transition rates\(^\dagger\) and definition and computation of the program performance indices.

5.1. Transition rates

The target system we consider is geographically distributed. The quantitative parameters have been determined with reference to a typical configuration, composed of departmental local area networks (LAN), and of remote links over dedicated connections from the departments to the central site. All timing parameters are normalized to the local communication time. These parameters have been defined with the help of the specialists of the administrative centre of the university. Each client eventually represents the request originating from a small local area network of PCs, workstations or terminals in a same room.

The following quantities have been identified as relevant parameters of our system, with \texttt{T_LC}, unit of local communication, being the reference parameter.

\( T^\dagger \) A transition rate is the parameter of the exponential distribution of the delay associated with the transition.
Figure 5. Throughput of processes for the program with one department and two clients.

\[ T_{RC} = K \cdot T_{LC}, \] unit of remote communication; the reference value for \( K \) is 10.

\[ EL_A = K_A \cdot T_{LC}, \] elaboration time for server A; the reference value for \( K_A \) is 30.

\[ EL_B = K_B \cdot T_{LC}, \] elaboration time for server B; the reference value for \( K_B \) is 40.

\[ EL_C = K_C \cdot T_{LC}, \] elaboration time for server C; the reference value for \( K_C \) is 60.

\[ EL_{Cl} = 2 \cdot T_{LC}, \] elaboration of the client at the central site.

\[ EL_{DCl} = 2 \cdot T_{LC}, \] elaboration of the client at a department site.

\[ Th_{Cl} = K_{Cl} \cdot T_{LC}, \] think time of client at central site.

\[ Th_{DCl} = K_{DCl} \cdot T_{LC}, \] think time of client at department site.

The think time parameter models the amount of time necessary for a client to generate the next request to the servers, after having received a reply, while all other statements in the program are considered as execution activities of the appropriate type.

The \( EL_X \) parameter represents the elaboration activity of process \( X \). For simplicity, the elaboration activity of the process has been summarized in the code presented in the appendices as a ‘printf’ or as an ‘elaborate’ command.

Parameters have been assigned to the transitions according to the type of activity modelled by the transition.

5.2. Definition and computation of performance indices

EPOCA supports the automatic definition and computation of performance indices to compute the program phases (subset of processes that can be active together—a useful indication for allocation purposes), the communication costs between processes, and between processes on a given channel, the interference cost (probability of two processes being computing, and therefore using the cpu, at the same time), and the processes’ completion times.

We have selected two program configurations for the analysis: an application with one Dept process and two DClient processes (application 1-2), and one with two Dept processes, with one DClient each (application 2-1). The state space construction algorithm reveals that the two models lead to an ergodic Markov chain, and that the size of the first GSPN model is 161 178 and of the second one is 90 602. The solution (reachability graph construction, Markov chain construction and Markov chain solution) of the models took less than a minute on a Sun10 workstation.

For each configuration we show two performance indices: the throughput (number of completions per unit time) of each process, and the mean completion time of each process. The first measure is a ‘classical’ GreatSPN statistic, while the second one has been computed using the EPOCA performance indices postprocessor [5]. Starting from the steady state solution of the GSPN computed by the GreatSPN solver, which is in terms of places and transitions, the postprocessor of EPOCA computes some relevant program performance indices, such as the one listed above, that are instead in terms of processes and channels.

Figure 5 shows the throughput of the transitions representing the while(true) statement of each process for application 2-1 for various values of the ratio between remote communication and local communication.
Client–server programs analysis in the EPOCA environment

Figure 6. Completion time of processes for the program with one department and two clients.

remote and local communication rate. We have used the acronyms CM, Ds, CI, SerA, serB, serC, and DCl for processes CommMgr, Dispatcher, Client, ServerA, ServerB, ServerC and DClient respectively.

The two DClient have the same throughput, which is quite obvious considering that they have the same behaviour and the same transition rates, and therefore only one of the two has been shown. Due to the peculiar shape of the DClient, that executes an even number of requests to ServerB and ServerC, also the throughput of the two servers is identical, and is shown superposed on the figure.

Figure 6 shows the completion time for the same application, again for varying values of the remote communication rate. This is more immediately interpreted as a program performance index: indeed when increasing the remote communication rate from 0.01 to 10 (which means a time for a single remote communication from 100 times a local communication down to one tenth of a local communication), we observe a decrease in the completion time of the DClient processes from 1800 to slightly less than 1000. The time unit here is the local communication time, considered equal to one.

It is also interesting to note that there is very small variation in the completion time for the range 0.1 to 10 of the rate of communication, which means that, due to the highly synchronized behaviour of the implementation, there is no gain in increasing the speed of the remote communication medium above a certain threshold.

Application 2-1 has instead been studied for varying values of the Client and DClient think times. Considering a situation in which the remote communication medium is 10 times slower than the local communication, we have considered four values of the client think times rates. By increasing the think time rate, we decrease the time necessary for a client to prepare the next request after the previous one has been satisfied. As can be observed from the curves plotted in figures 7 and 8, the observed indices are rather insensitive to the client speed, presumably because the system is already overloaded for a think time value equal to 0.001 (a rate of 1000).

