Structured analysis for Component-based systems: an EJB/CORBA Application

Malika Ioualalen

LSI-USTHB - Université des Sciences et Technologie
BP 32, El-Alia, Bab-Ezzouar, 16111 Alger, Algérie
ioualalen@lsi-usthb.dz

Patrice Moreaux

LISTIC - Université de Savoie
BP 80449, 74944 Annecy le Vieux Cedex, France
patrice.moreaux@univ-savoie.fr

Nabila Salmi

LISTIC - Université de Savoie
BP 80449, 74944 Annecy le Vieux Cedex, France
LSI-USTHB - Université des Sciences et Technologie
BP 32, El-Alia, Bab-Ezzouar, 16111 Alger, Algérie
nabila.salmi@univ-savoie.fr

Abstract

Over the last years, modern software system design has been turning into the use of the Component Based System paradigm. Components are developed in isolation or pre-exist, and are then assembled to build a system. In this paper, we present an example of performance analysis of an Enterprise JavaBeans /Common Object Request Broker infrastructure. with an approach exploiting the component architecture of the system. This approach starts from the definition of the components and their interactions, and applies a structurally-driven way for performance analysis of the whole system. Components are modelled with Stochastic Well-formed Net, a high level model of Stochastic Petri Nets which have proved to be efficient for performance analysis of complex systems with symmetrical behaviours. Structured interconnections of SWN allow an efficient analysis of the global system based on our previous work on decomposable SWN.

Keywords: Performance analysis, Component-Based Systems, Compositionality, SWN, Component, Interface, Interconnection

1. INTRODUCTION

Recent software systems are more and more being designed as a set of components assembled together and interacting to achieve a common goal [3]. This compositional approach has captivated many industrial developers as it allows higher maintainability, reuse, easier upgrade and dynamic reconfiguration. A system is therefore seen as an assembly of components. Such architecture is known as Component Based Systems (CBS). Since the mid'70, several component models have been proposed in the literature [10, 15, 5, 14], and some of them are used in industry.

In this context, a software *component* can be a source unit (consumed at development time and architectural design), or a unit of deployment (machine executable). It can also be a unit of versioning and replacement [16]. It is defined as a unit of composition with contractually specified *interfaces* and explicit context dependencies only.

Although there is no unique definition of what is a component, authors consider that the definition of a component is made of a behaviour (functionalities) and one or several interfaces. Interfaces are used to assemble components, depending on their interaction, specifying required or offered services. Components can be assembled or composed to form a software application or system, according to their specified *interactions*. Adaptation of the interfaces of two components may require a *connector* which is a special kind of component.

With the increasing of applications complexity, analysis of software systems remains an important topic. Some works addressed verification of behavioural properties using formal models, such as Labelled Transition Systems (LTS) and Coloured Petri nets [7, 1]. In the present work, we are interested in performance analysis of CBS, which is usually carried out through measures on existing systems. We use the Stochastic Well formed Net [6] (SWN) model. The SWN is a high level model of Stochastic Petri Nets which have proved to be efficient for performance analysis of complex systems with symmetrical behaviours. However, even with such high level modelling tools, computation of performance indices of CBS are still difficult, mainly due to the huge state spaces of these systems.

In previous works [8, 9], we introduced a decomposition approach for SWN models. This approach starts from the global SWN of a system, decomposes it into several subnets, and applies a structured analysis of the SWN based on analysis of its subnets. In the present work, we adopt a reverse approach fitting component architecture of CBS: we compose the SWN models of the components rather than starting from the global net.

In this paper, we exemplify our adapted structured approach for performance analysis with an example of Enterprise Java Bean (EJB) /Common Object Request Broker (CORBA) system. We start from the components and their SWN models. We also model possible connectors as SWN. Then, we show how we can provide a performance analysis of the CBS while keeping as much as possible, benefits of a structured analysis as demonstrated in our previous results.

The outline of the paper is as follows. In section 2, we describe the EJB/CORBA system under study and the SWN models of its components. We then present our structured analysis methodology in section 3 and its application to our example. Section 4 gives results and comments of the analysis. We conclude in section 5.

