A Formal Framework for Middleware Behavioural Specification

Nelson Souto Rosa and Paulo Roberto Freire Cunha
Universidade Federal de Pernambuco, Centro de Informática
Recife, Pernambuco
{nsr,prfc}@cin.ufpe.br

Abstract

The number of open specifications of middleware systems and middleware services is increasing. Despite their complexity, they are traditionally described through APIs (the operation signatures) and informal prose (the behaviour). This fact often leads to ambiguities, whilst making difficult a better understanding of what is actually described. This paper presents a formal framework, specified in LOTOS (Language Of Temporal Ordering Specification), for the specification of middleware systems. The framework consists of a set of basic/common middleware components and some guidelines on how to compose them. The components of the framework facilitate the formal specification of different middleware systems. In order to illustrate how the framework may be used, it is adopted to specify procedural (synchronous) and message-oriented (asynchronous) middleware systems.

Keywords: Middleware, Formal Specification, LOTOS, Framework

1 Introduction

Middleware specifications are not trivial to be understood, as the middleware itself is usually very complex [7]. Firstly, middleware systems have to hide the complexity of underlying network mechanisms from the application. Secondly, the number of services provided by the middleware is increasing, e.g., the CORBA specification contains nineteen services [15][14]. Finally, in addition to hide communication mechanisms, the middleware also have to hide fails, mobility, changes in the network traffic conditions and so on. On the point of view of application developers, they very often do not know how the middleware actually works. On the point of view of middleware developers, the complexity places many challenges that include: how to integrate services in a single product [21], how to satisfy new requirements of emerging applications [5], how to understand the middleware behaviour prior to implementing it.

The main objective of this paper is to propose a framework that helps to formally describe middleware behaviour by providing a set of basic abstractions. These abstractions are specified in LOTOS and basic/common in the sense that may be combined in different ways in order to specify diverse middleware systems. Main in our approach is the fact that the abstractions are defined and organised according to their role in relation to the message request. Hence, instead of adopting the traditional approach of organising middleware systems in layers [17], the proposed abstractions are defined considering their role in the message request. Hence, the abstractions are grouped into categories related to storage, communication, dispatching, and mapping of message requests. A message request is any message that an application (e.g., client, server, sender, transmitter) sends to another application.

This paper is organised as follows: Section 2 introduces basic concepts of LOTOS and properties that may be checked using this process algebra. Next, Section 3 presents the proposed framework. Section 4 presents how the proposed framework may be used to specify client-server and message-oriented middleware systems. Section 5 present some related work. Finally, last section presents an evaluation of the research until now and some future work.

2 Background

Prior to present the proposed framework, next section introduces some basic concepts of LOTOS. Additionally, we present the temporal logic used to express the temporal properties of LOTOS specifications.

2.1 LOTOS

A LOTOS [11][6] specification describes a system through a hierarchy of active components, or processes. A process is an entity able to realize non-observable internal actions, and also interact with other processes through externally observable actions. The unit of atomic interaction among processes is called an event. Events correspond to a synchronous communication that may occur among processes able to interact with one another. Events are atomic, in the sense that they happen instantaneously and are not time consuming. The point where an event interaction occurs is known as a port. Such event may or may not actually involve the exchange of values. A non-observable action is referred to as an internal action or internal event. A process has a finite set of ports that can be shared.

An essential component of a specification or process definition is its behaviour expression. A behaviour expression is built by applying an operator (e.g., parallel operator "`||`") to other behaviour expressions. A behaviour expression may also include instantiations of other processes, whose definitions are provided in the "where" clause following the expression [7]. The complete list of basic-LOTOS behaviour expressions is given in Table 1, which includes all basic-LOTOS operators. Symbols '`B`', '`B1`', '`B2`' in the table stand for any behaviour expression, and "`i`" is an internal action.

