# Formal Modeling of a Generic Middleware to Ensure Invariant Properties*

Xavier Renault[1], Jérôme Hugues[2], and Fabrice Kordon[1]

[1] Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
xavier.renault@lip6.fr,fabrice.kordon@lip6.fr
[2] GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
jerome.hugues@enst.fr

**Abstract.** The complexity of middleware leads to complex Application Programming Interfaces (APIs) and semantics, supported by configurable components in the middleware. Those components are selected to provide the desired semantics. Yet, incorrect configuration can lead to faulty middleware executions, detected late in the development cycle.
We use formals methods to tackle this problem. They allow us to find appropriate composition of middleware components and the use of their APIs, and to detect valid or faulty sequences. To provide reusable results, we modeled a canonical middleware architecture using Z.
We propose a validation scenario to verify middleware's invariants. We define invariants to exhibit inconsistent usage of these APIs. The specification has been checked with the Z/EVES [13] theorem prover.

## 1 Introduction

Middleware is now a central piece of many applications. Therefore, expectations about middleware reliability increase. To meet this goal, their architecture is being revisited, cleaned to exhibit structuring patterns. For example, TAO [12] or Zen [10] take advantage of Design Patterns to provide a safer design. Based on this architecture, the middleware can be tuned to operate dedicated policies (tasking management, memory allocation, etc.).

Yet, these complex assemblies of design patterns has never been formally proved in an easy and efficient way. Thus, components interactions may lead to faulty configurations lately detected. A way to tackle this problem is to define and check invariants [5]. This is a way to express stability conditions on software components. For example, one can use OCL [9] to declare such invariants.

Should the middleware architecture be formally specified and its invariants formally expressed, one can define use case scenarios to verify the system. We aim to model middleware's components to ensure their composition. We specify invariants (*e.g.* array size of some internal structures, unicity of identifiers) for each component and check if they are verified for the selected assembly.

* This work is funded in part by the ANR/RNTL Flex-eWare project.

Z [13] is an algebraic specification language, based on the mathematical typed set theory. It has a schema notation, which allow one to specify structures in the system. It relies on a decidable type system, allowing one to automatically perform well-formedness checking (e.g. interface matching, resource usage).

In this paper we use Z to specify middleware services. We compose them as a middleware developer or user would do. Then we formally prove that query identifiers are consistent.

Section 2 sketches the use of formal methods for middleware. Then Section 3 presents the canonical middleware we selected. Section 4 details its modeling with Z, and section 5 shows how we prove important properties on the system.

## 2 Applying Formal Methods to Middleware

*Middleware* provides a set of mechanisms to support distribution functions. Its typical architecture is made of components, each of which supports a particular step in the processing of interactions. Interactions are supported by the exchange of messages between a *client* and a *server*, representing the caller and the callee.

Middleware's architecture is a set of components supporting the different steps of this interaction. The use of components (and their variations, implementing different configurations or policies) allows developers to tune or to select configurations/policies to configure and deploy a middleware that meet application requirements. A *middleware configuration* is defined as a set of components implementation selected to fulfill requirements.

So far, middleware are described through their components and the service they implement. Components are often described with semi-formal notation such as UML. These notations allow to express invariants (e.g. OCL in UML).

However, very few middleware specification is based on formal methods. This is a problem because formal specification of components and their related properties is needed to formally ensure that invariants are still valid for a given configuration (*e.g.* the selected implementation of components respect the middleware invariants).

**Related Works** There are two main approaches using formal methods in such context: dynamic and static.

*Dynamic Verification* deals with the system's behavior. It is the enumeration and analysis of all the states the system can reach during its execution. It is of interest to check if the system is deadlock or livelock free (model-checking). For example, in [11], LOTOS is used to build a formal framework to specify a middleware behavior based on its architecture [2]. Petri Nets [14] are also used to verify that PolyORB's core is both livelock and deadlock free [6].

*Static Verification* deals with structural aspects of a system and relies on predicate logic [5]. It is appropriate to analyse systems architectures, such as composition of interfaces from a typing theory point of view. It is also useful to check that invariants defined in the specification remains when components are composed. For example, in [1] the Z algebraic notation is used to verify the CORBA [4] object model: this study exhibits inconsistencies in the OMG CORBA Security Services. Similar work has been achieved on the CORBA Naming Service [7] or on parts of the POSIX real-time standard [3]. They both characterize potential problems in the use of components (such as logical naming or messages typing). Z is also used in [8] to specify the architecture of a cognitive application for redesign.

However there is no approach that really deals with middleware architecture issues. They only describe high-level services or behavior.

**Our Approach** We first analyse the architecture of a middleware to find relevant abstractions of the system. We also specify properties: 1) inside components, 2) at components interfaces, and 3) at the composition level.

