Revisiting COTS middleware for DRE systems

Jérômes HUGUES, Laurent PAUTET
GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
jerome.hugues@enst.fr, laurent.pautet@enst.fr

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

Abstract

Distributed Real-Time Embedded systems (DRE) increasingly rely on COTS middleware to meet their distribution needs. Yet, there is a technology gap between the design of COTS middleware and the high-integrity constraints of real-time engineering. This puts a limit on the adoption of middleware by system families such as space or avionics. In this paper, we present our current work on the "schizophrenic middleware architecture", a highly tailorable middleware architecture, and its implementation PolyORB. We illustrate how it allows for support of real-time engineering guidelines, enforces determinism, allows for modeling and verification.

1 Introduction: Middleware & DRE systems

Several projects, such as Bold Stroke [24], evaluate the use of "Commercial Off-The-Shelf" (COTS) middleware to build Distributed Real-Time Embedded systems (DRE) for space or avionics applications. Since there is no "one size fits all" middleware, the choice of a particular middleware and its configuration are key design issues that may dramatically impact the design and behavior of an application. Such an impact is difficult to evaluate by non middleware experts. So, there is a need to tailor the middleware and then evaluate its properties.

The IST ASSERT project[1] aims at developing new system engineering methods for DRE systems. It focuses on advanced technologies such as system families architectures, use of formal methods to proof system properties, and reuse based on building blocks that can be composed, tailored and verified in open frameworks. As a milestone, the ASSERT project defines requirements for middleware to be integrated in a full engineering process that ensures and preserves the properties of the system. As an illustration, the European Space Agency defines the following scenario for middleware: ground stations interacting with satellites as well as fleets of collaborating satellites or drones; and require multiple distribution mechanisms to handle variations in communication channels, flexible resource management, and to ensure autonomy for long missions. Thus, DRE systems must address simultaneously Distribution, Real-Time and Embedded issues. We note the following:

Distribution cannot remain hidden from the developer. The semantics of the distribution models must be adapted to real-time application needs; communication channels may support some form of Quality of Service (QoS). Their impact on the timeliness of the application must be measured.

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 [3]. Finally, a methodological guide and support tools help to tailor the middleware with respect to the application requirements.

Embedded targets have strong constraints on their resources (e.g. CPU, memory) 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.

Hence, there is a need for COTS middleware targeting DRE systems *1)* that supports an ever-increasing set of functions; *2)* whose properties are verified or validated; *3)* that can be deployed on targets with limited resources.

In the remainder of the paper, we present our current

[1] ASSERT is part of the Sixth Framework Programme IST of the European Union, see http://www.assert-online.net

work on middleware architecture. We first review existing middleware architectures, and note they partially support the requirements of DRE systems. Then, we present the schizophrenic middleware architecture as a solution to address these requirements in one unified and tailorable architecture. Finally, we present how we take advantage of its architecture to support the requirements of DRE systems; and we discuss tool supports to assist the construction and deployment of tailored and verified middleware.

2 Middleware architectures for DRE systems

In this section, we review existing middleware implementation and assess their support for DRE systems needs.

2.1 Existing COTS middleware

DRE systems may be classified into *system families*, such as space, avionics, automotive. Each family lists generic requirements and common APIs along with their semantics. Domain-specific application execution environment have been defined, e.g. OSEK/VDX [6] for automotive applications, or ARINC 653 [1] for avionics. They define some primitives to support communications on top of industrial buses such as CAN or MIL-STD 1553, and low-level primitives for distribution.

Distribution mechanisms have been defined to meet wider requirements: Rajkumar et al. [20] advertised Message Passing as a solution for DRE systems, and proposed the Real-Time Publisher/Subscriber service. RT-CORBA extends CORBA's Distributed Object mechanisms for real-time systems and integrates support for many QoS policies [21]. Besides, we claim that COTS middleware should not be restricted to one service or distribution semantics.

