Refining middleware functions for verification purpose

Jérôme Hugues, Laurent Pautet
{hugues, pautet}@enst.fr
École Nationale Supérieure
des Télécommunications
CS & Networks Department
46, rue Barrault
F-75634 Paris CEDEX 13, France

Fabrice Kordon
Fabrice.Kordon@lip6.fr
Laboratoire d'Informatique de Paris 6/SRC
Université Pierre & Marie Curie
4, place Jussieu
F-75252 Paris CEDEX 05, France

Abstract

The development of real-time, dependable or scalable distributed applications requires specific middleware that enables the formal verification of domain-specific properties. So far, typical middleware implementations do not directly address these issues. They focus on patterns and frameworks to meet application-specific requirements.

Patterns propose a high-level methodology adapted to the description of software components. However, their semantics does not clearly address verification of static or run-time properties. Such issues can be addressed by other formalisms, at the cost of a more refined description.

In this paper, we present our current effort to combine both patterns and Petri Nets to refine and then to verify middleware. Our contribution details steps to build Petri Net models from the Broker architectural pattern. This provides a model of middleware and is a first step towards formal middleware verification.

1 Issues in middleware development

Distribution middleware provides description methods, services and guidelines to ease the development of distributed applications. Middleware specifications describe the semantics and runtime supports for distribution.

Successful implementations of solutions such as CORBA, Java Message Service (JMS) or SOAP demonstrate that distributed applications require very different distribution models: Remote Procedure Call (RPC), Distributed Objects Computing (DOC), Message Passing (MP) or Distributed Shared Memory (DSM).

Besides, there is a rising demand for a wider range of runtime and platform support: embedded, mobile, real-time, multimedia, etc. These new criteria increase complexity in both middleware development and use. Middleware implementations should be versatile enough to handle different (and potentially antagonist) platform requirements; application must abide to complex middleware semantics.

Current middleware implementations rely on *patterns* to enable configurability and then to meet user requirements for one specific distribution model. Architectural and design patterns are introduced to describe specific solution to recurrent design problems (request demultiplexing, buffers allocations, concurrent execution, etc). Middleware is described by means of a language pattern that weaves together a set of related patterns. This approach proved its efficiency in various industrial projects [1]. Hence, the combination of patterns provides a high-level description of middleware. Yet, weak pattern descriptions may lead to slightly different implementations or implementations that interleave different patterns concerns. This impedes implementations verification.

Moreover, patterns are only descriptive. They do not provide any verification guidelines. Thus, implementations rely only on simple verification methods to verify behavioral-only properties: the use of some middleware functions and the execution of predefined test cases. But this approach lacks generality: it can only test a restricted subset of the infrastructure properties.

As middleware use evolves toward real-time and dependable applications, there is a strong need for formal verification of middleware with respect to explicitly defined properties. Yet, verification process is a complex task. The choice of a verification mechanisms is thus significant.

In the remainder of this paper, we detail our current effort on middleware verification. We focus on remote invocation models (RPC, DOC or MP) and exclude DSM. We present a middleware typical architecture, built around the *Broker* architectural pattern and show limits that prevent verification. Then we introduce our work to fill this gap and detail the

modeling notations and process we use to verify the *Broker*. We also detail our specific middleware architecture, implemented by PolyORB[1] and demonstrate how separation of concerns eases verification. Then we explain how we refine the *Broker* and detail how to formally verify it.

2 Broker: a key middleware pattern

Basically, middleware provides mechanisms to enable transparent interaction between application nodes using messages. Reception of a message triggers: *1)* resources allocation, *2)* specific processing for the distribution model and then *3)* execution of application-specific code. A response may be sent back to the message initiator, following the same path. The architectural pattern *Broker* provides a view of the components involved in this process.

We first present this pattern; we show its importance for middleware specifications, performance and runtime; then we discuss limitations when coming to verification.

2.1 Overview of the Broker

The *Broker* architectural pattern provides a synthetic description of the role of middleware [2]: *" [..] (to) structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions."*.

The *Broker* pattern prescribes the use of many different objects: proxy, client and server, repository, bridge. These objects cooperate in the following way: *Servers* register themselves with the broker through the *Repository*, and make their services available to *Clients* through method interfaces; *Clients* access servers by sending requests via the *Broker*. *Broker* exchanges requests between nodes by locating the appropriate server, forwarding the request to it and transmitting results back to the client. *Proxy* and *Bridge* handle communication mechanisms and enable data exchange across heterogeneous platform.