The chosen architecture (presented in Section 3) is modular and versatile. It provides de facto a set of interesting abstractions for formal description. This work is complementary to the state of the art because we focus on the verification of properties for a given components assembly.

In an idealistic middleware development process, architecture verification appears after static verification (that deals with service specification) and before the dynamic verification (that ensures behavior of the system). It aims at building a middleware correct *by construction*. For example, we want to ensure that a configuration (i.e. a specific instantiation of selected components expressed by their formal specification) will not lead to a non-functionnal middleware (for instance one that cannot process requests).

## 3 Middleware Architectures

New architectures based on design patterns and genericity ease middleware adaptation by enhancing their configurability capabilities[12][10]. However their development process is not clearly specified and remains complex because it requires the implementation of most of the middleware functions.

We present in this section the characteristics of a specific middleware architecture we chose for our study.

**The *Schizophrenic* architecture** [15] is based on a Neutral Core Middleware (NCM), on which we can plug application-level and protocol-level personalities.

*The Neutral Core Middleware (NCM)* provides a set of canonical services on which we can build higher level services. The former are generic components for which a general implementation is provided. They are sufficient to describe various distribution models.

*Application personalities* are the adaptation layer between application components and middleware through a dedicated API or code generator. They register application components within the NCM; they interact with it to enable the exchange of requests between entities at the application-level.

*Protocol personalities* handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages transmitted through a communication channel.

We chose the Schizophrenic architecture because the core of the middleware fits to formally specify services and invariants in the system: main services are grouped in this core, easing the formal analysis of the middleware.

**PolyORB** implements a schizophrenic architecture presented in [15]. It demonstrates the concept is sound: it implements multiple personalities like CORBA, GIOP, Distributed Systems Annex of Ada, a Message Oriented Middleware derived from Java's JMS and SOAP. From the NCM, we identify seven steps in the processing of a request, each of which is defined as a fundamental service. Their associated level is the middleware is depicted in Figure 1.

We present these fundamental services and their specific roles: first, the client looks up server's reference using the *addressing* service (a). Then, it uses the *binding* factory (b) to establish a connection with the server, using one communication channels (*e.g.* sockets, protocol stack). Request parameters are mapped onto a representation suitable for transmission over network, using the *representation* service (c) (e.g. CORBA CDR). A *protocol* (d) is implemented for transmissions between the client and the server nodes, through the *transport* (e) service; it establishes a communication channel between the two nodes. Then the request is sent through the network and unmarshalled by the server. Upon the reception of a request, the middleware instance ensures that a concrete entity is available to execute the request, using the *activation* service (f). Finally, the *execution* service (g) assigns execution resources to process the request.

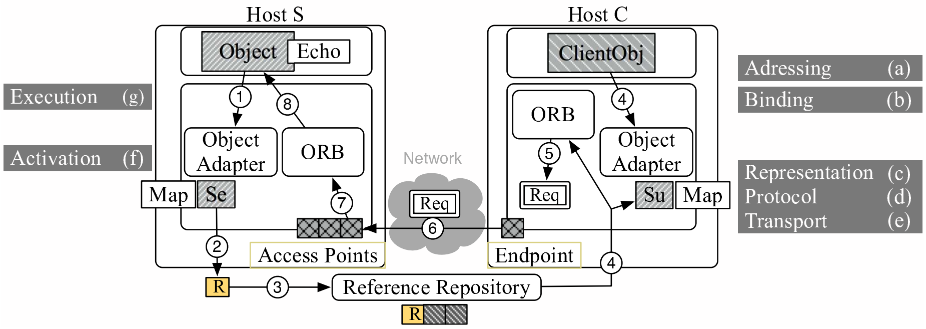

**Fig. 1.** Inside a schizophrenic middleware: PolyORB

Figure 1 also depicts the standard interaction between these fundamental services. It presents two hosts, each having the same underlying middleware: PolyORB. We present the main interactions between services from the export of a service from the server, to the sending of a request from the client. The object “Se” is called a *Servant* (from server side) or a Surrogate (object “Su” from client side). It belongs to the Binding service and is, for the client, the local interface of the remote service.

This object is created (1) when an object *Object*, which belongs to an application personnality, wants to provide a service. It is registered in a map managed by an *Object Adapter* (OA), which is associated to an *Object Request Broker* (ORB). In order to share this service, a *Reference* R is built (2) and shared (3) to the system. A Reference contains information to locate and identify a service in the network. When a client object wants to invoke this service, the client ORB should get the specific reference (4) and extract from it a Surrogate, managed by the client OA. Then, a *Request* is built (5) from this Reference, and send through the local *Endpoint* managed by the client ORB. It uses protocol chosen from the Reference. This protocol belongs to a protocol personnality. The Request is received (6) through an *Access Point*, managed by the server ORB. The Request is then analysed (7): if it is valid, the Activation service is involved (8), using the OA, to select the local object which provides the service. Finally, the Execution service allocates needed resources.