Name	**Syntax**	**Semantics**
inaction	`Stop`	It cannot offer anything to the environment, nor it can perform internal actions.

action prefix
- unobservable
- observable
	`i ; B`
	`g; B`
		It is capable of performing action i (g) and transform into process B.

choice	B_1`[]`B_2	It denotes a process that behaves either like B_1 or like B_2.
parallel composition - general case - pure interleaving - full synchronization	B_1 **|[** $g_1,..,g_n$ **]|** B_2 B_1 **|||** B_2 B_1 **||** B_2	A parallel composition expression is able to perform any action that either component expression is ready to perform at a gate (not in $g_1,…,g_n$) or any action that both components are ready to perform at a gate in $[g_1,…,g_n]$.
hiding	**hide** $g_1,...,g_n$ **in** B	Hiding allows one to transform some observable actions of a process into unobservable ones.
process instantiation	P[$g_1,...,g_n$]	It is used to express infinite behaviours.
successful termination	**Exit**	exit is a process whose purpose is solely that of performing the successful termination
sequential composition (enabling)	B_1 **>>** B_2	B_2 is enabled only if and when B_1 terminates successfully.
disabling	B_1 **[>** B_2	B_1 may or may not be interrupted by the first action of process B_2.

Table 1 - Syntax of behaviour expressions in LOTOS [6]

Next, we present the LOTOS specification of a simple client-server system:

```
(1) specification Client_Server [request, reply]
                                     : noexit
(2) behaviour
(3)  Client [request, reply]
     || Server [request, reply]
(4) where
(5)   process Client [request, reply] : noexit :=
(6)     request; reply; Client [request, reply]
(7)   endproc
(8)  process Server [request, reply] : noexit :=
(9)    hide processRequest in
(10)      request;
(11)      processRequest;
(12)      reply;
(13)      Server [request, reply]
(14)   endproc
(15)endspec
```

The top-level specification (3) is a parallel composition (operator '`||`') of the processes `Client` and `Server`, i.e., every action externally observable executed by the process `Client` must be synchronised to the process Server. The process `Client` (5) performs two actions, namely request and `reply` (6), and then re-instantiate. The action-prefix operator ('`;`') defines the temporal ordering of the actions request and reply (the action request occurs before the action reply) in the `Client`. Informally, the `Server` (8) receives a request (10), processes it (11) and then sends a `reply` (12) to the process `Client`.

It is worth pointing out that LOTOS specifications may be compared in order to check their behavioural equivalences such as strong, observational and safety equivalences. All of them are checked through the CADP Toolbox[1].

3 LOTOS Specifications of Middleware Components

As mentioned before, the proposed framework consists of a set of abstractions that addresses a number of common functionalities of middleware systems. The framework also defines how these abstractions work together to formalise different middleware models. For example, the abstractions may be combined to produce the specification of a message-oriented middleware, whilst they also may be combined to define a procedural middleware (client-server applications) or a tuple space-based middleware.

The whole framework is "message-centric" in the sense that basic elements of the framework are grouped according to how they act on the message. Figure 1 shows a general overview of the proposed approach in which the message is intercepted by both middleware elements on the transmitter and receiver sides. It is worth observing that the message may be either a request in which the transmitter ask for the execution of a task on the receiver side or a simple information between loosely-coupled applications.

[1] http://www.inrialpes.fr/vasy/cadp/

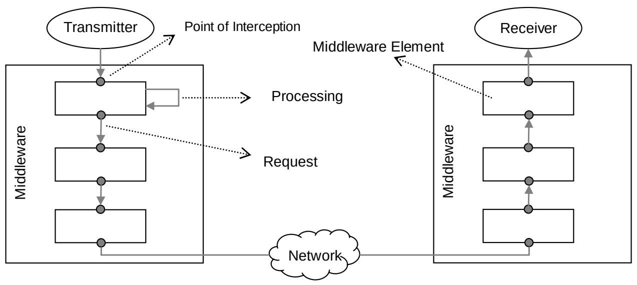

Figure 1 – Message-centric approach

The abstractions of the framework are categorised into four classes: mappers (e.g., stub and skeletons), multiplexers (e.g., dispatcher), communication (e.g., communication channel), and storage (e.g., queue and topic). Whatever the class of abstraction, it intercepts the message, processes it and forwards the message to the next element. The next element may be a local or remote one. However, only communication elements may forward the message to a remote element, i.e., an element only accessible through the network. A non-communication element may need to communicate with a remote element to carry out its task, but it does not send the message directly to a remote element. For example, a transaction service may need to obtain a remote lock before pass the request to the next element of the middleware. In this case, the transaction service uses a communication element to send the message and obtains the lock.