We can note that the *Broker* specification presented above provides a complete view of an architecture to achieve remote service invocation. It covers all functions involved in middleware execution: protocol stack, data representation for transmission through network, resource allocation, etc. Hence, its precise definition and study will provide a first analysis of a middleware architecture.

2.2 Broker within middleware architectures

The *Broker* pattern has a key role in middleware architecture and implementations. Moreover, similar patterns [2] propose simpler views of the *Broker* adapted to specific cases. Variations of this pattern are used by middleware norms or specifications: e.g. for CORBA or Microsoft .NET specifications. Some implementations detail variations that support different distribution models or enable precise tuning of middleware performance:

- The Advanced Communication Toolkit (ACT) [3] provides a flexible implementation of the *Broker* pattern. It allows a precise description of resource allocation policies for multi-threading or data marshalling. ACT shows the *Broker* may serve as a basis to implement various distributions models. It supports CORBA (DOC) and cBus (Message Oriented Middleware, MOM).

- The ACE ORB (TAO) [4] demonstrates how the *Broker* pattern can be extended and then adapted to several concurrent executions policies. TAO proposes multiple patterns to control concurrent execution of *Broker* instances. A performance analysis revealed these different patterns enable great flexibility in configuration and good performance.

Hence, the *Broker* pattern is used under multiple forms at the core of most middleware architectures. Its precise definition and analysis would provide significant information about resource use, execution flow, middleware faults and performance analysis; but also to detect incorrect design.

So far, middleware implementations rely on slightly different behavioral descriptions of the *Broker*. This pattern definition is not sufficiently detailed and may lead to many fine variations introduced by implementation choices. Moreover, this pattern interleaves multiple functions: protocol, resource allocation, etc. Such a description impedes verification: it covers many complex functions. Thus, the *Broker* architectural pattern provides a specification of middleware architecture not suitable for formal verification.

However, patterns provide an elegant way to describe a component. We now present how we extract a precise specification of middleware components from the *Broker* architectural pattern that is suitable for verification.

3 Analysis guidelines

A complete analysis of the *Broker* pattern requires first a clear description of the component interface, related semantics and expected properties. Ultimately, this description is expressed using a notation that enables formal verification. In between, several transformations may be required to go from high-level unformal specification to strongly formalized description of a component.

This raises the question of the most adequate modeling notation (or set of notations) to achieve this process. We

[1] `http://libre.act-europe.fr/polyorb`

contemplate using one or more models among automata, UML diagrams (and derived stereotypes), Petri nets (colored, stochastic, etc) and architecture description languages (ADL). These are the most used notations for modeling software components.

We can note that none of these notation is supported by a complete specification and verification cycle. Each notation only covers a restricted part of the software life cycle: UML diagrams focus on system specifications and modeling; Petri nets on formal verification of controlled systems; ADL on the description of system architectures.

Thus, one has to use two or more notation to cover both specification and verification. Current research activities focus on the combination of different description models to provide a complete description of a component :

- For instance one can derive Petri Nets from UML state machines and diagrams [5] to achieve verification, yet there is no fully automated tool to complete this task.
- Another possibility is to rely on domain specific notations such as the Avionics Architectural Description Language (AADL) [6] or LfP [7]. They detail how new notations can be defined by extending and combining multiple models. Yet, they are still at an early definition stage and not fully supported by tools.

These studies provide guidelines for the formal specification and verification of components. They follow a top/down approach from high level specifications to formal one, enabling verification.

We propose to follow a similar approach, adapted to a very specific problem: verifying the *Broker* pattern. We present the different steps in the following sections. First we propose a middleware architecture that eases verification; then we refine the *Broker* pattern and define it with respect to our middleware architecture. Finally, we present the formal model of the *Broker* we produced using Petri Nets.

4 Middleware architecture for verification

In this section, we present how specific middleware architectures enable formal verification. We introduce our proposal, the *schizophrenic* architecture, and our implementation: PolyORB.

4.1 Rationale

We stated in section 2.2 that the interleaving of many high-level functions is a major limitation for the verification of middleware architectures based on the *Broker* pattern. To solve this problem we have to propose a comprehensive definition and then separation of middleware functions.

