Adopting LOTOS and Software Architecture Principles for Formalising Middleware Behaviour

Nelson S. Rosa, Paulo R. F. Cunha

Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7851 50740-540 – Recife – PE – Brazil

{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 and makes difficult a better understanding of what is really described. In this paper, we adopt software architecture principles for structuring middleware together the LOTOS language for formalising their behaviour. The adoption of software architecture principles makes explicit structural aspects of the middleware. Meanwhile, the formalisation enables us to check behavioural properties of the middleware. In order to illustrate our approach, we present a LOTOS specification of the well-known object-oriented middleware CORBA and its transaction service.*

Resumo. *O número especificações abertas de sistemas de middleware e serviços de middleware tem crescido muito rapidamente. Apesar da complexidade, estes elementos são tradicionalmente descritos através de APIs (assinatura das operações) e de um texto em linguagem natural que descreve o comportamento destas operações. A ausência de formalismo normalmente leva a especificações ambíguas e difíceis de serem entendidas. Neste artigo, nós propomos o uso de arquitetura de software para estruturação das especificações e LOTOS para a formalização do comportamento do middleware. A adoção de elementos arquiteturais permite explicitar aspectos estruturais do middleware. Ao mesmo tempo, a formalização permite a verificação de propriedades comportamentais do middleware antes de sua implementação. Para ilustrarmos esta abordagem, nós apresentamos a arquitetura de software de CORBA e a formalização do seu comportamento em LOTOS.*

1. Introduction

The number of open specifications of middleware systems is rapidly increasing. Those specifications are usually implemented according to open standards such as DCE (Distributed Computing Environment) [Rosenberry 93], RM-ODP (Reference Model – Open Distributed Processing) [ISO 95], EJB (Enterprise Java Beans) [Matena 98] and CORBA (Common Object Request Broker Architecture) [OMG 02]. The open specifications of middleware services have also been popular through the JTS (Java Transaction Service) and JMS (Java Message Service) [Sun 02].

Middleware specifications are not trivial to be understood, as the middleware itself is usually very complex [Campbell 99]. Firstly, they 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 fourteen services. 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 [Venkatasubramanian 02] or how to satisfy new requirements of emerging applications [Blair 98].

The aforementioned specifications are usually described through APIs. Essentially, the service's operations signatures are described in IDL (Interface Definition Language) and the behaviour of each operation is described by informal prose. For example, the CORBA common object services (e.g., security, transaction) are described in IDL CORBA and informal text [OMG 98]. In practical terms, developers who want to implement those services have a hard task to produce a final product by interpreting what the specifications actually describe.

In this context, we present an approach for defining the software architecture of middleware systems. Meanwhile, we propose the adoption of LOTOS [ISO 01] for describing the behaviour of those software architectures. Initially, the middleware architecture is defined in terms of software architecture elements such as components and connectors. Next, the LOTOS language is used as an ADL (Architecture Description Language) [Medvidovic 00] in which the middleware behaviour is formalised. It is worth observing that we are not interested in any particular middleware product or middleware model.

On one hand, the adoption of software architecture principles is interesting as it treats with the system complexity by explicitly separating communication and computation aspects. Additionally, the software architecture enables us to have a better structural view of the middleware. On the other hand, the use of LOTOS allows the checking of particular behavioural properties of middleware systems, e.g., deadlock, livelock and execution sequences. Additionally, LOTOS enables us to automatically generate tests and check the behavioural equivalence (e.g., strong equivalence, branching equivalence, weak equivalence) between different middleware models and different middleware service compositions. For example, if one desires to replace a transactional middleware by a procedural middleware, it is possible to check if their behaviours are equivalent. Finally, a formal specification eliminates ambiguities in the middleware specification and provides a better understanding of what is actually described.

