Behavioural Specification of Middleware Systems

Nelson Souto Rosa & Paulo Roberto Freire Cunha

Universidade Federal de Pernambuco
Centro de Informática - 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, whilst making difficult a better understanding of what is actually described. In this paper, we adopt software architecture principles for structuring middleware specifications together with LOTOS 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..

Keywords: Middleware, LOTOS, Software Architecture, Formalisation.

1. Introduction

The number of open specifications of middleware systems [5][27] is rapidly increasing. Those specifications are usually implemented according to open standards such as DCE (Distributed Computing Environment) [21], RM-ODP (Reference Model – Open Distributed Processing) [11], EJB (Enterprise Java Beans) [14] and CORBA (Common Object Request Broker Architecture) [18][18]. The open specifications of middleware services have also been popular through the JTS (Java Transaction Service) [25] and JMS (Java Message Service) [24].

Middleware specifications are not trivial to be understood, as the middleware itself is usually very complex [8]. 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 includes fourteen services. Finally, in addition to hiding communication mechanisms, the middleware also has to hide failures, mobility, changes in network traffic conditions, and so on. From the point of view of application developers, they very often do not know how the middleware actually works. From the point of view of middleware developers, the complexity places many challenges that include how to integrate services in a single product [26] or how to satisfy new requirements of emerging applications [6].

The aforementioned specifications are usually described through APIs. Essentially, the service's operation signatures are described in CORBA 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 [19]. 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 structuring the middleware architecture using software

architecture principles [22]. 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) [16] that allows to formally specify the behaviour of middleware software architectures. It is worth observing that we are not interested in any particular middleware model [9] or middleware product.

On the one hand, the adoption of software architecture principles is interesting as it treats with the system complexity by separating communication and computation as-pects. Additionally, the software architecture enables us to have a structural view of the middleware. On the other hand, the use of LOTOS allows the checking (by using tools) of particular behavioural properties of middleware systems, e.g., deadlock freedom, liveness and safety. It also makes possible to check the behavioural equivalence either between the specifications of different middleware models or between two specifications during the refinement process. For the first case, if one desires to replace a transactional middleware with a procedural one, it is possible to check if their behaviours are equivalent. Furthermore, a formal specification eliminates ambiguities in the middleware specification and provides a better understanding of what is actually described. Finally, the formalisation creates the possibility of automatic generation of tests.

Formal description techniques have been used together middleware in the RM-ODP [11], in which the trader service is formally specified in E-LOTOS [12]. Most recently, the Z notation and High Level Petri Nests have been adopted for specifying CORBA services [3], Naming service [13], Event service [4] and Security service [2]. All those works, however, do not adopt software architecture principles for structuring the service descriptions. In terms of software architecture, a few ADLs like Wright (a CSP-based ADL) [1] include the possibility of describing the behaviour of the software architecture. However, there are not tools available for manipulating Wright specifications. Medvidovic [15] has observed the convergence of middleware and software architecture principles in an informal way. Finally, it is possible to note that the software architecture principles are widely adopted to build distributed applications, but its benefits are rarely applied to the middleware systems.

This paper is organised as following: Section 2 introduces basic concepts useful for understanding the rest of this paper. Section 3 presents how architectural elements (components, connectors and configuration) are defined in LOTOS. Next, Section 4 presents how architectural elements are put together to define middleware software architectures in LOTOS. Section 5 illustrates the proposed approach by specifying the software architecture of CORBA. Finally, the last section presents the conclusions and some directions for future work.

2. Basic Concepts

Prior to presenting the structure and the behavioural description of middleware software architectures, next sections introduce some basic concepts of software architecture and LOTOS. Additionally, we present the temporal logic used to express the temporal properties of LOTOS specifications.

2.1. Software Architecture

The definition of software architectures involves the use of three basic abstractions: components, connectors and configurations [22][16]. 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 calls and so on. A connector has an interface that contains interaction points between the connector and the component and other connectors attached to it. Finally, the configuration describes how components and connectors are wired together.

2.2. 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 other processes through externally observable actions. The unit of atomic interaction among processes is called an event. Events correspond to a syn-chronous 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 an 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', 'B_1', 'B_2' in the table stand for any behaviour expression, and "i" is a internal action.

