A Factory To Design and Build Tailorable and Verifiable Middleware

Jérôme Hugues[1], Fabrice Kordon[2], Laurent Pautet[1], and Thomas Vergnaud[1]

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

Abstract. Heterogeneous non-functional requirements of Distributed Real-Time Embedded (DRE) system put a limit on middleware engineering: the middleware must reflect application requirements, with limited runtime impact. Thus, building an application-tailored middleware is both a requirement and a challenge.
In this paper, we provide an overview of our work on the construction of middleware. We focus on two complementary projects: the definition of middleware that provides strong support for both tailorability and verification of its internals; the definition of a methodology that enables the automatizing of key steps of middleware construction.
We illustrate how our current work on PolyORB, Ocarina and the use of Petri Nets allows designer to build the middleware that precisely matches its application requirements and comes with precise proof of its properties.

1 Introduction

Middleware first emerged as a general solution to build distributed applications. Models and abstractions such as RPC, distributed objects hide the intrinsic of distribution from the user, and provide a programming model close to the local case.

In the meantime, the need for Distributed Real-Time Embedded systems (DRE) increases regularly. Such systems require execution infrastructures that have specific capabilities, some of which conflict with "plain old middleware technology":

- *Distribution* cannot remain hidden from the developer. The semantics of the distribution models must be adapted to real-time application needs. For instance, the application entity should be well adapted to scheduling analysis such as the publish/subscribe model [RGS95]. Besides the impact of runtime entities (e.g. communication channels, memory management) on timeliness or determinism must be fully assessed.
- *Real-Time* engineering guidelines must be supported by the middleware. This middleware follows a clear and precise design so as to guarantee its determinism and its temporal properties; it comes with complete proofs that it does not withdraw the properties of the application [Bud03]. Finally, a methodological guide, support

tools and Quality of Service (QoS) policies help to tailor the middleware with respect to the application requirements.
– *Embedded* targets that have strong constraints on their resources (e.g. CPU, memory, bandwidth) or limited run-time support by a real-time kernel (no exception, no dynamic memory, limited number of threads, etc). So, middleware must cope with strong limitations; and scale down to small targets. In some cases, new functions or QoS policies are added to cope with platform limitations, e.g. data compression for systems with a narrow bandwidth.

So, there is a need to 1/ make available to the developer some internals of the middleware to allow its tailoring and adaptation; 2/ define a development process and supporting tools to ease this adaptation and ensure its is correct with respect to middleware constraints.

Let us note a DRE is usually composed of several components for with different requirements. Therefore, both functional interoperability and compatibility of non-functional policies must be contemplated. Such assessment capability is seldom contemplated by middleware architects.

Another common pitfall when designing DRE is the use of "Commercial Off-The-Shelf" (COTS) components. This allows to reduce costs and potential errors by reusing already tested components. But this puts a strong limit on middleware tuning, verification and performance capabilities.

Engineers of DRE systems require middleware that have good performance (including efficient marshaling), real time (use only deterministic constructs), fit embedded constraints. Besides, they also need to ensure their use of the middleware is correct (no deadlock, deadline are respected, etc). Hence, this calls not only for a middleware, but also for a design process and tools that allow the user to carefully tune the middleware it to needs, instead of selecting a "best effort" middleware.

The objective of the PolyORB project is to elaborate both a middleware and a design process. We propose an innovative architecture that aims at providing better control on the configuration of the middleware, and enables the careful analysis of its properties. This paper presents an overview of our work in this area for the past years.

In the next section, we motivate our work by reviewing major issues when designing middleware for DRE systems, revolving around tailorability and verification concerns. Then, we present our current results in middleware architecture, and how we efficiently address both concerns by defining an original architecture. We note that another limit to the adoption of middleware is the lack of tool support; we then discuss our current research work around Architecture Description Language to build tool that help building and verifying application-specific middleware configurations.

2 Tailorable and Verifiable Middleware: State of the Art

In this section, we discuss limits and trade-offs when considering tailorable and verifiable middleware. Even though both capabilities are of interest for the application designers, we note that there is usually little support provided by the middleware.

2.1 From Tailorability to Verification

The many and heterogeneous constraints of distributed applications deeply impact the development of distribution middleware. Middleware should support developers when designing, implementing and deploying such systems in heterogeneous environments and evaluate so called "non functional" requirements (such as QoS or reliability).

The design and implementation of tailorable middleware is now a (almost) mastered topic. Design patterns, frameworks have proved their value to adapt middleware to a wide family of requirements [SB03].

In the mean time, middleware platforms have shown in various projects they can meet stringent requirements. They are now used in many mission-critical applications, including space, aeronautics and transportation.

Building distribution platform for such systems is a complex task. One has to cope with the restrictions enforced to achieve high integrity standards, or to meet certification requirements, such as DO-178B. Thus, one has to be able to assert middleware properties, e.g. functional behavioral properties such as *absence of deadlocks*, *request fairness*, or *correct resource dimensioning*; but also *temporal* properties.

Hence, verifying middleware is now becoming a stringent requirement in many DRE systems. The developer must ensure beforehand that its application design is compatible with middleware capabilities.

We claim middleware engineering should now provide provisions for some verification mechanisms as defined by the ISO committee [ISO94] as *"[the] confirmation by examination and provision of objective evidence that specified requirements have been fulfilled. Objective evidence is information which can be proved true, based on facts obtained through observation, measurement, test or other means."*

However, we note there is a double combinatorial explosion when considering middleware as a whole: the number of possible execution scenarios for one middleware configuration increases with the interleaving of threads and requests; the number of possible configurations increases with middleware adaptability and versatility. Finally, the behavior of a middleware highly depends on the configuration parameters selected by the user. Thus, verifying a middleware is a complex task.

Some projects consider testing some scenarios, on multiple target platforms. The Skoll Distributed Continuous Q&A project [MPY+04] relies on the concepts of distributed computing to test TAO many configurations and scenarios on computers around the world. This provides some hints on the behavior of the middleware, but cannot serve as a definite proof of its properties.

One may instead contemplate the verification of middleware properties. Yet this is usually done on a limited scale, restricted to the very specific scenarios of the application to be delivered and the semantics of the distribution model used (e.g. RT CORBA), for instance using the Bogor model checker [DDH+03]. But the middleware must be considered as part of the application and must not be discarded from the verification process as a blackbox would be.