2. AN EJB/CORBA PLATFORM

2.1. System architectures

In the context of software architectures, which is the topic of this paper, Enterprise JavaBeans (EJB) and Common Object Request Broker Architectures (CORBA) are two of the most well known proposals and implementations of CBS.

EJB technology (<http://java.sun.com/javaee/index.jsp>) is a software server-side component architecture, developed for the Java 2 Enterprise Edition (J2EE) platform. It enables development of distributed, transactional, secure and portable applications, based on Java technology. It is frequently used for developing business applications.

CORBA (<http://www.omg.org/>) is an object-based software architecture specification promoted by the Object Management Group (OMG). CORBA systems allow interaction between components developed in various languages, through an independent description language. CORBA execution systems are built around one (or several) Object Request Broker (ORB), a kind of software bus. An ORB can communicate with another ORB or with a non CORBA system through the Internet Inter-ORB protocol.

Several companies have existing applications developed with different object oriented languages connected through ORBs. Since they also develop new applications with J2EE systems, there is a need for interconnecting existing CORBA systems with J2EE systems.

Figure 1: EJB/CORBA System (left) and stand-alone CORBA client (right) architectures

A particular interest has been focused on interconnecting EJB components with a CORBA system. Although such an integration seems more or less easy, it is also important to examine performance characteristics of this integration solution. In this context, we exemplify our structured analysis approach by revisiting the work presented in [4]. The goal of this work was to evaluate the performance of the J2EE-CORBA interoperability portion of an end-to-end J2EE-CORBA architecture. Performance has been measured on two kinds of systems (figure 1): a “stand-alone” system made of a client directly invoking methods of CORBA managed objects, and an EJB/CORBA system, where clients call methods of EJB, the J2EE server being able to process the request, or to delegate it to a CORBA system. Results were obtained (measured) by a battery of tests designed for the two cases. In contrast with this previous work, we develop first a model of these systems and then compute performance indices to compare them. Such an approach is adapted to the design phase of an application.

Let us present the elements of an EJB/CORBA-based middleware infrastructure.

A typical CORBA system includes a client communicating with a CORBA system using IIOIP, and a “server” hosting service objects communicating with a database. In the J2EE framework, clients communicate through RMI with the EJB components hosted by the J2EE server. When the J2EE server is interconnected with a CORBA system, it communicates with the CORBA server using IIOIP, and so becomes a CORBA client. In order to allow communication, the architecture is supplied with a service directory implemented by a naming service.

A request scenario is the following: a client application sends an RMI request to the EJB Container, specifying the targeted object and the required interface method. In fact, the client begins by looking for the interface associated to an EJB, and once a reference to the object obtained, it invokes the request. As we concentrate on the interoperability between the EJB component and the CORBA Server, we abstract this step.

Our interest here is to see what is the response time for a client request in the framework of this architecture. As already done in [4], we want to compare the results of two case studies:

1. a stand-alone CORBA client's request : This case study is composed of two components : the client and the CORBA System (figure 1, right).
2. a J2EE application server requesting CORBA services : Three components are specified here : the client, the EJB application server and the CORBA System (figure 1, left).

In both cases, each component must be provided with one or more interfaces. The interfaces are used to connect or assemble components according to their interactions. We can see here that the client interacts with the CORBA System in the first case, and with the EJB Application Server in the second case. The EJB Application Server interacts with the CORBA System too. Therefore, we deduce the following interfaces:

1. The client needs an interface whose role is to invoke a request (either to the CORBA System in case 1, or to the J2EE Application Server in case 2).
2. The EJB Application Server has two interfaces : one for receiving the client request and sending its results, and another one for transferring the request to the CORBA System.

3. The CORBA system is supplied with one interface on which it receives invocation methods and replies to these invocations once the request processed.

A connector will also be required to adapt the interfaces of the client and the CORBA system in the stand-alone case, as it is explained later.

2.2. Modelling components and connectors of the architectures

We model each component of the two systems with SWN. Petri net is essentially a "flat" model, the whole system being modelled by an unique net. However, there have been many proposals to introduce some structure in the PN models when we are dealing with large systems. In the PN framework, interfaces of net components are usually places and/or transitions and interconnections are translated into places or transitions fusion, that is merging places or transitions of several components and adapting arcs in a coherent manner. We follow this approach which is well suited to systems with loosely coupled components.