Formal description techniques have been used together middleware in the RM-ODP [ISO 95], in which the trader service is formally specified in E-LOTOS [ISO 01]. Most recently, the Z notation and High Level Petri Nests have been adopted for specifying CORBA services [Bastide 00], Naming service [Kreuz 98], Event service [Bastide 00] and Security service [Basin 02]. All those works, however, do not adopt software architecture principles for structuring the service descriptions. In terms of software architecture, a few ADLs include the possibility of describing behaviour

[Allen 97]. However, there are not tools that allow to check behaviour properties. Medvidovic [Medvidovic 02] has observed the convergence of middleware and software architecture principles. However, he does not adopt a formal approach. Finally, it is possible to note that the software architecture principles are widely adopted to build distributed applications (client and servers), but its benefits are rarely applied to middleware that connect them.

This paper is organised as following: Section 2 presents how the middleware architecture is defined in terms of software architecture elements. Next, Section 3 presents the use of LOTOS for describing the middleware software architecture. In Section 4, we adopt our approach for specifying CORBA. Finally, the last section presents the conclusions and some directions for future work.

2. Middleware Software Architecture

Prior to describe our approach on how to define middleware software architectures, we present the notion of middleware and middleware services.

The middleware layer is placed between the application and the operating system in order to hide the complexity of underlying network mechanisms [Bernestein 96, Vinoski 02]. This fact enormously facilitates the task of distributed application developers. For middleware developers, the middleware is viewed as a collection of distributed services (or middleware services) that takes the primary responsibility of communicating distributed applications. The middleware often also provides additional services such as security, transaction, naming and events, which "aggregate" value to the communication between distributed applications.

A middleware service is defined as a black box where it is known what is provided at the interface but not how it is actually implemented. The middleware service is specified by a set of interfaces (APIs) and protocols it supports. The APIs are usually defined through IDLs, whilst the service's behaviour is informally described by prose. In terms of implementation, a middleware service is distributed and includes entities (e.g., objects, components) that make up a client part, which supports the service's API running in the application's address space, and a server part that actually implements the service.

The definition of software architectures involves the use of three basic abstractions: components, connectors and configurations. A component is a unit of computation or a data store. Components represent a wide range of different elements, from a simple procedure to an entire application, and have an interface used to communicate the component with the external environment. A connector is an architectural building block used to model interactions among components and rules that govern those interactions. Some examples of connectors include client-server protocols, variables, buffers, sequence of procedure call and so on. A connector has an interface that contains interactions points between the connector and the component and other connectors attached to it. Finally, the configuration describes how components and connectors are wired together [Medvidovic 00].

Using the aforementioned basic elements, the middleware software architecture is defined according to the following principles:

- The middleware is viewed at three different levels of abstractions: a single connector that enables the interactions between distributed applications (Figure 1 (a)); the middleware abstract software architecture; and the middleware concrete software architecture. The first level is usually used/understood by application developers, the second one is typically adopted in the open specifications and the last one is required by middleware developers;
- The middleware abstract software architecture (see Figure 1 (b)) is defined from the services specifications (APIs), whilst the middleware concrete software architecture (see Figure 1 (c)) is a refinement of the abstract one;
- Each service provided by the middleware (e.g., security, event) defines a component in the abstract software architecture, except the communication service, which is modelled as a connector. Meanwhile, according to software architecture principles, a connector must always exist between any two components (service);
- The communication service, whatever the middleware model or product, is the only mandatory service. Thus, it is explicitly defined in the middleware software architecture (see Figure 1). Whether the middleware has additional services or not, it depends on the middleware specification;
- The services in the middleware concrete software architecture are defined through two parts, namely client (or sender) and server (or receiver) parts. Additionally, each service may be defined as a composition of fine-grained components. For example, the CORBA security service is made up of a principal authenticator and a component responsible for the cryptography. Both are accessible remotely;
- The underlying communication layers (e.g., transport and network layers) are defined as a connector in the software architecture; and
- The dashed line connector is a virtual connector [Medvidovic 02] that models protocols between the client/sender and server/receiver parts of the service. For example, the two-phase commit protocol commonly used in the transaction service.