## 4 Modeling Middleware with Z

We present the modeling in Z of the *Neutral Core Middleware* and related services, as presented in the previous section.[3]

**The System** Several ORBs run into an heterogeneous system. At the specification level, the system is a collection of ORBs. Each ORB has to have unique name in this system. Furthermore, References on available services are shared through the system. The following Z schema specifies this system:

$$
\begin{array}{|l}
\hline
\mathit{ORB\_System} \\
\mathit{orbs} : \mathbb{F}\, \mathit{ORB} \\
\mathit{ref\_repository} : \mathbb{P}\, \mathit{Reference\_Info} \\
\hline
\#\mathit{orbs} \leq 1 \Rightarrow \\
\quad (\forall\, o1, o2 : \mathit{ORB} \\
\qquad \mid o1 \neq o2 \wedge \{o1, o2\} \subseteq \mathit{orbs} \\
\qquad \bullet\ o1.\mathit{ORB\_TSAP} \neq o2.\mathit{ORB\_TSAP}) \\
\hline
\end{array}
$$

A Z schema is divided into two parts (with an horizontal line as separator): the first one presents attributes names and type of a schema. The second presents invariants on these attributes. Schemas are a modular way to realize a specification, allowing one to split into small parts a system, and to compose them.

[3] An extended version of this work is available at `http://pagesperso-systeme.lip6.fr/Xavier.Renault/pub/research/techreport/ZPolyORB_08.pdf`

The invariant of the ORB_System schema states that if there is more than one ORB in the system, they must have different names (ORB_TSAP, see the definition of the ORB schema below).

The orbs set is theoretically infinite, but to specify the invariant related to names (which involves the cardinality), it is finite ($\mathbb{F}$ symbol). The ref_repository attribute is a infinite set of References. Each reference is unique.

**Neutral Core Middleware Components** We present the main components of the NCM: the ORB and its associated Object Adapter.

*The ORB* manages several resources in the system: resources allocation, task scheduling, event priorities and request handlers. On one side, the ORB has to serve requests to applications entities, using an "Object Adapter". On the other side, it has to send and receive requests from other nodes, through Transport Access Points (TAP). Each TAP is bound to a specific protocol. Furthermore, the ORB is uniquely identified over the network. Hence the following Z schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{ORB}\ \rule{22em}{0.4pt}} \\
\mathit{ORB\_TSAP} : \mathit{TSAPType} \\
\mathit{Request\_Queue} : \mathrm{seq}\, \mathit{Request} \\
\mathit{Transport\_Access\_Points} : \mathit{Transport\_Access\_Point} \text{ ⇸ } \mathit{ProtocolStack} \\
\mathit{RootPOA} : \mathit{Object\_Adapter} \\
\hline
\end{array}$$

The ORB_TSAP attribute is the network identifier of each ORB.

Each ORB has a request queue, which is in our specification a sequence of Request. This allow us to keep the order of incoming requests, since it is an intrinsic properties of sequences in the Z notation.

The Transport_Access_Points attribute is the set of TAPs, each bounded to a specific ProtocolStack. It is a partial function of TAPs to ProtocolStack.

*The Object Adapter* (the RootPOA attribute) manages all objects which export services (named Servant), provided by the application. These servants have a unique identifier on their host. The Object_Adapter (OA) manages the mapping between these servants and their identifiers. It affects a unique identifier to each registered servant:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{Object\_Adapter}\ \rule{20em}{0.4pt}} \\
\mathit{Objects\_Map} : \mathit{ServantType} \rightarrow \mathit{Ident} \\
\hline
\forall\, s : \mathrm{dom}\, \mathit{Objects\_Map} \\
\qquad \bullet\ \mathit{Objects\_Map}(s) \neq \mathit{NULLId} \\
\hline
\end{array}$$

In this Object_Adapter schema, Objects_Map models the binding between some servant to a unique identifiant. This is a total function between sets ServantType and Ident (symbolized by →).

The invariant part of this schema states that it is not allowed to have entries in the map which have a NULLId (which means the Servant may not have been initialized for example).