However, middleware implementations of the same specifications may behave differently [BSPN00]. Some properties may be withdrawn by implementation issues, such as the use of COTS, that are hidden by this modeling process, or by different interpretation of the same specifications. Besides, such a verification process usually does

not take into account implementation-defined configuration options, and target capabilities. Finally, such methods may be limited by combinatorial explosion that arise when building the system state-space.

Thus, we note it is hard (if not infeasible) to verify existing middleware as a whole. One should go forward and integrate verification to the design of middleware.

2.2 Addressing Verification Concerns

The formal-based verification of distributed application behavioral properties is usually the domain of verification-domain experts, using specific verification techniques, e.g. calculi, formal methods. However, such a verification process is usually used only to verify the semantics of the application (e.g. set of correct message sequences) [Jon94].

Turtle-P [AdSSK03] defines a UML profile for the validation of distributed applications, linked with code generation engines and validation tools built around RT-LOTOS [LAA04]. Validation is done either through simulation or verification of timed automata. However, this provides no information on the underlying distribution framework or middleware integrated to the system; and thus reduces the scope of the properties proved for the application under study.

Finally, it should be noted that complex semantics of distribution models is difficult to model and usually reduced: complex request dispatching policies, I/O or memory management are simplified. This reduces verification cost but also interest in the middleware modeled that looses many configuration capabilities.

Thus, we claim the verification process of a distributed application should also focus on the middleware as a building block, and thus middleware architecture should be made verification-ready so as to ease this process, without impeding its configurability.

Still, this increases the complexity of the verification process: one should focus on the actual configuration being used. This means that models of the configuration should be built “on demand”, and that a strong link between model and implementation exists.

From the previous analysis, we conclude that a dedicated process to build and verify tailorable middleware is required. This process should be defined around well-grounded engineering methods and foster reusable and tailorable software components. Besides, verification techniques should be included in the process to assert strong properties of complex configurations, using the most suitable methods, depending on the nature of the property (causal, time, dependability, etc.).

3 The Schizophrenic Architecture: a Tailorable and Verifiable Middleware

In this section, we discuss our approach to design middleware dedicated to the requirements of a given application. This approach can be viewed as a co-design between the application and its supporting middleware. As an illustration of the feasibility of this design process, we provide a highly generic middleware architecture (also known as the “schizophrenic” architecture) and a methodological guide to instantiate it.

3.1 From system requirements to a dedicated middleware

Actual middleware has to fulfill the system requirements. Some solutions are based on standardized “rigid” specifications; this is the case for most CORBA implementations and its many extensions (RT-, FT-, minimum CORBA…). Such middleware architectures are targeted to a certain application domain, and usually add many configuration parameters to partially control its resource or request processing policies.

Yet, implementations are not as efficient as specifically designed middleware [KP05]. The cost to deploy specific features is high due to the API to manipulate. Many optimization options cannot be implemented because of the heterogeneity of requirements and the number of (possibly useless) functions to support. Besides, verification or testing is not addressed and under the control of the middleware vendor. It is a direct consequence of the absence of a “one size fits all" middleware architecture.

Therefore, one should not contemplate middleware as a whole, but instead design middleware components and the process to combine them as a safe and affordable solution to system requirements. Thus, it becomes possible to build the distribution infrastructure built for specific requirements.

In the following, we describe the different steps we followed to define one such process built around a highly tailorable middleware architecture, a set of middleware components.

3.2 Defining a new tailorable middleware architecture

Solutions have been proposed to design tailorable middleware. *Configurable* middleware defines an architecture centered on a given distribution model [SLM98] (e.g. distributed objects, message passing, etc.); this architecture can be tuned (tasking policy, etc.). *Generic* middleware [DHTS98] provides a general framework, which components have to be instantiated to create middleware implementations. Those implementations are called *personalities*. Generic middleware is not bound to a particular middleware model; however, various personalities seldom share a large amount of code.

Generic functions propose a coarse grain parametrization (selection of components). Configuration is fine grain parametrization (customization of a component). Verification is possible through behavioral descriptions (attached to components).

Configurable and generic middleware architectures address the tailorability issue, as they ease middleware adaptation. However, they do not provide complete solutions, as they are either restricted to a class of distribution model; or their adaptation requires many implementation levels, thus becomes too expensive.

3.3 Decoupling middleware components

To enhance middleware adaptation at a reduced implementation cost, we proposed the “schizophrenic middleware architecture” [VHPK04]. Its architecture separates concerns between distribution models, API, communication protocols, and their implementations by refining the definition and role of personalities.

The schizophrenic architecture consists of three layers: *application* and *protocol* personalities built around a *neutral* core. Application interacts with application personalities; protocol personalities operate with the network.

Application personalities constitute the adaptation layer between application components and middleware through a dedicated API or code generator. They provide APIs to interface application components with the core middleware; they interact with the core layer in order to allow the exchange of requests between entities. Application personalities can either support specifications such as CORBA, the Java Message Service (JMS), etc. or dedicated API for specific needs.

Protocol personalities handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages exchanged using a chosen communication network and protocol. Protocol personalities can instantiate middleware protocols such as IIOP (for CORBA), SOAP (for Web Services), etc.

The neutral core layer 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.

The neutral core layer enables the selection of any combination of application and/or protocol personalities. Several personalities can be collocated and cooperate in a given middleware instance, leading to its "schizophrenic" nature.

3.4 PolyORB, a schizophrenic middleware

In [VHPK04], we present PolyORB our implementation of a schizophrenic middleware. PolyORB a free software middleware supported by AdaCore[3], PolyORB's research activities are hosted by the ObjectWeb consortium[4].

We assessed its suitability as a middleware platform to support general specifications (CORBA, DDS, Ada Distributed Systems Annex, Web Applications, Ada Messaging Service close to Sun's JMS) as well as profiled personalities (RT-CORBA, FT-CORBA) and as a COTS for industry projects.

In the remainder of this section, we provide a review of the key elements of PolyORB's architecture, implementation, and its capabilities to address middleware tailorability and verification.