3.1 Basic Abstractions

Mapper Elements Mapper elements typically represent remote objects, serve as input points of the middleware, their basic function is to (un)marshal application data (arguments and results) into a common packet-level (e.g., GIIOP request), and are usually found in middleware systems that support request/reply applications in heterogeneous environments. Additionally, non-conventional mappers may also compress data. The specification of a typical mapper, namely `Stub`, is defined as shown bellow:

```
(1) process Stub [iStub, oStub] : noexit :=
(2)  iStub ?m : Message;
(3)   oStub !marshalling (m);
(4)    iStub ?m : Message;
(5)     oStub !unmarshalling (m);
(6)      Stub [iStub, oStub]
(7) endproc
```

In this specification, the input (`iStub`) and output (`oStub`) ports serves as interception points of the stub. Information sent to the `Stub` from another abstraction is intercepted in the port `iStub`, whilst information the Stub sends to another abstraction is passed through the `oStub`. Hence, `Stub` receives a message sent by the transmitter and intercepted by the middleware (2), marshals it (3), passes it to the next element (4), and then waits for the reply from the receiver. The reply is also intercepted by the middleware and passed to the `Stub` (4) that takes responsibility of unmarshalling the reply (5).

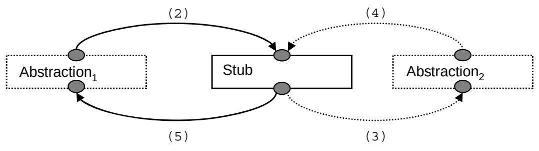

Figure 2 – Stub

Figure 2 depicts an intuitive view on how the stub actually works. The numbering refers to the lines of the Stub specification. Unlike other abstractions, the Stub both sends and receives information. Additionally, Abstraction$_1$ and Abstraction$_2$ are local abstractions in the sense that both are placed in the same local as the transmitter.

Communication Elements Communication elements get a message and communicate it to a remote element. They act as an interface between the middleware and the operating system. The structure of a communication element, named `Channel`, as follows.

```
(1) process Channel [iCh, oCh, comm] : noexit :=
(2)   Send[iCh,oCh,comm]|||Receive[iCh,oCh,comm]
(3)   where
(4)     process Send [iCh, oCh, comm] : noexit :=
(5)       iCh ?m : Message;
(6)        comm !m;
(7)         oCh;
(8)          Send [iCh, oCh, comm]
(9)     endproc
(10)    process Receive[iCh, oCh, comm]:noexit :=
(11)      iCh;
(12)       comm ?m : Message;
(13)        oCh !m;
(14)         Receive [iCh, oCh, comm]
(15)    endproc
(16) endproc
```

Similarly to the `Stub`, the input (`iCh`) and output (`oCh`) ports serves as interception points of the element. However, communication elements have an additional port, named `comm`, used to communicate the message to a remote element. Additionally, the `Channel` is composed by the processes `Send` and `Receive` that are responsible to send and receive messages, respectively. In this case, the `Channel` receives a message sent by the local element (5) and then communicates it to a remote element (6). Next, the reply message is received by the `Channel` (12) from a remote element and then it passes the message to the local element (13). Figure 3 illustrates how the Channel works.

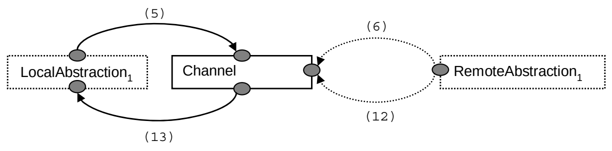

Figure 3 – Channel

Dispatchers Dispatchers get the request and forward it to the right object (service). The destination object is defined by inspecting the message, in which the destination has been set during the binding. In practical terms, the dispatcher acts as a multiplexer inside the middleware. The general structure of a dispatcher is depicted bellow.

```
(1) process Dispatcher [iDis, oDis] : noexit :=
(2)   iDis ?m : Message;
(3)    oDis !m ! multiplexer(m);
(4)     Dispatcher [iDis, oDis]
(5) endproc
```

The dispatcher receives a message (2) and inspects it, through the function multiplexer, to define the destination object (3). Figure 4 illustrates how the multiplex work considering two different destination objects.