By observing those guidelines, some points have to be taken in account. Firstly, since middleware systems do not perform any application-specific computation, they are naturally modelled as connectors. Secondly, the middleware provides further services in addition to the communication one. Hence, it may not be only considered as a simple connector. In the software architecture discipline, however, only components (no connectors) are traditionally decomposed into smaller elements. Finally, it is worth observing that the communication service enables other services (components) and applications to interact. Hence, it is also naturally differentiated and modelled as a connector.

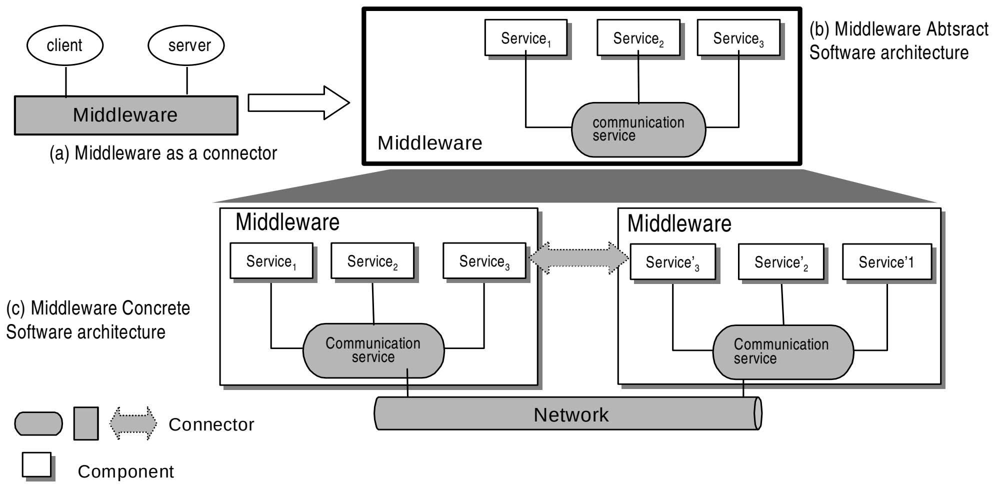


Figure 1. Different Views of the Middleware Software Architecture

Next, we present how those guidelines are followed in the definition of the middleware software architecture behaviour.

3. Middleware Software Architecture in LOTOS

A LOTOS 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 others 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 of 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.

In order to model the middleware software architecture in LOTOS, the basic architectural elements, namely components and connectors, are modelled through the basic LOTOS abstraction, namely process. The top-level specification defines the software architecture configuration. Additionally, the service specification consists of the temporal ordering of events executed at the service's interface. Each service specification S at the middleware software architecture is in monolithic style defined as $S = \Sigma\ a_i; A_i \mid i \in I$ for some finite index set I where each A_i is either process identifier or an expression in action prefix form.

3.1. Middleware as a Connector

As mentioned in Section 2, the simplest architectural view of the middleware is one that considers the middleware as a connector (Figure 1 (a)). In this case, the middleware is viewed as a black-box connector that has the role of the communication service.

The LOTOS specification at the top-level of Figure 1 (a) is a parallel composition (parallel operator `||`) of the client (`Client` component), the server

(`Server` component) and the Middleware (`Middleware` connector). The `Client` (4) communicates with the `Server` (8) through the connector `Middleware` as shown in the following:

```
(1) specification Middleware_Connector [invClt,terClt,invSrv,terSrv] : noexit
(2)   library OPER, RESULT, SERVICES, BOOLEAN endlib
(3) behaviour
(4)  Client [invClt, terClt]
(5)   |[invClt, terClt]|
(6)   Middleware [invClt, terClt, invSrv, terSrv]
(7)   |[invSrv, terSrv]|
(8)   Server [invSrv, terSrv]
(9)  where
(10)    …
(11) endspec
```