*A Reference* is used to identify an application entity within the global system, as CORBA's IOR. It is a finite collection of Profiles. A profile carries the identifier of the target host in the system, and the identifier of the service within this host. It defines available protocols to contact the remote target, it is a subset of the protocols managed by both the client and the server.

$$\begin{array}{l} \textit{Reference\_Info} \\ \hline \textit{Profiles} : \mathbb{F}(\textit{Profile}) \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{NULLRefSet} : \mathbb{P}\ \textit{Reference\_Info} \\ \hline \textit{NULLRefSet} = \\ \{ r : \textit{Reference\_Info} \\ \qquad \mid r.\textit{Profiles} = \emptyset \} \end{array}$$

$$\begin{array}{l} \textit{Profile} \\ \hline \textit{TSAPName} : \textit{TSAPType} \\ \textit{ObjectId} : \textit{Ident} \\ \textit{Continuation} : \textit{ProtocolStack} \\ \hline \textit{ObjectId} \neq \textit{NULLId} \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{NULLProfileSet} : \mathbb{P}\ \textit{Profile} \\ \hline \textit{NULLProfileSet} = \\ \{ p : \textit{Profile} \mid \\ p.\textit{TSAPName} = \textit{NULLTSAP} \vee \\ p.\textit{Continuation} = \textit{NULLPStack} \} \end{array}$$

A Profile could not be bound to a servant whose identifiant is NULL. For specification purpose, we define the set of all null references (which have an empty set of Profiles) and the set of null profiles.

A Reference is built by a server host: a client has read-only access to this kind of resource: by construction, a Reference refers only to one service, and lists the different ways to reach it.

**Neutral Core Middleware Services** To provide services as described in Section 3, different functions have to be defined in the middleware internals. We present them in the following order: first, from the server side, we define operations to share a service in the system; second, the functions used by a client to get information and send a request to the remote server; third, we present functions used by the server host to handle an incoming request.

From a server application, there are three mains steps to share and provide services in the network:
– The middleware exports the servant (Export procedure);
– The middleware produces a reference for this servant (Create_Reference);
– The middleware notifies other nodes about the availability of the service to the system, using a naming service for example.

**Exporting a service (Server Side)** The Export procedure registers Servants in the Object Adapter (OA). This procedure should respect the fact that Servants are not managed twice in the OA map. Given a Servant, it creates a new entry in the map with an available identifier. This procedure fails if the Servant is already managed by the OA.

$$
\begin{array}{|l}
\hline
\mathit{Export\_OK} \\
\Delta \mathit{ORB\_System} \\
O? : \mathit{ORB} \\
\mathit{Obj}? : \mathit{ServantType} \\
\mathit{Oid}! : \mathit{Ident} \\
\hline
O? \notin \mathit{NULLORBSet} \\
\wedge\ O? \in \mathit{orbs} \\
\wedge\ \mathit{Obj}? \neq \mathit{NULLServant} \\
\wedge\ \mathit{Obj}? \notin \operatorname{dom} O?.\mathit{RootPOA}.\mathit{Objects\_Map} \\
\mathit{Oid}! \notin \operatorname{ran} O?.\mathit{RootPOA}.\mathit{Objects\_Map} \wedge \mathit{Oid}! \neq \mathit{NULLId} \\
\\
\mathit{orbs}' = (\mathit{orbs} \setminus \{O?\}) \cup \{\theta \mathit{ORB}[ \\
\quad \mathit{ORB\_TSAP} := O?.\mathit{ORB\_TSAP}, \\
\quad \mathit{RootPOA} := \theta \mathit{Object\_Adapter}[ \\
\qquad \mathit{Objects\_Map} := \{\mathit{Obj}? \mapsto \mathit{Oid}!\} \oplus O?.\mathit{RootPOA}.\mathit{Objects\_Map}], \\
\quad \mathit{Transport\_Access\_Points} := O?.\mathit{Transport\_Access\_Points}, \\
\quad \mathit{Request\_Queue} := O?.\mathit{Request\_Queue}]\} \\
\hline
\end{array}
$$

The $\Delta$ symbol indicates that Export_OK changes the state of ORB_System. Modified attributes of ORB_System are decorated with a single quote, as orbs'.

A Z schema can also be parametrized: input variables are decorated with a '?' symbol; output variables are decorated with the '!' symbol.

The $\theta$ symbol indicate an instantiation of a Z schema with specific values, affected by the ':=' symbol. Here, the Export_OK operator removes the input ORB variable from the system and inserts a new one with modified attributes. The intrinsic type of a total function $X \rightarrow Y$ is a set of pairs $\mathbb{P}(X \times Y)$. The $\oplus$ symbol adds a pair $(x, y)$ into the set if there is no already existing pair as $(x, z)$. In the later case, it overrides the old pair.

Notice the name of the presented operator ends with the "_OK" suffix. In order to avoid undefined predicates in the specification, each operator schema is divided into two schemas: one specifying preconditions to a successfull application of the operator (name ending with "_OK"), and the other specifying the dual preconditions of the first schema (name ending with "_ERR"). The final operator is build as a combination of these two schemas, and is then called *robust*. All operators in our specification are robust, but for sake of clarity we only present the "_OK" part of each of them.