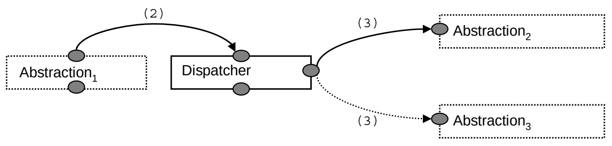

Figure 4 – Dispatcher

Storage Elements Storage elements express the need of some middleware systems of store the message prior to send it, e.g., for asynchronous communication or to keep a copy of the message for recovery reasons. The general structure of a `Storage` element is shown bellow.

```
(1) process Storage[iSto, oSto](q: Queue):
                                  noexit :=
(2)  hide enq, fst, empt, deq in
(3)    Manager [iSto, oSto, enq, fst, empt, deq]
(4)         |[enq, fst, empt, deq]|
(5)    Queue [enq, fst, empt, deq] (q)
(6)    where
(7)      process Manager [iSto, oSto] : noexit :=
(8)        iSto ?m : Message;
(9)        enq !m;
(10)       oSto;
(11)       Manager [iSto, oSto]
(12)     endproc
(13)     process Queue [enq, fst, empt, deq]
                       (q : Queue) : noexit :=
(14)        enq ?n : Nat;
(15)        Queue[enq,fst,empt, deq]enqueue(q,n))
(16)        []
(17)        fst !first (q);
(18)        Queue [enq, fst, empt, deq] (q)
(19)        []
(20)        deq;
(21)        Queue[enq,fst,empt,deq](dequeue (q))
(22)       endproc
(23) endproc
```

The storage element is modelled as a `Queue` that is administered by the `Manager`. In this particular case, the `Manager` receives a message (8) and then puts it in the `Queue` (9) which is inside the `Storage`. In particular, the queue has a traditional structure as shown in the specification that includes traditional queue operations such as: enqueue (14), to push the first element of the queue (17) and dequeue (20). Figure 5 shows the basic functioning of the `Storage`.

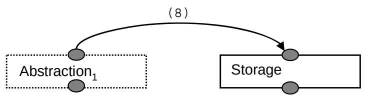

Figure 5 – Storage

It is worth observing that with minor changes to the storage element, it may be defined as a buffer or a file.

3.2 Putting the Basic Abstractions Together

By using the basic abstractions defined in the previous section, middleware systems may be specified by composing them according to the desired distribution model. The general structure of any specification according to the framework is defined as follows:

```
specification CompleteSystem [invC, terC,
                   invS, terS, comm] : noexit

   (* abstract data type definitions *)

  behaviour
    (Transmitter[invC,terC]
        |[invC,terC]|
     LocalMiddleware[invC,terC, comm])
       |[comm]|
     RemoteMiddleware [invS,terS,comm]
       |[invS,terS]|
     Receiver[invS,terS])
     where
       (* behavioural specification *)
Endspec
```

where a `Transmitter` sends a message to the `Receiver` through the middleware, which is made up of a local (`LocalMidleware`) and remote middleware (`RemoteMidleware`) that communicates through the port `comm` (e.g., it abstracts the whole network). Whatever the middleware model, its internal structure is defined as follows (except for the number of components):

```
process LocalMiddleware[invC,terC,comm]:noexit:=
   hide iA1, oA1, iA2, oA2 in
    (A1 [iA1,oA1] ||| A2 [iA2,oA2,comm])
          |[iA1, oA1, iA2, oA2]|
     Interceptor [invC, terC, iA1, oA1, iA2, oA2]
   where
       (* behavioural specification *)
endproc
```