3.5 A Canonical Middleware Architecture

Our experiments show that a reduced set of services can describe various distribution models. We identify seven steps in the processing of a request, each of which is defined as one fundamental service. Services are generic components for which a basic implementation is provided. Alternate implementation may be used to match more precise semantics. Such an implementation may also come with its behavioural description for verification purposes. Each middleware instance is one coherent assembling of these entities. The μBroker component coordinates the services : it is responsible for the correct propagation of the request in the middleware instance. Figure 1 illustrates the cooperation between PolyORB services.

[3] `http://www.adacore.com`
[4] `http://polyorb.objectweb.org`

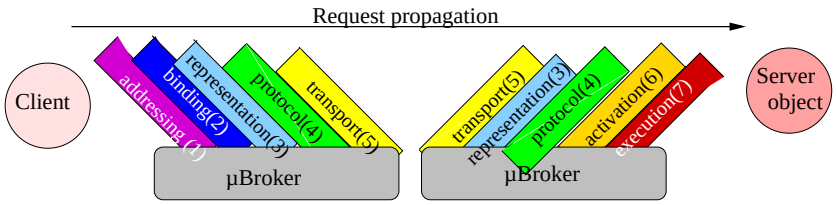


Fig. 1. Request propagation in the schizophrenic middleware architecture

First, the client looks up server's reference using the *addressing* service (1), a dictionary. Then, it uses the *binding* factory (2) 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 (3), this is a mathematical mapping that convert a data into a byte stream (e.g. CORBA CDR).

A *protocol* (4) supports transmissions between the two nodes, through the *transport* (5) service; it establishes a communication channel between the two nodes. Both can be reduced to *finite-state automata*. 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 (6). Finally, the *execution* service (7) assigns execution resources to process the request. These services rely on the *factory* and *resource management* patterns.

Hence, services in our middleware architecture are *pipes and filters*: they compute a value and pass it to another component. Our experiments with PolyORB showed all implementations follow the same semantics, they are only adapted to match precise specifications. They can be reduced to well-known abstractions.

The μBroker handles the coordination of these services: it allocates resources and ensures the propagation of data through middleware. Besides, it is the only component that controls the whole middleware: it manipulates critical resources such as tasks and I/Os or global locks. It holds middleware behavioral properties.

Hence, the schizophrenic middleware architecture provides a comprehensive description of middleware. This architecture separates a set of generic services dedicated to request processing from the μBroker.

3.6 μBroker: core of the schizophrenic architecture

The μBroker component is the core of the PolyORB middleware. It is a refinement of the Broker architectural pattern defined in [BMR+96]. The Broker pattern defines the architecture of a middleware, describing all elements from protocol stack to request processing and servant registration.

The μBroker relies on a narrower view of middleware internals: the μBroker cooperates with other middleware services to achieve request processing. It interacts with the *addressing* and *binding* services to route the request. It receives incoming requests

from remote nodes through the *transport* service; *activation* and *execution* services ensure request completion.

Hence, the µBroker *manages resources and coordinates middleware services to enable communication between nodes and the processing of incoming requests*. Specific middleware functions are delegated to the seven services we presented in previous section. The µBroker is the dispatcher of our middleware architecture.

Several "strategies" have been defined to create and use middleware resources: in [PSCS01], the authors present different request processing policies implemented in TAO; the CARISM project [KP04], allows for the dynamic reconfiguration of communication channels. Accordingly, the µBroker is configurable and provides a clear design to enable verification. Figure 2 describes the basic elements of the µBroker.

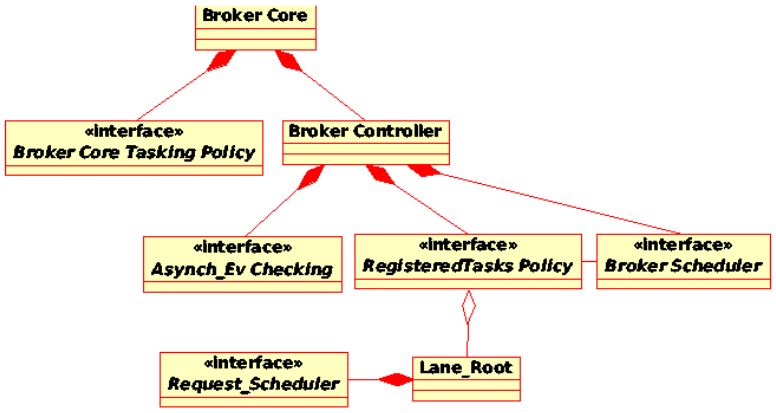


Fig. 2. Overview of the µBroker

The *µBroker Core API* handles interactions with other middleware services.

The *µBroker Tasking Policy* controls task creation in response to specific events within the middleware, e.g. new connection, incoming requests;

The *µBroker Controller* manages the state automaton associated to the µBroker. It grants access to middleware internals (tasks, I/O and queues) and schedules tasks to process requests or run functions in the *µBroker Core*. Several policies control it: the *Asynchronous Event Checking* policy sets up the polling and data read strategies to retrieve events from I/O sources; the *Broker Scheduler* schedules tasks to process middleware jobs (polling, processing an event on a source or a request). The *Request Scheduler* controls the specific scheduling of requests; the *Lane_Root* controls request queueing; the *Request Scheduler* controls thread dispatching to execute requests.

These elements are defined by their interface and a common high-level behavioral contract. They may have multiple instances, each of which refines their behavior, allowing for fine tuning. We implemented several instances of these policies to support well-known synchronization patterns.

The schizophrenic middleware architecture proposes one comprehensive view of one middleware architecture. This architecture is defined around a set of canonical components, one per key middleware's function, and the μBroker component that coordinate and allocates resources to actually execute them.

This allows for an iterative process to build new distribution feature and support new models: one can build new services and bind them to the μBroker. These services form the root of the distribution feature, exported to the user through dedicated API or code generator. We detail the later in the next section.