Generic middleware proposes such a separation: they assert that middleware implementations have similar design. Hence, distribution models may be built from a set of generic elements using a functionality-oriented approach. Then, these elements are instantiated to conform to a specific distribution model.

Several projects demonstrate how middleware functionalities can be described by a set of generic services, independent from any distribution model. They propose a set of abstract interfaces. Distribution models are implemented by combining the concrete modules that implement these interfaces and provide access to generic middleware services.

- Quarterware [8] is generic middleware from which CORBA, RMI and MPI instances have been produced. These models have been implemented using a restricted set of components that can be extended to implement a specific model; or specialized for optimization and high-performance.
- Jonathan [9] architecture emphasizes on instances as adaptations of the core system Jonathan. Jonathan is a framework of configurable components and abstract interfaces. Dedicated instantiations provides CORBA (David), Java RMI (Jeremie) distribution models, or specialized ones for multimedia.

These different projects provide incomplete solutions for verification. They enable the implementation of distribution models as instances or *personalities* of a generic set of components. But personalities implementation interleaves instantiated components: this impedes verification.

Thus, we have to clearly separate middleware functions at both the definition and implementation levels. We now present our solution: *schizophrenic middleware*.

4.2 Schizophrenic middleware

Schizophrenic middleware refines the definition and role of personalities to increase separation of concerns. It introduces application level, protocol level personalities and a Neutral Core Middleware. The latter allows for interaction between personalities. Figure 1 presents interaction capabilities between personalities available in our implementation of schizophrenic middleware: PolyORB.

Application personalities constitute the adaptation layer between application components and middleware through a dedicated API or code generator. They register application components with the core middleware; and 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 exchanged through a chosen

communication network and protocol. Requests can be received either from application entities (through an application personality and the neutral core) or from another node of the distributed application. They can also be received from another protocol personality: in this case the application node acts as a proxy performing protocol translation between third-party nodes.

The *Neutral Core Middleware* acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides the necessary abstractions to transparently pass requests between protocol and application personalities in a neutral way. It is completely independent from both application and protocol personalities: this enables the selection and interaction of any combination of application and protocol personalities.

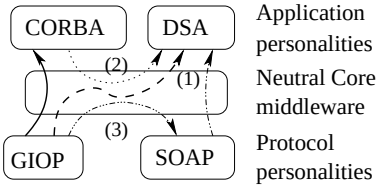

Figure 1. PolyORB's interacting personalities

Personalities implement a specific aspect of a distribution model. The Neutral Core Middleware enables the presence and interaction of multiple application and protocol personalities within the same middleware instance, leading to its "schizophrenic" nature (see [10] for more details).

Hence, this architecture separates three main components of middleware: protocol-side, application-side and internals. This reduces components interleaving. We now detail their interactions.

4.3 Separating middleware functions

Personalities implement middleware functions with respect to a specific semantics. Yet, most of these functions are notionally similar and can be defined as instances of some generic services.

Hence, schizophrenic middleware define generic services that express key middleware functionalities based on an analysis of multiple implementations. These services are focused on the completion of interactions between two nodes of a distributed application. One can combine particular services instances to implement the *neutral core middleware* and *application* or *protocol* personalities.

- **Addressing** Each entity is given a unique identifier within the entire distributed application.

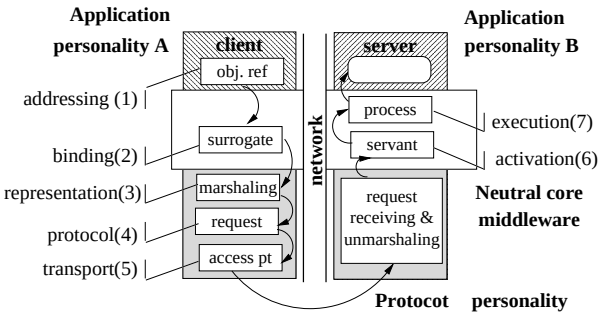

Figure 2. PolyORB's services

- **Binding** Middleware establishes and maintains associations between interacting objects and resources allowing this interaction (e.g. a socket, a protocol stack).

- **Representation** Request must be translated into a representation suitable for transmission over network.

- **Protocol** Middleware implements a protocol for the transmission of requests amongst nodes.

- **Transport** A communication channel is established between a node and an object to transmit messages.