**Creating a Reference (Server Side)** As previously defined, a Reference contains all information to contact and identify a service in the system. It carries information on the different protocols available to contact the remote node. These protocol are bound to a particular Transport Access Point. To create a reference, we need to build all information for each TAP. We need to set the local identifier of the service, and the global identifier of the host in the system. Figure 4 presents the related operator schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Create\_Reference\_OK}} \\
\hline
O? : \mathit{ORB} \\
\mathit{Oid}? : \mathit{Ident} \\
\mathit{PSet}! : \mathit{Reference\_Info} \\
\hline
O? \notin \mathit{NULLORBSet} \\
\mathit{Oid}? \neq \mathit{NULLId} \\
\land\ \mathit{Oid}? \in \operatorname{ran} O?.\mathit{RootPOA}.\mathit{Objects\_Map} \\
\mathit{PSet}!.\mathit{Profiles} = \{ \\
\qquad \mathit{TAP} : \operatorname{dom} O?.\mathit{Transport\_Access\_Points} \\
\qquad\qquad \bullet\ \theta \mathit{Profile}[ \\
\qquad\qquad\qquad \mathit{TSAPName} := O?.\mathit{ORB\_TSAP}, \\
\qquad\qquad\qquad \mathit{ObjectId} := \mathit{Oid}?, \\
\qquad\qquad\qquad \mathit{Continuation} := O?.\mathit{Transport\_Access\_Points}(\mathit{TAP})]\} \\
\hline
\end{array}$$

**Fig. 2.** The Create Reference operator

Since PSet!.Profiles is a set of profile, we can use the Z notation to build a *set comprehension*: the pattern is $\{x : T \mid C \bullet P\}$, where x is of type T. For each x under the condition C, then the predicate P holds. Here, a Profile set is built as follow: for each TAP in the input ORB's TAP, an instantiation of a Profile schema is made with specific value.

This operator fails if no Servant is bound to the input local identifier.

**Sharing a Service (Server Side)** Sharing a service is to notify the system that a new service is available.

Broadcast_Reference_Ok adds the input reference to the system's references repository. Each ORB in the system may then access to this repository to retrieve specific references.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Broadcast\_Reference\_OK}} \\
\hline
\Delta \mathit{ORB\_System} \\
\mathit{PSet}? : \mathit{Reference\_Info} \\
\hline
\mathit{ref\_repository}' = \mathit{ref\_repository} \cup \{\mathit{PSet}?\} \\
\hline
\end{array}$$

**Sending a request (Client Side)** We present operations that a client has to do in order to send a request to a remote server. The client should 1) get a

Reference on the targeted service; 2) build the Request; 3) select the appropriate Profile and associated Protocol; 4) send its request to the remote host.

The Reference may be get using the reference repository, or with alternative mecanisms such as IORs.

*The Request* is the structure containing information to invoke the remote service. It is sent through the network, and carries a Reference on the targeted server object and the identifier for this server service.

It is a simplified view of a request, which may contain more parameters and additional QoS information.

$$\begin{array}{|l}
\textit{Request} \\
\hline
\textit{Target} : \textit{Reference\_Info} \\
\textit{Operation} : \textit{OperationIDType} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Create\_Request\_OK} \\
\hline
\textit{Target}? : \textit{Reference\_Info} \\
\textit{Operation}? : \textit{OperationIDType} \\
\textit{Req}! : \textit{Request} \\
\hline
\textit{Req}! = \theta \textit{Request}[ \\
\qquad \textit{Target} := \textit{Target}?, \\
\qquad \textit{Operation} := \textit{Operation}?] \\
\hline
\end{array}$$

The Create_Request_OK operator acts as a request factory: given a reference and a service identifiant, it builds the appropriate request.

*Request Send* operation first selects the appropriate profile (and the associated protocol) to contact the remote host; then it sends this request using the selected profile, adding the request to the targeted ORB queue. The send operator fails if the Request is malformed.