3.7 A methodology to design new personalities

A methodological guide details the different steps to instantiate PolyORB (figure 3) from a specific set of application requirements and the implied distribution model (step 1). It is intended to give the user the proper knowledge to tailor PolyORB. There are several ways to adapt PolyORB to the application requirements (step 2):

- Use an existing personality. PolyORB already comes with CORBA, RT-CORBA, DSA, MOMA (Ada-like JMS), DDS and the existing configuration parameters;
- Design a new personality: design or refine some of the fundamental components, by re-using fundamental components already developed from existing personalities or from the neutral core; overloading them or designing new variant of fundamental components from scratch.
  Note that when a new personality is designed, we get back to the generic architecture (step 3) to decide whether the new features would be useful for other personalities. In this case, there are two possible policies:
- This feature has a simple and generic enough implementation that can be reused by other personalities, then the feature is integrated in the pool of neutral core layer components, e.g. concurrency policies, low-level transport;
- This feature is intrinsically specific to a personality, the implementation enhancement is kept at the level of the protocol or application personalities, e.g. GIOP message management, DDS specific API.

Finally the user derives one assembly of components: the fine-tuned middleware adapted to its initial needs (step 4).

This procedure may also be repeated to adapt more precisely components, allowing for evolving design of some core elements without impeding the whole assembly.

In this section, we have defined the middleware architecture and associated methodology used to implement middleware. We enforce a strong separation of concerns between the different functions involved in the middleware and we combine them to form the required implementation. Such a process proved its efficiency when implementing DDS on top of PolyORB [HKP06].

3.8 Formal verification

In this section, we discuss the formal techniques used to model the μBroker, and then verify some of its expected properties using model-checking.

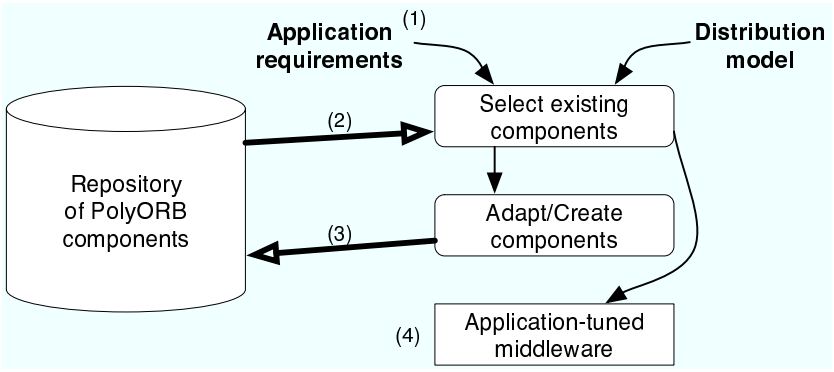


Fig. 3. Designing new personalities

Modeling one middleware configuration We propose to use formal methods to model and then verify our system. We selected *Well-formed colored Petri nets* [CDFH91] as an input language for model checking. They are high-level Petri nets, in which tokens are typed data holders. This allows for a concise and parametric definition of a system, while preserving its semantics. Using these methods, we can now model our architecture using Petri nets as a language for system modeling and verification (figure 4).

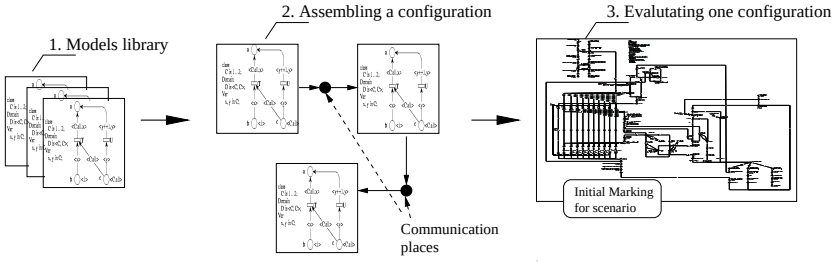


Fig. 4. Steps of the μBroker modeling

Step 1: we build one Petri net for each middleware components variation. Petri net transitions represent atomic actions; Petri net places are either middleware states or resources. Common places between different modules define interactions between Petri nets modules, they act as *channel places* [Sou89].

Step 2: for one configuration of the μBroker, some Petri net modules are selected to produce the complete model. Communications places (outlined in black) represent links to other μBroker functions or to middleware services.

Step 3: the selected modules are merged to produce a global model, it represents one middleware configuration. This model and one initial marking enable the verification of the middleware properties.

Then, middleware functions can be separately verified and then combined to form the complete Petri net model. Many models can be assembled from a common library of models. Thus, we can test for specific conditions (policies and settings).

The initial marking of the Petri Net defines available resources (e.g. threads, I/Os); or sets up internal counters. Its state space covers all possible interleaving of atomic actions; thus all possible execution orders are tested.

μBroker configurations and models In this section, we review the key parameters that characterize the μBroker, and some of the properties one might expect from such a component.

The μBroker is defined by the set of policies and the resources it uses. These settings are common to a large class of applications. We consider one middleware instance, in server mode, that processes all incoming requests. We study two configurations of the μBroker: *Mono-Tasking* (one main environment task) and *Multi-Tasking* (multiple tasks, using the Leader/Followers policy described in [PSCS01]). The latter allows for parallel request processing.

We assume that middleware resources are pre-allocated: we consider a static pool of threads; a bounded number of I/O sources and one pre-allocated memory pool to store requests. This hypothesis is acceptable: it corresponds to typical engineering practices in the context of critical systems. Our implementations and the corresponding models are controlled by three parameters:

S_{max} is the upper bound of I/O Sources listening for incoming data;

T_{max} is the number of Threads available within the middleware;

B_{size} is the size of the Buffer allocated to read data from I/O sources.

S_{max} and T_{max} define a workload profile for the middleware node, B_{size} defines constraints on the memory allocated by the μBroker to process requests. These parameters control middleware throughput and execution correctness.

We list three essential properties of our component. They represent basic key properties our component must verify to fulfill its role.

$P1$, *no deadlock* the system process all incoming requests;

$P2$, *consistency* there is no buffer overflow;

$P3$, *fairness* every event on a source is detected and processed.

$P1$, $P3$ are difficult to verify only through the execution of some test cases: one has to examine all possible execution orders. This may not be affordable or even possible due to threads and requests interleaving. Besides, the adequate dimensioning of static resources to ensure consistency ($P2$) is a strong requirement for DRE systems, yet it is a hard problem for open systems such as middleware. Thus, we propose to verify them for some configuration of the μBroker: each property is expressed as a LTL formula, then verified by model-checker tools.