As mentioned before, the middleware behaviour is defined through the temporal ordering of invocation operations in middleware interface as follows:

```
(1)   process Middleware [invClt, terClt, invSrv, terSrv] : noexit :=
(2)      invClt ? s : SERVICE ? op : OPER;
(3)         invSrv ! s ! op;
(4)            terSrv ! s ? r : RESULT;
(5)               terClt ! s ! r;
(6)                  Middleware [invClt, terClt, invSrv, terSrv]
(7)   endproc
```

In this case, the middleware receives an invocation from the server (2) that contains both the name of the requested service and operation being requested (`invClt ? s : SERVICE ? op : OPER;`). The middleware passes both of them to the server (3) and wait for the reply (4). Finally, the middleware passes the reply containing the result to the client (5).

The behaviour of those components and connectors together is shown in the following trace obtained by simulation in the CADP Toolbox[1].

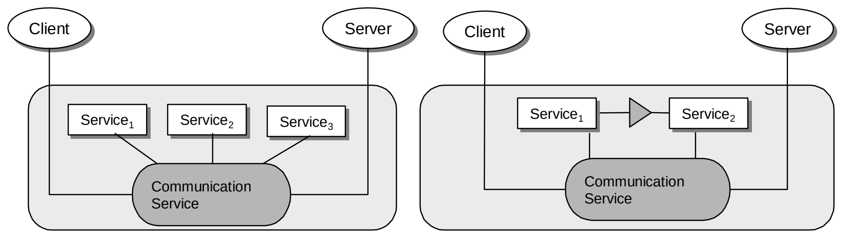
[1] Available at http://www.inrialpes.fr/vasy/cadp/.

```
(1) <initial state>
(2) "i" (BIND_TO_SERVER [16])       /* the client binds to the server */
(3) "INVCLT !`Service' !`op1'"    /* the client passes the request to the middleware */
(4) "INVSRV !`Service' !`op1'"    /* the middleware passes the request to the server */
(5) "i" (PROCESSOP1 [35])         /* the server processes the request */
(6) "i" (SA [24])                 /* the server updates it internal state */
(7) "TERSRV !`Service1' !`ok'"    /* server passes the reply to the middleware */
(8) "TERCLT !`Service1' !`ok'"  /* the middleware passes the reply to the client */
(9) <goal state>
```

Next, we present the middleware abstract software architecture.

3.2 Middleware Abstract Software Architecture

According to guidelines presented in Section 2, the middleware abstract software architecture is defined as a collection of services. Figure 2 (a)(b) depicts the components and connectors involved in a communication through the middleware abstract software architecture. It is worth observing that both the number of available middleware services and the way they may be composed depended on the particular middleware being considered.



(a) Usual middleware service composition (b) Alternative middleware service composition

Figure 2. Middleware Abstract Software Architecture

In order to specify the middleware abstract software architecture, we assume the configuration depicted in Figure 2 (a) that is composed by three components and a single connector. The LOTOS specification at the top-level of this software architecture is a parallel composition (operators `|||` and `||`) of the services (defined as components) provided by the middleware, namely `Service1`, `Service2`, `Service3` (3) and `CommunicationService` (7).

```
(1)  process Middleware [invClt, terClt, invSrv, terSrv] : noexit :=
(2)    hide inv, ter in
(3)     ((Service1 [inv, ter] ||| Service2 [inv, ter] ||| Service3 [inv, ter])
(4)            ||
(5)       ServiceOrdering [inv, ter])
(6)         |[inv, ter]|
(7)         CommunicationService [inv, ter, invClt, terClt, invSrv, terSrv]
(8)    where
(9)      …
```

```
(10) endproc
```