- **Activation** Middleware ensures a concrete entity implementing objects is available to execute the request.

- **Execution** Middleware assigns execution resource to process every incoming request.

Figure 2 illustrates how PolyORB's services cooperate to transmit one request from one application personality to another, on two separate nodes using a common protocol.

The client (application personality 'A') gets a reference on the object from the *"addressing"* service (1); the core middleware creates a binding object (2, *"binding"* service): a dynamic gateway to the remote object through which the client communicates; a message is built from the request (3 and 4, resp. *"representation"*, *"protocol"* services). and sent to the remote node (5, *"transport"* service). Upon reception, remote node middleware ensures that a concrete entity implementing the object is available to execute the request (6, *"activation"* service) and assigns execution resources (7, *"dispatching"*) leading to the execution of application code by application personality 'B'.

So far, we demonstrated in [11] how composing and reusing services enable the rapid prototyping of RPC (distributed system annex of Ada), DOC (CORBA) and MOM (based on Sun's JMS API) middleware. PolyORB allows implementors to design multiple distribution model from a common set of services. Code reuse ratio reaches 70%; it is less than 30 % for generic middleware. We show how

the instanciation of PolyORB services to build personnalities preserve separation of middleware functions. Finally, benchmarks show performance are correct for such a middleware, when compared to generic middleware.

Per construction, each service encompasses a restricted, well-defined set of functionalities. This separation of concerns enables separate analysis and verification of each middleware component. It is a first step towards the verification of the whole middleware. Thus, schizophrenic middleware architecture provides foundations for formal verification.

5 Refining the Broker pattern

The *Broker* architectural pattern interleaves functions that define middleware architecture. We now detail how we refine this pattern specification to separate its functionalities and thus facilitate verification.

5.1 From architectural to design pattern

The *Broker* pattern role is to coordinate distributed applications and handle request exchange between nodes. Its initial definition as an *architectural* pattern (section 2.1) encompasses protocol services, resource allocation, request execution, etc. It gathers many components that should be delegated to separate components.

We propose to refine this architectural pattern and to define a *Broker* design pattern, i.e. a component that interacts with other services to provide the same functionalities than initial architectural pattern. This definition provides greater separation of components involved in middleware.

PolyORB's architecture enables function delegation to a specific service. Thus, middleware are combination of these services. In this respect, we define the *Broker* design pattern whose unique role is to coordinate communications between nodes and to transmit requests.

This design pattern cooperates with other PolyORB's services. It sends requests from a node to another using both *addressing* and *binding* services. It receives incoming requests from remote nodes through *transport* service; *activation* service ensures request completion.

Hence, the *Broker* design pattern along with PolyORB's services is notionally equivalent to the *Broker* architectural pattern. Yet, it clearly separates functions into separate elements. We now define each of them.

5.2 Elements of the Broker design pattern

We focus on elementary abstractions to express the *Broker* design pattern interfaces. These abstractions interact with other PolyORB's services to form the complete middleware. The elements to describe this pattern are:

- **Asynchronous Event Sources:** enable waiting on external event sources, e.g. incoming data on TCP sockets. These sources are input points used by remote nodes to interact with this *Broker* instance. An API to manipulate and check sources is provided.
- **Job Queue:** stores all incoming *jobs* the *Broker* will process. Elementary *Broker* actions are defined as *jobs* to be processed by tasks running *Broker*; e.g. monitoring an event source, binding, processing incoming data, executing a request.
- **Broker:** enables services to access *Broker* functionalities either to process *jobs*, or to receive incoming data from remote nodes through *asynchronous event sources*. We distinguish the *Broker* main loop procedure that monitors sources or process jobs until a given exit condition is met; from the *Broker* API that allows for interaction with event sources (protocol level) and the job queue (application level).
- **Scheduling Policy:** allocates existing tasks to run the *Broker* main loop. Allocation is done upon the notification of event occurrences within middleware.

Figure 3 details communication between these blocks.

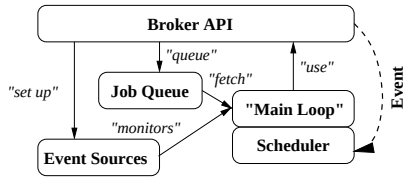

Figure 3. Interactions in *Broker* design pattern

This specification, and the unformal descriptions above provide a first overview of the *Broker* design pattern. It implies interactions among its sub-components. We now briefly present them in order to determine coupling between them.