$$\begin{array}{|l}
\textit{Send\_Request\_OK} \\
\hline
\Delta \textit{ORB\_System} \\
\textit{Req}? : \textit{Request} \\
P? : \textit{Profile} \\
\hline
\textit{Req}? \notin \textit{NULLReqSet} \\
\exists\, \textit{var} : \textit{ORB} \\
\mid \textit{var} \in \textit{orbs} \\
\wedge\ \textit{var}.\textit{ORB\_TSAP} = P?.\textit{TSAPName} \\
\bullet\ \textit{orbs}' = (\textit{orbs} \setminus \{\textit{var}\}) \cup \{\theta \textit{ORB}[ \\
\textit{ORB\_TSAP} := \textit{var}.\textit{ORB\_TSAP}, \\
\textit{RootPOA} := \textit{var}.\textit{RootPOA}, \\
\textit{Transport\_Access\_Points} := \\
\qquad \textit{var}.\textit{Transport\_Access\_Points}, \\
\textit{Request\_Queue} := \\
\qquad \textit{var}.\textit{Request\_Queue} \frown \langle \textit{Req}? \rangle]\} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{Select\_Profile\_OK} \\
\hline
\textit{Req}? : \textit{Request} \\
P! : \textit{Profile} \\
\hline
\textit{Req}? \notin \textit{NULLReqSet} \\
\textit{Req}?.\textit{Target}.\textit{Profiles} \neq \emptyset \\
\exists\, p : \textit{Profile} \\
\qquad \mid p \in \textit{Req}?.\textit{Target}.\textit{Profiles} \\
\qquad \bullet\ P! = p \\
\hline
\end{array}$$

The Select_Profile_OK operator specifies a simple selection algorithm: it randomly picks one Profile among available ones.

In the Send_Request_OK operator, an incoming request is added to the targeted ORB queue (using sequence concatenation operator $\frown$).

**Receive and process requests (Server Side)** When receiving a request, the ORB checks, using the Activation service, that the request is valid and the targeted object is managed by its Object Adapter. Then, the Execution Service allocates resources for request execution.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Process\_OK}} \\
\Delta \mathit{ORB\_System} \\
O? : \mathit{ORB} \\
\mathit{FOid}! : \mathit{Ident} \\
\mathit{var} : \mathit{Request} \\
\hline
O? \notin \mathit{NULLORBSet} \\
\land\ O? \in \mathit{orbs} \\
\land\ O?.\mathit{Request\_Queue} \neq \langle\,\rangle \\
\mathit{var} = \mathit{head}\ O?.\mathit{Request\_Queue} \\
\exists\, p : \mathit{Profile} \\
\qquad \mid p \in \mathit{var}.\mathit{Target}.\mathit{Profiles} \\
\qquad \land\ p.\mathit{ObjectId} \in \operatorname{ran} O?.\mathit{RootPOA}.\mathit{Objects\_Map} \\
\qquad \bullet\ \mathit{FOid}! = p.\mathit{ObjectId} \\
\\
\mathit{orbs}' = (\mathit{orbs} \setminus \{O?\}) \cup \{\theta \mathit{ORB}[ \\
\qquad \mathit{ORB\_TSAP} := O?.\mathit{ORB\_TSAP}, \\
\qquad \mathit{RootPOA} := O?.\mathit{RootPOA}, \\
\qquad \mathit{Transport\_Access\_Points} := O?.\mathit{Transport\_Access\_Points}, \\
\qquad \mathit{Request\_Queue} := \mathit{tail}\ O?.\mathit{Request\_Queue}]\} \\
\hline
\end{array}
$$

This operator dequeues an incoming request (using sequence operator like “head” and “tail”, which respectively return the first element of a sequence and all the sequence but the first one).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Find\_Servant\_OK}} \\
\mathit{OA}? : \mathit{Object\_Adapter} \\
\mathit{id}? : \mathit{Ident} \\
s! : \mathit{ServantType} \\
\hline
\mathit{OA}? \notin \mathit{NULLOASet} \\
\land\ \mathit{id}? \neq \mathit{NULLId} \\
\land\ \mathit{id}? \in \operatorname{ran} \mathit{OA}?.\mathit{Objects\_Map} \\
\exists\, \mathit{ss} : \mathit{ServantType} \\
\qquad \mid \mathit{ss} \in \operatorname{dom} \mathit{OA}?.\mathit{Objects\_Map} \\
\qquad \land\ \mathit{OA}?.\mathit{Objects\_Map}(\mathit{ss}) = \mathit{id}? \\
\qquad \bullet\ s! = \mathit{ss} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Find\_Servant\_ERR}} \\
\mathit{OA}? : \mathit{Object\_Adapter} \\
\mathit{id}? : \mathit{Ident} \\
s! : \mathit{ServantType} \\
E! : \mathit{Exception} \\
\hline
\mathit{OA}? \in \mathit{NULLOASet} \\
\lor\ \mathit{id}? = \mathit{NULLId} \\
\lor\ \mathit{id}? \notin \operatorname{ran} \mathit{OA}?.\mathit{Objects\_Map} \\
s! = \mathit{NULLServant} \\
E! = \mathit{FAILURE} \\
\hline
\end{array}
$$

The Find_Servant operator checks if the input identifier is related to a managed servant, and activates it. We have shown for the later operator the robust schema, including its "_OK" and "_ERR" parts. We present the robust Find_Servant operator (a combination of the two prevous ones):

$$\mathit{Find\_Servant} \mathrel{\widehat=} (\mathit{Find\_Servant\_OK} \land \mathit{Success}) \lor \mathit{Find\_Servant\_ERR}$$

## 5 Verifying Invariants in the Middleware Specification

We have defined the main components of a middleware. For the sake of place, this section presents only one system configuration, where only one client host and one server host are set. We present their associated initialization schema and then a scenario where the client host sends a oneway request to the server host. Such a request is sent with the "best-effort" semantic, for which the client host does not wait for any answer. In this scenario, we expose that some properties hold in the system, ensuring its consistency.