In addition, as interfaces are used to assemble components, we classify interfaces into two kinds, depending on components interactions :

1. A *synchronous* interface, when assembled components are synchronized through a client/server mechanism, leading to one or more common activities.
2. An *asynchronous* interface corresponding to an asynchronous call method.

The stand-alone CORBA and EJB/CORBA systems are composed of three components: the client, the EJB component (adapter object) and the CORBA system. Another - technical-component, the connector, will be explained below. A fourth component should be quoted : the Naming service. But, since we are interested to compute performances on the interoperability of the EJB Server with the CORBA System, we abstract this component. Note however, that the designer could look for more detailed performance analysis. In this case, he has to provide a functional description of the naming service and its interconnections with the other components. From these descriptions, an SWN may be derived and integrated in the model of the whole system.

The client component The client component (figure 2, up, left) describes a request for executing a service. In the stand-alone scenario, the client requests an Object CORBA Service, after obtaining a reference to it from the Naming service. The request specifies the target service class name ((basic) colour CLA), the method to execute (colour M) and the identification of the client (colour CLI). The client interacts *asynchronously* with the objects server (either the CORBA system in the stand-alone case, or with the EJB Server in the other case). It sends an RMI (or IIOP) request through the output transition *Request_RMI*. When the service system (EJB or CORBA) have processed this request, results are sent to the client and received in the input place *Results*. These two net elements (*Request_RMI* and *Results*) model the asynchronous interface together with the output arc of the transition.

The ejb component The EJB Application server component (figure 2, up, right) is modelled by a local execution of the client's request (see LocalExecute transition), or a request to a CORBA service. As in the standalone client, the EJB component needs to identify what service name is invoked, leading to the use of the Service names class SN. It receives RMI requests (as triplets (*Cl*, *ER*, *M*)) from the client in the input place *Requests*. After processing the request, the component either ends the communication with the client by the *EndLocal* transition (local processing of the request), or by the *EndDistant* transition (distant processing of the request). These two transitions with their output arcs and the *Requests* place is an asynchronous interface which is to be connected later with the client component. To invoke a CORBA service, the component synchronizes itself with the CORBA system through a *synchronous interaction*. This

Figure 2: The client (up, left), the CORBA (up, right) components and the EJB component of the EJB/CORBA System (bottom)

Figure 3: The connector for Client-CORBA system interconnection

synchronization is initiated by the *InvokeMethod* transition, waits to be processed in *T25*, and is ended by desynchronizing from the CORBA server with the *End_method* transition. So, a second interface is exposed by the ejb component, but a synchronous one composed of the *InvokeMethod*, *T25* and *EndMethod* transitions.

The CORBA System When the CORBA system (figure 2, bottom) is solicited for executing a business method, it synchronizes itself with the requestor of the method (which in turn is either the stand-alone client or the ejb component), through the *InvokeMethod* transition. Then, it gets the data (*GetData* and *T20* transitions). It finally ends the request by sending results to the partner of communication through the *End_method* transition. This leads to a synchronous interface provided by the CORBA component, composed of one input transition *InvokeMethod* and three continuation/end transitions *GetData*, *T20* and *EndMethod*. The synchronization is done with the Service Objects basic class *SO*. The CORBA system also binds its instantiated objects to the Naming Service by sending a *bind* request to this service. This is done through the *BindObject* transition.

The connector component Since the client requires an asynchronous interface, and the CORBA system exposes a synchronous service call, the designer must supply a software adapter, called a connector in the CBS context. The connector module simply receives the request from the client and call the CORBA service. It then returns the result to the client (figure 3).