The middleware is composed of a set of abstractions (e.g., `A1` and `A2`), depending on its complexity. The composition is expressed in the process `Interceptor`. As mentioned before, our approach is message-centric. Hence, each abstraction initially "intercepts" the request, processes it and then passes to the next one according to the constraints imposed by the process `Interceptor`. The Interceptor plays a key role of defining the order the request is intercepted by the abstractions. For example, the interceptor of a `LocalMiddleware` in a client-server communication may be defined as follows:

```
(1) process Interceptor [invC, terC, iStub,
                         oStub, iCha, oCha] :
                         noexit :=
(2)   invC ?m : Message;
(3)   iStub !m;
(4)   oStub  ?m1 : Message;
(5)   iCha !m1;
(6)   oCha;
(7)   iCha;
(8)   oCha ?m : Message;
(9)   iStub !m;
(10)  oStub ?m3 : Message;
(11)  terC !m3;
(12)  Interceptor [invC, terC, iStub,
                   oStub, iCha, oCha]
(13)      endproc
(14)   endproc
```

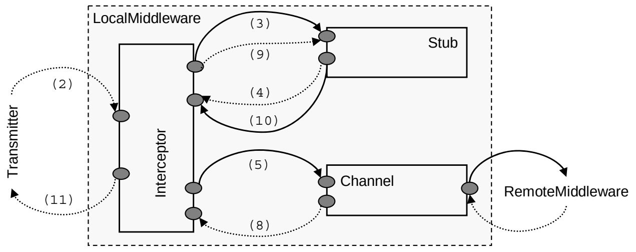

Figure 6 – Interceptor

Figure 6 shows the interaction among the elements that composes a particular `LocalMiddleware`. In this case, according to the interactions, the order imposed by the `Interceptor` defines that after the `LocalMiddleware` receives the request from a `Transmitter`, it is passed initially to the `Stub` and then to the `Channel`. When the reply message arrives in the `Channel` from the `RemoteMiddleware`, it is passed back to the `Stub` and then to the `Transmitter`.

4 Adopting the Framework Elements

In order to illustrate how the elements introduced in the previous session may be used to facilitate the middleware specification, we present the specification of a simple middleware (Figure 7) that has a structure similar to CORBA and a message-oriented middleware (Figure 8).

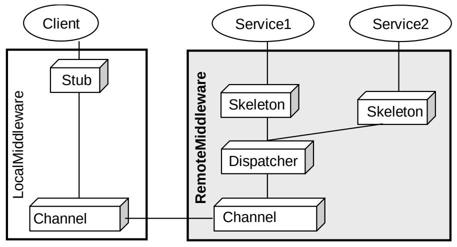

Figure 7 – Client-server middleware

According to Figure 7, the specification presents a client-server middleware where the local middleware is a composition of a stub and channels elements. On the server side (remote), the middleware is more complex, as it is composed by a communication element (`Channel`), a dispatcher (`Dispatcher`) that forwards the request to the proper skeleton, and some skeletons (`Skeleton`). It is worth observing that additional middleware elements are easily added to the middleware just including them in the parallel composition (`|||`) and changing the `Interceptor` element.

```
process RemoteMiddleware[invS,terS,comm]:noexit
:=
 hide iSkeleton,oSkeleton,iTcp,oTcp,iDis,oDis in
    ((Skeleton [iSkeleton, oSkeleton] (1)
  ||| Skeleton [iSkeleton, oSkeleton] (2)
  ||| Channel [iTcp, oTcp, comm]
  ||| Dispatcher [iDis, oDis])

     |[iSkeleton,oSkeleton,iTcp,oTcp,iDis,oDis]|

  Interceptor [invS, terS, iSkeleton, oSkeleton,
               iTcp, oTcp, iDis, oDis] )
  where
      (* behavioural specification *)
endproc
```

The adoption of LOTOS enables us to use tools to check properties of middleware specifications defined using the framework. In this particular case, the searching of deadlock has been carried out in the CADP Toolbox (Cesar) . The trace resulting from this evaluation is shown bellow

```
*** searching for sequence of the form:
<any>* . <deadlock>
*** using breadth-first search algorithm
*** no sequence found
*** no prefix of the sequence has been recognized
```

As mentioned before, the second middleware specification is a message-oriented middleware (MOM). A MOM is characterised by the use of a buffer to the asynchronous communication and it is widely adopted to communicate loosely coupled applications.