It is obvious that the efficiency of the proposed implementation is very low, but we should not forget that we have chosen a completely synchronous paradigm. Indeed the next step in the implementation will be the insertion of appropriate buffer processes to reduce the level of coupling between processes. The proposed integrated approach to program correctness and performance evaluation can surely be used to evaluate the correctness of the buffer implementation, while the performance evaluation phase can be used to choose an adequate buffer size.

6. Conclusions

In this paper we have presented our experience in using EPOCA for the development, qualitative and quantitative analysis of a client–server application. The goal of this experience was to push EPOCA to its limits by taking a real application of a well known class.

Our experience has shown that the use of EPOCA by users that are not experts with Petri nets is still not feasible, due to the need to manually model variables to avoid non-faults. The experience gained while modelling...
the variables in this application has strongly influenced the implementation that is under development in EPOCA to automatically translate variables.

When the application grows in size, it is also important to be able to automatically associate to each transition its rate parameter: this is a time consuming and error prone task.
work. A set of programs is already available with this aim, but it has not been fully integrated in EPOCA yet.

The positive aspect of this experience is that the automatic translation from the program code to the GSPN model is extremely important to avoid a 'biased' construction of the model, that is to say that the model is constructed based on what the programmer thinks the application does more than on what the application really does. We were also able to discover and correct, through the model analysis, some deadlock situations that have not been detected by program execution and monitoring, and one error in the program that, although it did not lead to deadlock, caused an incorrect behaviour.

Even for a GSPN expert, it may not be easy to correctly define the program performance indices in terms of the GSPN statistics. Another positive aspect of this experience was the use of the postprocessor of EPOCA that does this computation in an automatic manner, and this is of great help for the analysis.

The size of the state space for the GSPN models produced by EPOCA is a problem that has to be faced, since the analytical solution approach may not be feasible. For example we could only solve the case of two departments, while, indeed, the University of Naples has many more. Of course we could compute the performance indices using the GreatSPN simulator, assuming that we are able to build the model, which is not easy if we have to manually modify the model. This approach will undoubtedly become much more attractive when the automatic translation of program variables is available, since there will not even be the need to look at the GSPN model.

This is the first time that we have used EPOCA together with PARSE. Indeed PARSE provides a higher level model of our application, since it is a model of the design and not of the application. We plan in the near future to investigate the advantages and the costs of an integration between PARSE and EPOCA, to allow the possibility of evaluating the performance of an application already at the design level.

Acknowledgments

We would like to thank the anonymous referees for the very careful reading of the original manuscript, and for useful comments.

Appendix A. The code of the first Disc program

typedef struct { int sender;
    int type;
    char body[10]; } Message;

process root()
:: {
    par { local dom[2];
        process CentralSite(dom[0], dom[1]);
        process Department(dom[0], dom[1]); }
} endprocess

process CentralSite(input, output)
owned struct Message input;
used struct Message output;
:: {
    par { local request, reply, toSer, fromSer,
        toDis, fromDis;
        process Client(request, reply);
        process Dispatcher(request, reply, toSer,
            fromSer, toDis, fromDis);
        process ServerA(toSer, fromSer);
        process CommMgr(input, output, toDis,
            fromDis); }
} endprocess

process Department(output, input)
used struct Message output;
owned struct Message input;
:: {
    par { local req, rep, toSerB, toSerC, fromSerB,
            fromSerC, RemReq, RemRep;
        process DeptDispatcher(req, rep, toSerB,
            fromSerB, toSerC, fromSerC,
            RemReq, RemRep, input, output);
        process ServerB(toSerB, RemReq, RemRep,
            fromSerB);
        process ServerC(toSerC, fromSerC);
        process DClient(req, rep); }
} endprocess

process Dispatcher(request, reply, toSer, fromSer,
    toDis, fromDis)
owned struct Message request, reply, fromSer,
    toDis;
used struct Message fromDis, toSer;
:: {
    struct Message msg;
    while(TRUE) {
        alt { request ?? msg => {toSer !! msg;}
            /* request from local client */
            fromSer ?? msg => {
            /* reply from server */
                if(msg.type != 0) {
                /* was a remote request */
                    fromDis !! msg; } /* send reply
                    to ComMgr */
                else { reply !! msg; } }
            /* reply directly to
                local client */
            toDis ?? msg => {
            /* submit to server */
                toSer !! msg; }
            /* submit to server */
            on fail terminate(fail); }
        } /* endwhile */
    } endprocess

process DeptDispatcher(request, reply, toSerB,
    fromSerB, toSerC, fromSerC, RemReq,
    RemRep, input, output)
owned struct Message request, reply, fromSerB,
    fromSerC, input, RemReq, RemRep;
used struct Message output, toSerB, toSerC;
:: {
    struct Message msg;
    while(TRUE) {
        alt { request ?? msg => {
            /* remote request */
            toSerB !! msg;
        } /* submit to server */
        on fail terminate(fail); }
    } /* endwhile */
} endprocess