An important point of this specification is the ordering of composition of the middleware services (5). We adopt the LOTOS constraint-oriented specification style by defining the process `ServiceOrdering` that constrains the way the services are composed. As a consequence, this LOTOS process is not part of the software architecture itself, but a modelling element.

```
(1)      process ServiceOrdering [inv, ter] : noexit :=
(2)         inv ! Service1 ? op : OPER;
(3)            ter ! Service1 ? r : RESULT;
(4)               inv ! Service2 ? op : OPER;
(5)                  ter ! Service2 ? r : RESULT;
(6)                     inv ! Service3 ? op : OPER;
(7)                        ter ! Service3 ? r : RESULT;
(8)                           ServiceOrdering [inv, ter]
(9)      Endproc
```

In this particular case, according to the constraints imposed by `ServiceOrdering`, after the request gets in the middleware, it is passed to `Service1` (2-3) followed by `Service2` (4-5) and `Service3` (6-7). Then, the request is sent to `Server` where it is processed and sent back to `Client`. Following the RM-ODP [ISO 95] terminology, we call to invocation (`inv`) those actions to activate the service and termination (`ter`) to the action of return a result.

3.3. Middleware Concrete Software Architecture

The decomposition of the service component consist of explicitly define client and server parts. The client part is the service interface (remotely accessible), whilst the server part is the implementation of the service itself. As mentioned in Section 3.1, unlike other services, the communication service is a connector and it is not designed using this client/server approach. Hence, there is a part running in both sides of the architecture.

The LOTOS specification of the concrete software architecture at top-level is shown in the following:

```
(1)specification Middleware_Concrete [reqClt, repClt, reqSrv, repSrv] : noexit
(2)   library OPER, RESULT endlib
(3)behaviour
(4)   hide reqCN, repCN, reqSN, repSN in
(5)      (Client [reqClt, repClt]
(6)         |[reqClt, repClt]|
(7)       MiddlewareClient [reqClt, repClt, reqCN, repCN])
(8)             |[reqCN, repCN]|
(9)         Network [reqCN, repCN, reqSN, repSN]
(10)             |[reqSN, repSN]|
(11)      (MiddlewareServer [reqSrv, repSrv, reqSN, repSN]
             |[reqSrv, repSrv]|
             Server [reqSrv, repSrv])
     where
     …
     endspec
```

The middleware in the server (`MiddlewareServer`) and the middleware in the client (`MiddlewareClient`) have not the same behaviour. This is an interesting point to be observed as middleware products are different in both sides. This fact has a direct impact on how the middleware services are composed. Additionally, a service (or some of its components) may be present in the server and absent in the client. For example, the authentication component in the CORBA security service is not present in the client. Hence, the `ServiceOrdering` process and the set of services in both sides are different. For lack of space, we do not show the concrete middleware behaviour, but it is composed by 48 actions when the client request an operation followed by a reply.

4. Case study: CORBA

In order to illustrate our approach, we present how it may be applied to the middleware CORBA [OMG 03]. Three main reasons have motivated the adoption of CORBA: the number of middleware services available in CORBA is larger than any other middleware; the CORBA services are well detailed through API, which enables us a better understanding on how the service actually works; and CORBA has an explicit communication element (the ORB) that naturally acts as a connector.

Object-oriented middleware (OOM), such as RMI, EJB and CORBA, provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Two services are usually mandatory in the OOM, namely the naming service and the communication service. The naming service (known as "yellow pages") takes responsibility of registering business services provided by the servers. Clients that desire to make request to those services contact the naming service, which gives direction on how to find the server previously registered that provides the required service.

CORBA is a standard that has been widely adopted for building middleware products. According to the CORBA specification, in addition to the communication service known as ORB, fourteen distributed services should be provided by the middleware: persistence, externalisation, events, transactions, properties, concurrency, relationships, time, licensing, trader, query, collections, lifecycle and security [OMG 98]. All these services are not usually implemented in a single product, but at least the naming, life cycle and communication services are available in CORBA complaint products.

Figure 3 presents the CORBA software architecture at 3 different abstract levels, according to the guidelines presented in Section 2. Each CORBA Service, known as COS (Common Object Services), is modelled as a component in the CORBA software architecture. Additionally, the ORB (communication service) is defined as a connector.