3. STRUCTURE DIRECTED ANALYSIS METHODOLOGY OF CBS

3.1. From CBS to structured SWN models

As we mentioned in the introduction, we aim at developing an analysis methodology based on the architecture of CBS. This means that we try to keep as much as possible, the component view of the system in order to benefit from structured qualitative and performance analysis methods we previously developed [11, 12, 8, 9]. Although (coloured) PN models of components could be assembled in many ways (see for instance [2] for a general solution), our approach is based on detection of structured interactions between components. We consider two specific possible interactions between components:

Figure 4: Synchronous (left) and asynchronous (right) interfaces and interconnection

- A "synchronous" interaction (figure 4, left) where a client and a service process complex activities involving both entities. This interaction is restricted to well identified parts of the SWNs of the client and the service. An SWN component may expose two kinds of synchronous interfaces, depending on the exchange of colours between components. When colours "cross" from one component's interface to another one in the interaction, the interface is said *two-phases*. Otherwise, it is called *anonymous* since there is no memory of the synchronization between colours. In the two-phases case, the interface is said *client-side* when it does not interact with its component core through coloured tokens. It is called *server-side* when it is allowed to do so, but there could not be client-side colour involved in the synchronization phase, in order to apply the structural analysis method. Note that the terms "client" and "server" differ from those of a client/server architecture. In such architectures, a client requests services from the server side, which performs the service and provides results to the client. Whereas in our case, we should quote two points : On one hand, the client (respectively server) component is not necessarily an SWN-synchronized client-side (resp. server-side) : the service part belongs to the client core, as it is required by the analysis method (see example in 3.3.1). On the other hand, depending on the kind of request, client-server systems may be modelled as synchronous or asynchronous interaction (or none of them) in our terminology .
- An "asynchronous" interaction (figure 4, right) which corresponds to an asynchronous method call or a message sending and receiving between the client and the service. During processing of the request, the client does nothing else. An asynchronous interaction is initiated by one single component, and is propagated through a chain of other components in a sequential way, ending in the initiating component. Basic colour classes corresponding to entities moving from module to module are called *global* colour classes, and other colour classes are named *local* colour classes.

These two structured interactions are adapted to loosely coupled CBS. However, there are many cases where we cannot derive such interconnections; this is generally due to complex (and not loosely coupled) interactions. In these cases, the technical conditions imposed to structured

interconnection are not fulfilled, and we can simply merge the components in the standard way (places and transitions fusion) as in [2]. We then get a new SWN and we can try to apply structured interconnection of this SWN with the other ones. Obviously, a fully unstructured system leads to a global SWN without any structure, and our method is then useless. Nevertheless, this kind of unstructured system can be studied through the analysis of its global sWN.

Having described our system as a set of interacting SWNs, we are able to carry out its analysis, also in a structured way.

3.2. Structured analysis

A general approach for analyzing a CBS can be to connect the SWN components interfaces, leading to a global SWN, and, then to analyse the global net with an appropriate tool. This is the method proposed by [2] implemented in the Algebra tool. This method merges places or/and transitions of two or more components, on the basis of fusing elements with same labels, and uses the GreatSPN tool [13] on the obtained global SWN for standard SWN analysis.

In contrast, we propose to keep structure information about the CBS to analyze its model. We follow an approach which enables us to apply a modified version of the structural composition analysis method we developed previously [8, 9], defined for either synchronous or asynchronous decomposition of SWNs. This technique avoids the construction of the whole Markov Chain corresponding to the global net, using a tensorial representation of the symbolic reachability graph (SRG), hence enabling important memory and computation time savings, and so, reducing analysis complexity.

3.2.1. Principle of structured analysis method

The initial decomposition method proceeds in several steps:

- Decomposition of the global SWN (say $\mathcal{N}$) into components in either a synchronous or an asynchronous way giving a set of SWNs (say $\mathcal{N}_k$).
- Checking applicability conditions for a structured representation of the SRG and its aggregated generator.
- Extension of the components SWNs $\mathcal{N}_k$ to autonomous SWNs $\tilde{\mathcal{N}}_k$.
- Generation of the SRGs of these extended SWNs.
- Computation of the synchronized product of these SRGs and of the tensorial representation of the generator of the underlying aggregated Markov chain.
- Computation of the steady state or transient distribution of the aggregated model and computation of the required performance indices.