Table 1: Syntax of behaviour expressions in basic LOTOS [7]

Name	Syntax	Semantics
inaction	stop	It cannot offer anything to the environment, nor it can perform internal actions.
action prefix - unobservable (internal) - 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 likeB_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 thateither 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 ofperforming the successful termination.
sequential composition (enabling)	B_1 >> B_2	B_2 is enabled only if and when B1 terminates successfully.
disabling	B_1 [> B_2	B_1 may or may not be interrupted by the first action of process B_2

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

```
(1)specification ClientServer
      [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 reinstantiate. 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 equivalence, observational equivalence and safety equivalence. All of them are checked through the CADP Toolbox .

2.3. Temporal Logic

A temporal property defined in this paper is expressed as a logic formula that is evaluated by a tool (the "evaluator" of the CADP Toolbox). The "evaluator" performs an on-the-fly verification of a property on a given LTS (Labelled Transition System) generated from the LOTOS specification. The temporal logic used to express the properties is called regular alternation-free mu-calculus and it is an extension of the alternation-free fragment of the modal mu-calculus with action predicates and regular expressions over action sequences.

The logic is built from three types of formulas: action formula (A), regular formula (R) and state formula (F). An action formula is a logical formula built from basic action predicates and Boolean connectives. A regular formula is a logical formula built from action formulas and traditional expression operators. A regular formula R denotes a sequence of (consecutive) LTS transitions such that the word obtained by concatenating their labels belongs to the regular language defined by R. Finally, a state formula is a logical formula built from Boolean, modal and fixed-point operators. The axiom of the grammar is the F formula. These formulas enable us to define some interesting temporal properties of LOTOS specifications such as safety, liveness and fairness.

- Safety Properties

A safety property informally expresses that “something bad never happens”, e.g., deadlock, a reply without a request. An example of safety property is defined in the following state formula:

[**true***.”OPEN !1".(**not** “CLOSE !1”)*.”OPEN !2"] **false**

In this formula, the looping operator “[ ]” has been used to define an axiom. According to the semantics of this operator, a state of the LTS satisfies “[“ R “]” F iff all transition sequences starting at the state and satisfying R are leading to states satisfying F. Hence, the formula states that every time process 1 enters in its critical section (action “OPEN !1”) , it is not possible that process 2 also enters its critical section (action “OPEN !2”) before process 1 has left its critical section (action “CLOSE !1”).

- Liveness Properties

Liveness properties informally express that “something good eventually happens”, e.g., the reachability on a sequence. An example of liveness property is defined in the following state formula:

<**true*** . “GET ! 0”> **true**

The looping operator “< >” used in this formula has the following semantics: a state of the LTS satisfies “<“ R “>” F iff there is (at least) one transition sequence starting at the state, satisfying R, and leading to a state satisfying F. Hence, the previous formula states that there exists a sequence leading to a “GET !0” action after performing zero or more transitions.

- Fairness Properties

Fairness properties are similar to liveness properties, except that they express reachability of actions by considering only fair execution sequences. A sequence is fair iff it does not infinitely often enable the reachability of a certain state without infinitely often reaching it. An example of a fairness property is shown in the follows:

[**true*** . “SEND” . (**not** “RECV”)*] <(**not** “RECV”) * . “RECV”> **true**

This formula expresses that after every message emission (action “SEND”), all fair execution sequences will lead to the reception of the message “action “RECV” after a finite number of steps.

3. Software Architecture in LOTOS

As mentioned in Section 1, the LOTOS language is adopted as an ADL. In order to model software architectures in LOTOS, the basic architectural elements, namely components, connectors and configuration (defined in Section 2.1) must be represented by LOTOS elements. In fact, LOTOS has not been designed to be an ADL and its only abstraction is the process (see Section 2.2).

Next sections present how the behaviour of architectural elements is specified in LOTOS.

3.1. Components

According to Medvidovic [16], an ADL may include the following elements in order to describe a component: the component’s interface (set of interaction points between the component and the external world), the component’s type (for the reuse of the component), the component’s semantics (the component’s behaviour), some con-straints (a property of or assertion about a part of the system), evolution (ability to specify modification of component’s properties) and non-functional properties (e.g., security and fault-tolerance).

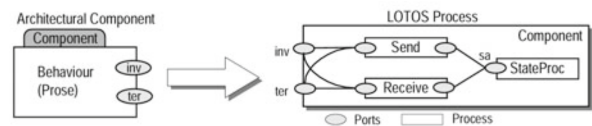

Figure 1: Structure of the component.

A component is modelled in LOTOS through the basic abstraction provided by the language, the process. Figure 1 shows informally how an architectural component is structured in LOTOS. The component’s interface is mapped to the set of ports of the LOTOS process (inv and ter), whilst the process behaviour refers to the component’s behaviour. The component behaviour is specified as a parallel composition of three LOTOS processes (3-5) as shown in the following:

```
(1) process Component [inv, ter]:
                noexit :=
(2)   hide sa in
(3)     (Send [inv, ter, sa]
          |||
        Receive [inv, ter, as])
(4)          |[sa]|
(5)     StateProc [sa]
(6)   where
(7)     process Send [inv, ter, sa]:
                noexit := … endproc
 (8)    process Receive[inv,ter,sa]:
                noexit := … endproc
 (9)    process StateProc [sa]:
                noexit := … endproc
 (10)endproc
```

The process `Send` (7) specifies the behaviour of the requests issued by the component (operations the component needs from the external world), whilst the process `Receive` (8) refers to the requests received by the component (operations the component provides to the external world). The process `StateProc` (9) represents the component state. The behaviour of the `Component` is a parallel composition of three processes, `Send`, `Receive`, and `StateProc`. In practical terms, the processes `Send` and `Receive` "execute" in parallel (parallel operator without synchronisation '|||'), but they have to syncronise with the process `StateProc` that mantains the component state.

It is worth observing that the ports `inv` (a short for "invocation") and `ter` (a short for "termination") are used for invocations from/to the component and for returning results to/from the component, respectively. For example, a client component in a client-server interaction makes requests through the port `inv` and waits for the reply in the port `ter`. Meanwhile, a server component receives the requests in the port `inv` and returns the result in the port `ter`.

3.2. Connectors

In a similar way to components, the connector specification includes the interface, types, semantics, constraints, evolution and non-functional properties. Despite the use of similar elements for describing the connector, the semantics of a connector is obviously different from the component. As mentioned before, the connector is responsible for defining how two (or more) components interact together. Figure2. depicts how an architectural connector is defined in LOTOS.

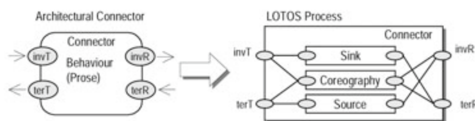

Figure 2: Structure of the connector

A connector is also modelled in LOTOS through the basic abstraction provided by the language, the process. The connector is made up with three processes, namely Source, Sink and Choreography, as follows:

```
(1)Process Connector[invT, terT,
           invR, terR] : noexit :=
(2)    (Source [invR, terT]
(3)         |||
(4)     Sink [invT, terR])
(5)        |[invT, terT, invR, terR]|
(6)     Choreography [invT, terT,
                      invR, terR]
(7)  where
(8)   process Source [invR, terT]:
                noexit := … endproc
 (9)   process Sink [invT, terR]:
                noexit := … endproc
 (10)  process Choreography [invT,
                             terT,
                             invR,
                             terR]:
                noexit :=
 (11)         …
 (12)  endproc
 (13)endproc
```

The process `Sink` (9) refers to the transport of messages the connector receives from the components plugged to it. The process `Source` (8) models the transport of messages that the connector has to send. The process `Choreography` (10-12) takes responsibility of ordering the messages the connector receives and sends, i.e., it coordinates the way the components plugged to the connector interact. In more practical terms, the process `Choreography` usually defines the communication protocol the connector implements.

3.3 Configuration

The architectural configuration consists of the composition of components and connectors together (see Figure3). The configuration is the top-level specification. It is made up of the composition of LOTOS processes, i.e., the components and connectors together (B1 || B2 || B3).

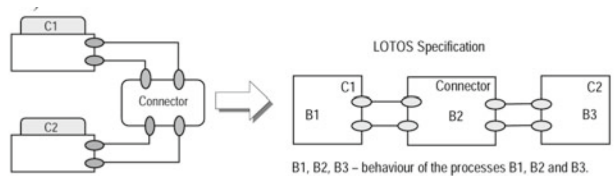

Figure 3: Structure of the configuration

A basic architectural rule must be followed to define the configuration: two components cannot be connected directly, i.e., there is a connector between any two components in the configuration. For example, a connector is necessary between the client and the server components in order to explicitly define how those elements interact.

Next, we present how those elements are used to specify middleware software architectures at three different abstraction levels.

4. Middleware Software Architecture in LOTOS

Since middleware systems neither perform any application-specific computation nor store data, they are naturally modelled as connectors. Unlike the usual connectors (see Section 2.1), the middleware provides services in addition to explicitly modeling the communication between components. In the software architecture discipline, however, only components (not connectors) are traditionally decomposed into smaller elements or provide services.

In order to define the middleware software architecture in LOTOS, we have adopted an approach in which the architecture is viewed at three different levels of abstractions: a simple connector that enables the interaction between distributed applications, a composite connector made up of services and a distributed composite con-nector. In fact, these levels represent refinement steps in the design of a middleware platform.

The middleware software architecture has been defined following some basic principles:

- Each service provided by the middleware (e.g., security, event, naming) defines a component in the composite connector. 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 communication service, whatever the middleware model or product, is the only mandatory service. Whether the middleware has additional services or not, it enormously depends on the middleware specification (or standard specification); and
- The services of the distributed composite connector are defined through two parts, namely client (or sender) and server (or receiver) parts. The un-derlying communication layers (e.g., transport and network layers) are also defined as a connector.

Next sections present how the middleware software architecture is defined in each abstraction level by adopting the abstractions defined in Section 3.

4.1. Simple Connector

The middleware as a simple connector is the highest abstraction view of the middleware. At this level, the middleware specification is commonly used/understood by application developers who are not interested in details of how the middleware actually works. In fact, the application developer views the middleware as a communica-tion element that transports messages between components.

Figure 4: Middleware as a simple connector

Figure4 shows the middleware as a simple connector (without services) that simply defines how the components C1 and C2 interact on the point of view of an external observer. In this particular case, the middleware receives an invocation from component C1 (1) that contains both the name of the requested service and the operation being requested. Next, the middleware passes both of them to C2 (2) and waits for the reply that comes from C2 (3). Finally, the middleware passes the reply containing the result to C1 (4).

The behaviour of those components and the middleware together is shown in the following trace obtained by simulation in the CADP Toolbox:

```
(1)<initial state>
(2) "INVT !`ServiceStd' !`op1'"
(3) "i" (SA [16])
(4) "INVR !`ServiceStd' !`op1'"
(5) "i" (IOP1 [62])
(6) "i" (SA [43])
(7) "TERR !`ServiceStd' !`ok'"
(8) "TERT !`ServiceStd' !`ok'"
(9) <goal state>
```

It is worth noting that at this level of abstraction, the description of the middleware behaviour is very simple/ abstract and it is not possible to know how a request is actually passed between C1 and C2. In practical terms, the behaviour of individual middleware products may not be differentiated (by an external observer) when the mid-dleware is viewed as a simple connector. The only exception occurs if two middleware systems have different communication models, e.g., OOM (Object-Oriented Middleware) and MOM (Message-oriented Middleware). For example, in the previous trace, the middleware transfer both the requests (2) from C1 to C2 and the replies from C2 to C1. However, in a MOM, the request (service and operations) is simply replaced by a message and the reply is not necessary.

4.2. Composite Connector

The middleware as a composite connector is typically adopted in open specifications such as CORBA, JMS and JTS. Unlike the application developer who views

the mid-dleware as simply as a communication element, in this case the middleware is viewed as a collection of services such as security, event, time, transaction and so on. Hence, the middleware "aggregates" value to the communication through these services.

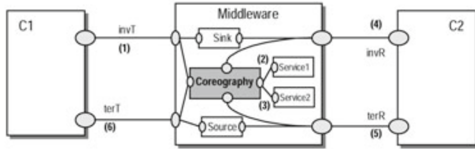

Figure 5: Middleware as a composite connector

In Figure5, the number of available services and the way they may be composed depend on the particular middleware being specified. For example, when a request gets in the middleware, it may firstly pass to the security service and then the transaction service before being forwarded to the remote component. Hence, an important point of this specification is the ordering of composition of the middleware services. For this particular purpose, we adopt the LOTOS constraint-oriented specification style in which the Choreography constrains both the component interactions and the way the services are composed.

The process Choreography of Figure5 is specified in the following:

```
(1) process Choreography[invT,
            terT, invR, terR,
            inv1, ter1, inv2, ter2]:
                     noexit :=
 (2)   invT ? s:SERVICE ? op : OPER;
 (3)   inv1 ! Service1 ! Op1S1;
 (4)   ter1 ! Service1 ? r : RESULT;
 (5)   inv2 ! Service2 ! Op1S2;
 (6)   ter2 ! Service2 ? r : RESULT;
 (7)   invR ! s ! op;
 (8)   terR ? s:SERVICE?r:RESULT;
 (9)   terT ! s ! r;
 (10)  Choreography[invT, terT,
                   invR, terR,
                   inv1, ter1,
                   inv2,ter2]
    Endproc
```

In this particular case, according to the constraints imposed by `Choreography`, after the request gets in the middleware (2), it is passed to `Service1` (3-4) followed by `Service2` (5-6). Then, the request is sent to `C2` (7) where it is processed and sent back to `C1` (8-9).

At this level of abstraction, the behaviour of distinct middleware systems still not differentiated by an external observer (observational equivalence), i.e., the application passes a request to the middleware and whatever the services found in the middleware, the application receives a reply without knowing the actual path taken by the request. However, a closer observation (strong equivalence) may reveal that the behaviours of two middleware systems that provide distinct services are not the same.

4.3. Distributed Composite Connector

Finally, the last view of middleware software architectures concerns to middleware developers, i.e., a detailed view of the middleware. The middleware is now defined as a distributed composite connector, which is decomposed into two parts, referred to as Middleware Transmitter and Middleware Receiver (see Figure6).

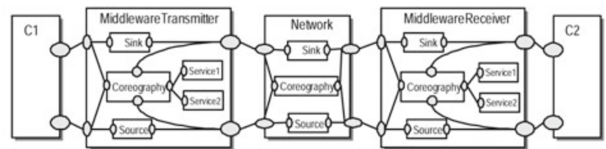

Figure 6: Middleware as a Distributed Composite Connector

The Middleware Transmitter receives requests from C1 and passes it to the Middleware Receiver through another connector (the network). The Middleware Receiver receives the request from the network, passes it to C2 and waits for the reply that must be sent to the C1 through the network and the Middleware Transmitter.

The top-level specification (1) of the software architecture presented in Figure6 is shown in the following:

```
(1) specification Configuration
                 [invT, terT,
                  invR, terR]:
             noexit
 (2) behaviour
 (3)   hide reqTN,repTN,
            reqRN,repRN in
 (4)       (
 (5)         C1 [invT, terT]
 (6)            |[invT, terT]|
 (7)         MiddlewareTransmitter
                  [invT, terT,
                   reqTN, repTN]
 (8)        )
 (9)           |[reqTN, repTN]|
 (10)        Network [reqTN, repTN,
                      reqRN, repRN]
 (11)          |[reqRN, repRN]|
 (12)      (
 (13)        MiddlewareReceiver
               [invR, terR,
                reqRN, repRN]
 (14)          |[invR, terR]|
 (15)         C2 [invR, terR]
 (16)       )
 (17) where
 (18) …
 (19) endspec
```

The middleware in the transmitter side, the process `MiddlewareTransmitter` (7) and the middleware in the receiver side, the process `MiddlewareReceiver` (14) 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. Hence, the process Choreography and the set of services in both sides may be different.

5. Case Study: CORBA

As widely known, CORBA is a standard that has been adopted for building middleware products. According to the CORBA specification, in addition to the communica-tion service (ORB), fourteen distributed services can be by the middleware: persistence, externalisation, events, transactions, properties, concurrency, relationships, time, licensing, trader, naming, query, collections, lifecycle and security [19]. All these services are not usually implemented in a single product, but some of them such as the naming, life cycle and communication services are usually available in any CORBA-complaint products.

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 [18]. 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 not explicit elements in the software architecture.

5.1. CORBA as a Simple Connector

The behaviour of CORBA as a simple connector is very similar to the one shown in Figure 4. At this level of abstraction, CORBA receives a request from the client and sends it to the server. After being processed, the reply is sent back to the client. The behaviour of the simple connector CORBA is specified as the temporal ordering of events executed in the CORBA interface. The CORBA interface is made up of the dynamic invocation, stub, ORB, static skeleton, dynamic skeleton and POA interfaces.

The specification of the choreography of CORBA that defines the way the invocations to CORBA are ordered is shown in the following:

```
process Choreography [invT, terT,
                      invR, terR]:
                     noexit :=
 invT ? s : SERVICE ? op : OPER;
 invR ! s ! op;
 terR ? s : SERVICE ? r : RESULT;
 terT ! s ! r;
 Choreography [invT, terT,
               invR, terR]
Endproc
```

Next section presents the CORBA as composite connector that provides a more detailed view of CORBA.

5.2. CORBA as a Composite Connector

CORBA as a composite connector is defined as a collection of services according to Section 4.2. Figure 7 (some services have been omitted in the figure for clarity) presents the CORBA software architecture, which is composed by COS services (components) and ORBCore (connector). The ORBCore is defined as a connector for three main reasons: it implements the communication service between applications, it is the communication channel between COS services and two or more components cannot communicate directly (architectural constraint presented in Section 2.1).

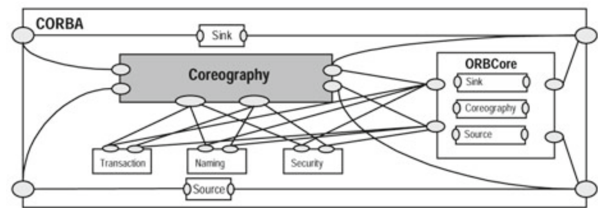

Figure 7: CORBA as a Composite Connector

The top-level specification is a parallel composition of fourteen different services (components), the process ORBCore, the process Choreography, the process Sink and the process Source:

```
process CORBA[invT, terT, invR,
              terR, inv, ter]:
               noexit :=
  (Source [invT, terT, invR, terR]
     |||
   Sink [invT, terT, invR, terR])
    |[invT, terT, invR, terR]|
   Choreography [invT, terT, invR,
                 terR, inv, ter]
       |[inv, ter]|
   ((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])
          |[inv, ter]|
     ORBCore [invT, terT,
              invR, terR, inv, ter])
where
  (* behaviour *)
endproc
```

As defined in Section 4.2, the LOTOS process Choreography takes responsibility for ordering the actions performed by the middleware and defining the way the services are composed. For example, an ordering constraint related to the naming service (referred to COSNaming) may define that every distributed service must be registered in the naming service before being used by clients. Additionally, clients must obtain an interface reference to the service in order to use it.

Next specification presents a possible choreography for the CORBA software architecture presented in Figure 7.

```
(1)Process Choreography[invT, terT,
           invR, terR, inv, ter]:
                noexit    :=
 (2)  invR ! COSnaming ! register;
 (3)  inv ! COSnaming ! register;
 (4)  ter ! COSnaming ? r : RESULT;
 (5)  terR ! COSnaming ! r;
 (6)  invT ! COSnaming ! lookup;
 (7)  inv ! COSnaming ! lookup;
 (8)  ter ! COSnaming ? r : RESULT;
 (9)  Loop [invT, terT, invR, terR]
 (10)where
 (11)  process Loop [invT, terT,
                    invR, terR,
                    inv, ter]:
               noexit :=
 (12)    invT?s:SERVICE?op:OPER;
 (13)    invR ! s ! op;
 (14)      terR?s:SERVICE?r:RESULT;
 (15)      terT ! s ! r;
 (16)     Loop[invT,terT,invR, terR]
 (17)    endproc
 (18) endproc
```

This Choreography defines that first possible action is a component asking for its own registration (operation “register” defined by the second parameter in invR ! COSnaming ! register;) with COSnaming (2). The request is passed to the COSnaming service (3) that performs a non-observable action and returns the result (4) to the Choreography and finally the result is sent back to the requester component (5). After the registration being ok, a component looks for the service just registered (6). Then, following the same steps of the operation register (6-8), the component is allowed to make as many requests as it desires to do. Finally, the Choreography enters in a loop that defines how two components interact to invoke an already registered service (12-15).

5.3. CORBA as a Distributed Composite Connector

The view of CORBA as a distributed composite connector is the most detailed one if compared to the views presented in Sections 5.1 and 5.2. Figure 8 presents the elements of the CORBA software architecture. For simplicity the service provided by the middleware have been grouped in the dashed box named Service. Additionally, the connectors Network and ORBCore (both in the client and server sides) have been simplified.

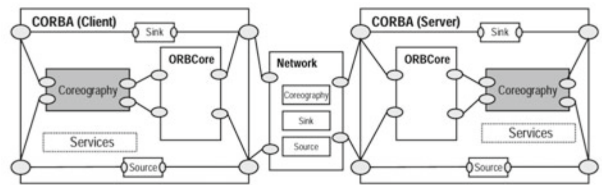

Figure 8: CORBA as a Distributed Composite Connector

The top-level specification of the CORBA distributed composite connector is shown in the following:

```
specification CS-CORBA[invT, terT,
                       invR, terR,
                       invS, terS,
                       invTN, terTN,
                       invRN,terRN]:
                    noexit
behaviour
   (Client [invT, terT]
     |[invT, terT]|
    CORBAClient[invT, terT, invTN,
                terTN, invS,terS])
     |[invTN, terTN]|
    Network [invTN, terTN,
             invRN, terRN]
     |[invRN, terRN]|
  (CORBAServer [invRN, terRN, invR,
               terR, invS, terS]
     |[invRN, terRN]|
   Server [invRN, terRN])
 where
 (* behaviour *)
endspec
```

Three parts compose this specification: the client side (the processes `Client` and `CORBAClient`), the Network (the process `Network`) and the server side (the processes `Server` and `CORBAServer`). A client's request gets in the CORBA through `CORBAClient`. The process `CORBAClient` receives the request passes it to the proper set of services and then sends it to `Network`. The process `Network` transports the request to `CORBAServer` which forwards the request to `Server`. The reply to `Client` takes a similar path in the reverse order.

5.4. Temporal Properties of CORBA Software Architectures

As mentioned before, the adoption of LOTOS as an ADL enables us to use tools both to check temporal properties and to compare CORBA software architecture

specifications. The checking is performed by defining properties in the temporal logic (see Section 2.3) and checking them against specifications. In relation to the second task, the behavioural equivalence of the CORBA software architectures presented in Sections 5.1, 5.2 and 5.3 have been verified. In particular, those specifications have been compared (in pairs) using a bisimulator and defining a relation of observational equivalence. In practical terms, those specifications are observationally equivalent, i.e., from the point of view of an external observer, the CORBA specifications are equivalent.

The temporal properties that have been defined for the CORBA software architectures are presented in the following:

- Deadlock freedom (safety property):

[**true***] <**true**> **true**

It states that every state has at least one successor.

- Safety property of the composite connector presented in Section 4.2: a middleware service terminates before any other service starts. This property is expressed as a state formula as follows

[**true*** . 'INV1 \(!.Service1.\) \(!.*\)' . (**not** 'TER1 \(!.Service1.\) \(!.*\)')* . 'INV2 \(!.Service2.\) \(!.*\)'] **false**

It states that every time Service1 starts (action 'INV1 \(!.Service1.\) \(!.*\)'), Service2 cannot start (action 'INV2 \(!.Service2.\) \(!.*\)') before Service1 terminates (action 'TER1 \(!.Service1.\) \(!.*\)').

In a similar way, the state formula in the following defines that if Service2 has started (action 'INV2 \(!.Service2.\) \(!.*\)'), Service1 cannot starts (action 'INV2 \(!.Service2.\) \(!.*\)') before Service2 terminates (action 'TER2 \(!.Service2.\'):

[**true*** . 'INV2 \(!.Service2.\) \(!.*\)' . (**not** 'TER2 \(!.Service2.\) \(!.*\)')* . 'INV1 \(!.Service1.\) \(!.*\)'] false

- Safety property of the composite connector: the composition of the services in the composite connector shown in Figure 5 defines that Service1 is performed before Service2. The state formula in the following expresses this property:

[(**not** 'INV1 \(!.Service1.\) \(!.*\)' **and true***) . 'INV2 \(!.Service2.\) \(!.*\)'] **false**

It states that it is not possible that Service1 starts (action 'INV1 \(!.Service1.\) \(!.*\)') after Service2 (action 'INV2 \(!.Service2.\) \(!.*\)').

- Fairness property of the distributed composite connector presented in Section 4.3:

[**true*** . 'INV1 \(!.Service1.\) \(!.*\)' . (**not** 'TER1 \(!.Service1.\) \(!.*\)')*] <(**not** 'TER1 \(!.Service1.\) \(!.*\)')* . 'TER1 \(!.Service1.\) \(!.*\)' > **true**

This formula expresses that after every invocation of Service1 (action 'INV1 \(!.Service1.\) \(!.*\)'), all fair execution sequences will lead to its termination (action 'TER1 \(!.Service1.\) \(!.*\)').

6. Related Work

We can identify two categories of researches in the literature that are mostly related to our paper. The first one concerns works that focus on formal aspects of ADLs. The second one encompasses all the works that have been done in the formalisation of middleware systems. It is worth observing that Medvidovic [15] has also observed the convergence of middleware and software architecture principles in an informal way. Formal aspects of ADLs include the use of formal ADLs such as Wright [1]. In terms of formal description of middleware systems, the basic idea is to use a formal description technique for specifying several aspects of middleware. In particular, formal description techniques such as E-LOTOS [12], Z notation [2], 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].

Wright [1] is an ADL that provides a formal basis for the description of architectural configurations and architectural styles. Those descriptions are carried out through a CSP-like that allows to specify the absctract behaviour of architectural components and connectors. Similarly to LOTOS, the formal basis of Wright provides a sound basis for reasoning about systems properties. Additionally, both of them address the formalisation of behavioural aspects of systems. As Wright has been specially designed to describe software architectures, it already includes component, connectors and configuration as first-class elements. As LOTOS has not these abstrac-tions, the only abstractions available to model component, connectors and configuration are the "process" and "specification". However, component (Send and Receive) and connector's (Source, Sink and Coreography) elements in LOTOS have been defined in such way that facilitates and respects the inherent differences between the roles (computation and communication, respectively) of theses elements. Finally, LOTOS is a standardised language and has a very good

tool support (absent in Wright) that enable us to easily check properties, to verify refinement, to generate tests and so on.