Client–server programs analysis in the EPOCA environment
/* request from client */
if(msg.type == 0) toSerC !! msg;
/* local request - send it to server C */
else if(msg.type == 1)
    /* mixed request - to server B */
    toSerB !! msg;
else output !! msg;
/* remote request - send to central site */
fromSerB ?? msg => {
    /* got reply from local server B */
    reply !! msg;
} /* send to client */
fromSerC ?? msg => {
    /* got reply from local server C */
    reply !! msg;
} /* send to client */
RemReq ?? msg => {
    /* got remote request from server B */
    output !! msg;
    /* send to central site */
} /* send to client */
/input ?? msg => {
    /* got remote reply from CommMgr */
    if(msg.type == 1) RemRep !! msg;
    /* send to server B */
    else
        /* msg.type == 2 */
        reply !! msg;
} /* send directly to client */
on fail terminate(fail); }
/* endwhile */
} /* endwhile */
process CommMgr(input, output, toDis, fromDis)
owned struct Message input;
used struct Message output;
used struct Message toDis;
owned struct Message fromDis;
::
struct Message msg;
while (TRUE) {
    alt { input ?? msg => { toDis !! msg;}
        fromDis ?? msg => { output !! msg; }
    } /* send to central site */
on fail terminate(fail); }
/* endwhile */
} /* endwhile */
process ServerA(request, reply)
owned struct Message request;
used struct Message reply;
::
struct Message msg;
while (TRUE) {
    request ?? msg on fail terminate(fail);
    printf("ServerA working \n");
    reply !! msg; }
} endprocess
process ServerB(request, RemReq, RemReply, reply)
owned struct Message request;
used struct Message reply, RemReq, RemReply;
::
struct Message msg;
while (TRUE) {
    request ?? msg on fail terminate(fail);
    printf("ServerB working \n");
    RemReq!! msg;
    RemReply ?? msg;
    printf("ServerB working \n");
    reply !! msg; }
} endprocess
process ServerC(request, reply)
owned struct Message request;
used struct Message reply;
::
struct Message msg;
while (TRUE) {
    request ?? msg on fail terminate(fail);
    printf("ServerC working \n");
    reply !! msg; }
} endprocess
process Client(cout, cin)
used struct Message cout;
used struct Message cin;
::
struct Message msg;
int i=0;
while(TRUE) {
    i++;
    msg.type = 0;
    printf("Client sending a local request\n");
    cout !! msg;
    cin ?? msg;
    printf("Client has got the reply\n"); }
} endprocess
process DClient(cout, cin)
used struct Message cout;
used struct Message cin;
::
int i=0;
struct Message msg;
while(TRUE) {
    i++;
    msg.type = i%3;
    printf("Department client: sending a request of type %d\n",msg.type);
    cout !! msg;
    cin ?? msg;
    printf("Department client has got the reply\n");
    i++;
} endprocess

Appendix B. The modified code
struct Message msg;
int busyB=0, busyC=0, busychan=0;
while(TRUE) {
    alt { (busyC==0) ;; requestC ?? msg =>
        /* request from client */
        {busyC=1;
     toSerC !! msg; } }
(busyB==0) ;; requestB ?? msg =>
/* request from client */
{busyB=1;
 toSerB !! msg; }
(busychan==0) ;; requestA ?? msg =>
{busychan = 1;
 msg.dom=id;
 output !! msg; }
fromSerB ?? msg =>
/* request from client */
{busyB=0;
 reply !! msg; }
fromSerC ?? msg =>
/* got reply from local server C */
{busyC=0;
 printf("DeptDispatcher:
 answer to server C\n");
 reply !! msg; }
/* send to client */
{busychan=0); ; RemReq ?? msg =>
/* got remote request from server B */
{busychan =1;
 msg.dom= id;
 output !! msg; }
/* send to central site */
input ?? msg =>
/* got remote reply from CommMgr */
elaborate;
if(msg.type == 1)
{busychan=0;
 RemRep !! msg; }
/* send to server B */
else
/* msg.type == 2 */
{reply !! msg;
 /* send directly to client */
 busychan=0; }
} on fail terminate(fail); }
} endwhile */
} endprocess

process DClient(coutA, coutB, coutC, cin)
used struct Message coutA, coutB, coutC;
used struct Message cin;
:: { }
struct Message msg;

Client–server programs analysis in the EPOCA environment

while (TRUE){
    create_request;
    msg.type = 0;
    coutC !! msg;
    elaborate;
    cin ?? msg;
    elaborate;
    msg.type = 1;
    coutB !!msg;
    cin ?? msg;
    msg.type = 2;
    coutA!!msg;
    cin ?? msg; }
} endprocess

References

vol 19