Middleware support one or more distribution mechanisms, and propose a set of services and configuration mechanisms to help in the construction of DRE systems. We review some of them, indicating their primary focus:

Reduced footprint: OSA+ [23] is geared towards microcontroller as target platform, it is designed so that it has a reduced memory footprint (around $60kB$). It is built around a few services that provide key middleware mechanisms, and a core, following a micro-kernel like approach. It supports an event-based distribution semantics. However, OSA+ lacks advanced services to support complex distribution needs, e.g. RPC-like semantics, or multiple protocols.

Fine-grained optimizations: the ACE framework and TAO [22] forms a complete implementation of CORBA and RT-CORBA. They rely on a set of fine-grained optimizations to ensure the determinism of key functions of the middleware. However, this is a local property of one component of the middleware; it is hard to complete the analysis of the timing of TAO as a whole because of the implied complexity of its object-oriented patterns.

Configuration and standards: Zen [15] is an implementation of the RT-CORBA specifications, on top of Real-Time Java. It relies on the design patterns and lessons learned from the ACE and TAO projects. Zen relies on multi-layer plug-ins so that unneeded components can be removed from the middleware instance: this reduces memory footprint. Such a modularity is interesting; but it also increases the cost of development.

COTS reuse: nORB [25] demonstrates the benefits of designing middleware in a bottom-up approach, selecting only the required components from several COTS: the ACE framework, the Kyokyu scheduler, etc. However, this approach is limited to the integration capabilities of many (and possibly heterogeneous) components, and requires extensive knowledge on each COTS. This impedes the benefits of this approach.

Vertical integration: CosMIC [10] proposes an integrated development framework to build and deploy distributed applications. It proposes a CASE tool to build DRE systems, using TAO and CIAO as building blocks. Yet this approach drags a lot of code: each building block is more that $100'000$ SLOCs. This leads to an exponential code size growing when integrating all of them. This leaves many issues for memory-constrained systems, or system properties verification and/or validation unsolved.

These projects focus on very specific implementation details of a middleware for DRE systems: resources consumption, configurability. They extend non-real time distribution mechanisms with QoS parameters; they add constraints on a non-deterministic middleware implementation so that it can meet real-time deadlines. This is a complex design in contradiction with guidelines for high-integrity systems: it is difficult to assess determinism or to evaluate precisely the Worst Case Execution Time (WCET) of request processing.

Moreover, the implementations have a significant runtime cost, making it unsuitable for high-integrity systems, or it cannot be deployed on small targets, e.g. the object model of CORBA. Or they lack generality as for OSA+, or focus on only one distribution semantics, usually Distributed Object Computing.

2.2 Towards a tailorable COTS middleware

We now sketch the requirements for a new middleware architecture for DRE that can meet a larger spectrum of requirements and still follow DRE guidelines.

R1) assessment: the middleware design should support a comprehensive validation and/or verification process at an affordable cost, allowing for the complete determination of the middleware properties. This is a key milestone to support deterministic and certifiable distributed applications;

R2) customizability: middleware should allow for extreme tailorability of its key components, at a limited cost;

R3) scalability: the middleware architecture should be scalable, i.e. supporting both limited distribution mechanisms up to full-featured and complex ones;

R4) resource preservation: the cost of a new feature, or its withdrawal should have a limited penalty on the overall middleware design, and on the resources used;

R5) standard support on demand: the middleware should support main industrial specifications and standards for both the targets (real-time kernels, hardware, etc) and the middleware functions it proposes. It should also support ad hoc APIs for restricted target.

Thus, we propose the "schizophrenic middleware" as a comprehensive middleware architecture to meet those heterogeneous requirements. As an additional requirement, this architecture should later serve as a base for modeling to ease middleware tailorability, verification and deployment.

3 Middleware architecture for DRE

We first present the schizophrenic middleware architecture, then we present its key component: the μBroker.

3.1 Decoupling middleware components

Middleware combines two complementary facets: (1) a framework to implement distributed systems, using the host and operating system resources (e.g. tasks, I/O); and (2) a set of services to build portable distributed applications.