In this paper, we extend this method to CBS. In the general case, CBS may present both synchronous and asynchronous interconnections between components. Hence, we are faced with two kinds of problems :

1. First, some interconnections may not fit the synchronous neither the asynchronous interconnection pattern. In this case, we must merge the corresponding component models and try to apply the method with the new set of SWNs made of the new merged SWN and the SWNs of the other components.
2. Second, without restriction, asynchronous and synchronous interconnections related to the same component forbid applicability conditions to be fulfilled. Some kind of autonomy is required between colours behaviours outside the synchronization part of a synchronous interaction between two components A and B . Intuitively, this autonomy may be broken by global colours moving from A to B in an asynchronous interconnection. We term *multi-synchronized*, component SWNs for which such a situation arises. If there are no pair of multi-synchronized SWNs in the composed system, we say that the system is a *non multi-synchronized* composition. In this case, we can develop a structured solution as detailed below. Note however, that for multi-synchronized systems, we can try to merge involved components in a unique SWN and apply if possible our method.

A final technical adaptation of the decomposition approach is required. We compose and interconnect SWN components rather than decomposing the global SWN of a system. As a consequence, name conflicts (on colours, places or transitions) may arise. So, we systematically rename conflicting names involved in interactions with unambiguous ones. In addition, colour class names corresponding to the same entities in the interconnected components must be mapped and renamed. We can see this phenomenon when interconnecting the client and EJB component modes: the client invokes a request to a certain service class with colour CLA. This colour is mapped with the ER class colour when interconnection is done.

Consequently to renaming colours, all analysis results and performance indices have to be restated in the initial context when computed.

3.2.2. Algorithm for structured analysis of components based SWN

The proposed method is summarized in the following algorithm. These steps are currently implemented in our tool *compSWN*.

1. Start from the components with their interfaces.
2. Define the interactions between pairs of components.
3. Check for name conflicts between components, and rename them when necessary.
4. Build the global net, connecting components.
5. Check conditions for structured analysis:
 if they are fulfilled, goto 6
 else //to be done by an SWN expert
 Merge components to new components
 trying to fulfill conditions.
 Goto 2.
6. Build the extension $\overline{\mathcal{N}}_k$ of each SWN component $\mathcal{N}_k$.
7. Study the extended nets $\overline{\mathcal{N}}_k$ in isolation, and build the associated SRGs.
8. Compute the “synchronized” product of the SRGs yielding a state space of tuples of symbolic markings and a tensorial representation of the generator of the aggregated Markov chain.
9. Compute performance indices, using this tensorial expression.
10. Provide results in the initial context of the components.

3.3. Application to the EJB/CORBA architectures

Let us apply our method to the stand-alone CORBA and EJB/CORBA systems presented in section 2. Since the required and exposed interfaces differ between clients and CORBA systems, we study successively the two architectures. In both cases, we connected components to check applicability conditions on the global net. We mapped colour names between components, and rename them for analysis requirements. We see that that conditions for a structured analysis are met in the two architectures. We also mention that there is no multi-synchronized components, particularly in the EJB/CORBA system, where two interactions are involved between the components : an synchronous interaction between the client and EJB components, and a synchronous one between the EJB and CORBA components.

Finally, we extend components in each case, and then carry out the analysis on extended nets.

3.3.1. Stand-alone client-CORBA system architecture

For the stand-alone case, we have to connect the client to the CORBA system with the help of a connector (adaptation) module. Since the connector is a very simple one, we choose to add it directly to the client component. Otherwise, when the connector is a complex subsystem, we can choose to consider it as a proper component: in this case, we follow the same approach than for the EJB/CORBA system (see below).

Figure 5: Client (with connector) extended net in the stand-alone case

Figure 6: CORBA extended net in the stand-alone and EJB/CORBA cases

Figure 7: EJB extended net (in the EJB/CORBA system)

The client (with composed connector) SWN is then extended (figure 5) in the context of a synchronous composition with the CORBA system.

The extended CORBA system SWN is given in figure 6. Note that, although the CORBA system is a server from the functional point of view, it has the client role in the synchronous composition with the connector (or the EJB component below). This is a classical “inverted” view, due to technical conditions for structured composition of SWN, but it does not impact the design of the CBS.