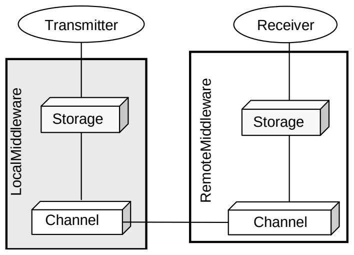

Figure 8 – Message-Oriented Middleware

This MOM (Figure 8) has two elements, namely `Channel` and `Storage`. The abstraction `Channel` is similar to Figure 7, whilst `Storage` is defined as presented in Section 3.1. MOMs that execute on the transmitter side are usually similar to one on the receiver (remote) side. The general structure of the MOM specification is shown bellow

```
Process LocalMiddleware[send,receive,comm]:
noexit :=
hide iSto, oSto, iCh, oCh in
 ((Storage[iSto,oSto] ||| Channel[iCh,oCh,comm])
        |[iSto, oSto, iCh, oCh]|
    Interceptor[send,receive,iSto,oSto,iCh,oCh])
where
        (* behavioural specification *)
endproc
```

5 Related Work

The basic idea of the formalisation of middleware systems is to use a formal description technique for specifying several aspects of middleware. In particular, formal description techniques such as E-LOTOS [11], Z notation [7], and Petri Nets [3] [4] have been used to specify functional aspects, whilst Petri nets have also been adopted to model middleware performance aspects [23][10].

In the RM-ODP [11], the trader service is formally specified through an extension of basic LOTOS named E-LOTOS [12]. By comparing with our approach, the main and significant difference is the absence of software architecture principles and abstraction levels of specification in order to structure the trader specification. This fact makes very difficult to understand the whole specification. E-LOTOS may effectively be adopted in the future due to its improvements to LOTOS, but there still having a lack of tools to support the automatic verification of properties and refinement.

Bastide [3][4] adopts the Cooperative Objects (CO) formalism to specify middle-ware behaviour. CO is a dialect of object-structured, high-level Petri nets used to generate tests and verify inconsistencies of the OMG specification of CORBA Event service. In a similar way to E-LOTOS, the basic difference of our approach refers to the use of software architecture principles and abstraction levels to threat with the complexity of middleware system specifications. Another point that may be mentioned is the better readability of LOTOS specification compared to Petri nets.

Basin [2] focuses on the uses the Z notation to analyse the CORBA Security Ser-vice. Having the main objective of taking advantage of formalisation to make proofs of properties, this approach concentrates on defining a formal-data model to the CORBA Security Service. There is a significant difference to our approach that refers to the fact we addresses behaviour aspects instead data. Hence, despite being formal, the objects being formalised are completely different.

Fernandes [10] and Souza [23] also adopt Petri nets for describing middleware aspects. However, their focus are on the generation of formal models that include performance elements. The proposed models do not serve to check properties such as deadlock freedom or safety, but only quantitative and qualitative properties. Hence, in a similar way to the use of Z notation, this approach has not the focus on the behaviour itself.

Finally, our previous work [17] presented an approach for structuring the middleware architecture using software architecture principles. The middleware software architecture is defined at three different levels of abstractions, which are usually adopted by application developers, standard bodies and middleware developers. At the same time, we propose the adoption of the LOTOS language [7] for describing the behaviour of these software architectures. In fact, LOTOS is used as an ADL (Architecture Description Language) that allows to formally specify the behaviour of middleware software architectures.

6 Conclusion and Future Work

This paper has presented a framework useful to formalise middleware behaviour based on LOTOS. The framework consists of a set of common elements usually found in the development of middleware systems. The framework is now being defined, but it is possible to observe that a formalisation approach centred on the message request instead of middleware layer facilitates the treatment of middleware complexity: simple abstractions are highly reusable (see abstraction Channel in Section 3) and easier to find specification errors and verify desired behaviour properties; and the way of composing middleware abstractions considering the order they intercept the message request enormously facilitate the composition of middleware abstractions.