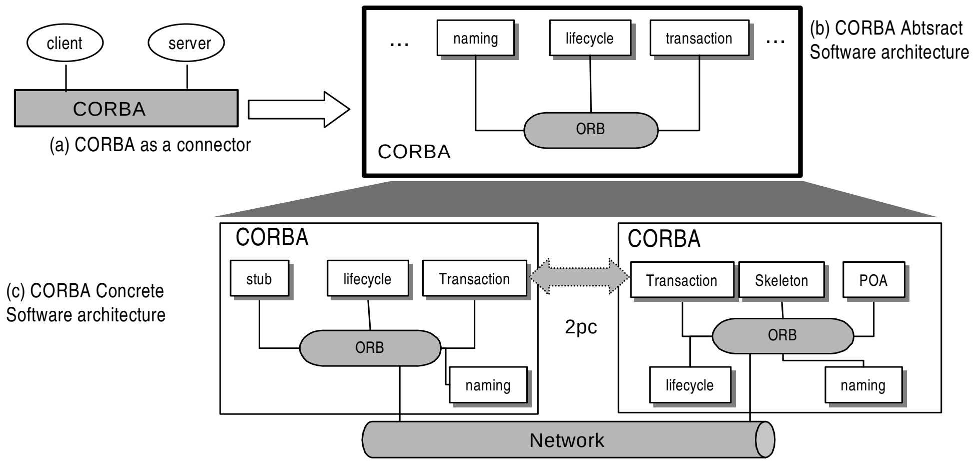


Figure 3. Object-oriented Middleware Model

Two points must be observed in the CORBA software architecture. Firstly, the CORBA standard defines that the COS services may be either inside or outside the ORB [OMG 03]. In this particular architecture, we adopt the second approach. Secondly, the stubs, skeletons and POA (Portable Object Adapter) have been incorporated by the ORB and are no explicit elements in abstract software architecture (application developers view). However, they are present in the concrete software architecture (middleware developers view).

After being defined the software architecture, next sections present how the principles described in Section 3.1, 3.2 and 3.3 are adopted to CORBA.

4.1. CORBA as a connector

The behaviour of the CORBA as a connector is very similar to one shown in Section 3.1. In this case, the CORBA receives a request from the server and sends it to client. After being processed, the reply is sent back to the client via the middleware. At this abstraction level, the software architecture does not present details on how this task is actually performed. The behaviour of the connector CORBA is specified as the temporal ordering of events executed in the CORBA interface. The CORBA interface is made up of several other interfaces such as dynamic invocation, stub, ORB, static skeleton, dynamic skeleton and POA interfaces.

In the following specification, the operations defined in each of the aforementioned interfaces are passed to the middleware through the event "`invClt ? s : Service ? op : OPER;`", where `s` is the name of service being request and `op` the operation.

```
process CORBA [invClt, terClt, invSrv, terSrv] : noexit :=
      invClt ? s : Service ? op : OPER;
         invSrv ! s ! op;
            terSrv ! s ? r : RESULT;
               terClt ! s ! r;
```

```
                  CORBA [invClt, terClt, invSrv, terSrv]
Endproc
```

Next, we present the CORBA abstract software architecture that provides a more detailed view of CORBA.

4.2. CORBA Abstract Software Architecture

The CORBA abstract software architecture is defined as a collection of services as mentioned before. The top specification is a parallel composition of fourteen different services (components) and the ORB (connector) as shown in the following:

```
process CORBA [invClt, terClt, invSrv, terSrv] : noexit :=
  hide inv, ter in
   (( Naming [inv, ter] ||| Event [inv, ter] |||
      Persistent [inv, ter] ||| LifeCycle [inv, ter] |||
      Concurrency [inv, ter] ||| Externalization [inv, ter] |||
      Relationship [inv, ter] ||| Transaction [inv, ter] |||
      Query [inv, ter] ||| Licensing [inv, ter] ||| Property [inv, ter] |||
      Time [inv, ter] ||| Security [inv, ter] ||| Trading [inv, ter] )
            ||
            ServiceOrdering [inv, ter] )
               |[inv, ter]|
           ORB [inv, ter, invClt, terClt, invSrv, terSrv] (0)
   where
      …
endspec
```