3.3.2. EJB/CORBA system architecture

For the EJB/CORBA system, the client SWN is simply extended in the context of an asynchronous composition with the EJB component. The extended SWN of the EJB component is given in figure 7, integrating an SWN-asynchronous interconnection with the client component, and an SWN-synchronous interconnection with the CORBA system. The extended CORBA system is the same as the one of the stand-alone case.

4. ANALYSIS AND RESULTS

We used our new *compSWN* tool to compute steady-state probabilities of the two configurations (stand-alone and EJB/CORBA systems). We also used the GreatSPN environment on the global net to compare results of both analysis methods.

4.1. Model parameters

We fixed the cardinalities of our basic colour classes to 3 clients, 3 EJB references, 3 methods and 3 server objects. Note that a colour may model a group of elementary entities, for instance a EJB reference colour can stand for 10, 100 or 1000 references; obviously, firing rates of transitions involving this colour should be adapted to the semantics of a colour (100 references provide a 100 times slower method request rate for instance). We also take initially homogenous rates of value 1 for all transitions. Then, we vary some transition rates, and study the evolution of response time from steady-state probabilities, in both configurations.

Figure 8: Response time versus client request'rate (up, left), requests processing rate (up, right) getting data rate (bottom, left) and method invocation rate (bottom, right)

However, in order to be coherent when comparing the two scenarii, we assume that a client request is most often processed by a distant CORBA service, rather than processed locally by the EJB container, since the stand-alone CORBA client invokes directly a distant service. So, we fixed the rate of the LocalExecute transition in the EJB/CORBA system, as 0.001 compared to the rate of the transition RequestCorbaService varying between 1 and 6 in our computations.

4.2. Response time variations and comparison between the two configurations

We interested mainly in computing variations of the response time in the two configurations with respect to several parameters:

- the load induced by client's requests;
- the rate of the method invocation;
- the rate of getting data in the CORBA system;
- the rate of effective processing of a request.

Particularly for the EJB component, we also computed these variations with respect to two other cases:

- the rate with which the EJB component prepares the request before sending it to the CORBA System.
- the impact of a lookup operation in the EJB component.

Figure 8 shows response time variations with respect to two parameters : the load of the system, and the duration of a request processing. In the first diagram, the two configurations present an increasing response time as far as the number of client requests increases, with much greater times for the EJB/CORBA architecture. This can be expected, because adding a third entity

Figure 9: Response time versus lookup rate (left), and request preparation rate (right)

between the client and the CORBA system implies a greater response time. But, in the other hand, response time of the EJB configuration evolves much faster than that of the stand-alone configuration. This can be explained by the fact that, when the EJB application server is overloaded by a great number of requests, it processes requests more slower, and this induces an additional time.

Whereas the second diagram shows a constant difference in response time variations of the two architectures. This means that the CORBA system processes requests identically in the two architectures.

We interested also in observing the evolution of response time with respect to the increasing of time required to bring back data from the database. If this time increases (see first diagram in figure 8), due to an increasing size of required data or to the low speed of the Database management system, a consequent response time is obtained. However, if the rate of getting data is too big, smaller response times are derived in both architectures. We can see also that a constant difference exists between response times of the two cases.

The same conclusions hold for the second diagram of this figure.

The last figure (9) shows in the first diagram the effect of a lookup request in processing a request, leading to greater response times corresponding to small lookup rates.

Finally, we computed response time variations with respect to the time needed in the EJB server to prepare a request for a CORBA service. Our objective here is to deduce what performances the J2EE application server must have in order to be as efficient as possible. We can see in the second diagram of figure 9 that the rate of preparing a request within the application server must be greater than 6 in order to reach response times of the standalone case.

We conclude by comparing our results to those of the paper [4]. Let us recall that the primary goal of this paper was to compare the two systems. For the most, our results indicate that response time of the EJB/CORBA system is slower than that of the stand-alone one. It seems coherent since the EJB mechanism introduce new processing steps. This behaviour is identical to the one of the experimental study. However, we quote that these conclusions are extended in the above paper to *almost all* experimental configurations: some of them give better results for EJB/CORBA systems than stand-alone systems. We are surprised of such results, which seems to derive from non comparable configurations (different ORB (Object Request Broker) for instance, used for communication with the CORBA system in the two studied architectures).