In [12], we introduced the "schizophrenic" middleware architecture: a unique architecture that advertises these two aspects, and enforces separation of concerns. "Personalities" are plugged into a "Neutral Core Middleware". Each personality implements one set of functions related to a distribution model: e.g. CORBA, SOAP. The Neutral Core Middleware can support multiple interacting personalities in one instance, hence its "schizophrenic" nature.

In [27], we present PolyORB our implementation of a schizophrenic middleware. PolyORB a free software middleware supported by AdaCore[2], PolyORB's research activities are hosted by the ObjectWeb consortium[3]. We assess its suitability as a middleware platform to support multiple specifications (CORBA, Ada Distributed Systems Annex, Web Applications, Ada Messaging Service close to Sun's JMS) and as a COTS for industry projects.

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. 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.

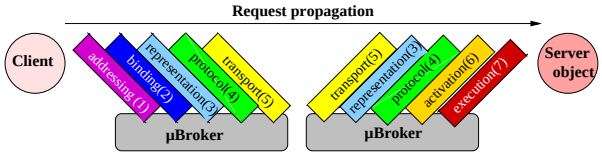

Figure 1. Request propagation in the schizophrenic middleware architecture

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.

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

3.2 µBroker: core of the middleware architecture

The µBroker component is the core of the PolyORB middleware. It is a refinement of the Broker architectural pattern defined in [4]. 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 shall cooperate 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 [19], the authors present different request processing policies implemented in TAO; the CARISM project [14], 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.

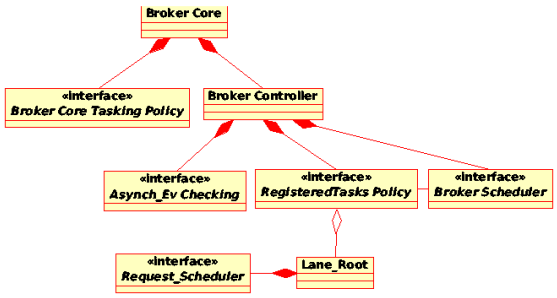

Figure 2. Overview of the µBroker

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.

Thus, the schizophrenic middleware architecture proposes a comprehensive view of one middleware architecture. This architecture is defined around a set of canonical components, one per middleware's function.

4 Schizophrenic architecture into action

The schizophrenic middleware architecture provides guidelines to build middleware. We present several experiments that are answers to the requirements of DRE systems we identified: 1) verification of PolyORB behavioral properties, 2) implementation following high-integrity engineering guidelines, 3) low memory footprint configuration and 4) support for the RT-CORBA specifications. These different experiments are made from the same middleware core implemented by PolyORB, viewed as COTS middleware.

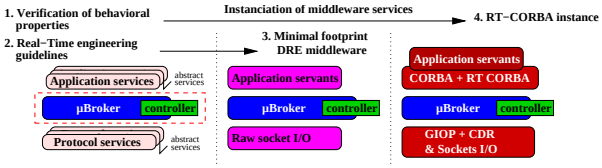

Figure 3. Adapting PolyORB to DRE

4.1 Verification

In [13], we detail how we use formal methods to model and verify our system. This is a preliminary work for the definition of a proof-based middleware, and an answer to requirement *R*1 *assessment* listed in section 2.2.

We selected *Well-formed coloured Petri nets* [5] 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 modeled some configurations of the µBroker.

The modularity of PolyORB, and the clear separation between middleware services (denoting functional components) and the µBroker (controlling the middleware behavior) provides guidelines to model our architecture: the mod-

eling of the μBroker allows us to verify the behavioral properties of our middleware.

We studied two configurations of the μBroker: *Mono-Tasking* (one main environment task) and *Multi-Tasking* (multiple tasks, using the Leader/Followers policy [19]). We tested for three properties expressed in LTL formulae: $P1$, *(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.

We justify the use of formal methods by the following: $P1$, $P3$ are difficult to validate 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 leading to a combinatorial explosion. The correct dimensioning of static resources ($P2$) is a strong requirement for DRE systems, yet it is a hard problem for open systems such as middleware.