Achieving formal analysis One known limit to the use of Petri Nets as model checker is the combinatorial explosion when exploring the system's state space.

We tackle this issue using recent works carried out at the LIP6. By detecting of the symmetries of a system [TMDM03], and exploiting the symmetries allowed by a property [BHI04]. In most favorable cases, these methods require exponentially smaller

memory space than traditional method based on full enumeration, and thus more amenable to computations within reasonable delays.

Thus, we claim that the analysis of PolyORB could not have been performed without the use of model checking because of the large number of states. As an illustration, even for common middleware configurations (up to 17 threads) the system presents over 6.56×10^{17} states, but we could compute and evaluate its properties on the model using advanced tools.

This verification experience is a proof of feasibility. New tools are a prerequisite to ease the structuring, and production of a formal specification of a middleware dedicated to application requirements. Such a specification would enable both the verification and the code generation the corresponding implementation

In the following, we illustrate how an architecture definition language such as the AADL enables us to define such a process and support tools.

4 A Process to Build Tailorable and Verifiable Middleware

The schizophrenic architecture allows for a fine tailoring of the middleware. It also permits formal verification on a given middleware instance. In order to help the configuration of the middleware, we need a way to capture the application needs and then build the corresponding middleware. In this section we explain our methodology to design and build a distributed application with its particular middleware, using the AADL.

4.1 Overview of the AADL

A few ADLs explicitly deal with real-time systems. Examples are ROOM [RSRS99] and AADL [Lew03]. An AADL model can incorporate non-architectural elements: embedded real-time characteristics of the components (execution time, memory footprint...), behavioral descriptions, etc. Hence it is possible to use AADL as a backbone to describe all the aspects of a system.

"AADL" stands for Architecture Analysis & Design Language. It aims at describing DRE systems [FLV00] by assembling blocks separately developed. In this section we describe the AADL and show how it can be used to describe application components.

The AADL [SAE04b] allows for the description of both software and hardware parts of a system. It focuses on the definition of clear block interfaces, and separates the implementations from these interfaces. It can be expressed using graphical and textual syntaxes; an XML representations is also defined to ease the interoperability between tools.

An AADL description is made of *components*. The AADL standard defines software components (data, threads, thread groups, subprograms, processes), execution platform components (memory, buses, processors, devices) and hybrid components (systems). Components model well identified elements of the actual architecture. *Subprograms* model procedures like in C or Ada. *Threads* model the active part of an application (such as POSIX threads). *Processes* are memory spaces that contain the *threads*.

Thread groups are used to create a hierarchy among threads. *Processors* model microprocessors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, etc. *Buses* model all kinds of networks, wires, etc. *Devices* model sensors, etc. Unlike other components, *systems* do not represent anything concrete; they actually create building blocks to help structure the description.

Component declarations have to be instantiated into subcomponents of other components in order to model an architecture. At the top-level, a system contains all the component instances. Most components can have subcomponents, so that an AADL description is hierarchical. A complete AADL description must provide a top-level system that will contain the other components, thus providing the root of the architecture tree. The architecture in itself is the instantiation of this system.

The interface of a component is called *component type*. It provides *features* (e.g. communication ports). Components communicate one with another by *connecting* their features. To a given component type correspond zero or several implementations. Each of them describe the internals of the components: subcomponents, connections between those subcomponents, etc. An implementation of a thread or a subprogram can specify *call sequences* to other subprograms, thus describing the execution flows in the architecture. Since there can be different implementations of a given component type, it is possible to select the actual components to put into the architecture, without having to change the other components, thus providing a convenient approach to configure applications.

The AADL defines the notion of *properties* that can be attached to most elements (components, connections, features, etc.). Properties are attributes used to specify constraints or characteristics that apply to the elements of the architecture: clock frequency of a processor, execution time of a thread, bandwidth of a bus, etc. Some standard properties are defined; but it is possible to define one's own properties.

Refining Architectures The AADL syntax allows for great flexibility in the precision of the descriptions. In the listing 1.1, we describe a process that receives messages (modeled by an event data port). Such a description is very vague, since we do not give any details about the actual structure of the process (e.g. how many threads?). Yet it is perfectly correct regarding the AADL syntax, and provides a first outline of the architecture specification.

```
1 data message
2 end message;
3
4 process receiver_process
5 features
6  msg : in event data port message;
7 end receiver_process;
```

Listing 1.1. Simple example of an AADL description

We can refine the architecture by providing an implementation of the process. Here we choose a very simple implementation, with one single thread that calls the user application (listing 1.2). We use an AADL standard property to indicate that the thread is dispatched aperiodically. The thread is to be executed upon the reception of a message.

We could also define other implementations, with several threads to process the incoming messages or perform other tasks. This facilitates the refinement of a given architecture: We can start by defining the outline of the architecture (listing 1.1), and then create implementations of the components (listing 1.2).

```
process implementation receiver_process.implem
subcomponents
  thr1 : thread receiver_thread.implem;
connections
  connect1 : event data port msg -> thr1.msg;
end receiver_process.implem;

thread receiver_thread
features
  msg : in event data port message;
properties
  dispatch_protocol => aperiodic;
end receiver_thread;

thread implementation receiver_thread.implem
calls
  {user_app : subprogram application};
connections
  parameter msg -> user_app.msg;
end receiver_thread.implem;

subprogram application
features
  msg : in parameter message;
end application;
```

Listing 1.2. Implementation of the process

Our model is partial and does not include any hardware component: we do not specify on what processor the process is running, etc. Such information should be provided when designing the complete architecture: the processes that send messages, the processors, associated memories and potential buses if there are several processors. The model is precise enough for the scope of this paper, though. In the following sections, we focus on the receiver thread.

4.2 Overview of the Methodology

Given its ability to describe both software and hardware components, the AADL perfectly fits our needs. We can use it to completely describe distributed architectures and capture all the necessary parameters. In addition, it has the ability to support a step-by-step design process based on the refinement of architecture. Thus it allows for a progressive approach in the architecture modeling.