- **Broker:** multiple tasks may concurrently execute *Broker*'s main loop, or other functions from the *Broker* API. This requires a precise locking policy.
- **Job Queue:** jobs can be queued by the *Broker* main loop upon the notification of an event on a source; or at the request of user code. This implies the definition of a specific queuing policy.

  Jobs are fetched in the *Broker* loop for processing.
- **Asynchronous event sources:** event sources list can be modified at any time; a traffic model defines how incoming data are received.

- **Scheduler:** Tasks executing the *Broker* main loop may be scheduled to achieve specific tasks. Figure 4 details the automaton used for scheduling. Current *Broker*'s global state determines the next action a task running the *Broker*'s main loop will perform: direct leaving if exit condition is met, processing jobs, blocking on event sources or going idle.

  The scheduler may be triggered at any time to ensure a task will process events: e.g. sources modification, new job queued, ...

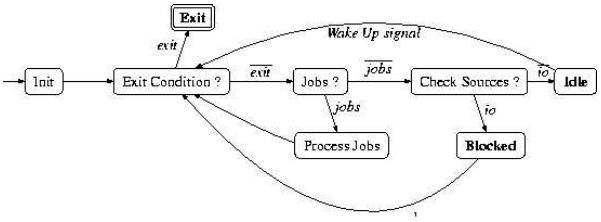

Figure 4. Task state automaton

These different elements show that this system is highly concurrent and driven by the occurrence of specific events. Thus we must choose carefully an appropriate modeling technique to model each *Broker*'s functions and their combination. We now detail how we move forward formal specification and then verification using Petri Nets.

6 Petri Net modeling of Broker

Petri Nets proved to be a rigorous formalism to verify concurrent event driven systems. Existing tools enable automatic verification of structural properties or model checking. In this section, we detail how we mapped previous specifications to Petri Nets.

We first briefly present well formed Petri nets. Then we discuss the modeling process of the *Broker*, and finally introduce the Petri net model of the *Broker*.

6.1 Unformal presentation of Petri nets

A well formed colored Petri net [12, 13] is a 5-uple $<P, T, Pre, Post, Types, M_0>$ where P is a set of places (depicted by circles), T is a set of transitions (depicted by rectangles), $Pre[t]$ is the precondition function for transition t, $Post[t]$ is the postcondition function for transition t, *Types* is the set of basic types (a basic type is a finite set) and M_0 is the initial marking.

To each place p, a domain $Dom(p)$ is associated: $Dom(p)$ is the cartesian product of some basic types. $Dom(p)$ corresponds to the set of token color that place p can possibly contain. In figure 5, basic classes is C. The domain of place **a** is the cartesian product of C with itself, the one of **b** and **c** is C.

A marking $M(p)$ is associated to each place p: $M(p)$ is a multi-set over $Dom(p)$. Therefore, a marking M is the function that associates a marking to each place p of P. An element of a marking in a place is called a token. In figure 5, the initial marking associates one token having value <1> in place **b** and two tokens having values <1> and <2> in place **c** (function <*C.all*> generates one token for any value of class C).

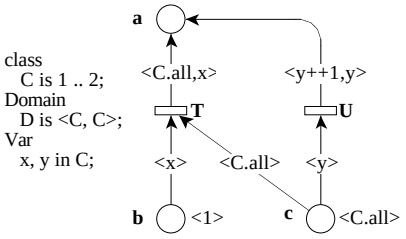

Figure 5. A small Petri net example.

Pre and *Post* functions describe how a marking is modified when an action is performed. Since actions are associated to transitions, instead of "an action is performed" we say: "a transition is fired". To each transition, a set of variables $Var(t)$ is associated. Each variable is defined over a basic type. In figure 5, $Var(\mathbf{T})$ = <*x*> and $Var(\mathbf{U})$ = <*y*>. The binding of a transition is the association of an actual value to each parameter. When a transition fires, the corresponding tokens are generated in its output places. Based on this evolution, the state space can be generated. The figure 6 provides the one of our example and explicitly presents all possible bindings (the double circle represents the initial state and black circles represent deadlock states).

Computing the state space allows behavioral analysis such as detection of specific states (e.g. "is a given situation possible") or causal relation between two states (e.g. "if I reach state S_1, will I eventually reach state S_2?").