As defined in Section 3.2, the LOTOS process `ServiceOrdering` is not an architectural component, but it is defined in order to constrain the way the services interact. In this particular case, the most important ordering constraint is one related to the naming service. As widely known, every distributed service must be registered in the naming before be used by clients (2). Additionally, the client must obtain an interface reference to the service to use it (3).

```
(1) process ServiceOrdering [inv, ter] : noexit :=
(2)         inv ! COSnaming ! register;
(3)            ter ! COSnaming ? r : RESULT;
(4)               inv ! COSnaming ! lookup;
(5)                  ter ! COSnaming ? r : RESULT;
(6)                     ServiceOrdering [inv, ter]
(7)      endproc
```

Next, we show the trace generated by the simulation of all those elements together. This traces reveals the constraint imposed by `ServiceOrdering` as the client and server make requests to the naming service (2-11) before use the `Service1` provided by the server (12-18).

```
(1) <initial state>
(2)"INVSRV !`COSnaming' !`register'"
(3)"i" (INV [68])
(4)"i" (OPREGISTER [106])
(5)"i" (TER [68])
(6)"TERSRV !`COSnaming' !`ok'"
(7)"INVCLT !`COSnaming' !`lookup'"
(8)"i" (INV [68])
(9)"i" (OPLOOKUP [106])
(10)"i" (TER [68])
(11)"TERCLT !`COSnaming' !`ok'"
(12)"INVCLT !`Service1' !`op1'"
(13)"INVSRV !`Service1' !`op1'"
(14)"i" (SA [28])
(15)"i" (PROCESSOP1 [43])
(16)"TERSRV !`Service1' !`ok'"
(17)"TERCLT !`Service1' !`ok'"
(18)"INVCLT !`Service1' !`op1'"
(19)<goal state>
```

4.3. CORBA Concrete Software Architecture

According to the COS Transaction specification [OMG 98], the transaction service is specified through 6 interfaces, namely Current, TransactionFactory, Terminator, Coordinator, RecoveryCoordinator and Resource. These interfaces allow multiple, distributed objects to cooperate to provide atomicity, consistency, isolation and durability properties. Each interface is modelled by a component in the software architecture

For lack of space, we only present the concrete software architecture of the CORBA transaction service. The top specification is very similar to one shown in Section 3.3 that is a parallel composition of `ServiceInterface` and `StateProc`. The process `ServiceInterface` models the interfaces of the transaction service, which is made up of is 6 other interface as mentioned before: `Current`, `TransactionFactory`, `Terminator`, `Coordinator`, `RecoveryCoordinator` and `Resource.`

```
process Transaction_Service [inv, ter] : noexit :=
      hide sa in
         ServiceInterface [inv, ter, sa]
         |[sa]|
         StateProc [sa]
   where
      process ServiceInterface [inv, ter, sa] : noexit :=
         Current [inv, ter, sa] ||| TransactionFactory [inv, ter, sa]
         ||| Control [inv, ter, sa] ||| Terminator [inv, ter, sa]
         ||| Coordinator [inv, ter, sa] ||| RecoveryCoordinator [inv, ter, sa]
         ||| Resource [inv, ter, sa]
      where
        …
endproc
```

The whole specification of the transaction service has approximately 1200 lines and it is not completely terminated.

5. Conclusions and Future Work

This paper has illustrated how to adopt LOTOS to describe the behaviour of middleware software architectures. The specification has been structured according to software architecture principles, i.e., all middleware model elements are viewed as components, connectors and configuration. This approach facilitates the understanding of the general structures of different middleware specification as it separates computation and communication elements.