Our specification is checked using the Z/EVES theorem prover. Z/EVES is an interactive system for composing and analysing Z specifications. It helps the modeler to prove theorems on the specification, but for complex specification it only provides guidelines for proving: the modeler has to finish the proof and guide Z/EVES to get a "true" or "false" result.

**System Instantiation** Both the client and the server hosts have an ORB. We present the initialization of one ORB, the server one. Since it is a transaction oriented architecture, server and client are identified only when one sends a request to the other. We introduce the global identifier of an ORB:

$$\begin{array}{|l} \mathit{SHost} : \mathit{TSAPType} \end{array}$$

The definition of an Object Adapter sets initially its map to an empty set (no servant managed).

$$\begin{array}{|l} \mathit{S\_OA} : \mathit{Object\_Adapter} \\ \hline \mathit{S\_OA.Objects\_Map} = \emptyset \end{array}$$

We define, for this case study, only one Transport Access Point and its associated Protocol Stack for each ORB:

$$\begin{array}{|l} \mathit{S\_TAP} : \mathit{Transport\_Access\_Point} \\ \mathit{S\_Protocol} : \mathit{ProtocolStack} \end{array}$$

We now instantiate an ORB, setting its name, Object Adapter, TAPs and Request_Queue:

$$
\begin{array}{|l}
S\_ORB : ORB \\
\hline
S\_ORB = \theta ORB[ \\
\quad ORB\_TSAP := SHost, \\
\quad Request\_Queue := \langle\,\rangle, \\
\quad RootPOA := S\_OA, \\
\quad Transport\_Access\_Points := \{S\_TAP \mapsto S\_Protocol\}]
\end{array}
$$

The previous initialization process holds for all hosts in the network. Once these hosts are set, we can initialize the whole environment: initializing the set of ORBs in the system and the references repository:

$$
\begin{array}{|l}
\hline
InitEnvironment \\
ORB\_System' \\
\hline
orbs' = \{S\_ORB\} \cup \{C\_ORB\} \\
ref\_repository' = \emptyset \\
\hline
\end{array}
$$

The system contains two running ORBs, with no request pending at the initialization time. Figure 5 introduces a Servant, named "Object", which will be managed by the server host for a transaction, and its associated service "Echo":

$$
\begin{array}{|l}
Object : ServantType
\end{array}
\qquad\qquad
\begin{array}{|l}
echo : OperationIDType
\end{array}
$$

**Fig. 3.** Z model of a Servant providing the Echo Service

**Validation scenario and associated proofs** We present a use-case scenario that corresponds to a typical activation of services. In this scenario, a server hosts an object which provides a service. The server application registers the object with its local middleware, creates a reference and shares it. A client host gets this reference, builds a request, and sends it to the remote entity. Finally, the server middleware handles the request.

As presented before, all operations that a server has to do in order to export a service are sequentialized as follow in a new schema Server_Op: it has to export the Servant to the Object Adapter, to create a reference on this Servant and to notify the system of the new service availability:

$$
\begin{array}{l}
Server\_OP \mathrel{\widehat=} \\
\quad Export[O? := S\_ORB, Obj? := Object] \\
\quad \gg Create\_Reference[O? := S\_ORB] \\
\quad \gg Broadcast\_Reference\_OK
\end{array}
$$

The $\gg$ symbol indicates operations are chained, where the output of one is the input of the other. These operations must be robust, to avoid undefined state.

In order to emit a request to the server host, a client should extract the remote object's reference, create a request targeting this object, adding request payload and select a profile to contact the remote host and finally send the request:

$$\begin{array}{l}
Client\_OP \mathrel{\widehat=} \\
\qquad GetReference \\
\qquad \gg Create\_Request[Target?, Operation? := echo] \\
\qquad \gg Select\_Profile \\
\qquad \gg Send\_Request
\end{array}$$

The GetReference schema is an operator defined in the scope of the case study. It is defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ GetReference\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi ORB\_System \\
Target! : Reference\_Info \\
\hline
Target! \in ref\_repository \\
\hline
\end{array}$$

This schema is pretty simple since in our case study there is only one reference in the repository.