Instead, formal methods provide a clear evidence of their validity: Petri nets analysis methods first build the full state-space of the system, then explore it to check the validity of a property. We note the size of the state space for the multi-tasking model (figure 4) increases exponentially with the number of threads and I/O sources: this denotes the complexity of the semantics of the system. We tackle combinatorial explosion by detecting the symmetries of a system [26], and exploiting the symmetries allowed by a property [2]. These methods are provided by the CPN-AMI CASE tools [16]. They build a *symbolic state space*, a quotient state space, in which nodes are equivalence classes of states, and arcs equivalence classes of events. In most favorable cases, the symbolic state space we analyze is exponentially smaller than the concrete state space, and thus more amenable to computations. Indeed, for our models, we note that the symbolic reachability graph is smaller by several powers of ten, and that this ratio increases with the number of threads. Thus, even for big configurations, the quotient state space remain within affordable bounds that allow for complete verification of $P1$, $P2$, $P3$.

For a typical middleware configuration (7 threads and 4 I/O sources), the system has more than 10^{11} states. We computed and evaluated its properties on the quotient reachability graph. Computations were completed in less than 10 hours for the biggest models, on a 2.6 GHz Pentium-4 computer with 512MB of memory, without swapping. These tools allows us to complete the analysis of our architecture, and provide a solution to verification needs of DRE systems.

4.2 Real-Time engineering

At the implementation level, we enforce the use of selected algorithms, patterns and run-time ensure properties of each building block of the middleware, and thus makes PolyORB deterministic. It corresponds to requirement *R2 customizability*.

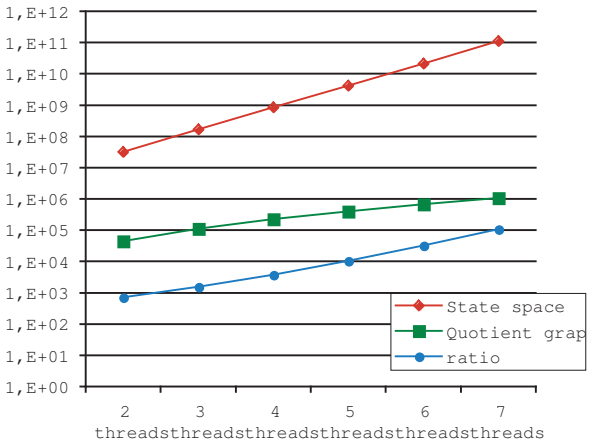

Figure 4. μBroker's state space

The Ravenscar profile [8] is a subset of the tasking and concurrency constructs of Ada 95. It has been designed so that the restricted form of tasking it allows can still be used in high-integrity programs and be certified. Basically, it explicitly forbids any dynamic behavior (dynamic priority change, task creation/destruction), and enforces scheduling techniques that allows for full schedulability analysis based on the Rate Monotonic Analysis (RMA).

As part of its configurability options, PolyORB implements one set of concurrency constructs compliant with the Ravenscar profile. This allows us to ensure the determinism of the tasking and concurrency elements we use.

In section 3.1, we classified middleware functions either as being involved in the control of middleware, such as the configurable μBroker component and concurrency patterns; or as middleware functional-only components (protocol, transport, activation, etc). We also reduced functional components of middleware to classical abstractions: pipes, filters, dictionaries, factories, finite-state automata.

PolyORB genericity enables the user to select the most adequate implementation of each services, using the previous assumptions as configuration and/or implementation guidelines. In this respect, middleware services can be described using well-known design patterns, that are completely defined and studied, for instance in [9]. There can be made real-time under some assumptions:

1) size of data to be manipulated: size of request parameters, number of application objects; bounds on the number of resources (e.g. threads, requests, I/O sources) to be used during middleware lifecycle. This helps to efficiently dimension resources and preallocate them;

2) memory allocation policies: the use of specific memory allocator (Ada's `Storage Pools`) to handle any tran-

sient overload in resource usage;

3) a priori knowledge on application entities: such as operation name, servant and POA hierarchy, etc, to efficiently use static and dynamic perfect hash tables [7] and then enable $O(1)$ look up time in dictionaries.