Petri nets also support structural analysis: properties such as invariants (e.g. "the number of tokens remains constant on a subset of places") are computed on the graph structure and thus do not require to compute the state space too. Thus, infinite systems may be verified [13].

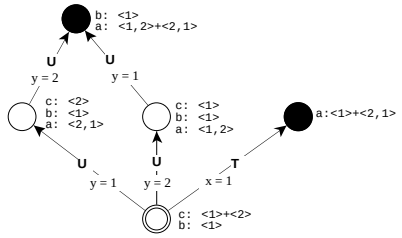

Figure 6. The corresponding state space.

6.2 The modeling process

Having unformally defined *Broker* subcomponents and their interactions, we now detail how to formally specify it.

We first transcribe the different components and interactions we presented into Petri Nets modules. We associate a specific action to each transition of the Petri Net; places represent states. Interaction between components is specified as common places between different modules. Hierarchy can also be used to provide partial views and helps in refining an initial Petri net model. However, this hierarchy should be flatten to use formal verification tools.

Then, Petri net modules are merged to produce a complete model, suitable for verification. To do so, we use the CPN-AMI[2] CASE environment that provides modeling facilities as well as model checking and structural analysis tools. Hence, the Petri Nets model for the *Broker* pattern is the aggregate of several Petri Net modules, each of which specifies one function of the *Broker*.

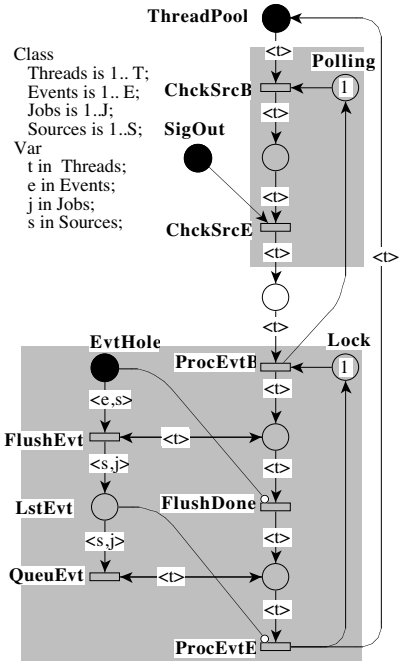

Figure 7. Petri Net for one *Broker* module

6.3 Petri Net of the Broker

By lack of space, we only present the Petri net module of one core function processed by the *Broker* main loop (see Figure 7). This procedure is triggered when a task is blocked, waiting for events. It consists of two phases: *1)* polling on event sources and *2)* processing events.

[2] `http://www-src.lip6.fr/logiciels/mars/cpn-ami`

- **Polling on event sources.** Place *ThreadPool* contains all threads scheduled to check event sources (see section 5.2). Only one task can actually check sources. When transition *ChckSrcB* fires, one available threads is selected to check sources. Transition *ChckSrcE* fires upon the notification of the presence of an event in the sources (place *SigOut*). Place *Polling* ensures only one thread can check sources.

- **Event processing.** Since we read from event sources (place *EvtHole*), this step has to be performed under critical section (ensured by place *Lock*). Transition *FlushEvt* is fired as long as there are events coming from any of the sources. Transition *FlushDone* fires when all events are consumed, which is enforced by the inhibitor arc from *EvtHole* to *FlushDone*. Events are stored in the Job Queue for further processing by others threads. The related functions are defined in other Petri net modules. When all events are queued, transition *ProcEvtE* fires to release the lock and restore the thread in the pool.

Places outlined in black have a special status in the Petri net module. They support communication with other Petri net modules. Their marking is generated by these modules and ensure an appropriate connection between the modules. They represent either other *Broker* functions or middleware modules interacting with the broker (i.e. behavior of PolyORB services). This typical composition technique is called *channel place* [14].

This Petri net module allows a stand-alone assessment of the modeled function. This can be done by changing the initial marking: each initial marking corresponds to a set of potential scenarios. This provides useful behavioral information on the correctness of the modeled function. All functions can be separately tested and then combined to form the complete Petri net model. Besides, module substitution allows us to define different scenarios that emulate specific conditions (e.g. queuing, locking or scheduling policies).