The adoption of LOTOS for describing the middleware enables us to check behaviour properties (we have used the CADP Toolbox) of each individual middleware and middleware service specification. This is not possible in the case an ADL is adopted instead LOTOS. We know that LOTOS has not been originally designed to be used like an ADL (e.g., ADLs have proper abstraction to model component and connectors), but its limitations are compensated by its powerful ability for describing behaviour.

The presented LOTOS specifications serve as basis for very interesting future work. We are now interested on the performance and reliability analysis of middleware models [Emmerich 00]. For this particular purpose, we are currently using the CADP Toolbox to generate Petri Nets. The Petri Nets specifications of the middleware models are more adequate to be analysed in terms of performance and reliability. Additionally, the proposed formalisation also opens a new track on how to compose middleware services, which is a basic task of adaptive middleware systems.

References

Allen, Robert J. (1997) “A Formal Approach to Software Architecture”, PhD Thesis, School of Computer science, Carnegie Mellon University.

Basin, David, Rittinger, Frank and Viganò, Luca (2002) “A Formal Analysis of the CORBA Security Service”, In: Lecture Notes in Computer Science, No. 2272, pp. 330-349.

Bastide, Rèmi, Palanque, Philippe, Sy, Ousmane and Navarre, David (2000) “Formal Specification of CORBA Services: Experience and Lessons Learned”, In: OOPSLA’00, p. 105-117.

Bastide, Rèmi, Sy, Ousmane, Navarre, David and Palanque, Philippe (2000) “A Formal Specification of the CORBA Event Service”, In: FMOODS’00, p. 371-396.

Bernstein, Philip A. (1996) “Middleware: A Model for Distributed System Services”, Communications of the ACM, Vol 39 (2), pp. 87-98, February.

Blair, Gordon, Coulson, G., Philippe, R. and Papathomas, M. (1998) “An Architecture for Next Generation Middleware”. In: Middleware’98, pp. 191-206.

Campbell, Andrew T., Coulson, Geoff and Kounavis, Michael E. (1999) “Managing Complexity: Middleware Explained”, IT Professional, IEEE Computer Society, Vol 1(5), pp. 22-28, October.

Emmerich, Wolfgang (2000) “Software Engineering and Middleware: A Roadmap”, Second International Workshop on Software Engineering and Middleware, Limerick. Ireland, pp. 119-129, June.

ISO 10476-1 (1995) “Reference Model of Open Distributed Processing (Part I) – Overview”, July.

ISO 15437 (2001) “Enhancements to LOTOS (E-LOTOS)”.

Kreuz, Detlef (1998) “Formal Specification of CORBA Services using Object-Z”, In: Second IEEE International Conference on Formal Engineering Methods, pp., December.

Matena, Vlada and Hapner, Mark, (1998) “Enterprise JavaBeans”, Sun Microsystems.

Medvidovic, Nenad (2002) On the Role of Middleware in Architecture-based Software Development. In: 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 299-306, 2002.

Medvidovic, Nenad and Taylor, Richard N. (2000) “A Classification and Comparison Framework for Software Architecture Description Languages”, IEEE Transactions on Software Engineering, Vol 26(1), pp. 70-93, January.

OMG (1998) “CORBAservices: Common Object Services Specification”, December.

OMG (2002) “Common Object Request Broker Architecture: Core Specification (CORBA 3.0)”, December.

Rosenberry, W., W. and Kenney, D. and Fisher, G., Understanding DCE, Ed.O'Reilly & Associates, 1993.

Sun Microsystems , Inc. (2002) “Java Message Service Specification”, http://java.sun.com/products/jms/, March.

Venkatasubramanian, Nalini (2002) “Safe Composability of Middleware Services”, Communications of the ACM, Vol 45(6), pp. 49-52, June.

Vinoski, Steve, (2002) “Where is Middleware?”, IEEE Internet Computing, Vol. 6(2), pp. 83-85.