5. CONCLUSION

In this paper, we have presented an analysis of two Component Based Systems (CBS) allowing a client's request processing by EJB based or CORBA based distant objects. Such systems

are loosely coupled and we applied our structure guided analysis method for their performance evaluation. Our approach is an extension of the decomposition method for Stochastic Well formed Nets (SWN) we previously defined. It consists in several steps: modelling the components and the connectors with SWN, interconnection of the components SWN, verification of the structural conditions for structured analysis, extension of the component SWNs and computation of the synchronized product of their underlying aggregated Markov chains. The method requires more applicability verification than the decomposition approach due to mixed synchronous and asynchronous interactions. Results were computed with our tool *compSWN* implementing our method. They show that EJB/CORBA systems introduce some overloading with respect to stand-alone CORBA systems.

Future works will apply our method to other examples of CBS.

Bibliography

- [1] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components: The vercors platform. In *3rd workshop on Formal Aspects of Component Systems*, Prague, Tcheque Republic, September 2006. ENTCS.
- [2] S Bernardi, S. Donatelli, and A. Horváth. Implementing compositionality for stochastic Petri nets. *Int. J. STTT*, (3):417–430, 2001.
- [3] Antonia Bertolino and Raffaella Mirandola. Software performance engineering of component-based systems. In *Proc. of the WOSP 04*, Redwood City, C.A., January 14–16 2004.
- [4] John Binu, Vu Trang, Michael Vernik, Stacie Wolny, and Sherry Yu. Performance evaluation of enterprise javabeans (ejb) corba adapter to corba server interoperability. <http://java.sun.com/developer/technicalArticles/ebeans/corba/>, January 2002.
- [5] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Proc. of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP02)*, June 2002.
- [6] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Transactions on Computers*, 42(11):1343–1360, November 1993.
- [7] L. Dias da Silva and A. Perkusich. Composition of software artifacts modelled using colored Petri nets. *Science of Computer Programming*, 56(1-2):171–189, April 2005.
- [8] C. Delamare, Y. Gardan, and P. Moreaux. Efficient implementation for performance evaluation of synchronous decomposition of high level stochastic Petri nets. In *On-site proceedings of the ICALP2003 Workshop on Stochastic Petri Nets and Related Formalisms*, pages 164–183, Eindhoven, Holland, June 21-22 2003. University of Dortmund, Germany.
- [9] C. Delamare, Y. Gardan, and P. Moreaux. Performance evaluation with asynchronously decomposable SWN: implementation and case study. In *Proc. of the 10th Int. Workshop on Petri nets and performance models (PNPM03)*, pages 20–29, Urbana-Champaign, IL, USA, September 2–5 2003. IEEE Comp. Soc. Press.
- [10] Object Management Group. Corba concurrency service 1.0. Technical report.
- [11] S. Haddad and P. Moreaux. Evaluation of high level Petri nets by means of aggregation and decomposition. In *Proc. of the 6th International Workshop on Petri Nets and Performance Models*, pages 11–20, Durham, NC, USA, October 3–6 1995. IEEE Computer Society Press.
- [12] S. Haddad and P. Moreaux. Asynchronous composition of high level petri nets : a quantitative approach. In Springer Verlag, editor, *Proc. of the 17th International Conference on Application and Theory of Petri Nets*, volume 1091, pages 193–211. LNCS, 1996.
- [13] P.E. Group. GreatSPN home page: <http://www.di.unito.it/~greatspn>, 2002.
- [14] L. Seinturier and F. Loiret. état de l'art sur les modèles de composants pour le temps réel et l'embarqué. Technical report, ANR-ARA Sécurité, Systèmes embarqués et Intelligence ambiante, <http://reve.futurs.inria.fr/livrables/T0+6/d111.pdf> (28/01/2007), June 2006.
- [15] Sun Microsystems. J2ee connector architecture. Technical report, Sun Microsystems, Inc., <http://java.sun.com/j2ee/connector>(jan. 2006), 2006.
- [16] C. Szyperski. Component technology - what, where, and how? In *Proc. 25th Int. Conf. on Software Engineering*, pages 684–693. IEEE, May 3–10 2003.