The figure 5 illustrates our approach to design the middleware. We use the AADL to describe the application. From the application description, we can deduce the required parameters for the middleware (scheduling policy, data types, etc.) and extract

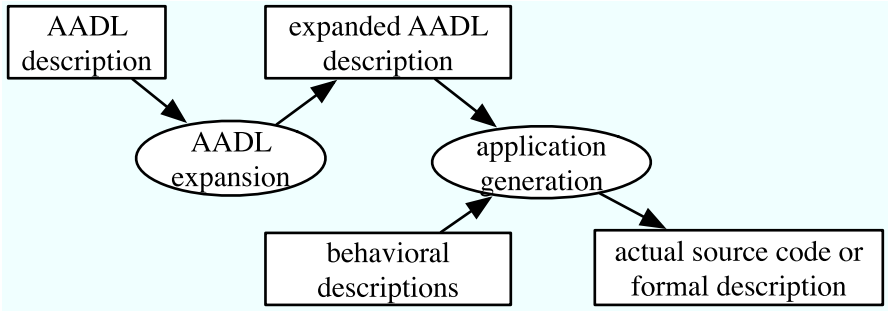


Fig. 5. Application generation based on the AADL

an adequate configuration; it is then possible to create an AADL description of the underlying middleware. We can then generate formal description from the AADL model and perform model checking. Once verifications have been performed, we can generate the code for the application and the middleware.

4.3 Modeling the middleware architecture using the AADL

The schizophrenic architecture provides a clear structure to create tailorable middleware. A notation such as the AADL syntax can be used to describe a schizophrenic middleware instance, in order to rapidly configure and deploy a tailored middleware that meets the application requirements.

Architectural description of the middleware components Middleware is the lower part of an application; it can be viewed as a software component (or a set of software components) on which the user application relies. Given its modular structure, the schizophrenic architecture shall be modeled by a set of AADL software components.

Overall design Middleware is a part of the application. Hence a middleware architecture shall be described using software components: a set of *subprograms* called by one or more *threads* (depending on the middleware configuration); *data* components model the data structures exchanged between the subprograms.

The subprograms should be organized so that they reflect the seven canonical services and the μBroker of the schizophrenic architecture.

Subprograms cannot be subcomponents of a system, since they do not model "autonomous" components. Hence the schizophrenic architecture cannot be represented as a set of systems. Consequently, the description is to be organized as a collection of packages containing subprograms and data; the packages should reflect the logical organization of the architecture.

Basically, the model should then have seven packages containing the subprograms associated with the seven basic services; the components of the μBroker, which constitutes the middleware "heart", should also be materialized as a package. Finally, the

different subprograms and data modeling the personalities should be defined into separate packages. Other “tools”, such as socket managers, could be defined into separate packages.

Each service can actually be modeled as a few main subprograms that are called from other parts of the architecture. Such subprograms shall be placed into the public sections of the packages, while more internal subprograms shall be defined into the private part.

Middleware configuration The middleware configuration is either given by its architectural description, or by some properties associated to the components.

The personalities to use for a given configuration are materialized by the actual packages and components used to describe the architecture. The actual number of threads to use is set by describing them in the architecture.

Some configuration elements such as the tasking policy deal with the behavioral description of the system, not its architecture; yet it is possible to specify them within the μBroker, using user-defined properties.

The configuration of some services can be specified by providing a particular component implementation. For example, the activation service can either be a mere list associating references to procedures, or or more evolved mechanism with priorities, like CORBA’s POA. Those two possibilities correspond to two different implementations of the same subprogram type.

4.4 Using AADL to verify the middleware

We now explain how to convert the AADL description into a Petri net and in source code; we show how to integrate existing behavioral descriptions associated with AADL components into the generated Petri net.

Using the AADL to support the construction of verifiable systems The AADL in itself only focuses on the description of the system architectures. Hence, unlike the UML, it does not aim at providing a complete and integrated set of syntaxes to describe all aspects of a model. Instead, the AADL facilitates the integration of other description paradigms within the architectural description, the latter one providing containers for the former ones. This allows for the reuse of “legacy” paradigms instead of imposing a specific syntax.

The integration of third-party languages within the AADL is done through properties or annexes. We privilege the use of AADL properties since it facilitates the use of a repository of behavioral descriptions that can be referenced by the AADL components. This allows for a clear separation between the architectural and behavioral descriptions.

Mappings must be defined in order to describe how to merge behavioral description into the AADL elements. The AADL standard defines mappings for Ada and C languages [SAE04a]. Translations have also been defined between the AADL error model and Petri nets [RKK06], thus allowing the use of existing verification and dependability evaluation tools.

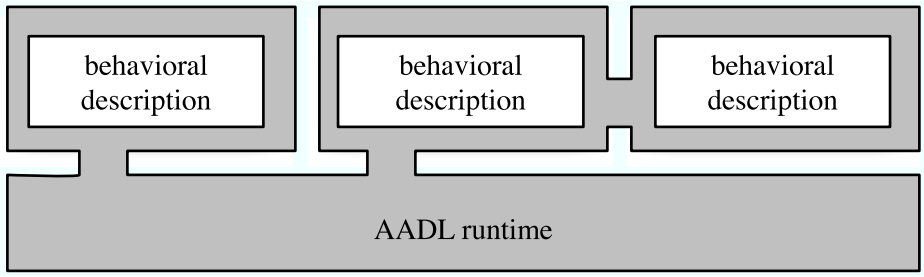


Fig. 6. Principle of an architecture-driven mapping for the AADL

Our approach focuses on the integration of behavioral descriptions within AADL architectures. Thus, behavioral implementations are controlled by the runtime built from AADL descriptions, which helps ensure the consistency between AADL model and resulting application. The figure 6 illustrates the principles of our mappings: behavioral descriptions (in white) are encapsulated by a runtime generated from the AADL description (in grey). We now give an overview of a mapping from AADL constructions to Petri nets and Ada.

Mapping AADL constructions to Petri Nets and source code We aim at using the AADL to coordinate formal verification and code generation. To do so, we defined rules to produce a Petri net or Ada code from AADL descriptions. Using these mappings we can generate a complete Petri net from the assembly of AADL components, each of them characterized by its own Petri net (such the nets described in section 3.8); once we ensure the architectural constructions are valid, we can generate the corresponding source code. This allows to perform verification on the whole system before code generation.