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 middleware behaviour. CO is a dialect of object-structured, high-level Petri nets used to generate tests and verify inconsistences 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 Secutity Service. 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 significative 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.

7. Conclusions and Future Work

This paper has illustrated how to adopt software architecture concepts together with the formal description technique LOTOS in order to describe the behaviour of middleware systems. The middleware itself is defined as a connector and its structure is defined through software architecture elements (component, connectors and configuration). Then, the middleware software architecture behaviour is described in LOTOS. Due to its complexity, the middleware software architecture is presented at three abstractions levels that represent traditional views of the middleware: middleware as a simple connector (application developer's view), middleware as a composite connector (standards' view) and middleware as a distributed composite connector (middleware developer's view). It is worth observing that this approach of point of views is close to the idea of RM-ODP viewpoints, i.e., each viewpoint captures a facet of design. However, our approach of "separation of concerns" only refers to architectural aspects. Hence, instead enterprise, technology, information, operational, and engineering aspects, it only covers architectural facets.

An approach based on software architecture concepts enables us to explicitly define the middleware internal structure and to separate computation and communication elements of middleware systems. The definition of the middleware software architecture facilitates the understanding of the complex structure of middleware systems as components and connectors have well-defined roles. Additionally, the software architecture serves as the basis for the implementation of middleware standards.

The adoption of LOTOS for describing the middleware software architecture behaviour enables us to check behaviour properties of each individual middleware systems and middleware services specifications. Furthermore, it makes it possible formally verify the behavioural equivalence of different middleware systems. This is not possible if conventional ADLs is adopted instead LOTOS. We know that LOTOS has not been originally designed to be used like an ADL (e.g., ADLs have proper abstractions to model component and connectors), but its limitations are compensated by its powerful ability for describing behaviour and the set of tools available.

The presented LOTOS specifications serve as a basis for very interesting future work. We are currently interested in the refinement of middleware specifications in which the refinement process follows the rules of the software architecture refinement. The proposed formalisation also opens a new track on how to compose middleware services, which is a basic task of adaptive middleware systems. Finally, LOTOS specifications can also be used to express and verify concurency models and real-time properties of middleware systems.

References

[1] R. J. Allen. A Formal Approach to Software Architecture. PhD Thesis, School of Computer science, Carnegie Mellon University, 1997.

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

[3] R. Bastide, P. Palanque, O. Sy, D. Navarre. Formal Specification of CORBA Services: Experience and Lessons Learned. In *Proceedings of OOPSLA*, pages 105-117, 2000.

[4] R. Bastide, O. Sy, D. Navarre, P. Palanque. A Formal Specification of the CORBA Event Service. In *Proceedings of FMOODS*, pages 371-396, 2000.

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

[6] G. Blair, G. Coulson, R. Philippe, M. Papathomas. An Architecture for Next Generation Middleware. *Proceedings of Middleware*, pages 191-206, 1998.

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

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

[9] W. Emmerich. Software Engineering and Middleware: A Roadmap. *Proceedings of Second International Workshop on Software Engineering and Middleware*, pages 119-129, Limerick, Ireland

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

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

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

[13] D. Kreuz. Formal Specification of CORBA Services Using Object-Z. *Proceedings of Second IEEE International Conference on Formal Engineering Methods*, 1998.

[14] V. Matena, M. Hapner. Enterprise JavaBeans. Sun Microsystems (1998)

[15] N. Medvidovic, E. Dashofy, R. Taylor. On the Role of Middleware in Architecturebased Software Development. *International Journal of Software Knowledge Engineering* , 13(4):367–393, 2003.

[16] N. Medvidovic, R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70-93, 2000.

[17] M. Moriconi, X. Qian, R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4): 356-372, 1995.

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

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

[20] F. Plasil, S. Visnovsky. Behavior Protocols for Software Component. *IEEE Transacitons on Software Engineering*, 28(11): 1056-1076, 2002.

[21] W. Rosenberry, D. Kenney, G. Fisher. Understanding DCE. Ed.O'Reilly & Associates, (1993)

[22] M. Shaw, D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Ed. Prentice Hall, (1996)

[23] F. N. Souza, R. D. Arteiro, N. S. Rosa, P. R. M. Maciel. Using Stochastic Petri Nets for Performance Modelling of Application Servers. *Proceedings of the Performance Modelling, Evaluation, and Optimisation of Parallel and Distributed Systems,* pages 1-8, 2006.

[24] Sun Microsystems Inc. Java Message Service Specification. http://java.sun.com/products/jms/, March (2002)X. B. Young. MyWWWPaperTile. http://www.complete.address, Jan. 1998

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

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

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