We are now extending the proposed set of abstractions including more sophisticated communication and concurrent elements. Meanwhile, it is also planned to include the specification of middleware services in such way that composition constraints may also consider middleware service composition.

References

[1] Basin, D., F. Rittinger, L. Viganò (2002): A Formal Analysis of the CORBA Security Service. Lecture Notes in Computer Science, 2272, pp. 330-349.

[2] Bastide, R., O. Sy, D. Navarre, P. Palanque (2000): A Formal Specification of the CORBA Event Service. In FMOODS – *Formal Methods for Open Object-Based Distributed Systems*, pp. 371-396.

[3] Bastide, R., P. Palanque, O. Sy, D. Navarre (2000): Formal Specification of CORBA Services: Experience and Lessons Learned. In OOPSLA – *Object-Oriented Programming, Systems, Languages, and Applications*, pp. 105-117.

[4] Bernstein, P. A. (1996): Middleware: A Model for Distributed System Services. Communications of the ACM, 39 (2), pp. 87-98.

[5] Blair, G., G. Coulson, R. Philippe, M. Papathomas (1998): An Architecture for Next Generation Middleware. In *Middleware*, pp. 191-206.

[6] Bolognesi, T., E. Brinksma (1987): Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems, 14(1), pp. 25-59.

[7] Campbell, A. T., G. Coulson, M. E. Kounavis, M. E. (1999) Managing Complexity: Middleware Explained. IT Professional, 1(5), pp. 22-28.

[8] Emmerich, W. (2000): Software Engineering and Middleware: A Roadmap. In *Second International Workshop on Software Engineering and Middleware*, pp. 119-129.

[9] Fernandes, S. F. L., W. J. Silva, M. J. C. Silva, N. S. Rosa, P. R. M. Maciel, D. F. Sadok. (2004): On the Generalised Stochastic Petri Net Modelling of Message-Oriented Middleware Systems. In *23rd IEEE International Performance, Computing, and Communications Conference*, pp. 783-788, 2004.

[10] ISO (1995): ISO 10476-1: Reference Model of Open Distributed Processing (Part I) – Overview.

[11] ISO (2001): ISO 15437: Enhancements to LOTOS (E-LOTOS).

[12] Kreuz, D. (1998): Formal Specification of CORBA Services Using Object-Z. In *Second IEEE International Conference on Formal Engineering Methods.*

[13] Matena, V., M. Hapner (1998): Enterprise JavaBeans. Sun

Microsystems.

[14] OMG (1998): CORBAservices: Common Object Services Specification.

[15] OMG (2002): Common Object Request Broker Architecture - Core Specification (CORBA 3.0)

[16] Plasil, F., S. Vinosky (2002): Behaviour. Protocols for Software Component. IEEE Transactions on Software Engineering, 28(11): 1056-1076.

[17] Rosa, Nelson S. and Paulo R. F. Cunha (2004): A Software Architecture-Based Approach for Formalising Middleware Behaviour, Electronic Notes in Theoretical Computer Science 108, pp. 39–51.

[18] Schmidt, Douglas and Buschmann, Frank (2003): Patterns, Frameworks, and Middleware: Their Synergistic Relationships, In *25th international conference on Software Engineering*, pp. 694-704.

[19] Souza, F. N., R. D. Arteiro, N. S. Rosa, P. R. M. Maciel (2006): Using Stochastic Petri Nets for Performance Modelling of Application Servers. *In Performance Modelling, Evaluation, and Optimisation of Parallel and Distributed Systems*, pp. 1-8.

[20] Sun Microsystems, Inc. (1999): JavaTM Transaction Service Specification. http://java.sun.com/products/jts/.

[21] Sun Microsystems, Inc. (2002): Java Message Service Specification. http://java.sun.com/products/jms/.

[22] Venkatasubramanian, N. (2002). Safe Composability of Middleware Services. Communications of the ACM, 45(6), pp. 49-52.

[23] Vinoski, S. (2002). Where is Middleware? IEEE Internet Computing, 6(2), pp. 83-85.