We restrict our case study to Oneway Request: for these later, the client does not wait for an answer. In this scenario, the server exports a service, the client sends a request, and the server checks this request before processing it.

$$ONEWAY \mathrel{\widehat=} InitEnvironment \fatsemi Server\_OP \fatsemi Client\_OP \fatsemi Process$$

The $\fatsemi$ symbol expresses a call sequence of operators.

In our scenario, we want to ensure that the request sent by the client contains the same id as the one exported by the server at the beginning.
To make this verification, we define the following test schema:

$$ONEWAYTest \mathrel{\widehat=} ONEWAY \gg Find\_Servant[OA? := S\_ORB.RootPOA, Foid?/id?]$$

The Process operator is invoked with the id sent with the process, and it returns the id of the activated object; we pipe it with the Find_Servant operator to check if it the same as Object . To verify this property, we express the theorem shown in Figure 4

| | |
|---|---|
| **theorem** tOneWayReliable | ⟨... Z/EVES output ...⟩ |
| $ONEWAYTest \Rightarrow s! = Object$ | Proving gives ... |
| | true |

**Fig. 4.** Theorem: a Oneway Request is reliable

*Analysis* For sake of clarity and readability, we do not present the whole interaction with Z/EVES. To achieve this proof, we needed to prove intermediates theorems, such as precondition reachability of each operators, schemas consistency and domain checking. We needed to set rules to help Z/EVES to finish the proof: typing related rules, transformation rules (predicate equivalence, etc.). Each rule has been proved in order to be used.

This global proof ensures that for this call sequence, invariants specified within each schema hold: names of ORBs are unique, no uninitialized objects are managed by Object_Adapters. Furthermore, preconditions for each operators are reachable and allow to produce valid postconditions as specified. These postconditions are checked and ensure that this combination of operators will lead to the seeked goal: identifiants consistency through a OneWay Request Process.

## 6 Conclusion and future work

In this paper, we presented the use of Z as formal notation to specify the architecture of a canonical middleware, based on a schizophrenic middleware architecture. This allows us to build abstraction of the middleware components, and to express properties and invariants upon each component of the system.

To elaborate the Z specification, we choose a well-structured architecture that relies on a canonical middleware core that concentrate all important services. Since these services are well-specified, it is possible to formally express them in Z. Moreover the execution path of these services is also well-identified by use-case scenarios that can serve as a basis for verification.

Once the canonical middleware core specified in Z, we have identified typical invariants for each components. These invariants are used to ensure that a given component configuration will not lead to inconsistencies in the middleware.

We experimented a well-identified use-case scenario on this architecture, and show its validity. Doing so with all use-case, we proved that our canonical architecture is consistent by construction.

One can enrich this specification, and add new contraints and invariants both deduced from a given implementation's caracteristics. Thus, our Z specification can serve as a framework to verify several variations based on our canonical middleware core.

The next step of our work is to express more invariants for each components, and to enrich the model with more details. We aim to analyse the impact of various QoS strategies on the middleware invariants. These QoS strategies will be expressed in Z to be bound to our current specification for analysis purpose. Categories of cases study will be defined and improved.

## References

1. D. Basin, F. Rittinger, and L .Viganò. A Formal Analysis of the CORBA Security Service. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB*, volume 2272 of *Lecture Notes in Computer Science*, pages 330–349. Springer, 2002.

2. T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.
3. L. Freitas. Posix 1003.21 standard – real time distributed systems communication (in Z/Eves). Technical report, University of York, 2006.
4. Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group, April 2006.
5. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 26(1):53–56, 1983.
6. J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Linz, Austria, September 2004.
7. D. Kreuz. Formal specification of corba services using object-z. In *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, page 180, Washington, DC, USA, 1998. IEEE Computer Society.
8. B. Milnes, G. Pelton, R. Doorenbos, M. Laird, P. Rosenbloom, and A. Newell. A specification of the soar cognitive architecture in z. Technical report, Pittsburgh, PA, USA, 1992.
9. OMG. *OCL 2.0 Specification - Version 2.0 ptc/2005-06-06*. OMG, June 2005.
10. K. Raman, Y. Zhang, M. Panahi, J. Colmenares, R. Klefstad, and T. Harmon. Rtzen: Highly predictable, real-time java middleware for distributed and embedded systems. 2005.
11. N. Rosa and P. Cunha. A formal framework for middleware behavioural specification. *SIGSOFT Softw. Eng. Notes*, 32(2):1–7, 2007.
12. D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 10 April 1998.
13. J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
14. R. Valk. *Basic definitions*, chapter 4, pages 41–51. Springer Verlag, Petri nets and system engineering (Claude Girault and Rudiger Valk Eds), first edition, 2003.
15. T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, volume LNCS 3063, pages 106 – 119, Palma de Mallorca, Spain, Jun 2004. Springer Verlag.