The AADL elements to map into Petri nets are the software components. Indeed, execution platform components are used to model the deployment of the software components; such deployment information is not to in the scope of Petri nets. AADL threads and AADL subprograms are the most important components, since they describe the actual execution flows in the architecture. AADL processes and systems are actually boxes containing threads or other components, and do not provide any “active” semantics; data components are not active components either.

The mapping for source code takes the same components into account. However, some components, such as AADL threads and processes, represent the AADL runtime. Thus they do not exactly correspond to code generation; the configuration of the AADL runtime is set from the information provided by these components. The table 1 lists the main rules of the mappings.

The Petri net mapping mainly consists of translating the AADL execution flows. Components that do not have any subcomponents nor call sequences are modeled by a transition that consumes inputs and produces outputs. Component features are modeled by places.

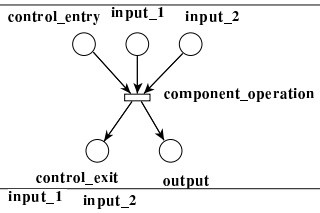
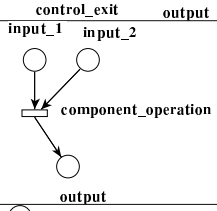
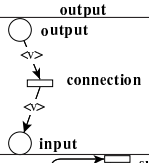
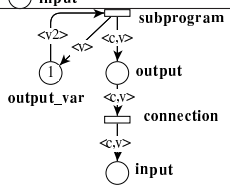
AADL	corresponding Petri net	corresponding Ada code
data data_type **end** data_type;	not translated in Petri nets	**type** data_type **is null record**;
subprogram a_subprogram **features** input_1 : **in parameter**; input_2 : **in parameter**; output : **out parameter**; **end** a_subprogram;		**procedure** a_subprogram (input_1 : **in** data_type; input_2 : **in** data_type; output : **out** data **type**) **is** **begin** **null**; **end**;
process a_process **features** input_1 : **in data port** data_type; input_2 : **in data port** data_type; output : **out data port** data_type; **end** a_process;		correspond to a middleware instance
connection : **data port** output −> input;		handled by the middleware
connection : connect : **parameter** output −> input;		−− *procedure subprogram_a (output: out data_type);* −− *procedure subprogram_b (input: in data_type);* subprogram_a (connect); subprogram_b (connect);

Table 1. Main patterns of the mapping between the AADL Petri nets and source code

We model a place per feature. This systematic approach help the user identify the translation between AADL models and corresponding Petri nets. In addition, it facilitates the expansions of the feature places. For example, we might want to describe the queue protocols defined by the AADL properties: in this case we would replace each place by Petri nets modeling FIFOs or whatever type of queue is specified by the AADL properties.

Connections between features are modeled by transitions. We distinguish connections between subprograms parameters and between other component ports.

Tokens stored in input features are to be consumed by component or connection transitions; tokens produced by component or connection transitions are stored in output features. Components that have subcomponents are modeled by merging the component transition with the subcomponent nets.

If an AADL port is connected to several other ports at a time, the Petri net transition shall be connected to all the corresponding places: a token will be sent to each target place, thus modeling the fact that each destination port receives the output of the initial port.

Call sequences are made of subprograms that are connected. We use an extra token to model the execution control. There is a single execution control token in each thread

or subprogram, thus reflecting the fact that there is no concurrency in call sequences, and in threads and subprograms in general.

It is important to note that this mapping only provides a solution to transform AADL construction into Petri nets. Therefore it cannot produce accurate description of the behaviors of the components, since it is out of the scope of the AADL. Proper behavioral description is achieved by inserting existing Petri nets into the framework generated from the AADL description. It consists of merging the descriptions of the components and the net generated, thus merging the transitions and places of the AADL threads and subprograms with the ones contained in the behavioral Petri net. The Petri net descriptions that corresponds to the behaviors of the AADL components should be set using AADL properties.

Defining a mapping between AADL constructions and Petri nets allows to perform verification on the structure of the architecture. Yet, it is mandatory to ensure the actual source code of the system will conform to the Petri net. This implies that the mapping between AADL and programming languages must be consistent with the Petri net mapping.

To ensure this consistency, the mapping we provide for source code relies on the same principles as for Petri nets [VZ06]. We only only give a very brief and incomplete overview of it in table 1. The source code mapping is basically a translation between the AADL subprogram constructions and Ada. Using both mappings in conjunction ensure that the Petri net used for the model checking of the AADL architecture effectively reflects the actual source code implementation of the architecture.

4.5 Using AADL to generate the middleware

We showed how the AADL and the definition of mappings from AADL to formal notations allow us to define a prototyping-based process of DRE system conception.

The initial AADL description can then be refined, according to the feedback provided by the model checking performed on the Petri nets. Once the behavior has been validated, we can generate the corresponding source code and then perform tests on the actual system. The AADL architecture can again be refined, according to the results of the tests.

In order to validate our approach, we created a complete AADL tool suite, Ocarina [VZ06], which can be used as a compiler for the AADL. As a support tool for verifying AADL model, Ocarina can take AADL descriptions as input and perform various operations, such as the expansion of architectural descriptions or the generation of Petri net description as well as compilable source code. It can also be integrated within other applications to provide AADL functionalities.

The code generator of Ocarina can produce Petri net models described in PetriScript [HR]. PetriScript is a text language that facilitates the description of Petri nets and allows to automate building operations, such as fusion of places or transitions, etc.

Ocarina can generate Ada source code that can be run by an instance of PolyORB. It also generates a tailored application personality and configures PolyORB to embed all the required features. We use PolyORB as an AADL runtime and allows one to build distributed applications defined as an AADL model.

Hence, Ocarina helps us to support the generation of tailored middleware, as illustrated on figure 5: from the AADL description of a distributed application, we can infer the description of the middleware instances for each application node, and then produce the corresponding Petri net and source code.

5 Conclusions and Perspectives

Although middleware is now a well-established technology that eases the development of distributed applications, many challenges remain opened. We noted that two key issues are the tailorability of the middleware to versatile application requirements, and the capability of the middleware to provide full proofs of its properties. In this paper, we provided an overview of our ongoing research work on these two aspects.

We first noted that middleware architecture impedes tailorability and verification. Therefore, we proposed and validated the "schizophrenic" middleware architecture. This architecture is a high-level model of middleware that gathers key concepts in middleware, addressing the definition of the key functions and the way to combine them.