Then, these different models of the *Broker* can be tested, providing information on resource consumption/ We may first test for any deadlock or livelocks situations. Then, we may compute resources needed to fullfill a specific scenario; we may look for stable states or compute shorter or longuest processing path. The main advantage of this technique is to enable verification targeted on the way the middleware is used. Thus, optimizations can be formally verified according to specific execution conditions.

7 Conclusion and future works

This paper presents the first steps towards middleware verification. We have detailed how to extract from an un-

formal specification components that can be verified; and proposed the use of Petri Net to verify these components.

Most efficient middleware architectures rely on design patterns as a language to express and then implement user requirements. Test cases are defined to validate this architecture. Yet, this approach lacks generality: it only tests the use of a restricted set of functions.

Then, we presented the *Broker* architectural pattern. It provides a complete and precise definition of all components involved in remote service invocation. It is of common use in middleware architecture. We noted its analysis would provide analysis of middleware implementations. But this pattern interleaves many high-level functions. This impedes verification. We thus looked for precise separation of all middleware functions to ease the verification process.

We detailed existing modeling notations. We showed at least two different formalisms are required to enable the complete definition and then verification of software components. We chose to rely on design patterns notations and Petri Nets to model the *Broker*.

We first presented the *schizophrenic* middleware architecture and its implementation PolyORB. We showed how its architecture clearly decouples middleware functions at both definition and implementation level. We refined *Broker* architectural pattern and define a *Broker* design pattern. It interacts with other PolyORB services to fulfill the same functions. Moreover, the *Broker* design pattern embeds less functionalities. This also eases verification.

Finally, we explained how we produced Petri net models of the *Broker*. We presented how to build specific scenarios to verify properties with respect to application needs. We contemplate verifying our implementation PolyORB.

We defined specific conditions to be tested. This will require future work to be completed. We expect it will provide more information on middleware behavior with respect to specific scenarios and lead to the formal validation of properties of our model, and then of our implementation.

References

[1] D. Schmidt and F. Buschmann, "Patterns frameworks and middleware: Their synergistic relationships," in *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal., *Pattern-Oriented Software Architecture: A System Of Patterns*. John Wiley and Sons Ltd., 1996.

[3] C. Francu and I. Marsic, "An Advanced Communication Toolkit for Implementing the Broker Pattern," in *Proceedings of ICDCS'99*. IEEE, June 1999.

[4] D. C. Schmidt, D. L. Levine, and S. Mungee, "The design of the TAO real-time object request broker," *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[5] J. Merseguer, J. Campos, S. Bernardi, and S. Donatelli, "A compositional semantics for UML state machines aimed at performance evaluation," in *Proceedings of the Sixth International Workshop on Discrete Event Systems*, october 2002.

[6] H. Feiler, B. Lewis, and S. Vestal, "The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering," in *RTAS 2003 Workshop on Model-Driven Embedded Systems*, May 2003.

[7] D. Regep and F. Kordon, "**L***f***P**: a specification language for rapid prototyping of concurrent systems," in *12th IEEE International Workshop on Rapid System Prototyping*, June 2001.

[8] A. Singhai, A. Sane, and R. Campbell, "Quarterware for Middleware," in *Proceedings of ICDCS'98*. IEEE, May 1998.

[9] F. D. Tran and J.-B. Stéfani, "Towards an extensible and modular ORB framework," in *Workshop of ECOOP'97*, Jyvaskyla, Finlande, Apr. 1997, http://sirac.inrialpes.fr/~bellissa/wecoop97/dangtran.ps.gz.

[10] T. Quinot, F. Kordon, and L. Pautet, "From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models," in *Proceedings of the 3rd Int'l Symposium on Distributed Objects and Applications (DOA'01)*, Sept. 2001.

[11] J. Hugues, L. Pautet, and F. Kordon, "Contributions to middleware architectures to prototype distribution infrastructures," in *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, San Diego, CA, USA, June 2003.

[12] G. Chiola, C. Dutheillet, G. Franceschini, and S. Haddad, "On Well-Formed Coloured Nets and their Symbolic Reachability Graph," *High-Level Petri Nets. Theory and Application, LNCS*, 1991.

[13] C. Girault and R. Valk, *Petri Nets for System Engineering*. Springer Verlag, Sept. 2002.

[14] Y. Soussy, "Compositions of Nets via a communication medium," in *10th International Conference on Application and theory of Petri Nets*, Bonn, germany, June 1989.