Besides, the configurability capabilities of PolyORB, and more specifically of the μBroker enables the implementation of real-time thread scheduling disciplines, event checking or concurrency strategies, e.g. deriving from existing concurrency policies as presented in [19]. Thus, PolyORB tailorability enables a complete adaptation of the middleware to very specific needs, e.g. real-time requirements

4.3 Limited footprint and runtime needs

DRE systems usually come with some restrictions on runtime capabilities such as limited tasking capabilities, no exception; or constraints on available resources such as CPU or memory. Thus, it is required to limit the resources required by the middleware; and also to control the runtime facilities the middleware depends on. It corresponds to requirements *R*2, *R*3 *scalability*, *R*4 *resource preservation*.

PolyORB extreme modularity enables the user to select only the set of components required for his application. The μBroker provides the minimal set of components to build a functional middleware: it provides all the circuitry to wait on I/O sources, store and dispatch incoming requests. The user may then add its own components to register its application entities (e.g. call back functions to process requests), type marshaling functions and protocol stacks.

The size of a minimal middleware built with PolyORB is about $12'000$ SLOCS, representing a memory footprint of $400kB$ for no-tasking configuration; and $13'000$ SLOCS and $500kB$ for a full-tasking one. The Ada runtime provided by the GNAT Pro 5.02a1 Ada compiler, on a GNU/Linux x86 target adds an overhead of $200kB$ up to $250kB$. As a comparison, OSA+ requires $60kB$, TAO+ACE requires more than $2MB$, and nORB+ACE approximately $340kB$.

Let us note this is a current work in progress: the whole code is compatible with the Ravenscar profiles, several components are pluggable (e.g. concurrency primitives, static and dynamic configurators). This concurs to reduce memory footprint. Still, many resource consuming components remains, such as types and generic marshalling procedures; more components can be made optional. An aggressive configuration and build process would allow to reduce their footprint. First results are encouraging, and we anticipate that the memory footprint can be reduced by $150kB$ without compromising our architecture.

4.4 Support for standard for DRE systems

So far, RT-CORBA [18] appears to be the dominant standard specifications for distributed real-time systems. We investigate the support of RT-CORBA in PolyORB; and detail benchmarks on PolyORB support for real-time distributed applications. It corresponds to requirement *R*5 *standard*.

The modularity and prototyping capabilities of PolyORB allows us to rapidly design a RT-CORBA implementation. We directly focus on the extension of middleware services introduced in section 3.1 to ensure determinism in request demultiplexing: the *activation* service to ensure a constant-in-time lookup for CORBA servants, the μBroker and *execution* services have been extended to support priorities. Finally, the protocol components supporting the GIOP stack have been extended to support priority transmission.

We experimented PolyORB on Ultra-5 workstations, running Solaris 9, with 128MB of RAM, operating one Ultra-Sparc IIi CPU at 333Mhz. This platform is a good compromise between all-purpose development platform that is easy to operate, and a real-time kernel. Solaris is known to have limited latency and good time properties with respect to its driver and protocol stack. Thus, it provides a first information on PolyORB capabilities to handle real-time application requirements.

1) request processing jitter: we measure the time required to process one CORBA request $1,000$ times, each request "echoes" an unsigned long, in three different configurations: *1) one local test*, client and servers are in the same process, using multi-tasking; *2) distributed tests*, using either multi-tasking, or no-tasking. Dispersion analysis (figure 5) shows that most of the samples are in an interval that is $100\mu s$ wide, representing less than 10% of the duration of one RPC. A few artifacts (less than 10% of measured values) denote the non-determinism of the underlying OS, linked to memory allocation or the jitter of the TCP/IP stack. They are negligible at that stage of the analysis.