Its genericity allows one to derive specific distribution models. PolyORB, our implementation demonstrates how this architecture can help designer to easily build middleware. This middleware is now used as a COTS in industrial projects, providing support for CORBA, DDS and still providing a high level of tailorability.

A methodological guide exists to help this adaptation work. Our measures show that the performance of the adapted middleware are close to existing middleware. Besides, the adaptation work is greatly reduced by the high-level of code reuse.

Finally, the schizophrenic architecture allows formal verification techniques. We illustrated how Petri nets allowed us to provide the first formal proofs of the behavioral properties of our COTS middleware. We consider that The middleware is not a blackbox that should be discarded from the verification process.

However, this remains a complex task that belongs to middleware or verification expert domains. Then, we noted that tools are required to conduct these two important steps in building tailored middleware.

We chose the AADL as a backbone language to help the user specify its application requirements. Dedicated tools are applied to the model to 1/ verify it is correct, 2/ generate the corresponding code and configuration of the support middleware. This provides a first step towards the definition of a "middleware factory" that would enable application designers to instantiate the middleware they actually need. This would reduce complexity in the design of distributed applications by removing the complexity in configuring and using middleware APIs.

Future work will complete and evaluate the benefits of such middleware factory as a supporting process to build specific middleware configuration for DRE systems.

References

[AdSSK03] L. Apvrille, P. de Saqui-Sannes, and F. Khendek. TURTLE-P: Un profil UML pour la validation d'architectures distribuees. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP)*, Pparis, France, October 2003. Hermes.

[BHI04] Soheib Baarir, Serge Haddad, and Jean-Michel Ilié. Exploiting Partial Symmetries in Well-formed nets for the Reachability and the linear Time Model Checking Problems. In *Proceedings of the 7th Workshop on Discrete Event Systems (WODES'04)*, Reims, France, septembre 2004.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, New York, 1996.

[BSPN00] R. Bastide, O. Sy, P. Palanque, and D. Navarre. Formal specifications of corba services: Experience and lessons learned. In *Proceedings of the ACM Conference on Object-Oriented Programmng, Systems, Languages and Applications (00PSLA'2000)*, Minneapolis, Minnesota, USA, 2000.

[Bud03] T. J. Budden. Decision Point: Will Using a COTS Component Help or Hinder Your DO-178B Certification Effort. *STSC CrossTalk, The Journal of Defense Software Engineering*, November 2003.

[CDFH91] 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.

[DDH+03] William Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Robby, and Gurdip Singh. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, March 2003.

[DHTS98] B. Dumant, F. Horn, F. Dang Tran, and J-B. Stefani. Jonathan: an open distributed processing environment in java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 1998.

[FLV00] P. H. Feiler, B. Lewis, and S. Vestal. Improving predictability in embedded real-time systems. Technical Report CMU/SEI-2000-SR-011, université Carnegie Mellon, December 2000. http://la.sei.cmu.edu/publications.

[HKP06] Jérôme Hugues, Fabrice Kordon, and Laurent Pautet. A framework for DRE middleware, an application to DDS. In *Proceedings of the 9th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'06)*, pages 224–231, Gyeongju, Korea, Avril 2006. IEEE.

[HR] A. Hamez and X. Renault. *PetriScript Reference Manual*. LIP6, `http://www-src.lip6.fr/logiciels/mars/CPNAMI/MANUAL_SERV`.

[ISO94] ISO. *Quality management and quality assurance - vocabulary*. ISO, 1994. ISO 8402:1994.

[Jon94] Bengt Jonsson. Compositional specification and verification of distributed systems. 1994.

[KP04] M. Kaddour and L. Pautet. A middleware for supporting disconnections and multi-network access in mobile environments. In *Proceedings of the Perware workshop at the 2nd Conference on Pervasive Computing (Percom)*, Orlando, Florida, USA, March 2004.

[KP05] F. Kordon and L. Pautet. Toward next-generation toward next-generation middleware? *IEEE Distributed Systems Online*, 5(1), 2005.

[LAA04] LAAS. The RT-LOTOS Project, 2004. `http://www.laas.fr/RT-LOTOS`.

[Lew03] B. Lewis. architecture based model driven software and system development for real-time embedded systems, 2003. available at `http://la.sei.cmu.edu/aadlinfosite/LinkedDocuments/`.

[MPY+04] Atif Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas C. Schmidt, and Bala Natarajan. Skoll: Distributed Continuous Quality Assurance. In *Pro-*

ceedings of the 26th IEEE/ACM International Conference on Software Engineering (ICSE), Edinburgh, Scotland, May 2004.

[PSCS01] I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and Optimizing Thread Pool Strategies for RT-CORBA. In *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*. ACM, 2001.

[RGS95] R. Rajkumar, M. Gagliardi, and L. Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proceeding of the 1st IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, USA, May 1995.

[RKK06] A.-E. Rugina, K. Kanoun, and M. Kaâniche. Aadl-based dependability modelling. Technical Report 06209, LAAS-CNRS, apr 2006.

[RSRS99] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schürr. UML + ROOM as a standard ADL? In *Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems*, 1999.

[SAE04a] SAE. Aadl, annex d: Language compliance and application program interface. available at `http://www.sae.org`, sep 2004.

[SAE04b] SAE. Architecture Analysis & Design Language (AS5506). available at `http://www.sae.org`, sep 2004.

[SB03] D.C. Schmidt and F. Buschmann. Patterns frameworks and middleware: Their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineer ing*, 2003.

[SLM98] D. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers. *Computer Communications*, 21, april 1998.

[Sou89] Y. Soussy. Compositions of Nets via a communication medium. In *10th International Conference on Application and theory of Petri Nets*, Bonn, Germany, June 1989.

[TMDM03] Y. Thierry-Mieg, C. Dutheillet, and I. Mounier. Automatic symmetry detection in well-formed nets. In *Proc. of ICATPN 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 82–101. Springer Verlag, juin 2003.

[VHPK04] 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)*, Palma de Mallorca, Spain, June 2004.

[VZ06] T. Vergnaud and B. Zalila. Ocarina: a Compiler for the AADL. Technical report, Télécom Paris, 2006. available at `http://ocarina.enst.fr`.