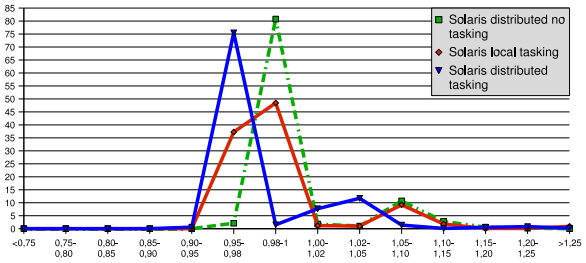

Figure 5. Dispersion of the processing time

2) compliance to RT-CORBA: we test the propagation of prioritized requests, and detect whether priority inversion occur. Three requests, representing respectively *low*, *medium*, *high* level alarms are sent, in random order. We

check that requests are propagated in correct order. An exhaustive analysis of the execution trace using a system logger and display utility shows there is no priority inversion at the level of requests propagation and processing on the server side, ensuring compliance with RT-CORBA.

Thus, the schizophrenic middleware architecture has been successfully adapted to build a real-time middleware, following the RT-CORBA specifications.

These experiments demonstrate that the schizophrenic architecture and its implementation PolyORB provide a solution to the requirements we identified.

5 Towards a "middleware factory"

In this section, we discuss the use of CASE tools to support the construction of verified middleware instances for DRE systems. This is a further step to increase middleware value as a COTS to be safely integrated into an application.

The schizophrenic middleware architecture can be used to build different middleware instances from a set of generic services, with interesting real-time properties. This demonstrates the versatility as well as the scalability of the schizophrenic middleware architecture.

The verification of selected configurations of the μBroker provides confidence in the code written. Yet, modeling is a non-trivial and error-prone work; there may be some differences between the actual system and its model. The proof's scope is limited to the component itself; we must also ensure all properties still hold when aggregating several components.

Nevertheless, our experiments demonstrate the value added of an architecture based on fundamental services, each being restricted to one specific function. We built various middleware instances, from restricted run-time up to full scale standards. We verify its behavioral properties and validated its real-time properties.

We propose to describe our architecture using Architecture Description Languages (ADLs) and their supporting tools to provide an integrated solution to the design and verification of tailorable middleware.

The *Architecture Analysis & Description Language* (AADL) [17] has been selected to serve as a foundation for the ASSERT project to model and then buid DRE systems. The AADL derives from the MetaH language [11]. The AADL allows for the modeling of the hardware and software components of a system; support tools verify this model for consistency, including scheduling analysis, resistance to faults or code generation.

We are currently developing Ocarina[4], a set of libraries to exploit AADL models. PolyORB and its architecture define a set of building blocks that can be modeled, and support a formal verification process. Several implementations of these building blocks have been made, each of which respond to specific needs. Combining the AADL and our work leads to the definition of a *middleware factory*, which create verified and precisely adjusted middleware instances.

Hence, to build customized proof-based middleware for DRE targets, we propose to: *1) model middleware entities*, using the AADL; *2) tailor components* to meet precisely the application needs and semantics; *3) assemble* components to form one middleware instance; *4) verify* its consistency with respect to family-specific guidelines and expected properties; *5) deploy* the application.

We are currently modeling our middleware using the AADL and Ocarina. The comprehensive description of our architecture leads to a one-to-one mapping between software components and their AADL models. The many levels of tailorability of PolyORB provides many hints on how to model a tailorable middleware. The existing know-how on MetaH tools, inherited by AADL, make us confident that the steps 3 to 5 defined above can be fulfilled.

Thus, the completion of this work will enable the integration of new software or hardware components as AADL components, their precise configuration, and finally the verification of the middleware instance. Compared to CosMIC presented in section 2.1, this approach ensures the properties of all components from hardware up to application components, including the middleware.

6 Conclusion

This paper discusses the definition of COTS middleware that address still unsolved issues of real-time engineering.

First, we presented existing middleware for DRE systems: multiple architectures support DRE functional and non-functional requirements. They usually focus on some issues of DRE engineering (e.g. footprint or QoS), but they do not address other critical issues like the complete verification of deterministic behavior.

We identify five strong requirements for DRE systems: 1) assessment of the middleware properties; 2) tailorability to meet application needs; 3) feature scalability; 4) resource preservation; 5) provision to support standards.

We presented the schizophrenic middleware architecture, and show it fits these requirements. The schizophrenic architecture emphasizes on the separation of concerns in middleware: a set of fundamental and tailorable services covers middleware functional components, a middleware main loop - the μBroker- coordinates them.

Then, we illustrated how this architecture can meet the requirements we identified. It enables the verification of properties. It follows real-time engineering guidelines. It targets small runtime and scales up to support standard specifications. PolyORB, our implementation has a formally assessed design; it can be adapted to real-time con-

[4] `http://eve.enst.fr/ocarina`

straints. Thus, we demonstrated how our architecture can be adapted to most stringent requirements of DRE systems, and to a large class of application domains. This is a milestone for the construction of COTS proof-based middleware for critical systems used in space or avionics.

Finally, we discussed the combination of our architecture and the Analysis Architecture & Design Language and CASE tools to foster the construction of a "middleware factory". Its goal is to combine middleware components to form one unique middleware that precisely meet application requirements, and is verified.

Later work will complete the adaptation and modeling of our middleware architecture to serve as a middleware for the ASSERT project; and concentrate of the definition of AADL CASE tools to exploit it. This will demonstrate its full compliance to a proof-based engineering process.

References

[1] ARINC. 653-1 Avionics Application Software Standard Interface, 2003.

[2] S. Baarir, S. Haddad, and J.-M. 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, Sept. 2004.

[3] 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*, Nov. 2003.

[4] 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.

[5] 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.

[6] O. Comitee. OSEK/VDX Home Page, 2004. `http://www.osek-vdx.org/`.

[7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, Aug. 1994.

[8] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, Nov. 1998.

[9] B. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-time systems*. Addison Wesley, Massachusetts, 2002.

[10] A. Gokhale, B. Natarjan, D. C. Schmidt, A. Nechypurenko, N. Wang, J. Gray, S. Neema, T. Bapty, and J. Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, Nov. 2002.

[11] Honeywell. *MetaH user's manual*, october 1998.

[12] 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.

[13] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Linz, Austria, Sept. 2004. To be published in Electronic Notes in Computer Science.

[14] 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.

[15] R. Klefstad, D. Schmidt, and C. O'Ryan. Towards highly configurable real-time object request brokers. In *Proceedings of ISORC'02*, pages 437–447, 2002.

[16] F. Kordon and E. Paviot-Adet. Using CPN-AMI to validate a safe channel protocol. In *Proceedings of the International Conference on Theory and Applications of Petri Nets - Tool presentation part*, Williamsburg, USA, June 1999.

[17] B. Lewis. Architecture based model driven software and system develoment for real-time embedded syste ms. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical innovations of software and systems engineeiring in the future*, volume 2941/2004, pages 249–248. Springer-Verlag, october 2002.

[18] OMG. *Real-Time CORBA Specification, dynamic scheduling, v2.0*. OMG, Apr. 2003. OMG Technical Document formal/2003-11-01.

[19] 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.

[20] 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 First IEEE Real-Time Technology and Applications Symposium*, Denvor, Colorado, USA, May 1995.

[21] D. Schmidt and F. Kuhns. An overview of the real-time CORBA specification. *IEEE Computer*, 33(6):56–63, 2000.

[22] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, Apr. 1998.

[23] E. Schneider, F. Picioroaga, and A. Béchina. Distributed Real-Time Computing for Microcontrollers - The OSA+ Approach. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 169. IEEE Computer Society, 2002.

[24] D. C. Sharp. Reducing avionics software cost through component based product line. In *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[25] V. Subramonian, G. Xing, C. D. Gill, and R. Cytron. The Design and Performance of Special Purpose Middleware: A

Sensor Networks Case Study, 2003. Technical report of the University of Washington Saint Louis 2003, # 6.

[26] 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, June 2003.

[27] 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.