Using the AADL to describe distributed applications from middleware to software components

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

[1] GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
thomas.vergnaud@enst.fr, laurent.pautet@enst.fr
[2] 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 (DRE) systems require the verification of their properties to ensure both reliability and conformance to initial requirements. Architecture description languages (ADLs) such as the AADL provide adequate syntax and semantics to express all those properties on each component of a system. DRE systems rely on a key component, the middleware, to address distribution issues. In order to build efficient and verifiable systems, the middleware has to be tailorable to meet the application needs, and to be easily modeled to support a verification process. We propose the schizophrenic architecture as a canonical solution to these concerns. We study how to describe the middleware architecture using the AADL. We also study how the AADL can be used to aggregate the different aspects of the modeling of a complete system: architecture, behavioral descriptions, deployment, etc.

1 Introduction

Distributed systems are widely used in various application domains such as embedded systems, business applications or web applications. Distribution has to address different requirements, either related to system constraints (execution time, memory footprint, limited bandwidth. . .) or related to the application design (reuse of legacy components, programming languages heterogeneity. . .).

An application can take advantage by the reuse of COTS to cut down development costs. Besides, architecture description languages (ADL) can capture the design of a complete application and of its key components. As they allow for a more abstract view of the application than programming languages, they help in identifying the structural components, and eventually expressing properties on the whole architecture. Large projects rely on an ADL to design embedded systems. In addition to architectural considerations, attention is focused on property verification to assess system reliability.

The ASSERT project[3], coordinated by the European Space Agency and the European Union, chose the Architecture Analysis & Design Language (AADL) as a support for modeling. The AADL is targeted to the description of real-time embedded systems.

[3] http://www.assert-online.org

It is based on component composition and provides very convenient facilities to specify properties on the architecture. AADL descriptions are tightly related to the implemented systems they represent, and the AADL provides support for system generation.

Distributed systems rely heavily on middleware to handle a large part of the distribution issues [1]. Compliance to constraints can only be verified once the system has been completely designed; in order to ensure property verification, the middleware has to be modelled as well as the other system components.

As a middleware is a complex piece of software, modelling it with an ADL may be a very tedious task. Moreover, middleware usually implements a given distribution model like CORBA [2], DSA [3] or Web Services [4]. Middleware may have a very different architecture depending on the distribution model it implements. To overcome these issues, we propose to focus on a middleware architecture which would be representative of most middlewares.

Schizophrenic middleware allows to instantiate a generic middleware for one or more distribution models. In other words, such a highly tailorable middleware can be adapted to meet the exact application requirements. In [5], we presented PolyORB, an implementation of the schizophrenic architecture.

The schizophrenic architecture can be decomposed into several well identified components that can be analyzed. Thus it eases the modeling of middleware; its clear structure facilitates its description using an ADL. This can ease property verification, configuration and deployment of an application.

Our long-term objective consists of extracting general properties from distributed real-time embedded (DRE) systems. In this paper, we aim at studying the ability of the AADL to describe such systems from middleware to application components. As a case study, we especially focus on the middleware, which represents the core component of a DRE system. Because of its clear structure and its versatility, the schizophrenic middleware architecture is a good candidate to evaluate AADL modelling capabilities.

This paper is structured as follows. We first give an overview of the AADL and its main features. We then describe the schizophrenic architecture and explain why it is a viable choice to model middleware. We give some elements on how to describe the architecture of a system based on a schizophrenic middleware using AADL. We finally study how the AADL can be used to federate all the aspects of a system description.

2 Modeling the architecture using the AADL

The AADL is an evolution of MetaH, [6] and thus they share many common features. The AADL aims at allowing for the description of DRE systems by assembling blocks developed separately. Thus, it focuses on the definition of clear block interfaces [7], and separates the implementations from those interfaces.

The AADL standard [8] is based on a textual syntax. It also provides a graphical notation. An XML notation [9] is also defined to ease interoperability between tools. It also defines a run-time and how to translate AADL constructions into programming languages [10]. Hence, the structure of an application can be automatically generated.

An AADL description consists of *components*. Each component has an interface providing *features* (e.g. communication ports), and zero, one or several implementations. The implementations give the internals of the component. Most component implementations can have *subcomponents*, so that an AADL description is hierarchical. The components communicate one with another by *connecting* their features. The AADL defines a set of standard *properties* that can be attached to most elements (components, connections, ports, etc.). In addition, it is possible to add user-defined properties, to specify specific description information.

2.1 Components

Basically, an AADL description is a set of components that represent the different elements of the whole architecture. The AADL standard defines software and hardware components; so it is possible to model a complete system.

Execution platform components represent all the components related to the computers and networks that are part of the whole system.

– *buses* are used to describe all kinds of networks, buses, etc;
– *memories* are used to represent any storage device: RAM, hard disk,. . . ;
– *processors* model micro-processors with schedulers: they are the general representation of a computer shipped with a basic operating system;
– *devices* represent components whose internals are not precisely known. Typical examples of such black boxes are sensors: the knowledge is limited to their external behavior and their interface. We do not control their structure.

Execution platform components are mostly hardware components. Yet, components like *devices* or *processors* may have software parts.

Software components allow for the description of pure software elements (no hardware is involved).

– *data* components are used to describe data structures that are stored in *memory* or exchanged between components;
– *threads* are the active components of a software application;
– *thread groups* gather several *threads*, thus allowing to describe a hierarchy among the *threads* of an application;
– *processes* correspond to memory spaces used to execute *threads*. A *thread* must execute within a *process*, and a *process* must have at least one *thread*;
– *subprograms* correspond to procedure calls in imperative programing languages such as Ada or C. They allow to model an entry point in a *thread* or a *data* component (which can be viewed as a class for object oriented languages) or can simply be used to model normal subprograms.

Systems are either used to make high-level descriptions or to add hierarchy in the description. They contain other components, and thus are neither pure software nor pure hardware components. *Systems* describe self-sufficient components. For instance, a *thread* cannot be directly put into a *system*, since a *thread* must be contained in a process.

The AADL introduces the notion of *component types* and *component implementations*. A *component type* corresponds to what is visible from the outside of the component, such as its interface (basically its inputs and outputs); a *component implementation* describes the internals of a component: its sub-components, the connections between them, etc. There can be several different *implementations* of a given *type*. The AADL also allows for the inheritance of *component types* and *component implementations*: a *type* or an *implementation* can extend another one.

Subcomponents are instantiations of component types or implementations, the same way as objects are instances of classes in object oriented languages.

2.2 Ports, subprograms and connections

Components communicate through *ports* and *subprogram calls,* that are provided as features of the component type.

Ports are used to model asynchronous communications:

- *data ports* are associated to a data component. They can be compared to the state of a port in an integrated circuit: the destination component may or may not listen to the data. If not, the information is lost;
- *event ports* can be seen like the signals of an operating system. Compared to *data ports,* they can trigger events in components, but do not carry data. Unlike data ports, a queue is associated with each event port;
- *data event ports* have the characteristics of the two former ports: they can trigger events and carry data. They are typically used to model the communications with message oriented middleware.

Event data ports can be used to model communications based on message passing. *Ports* can be declared *in*, *out* or *in out*.

Subprograms correspond to synchronous calls, like Remote Procedure Calls (RPC) or direct procedure call (as defined in programming languages) and accept *in*, *out* and *in out* parameters; *parameters* are comparable to *data* ports or *event data* ports, but are synchronous and dedicated to *subprograms*.

2.3 Properties and property sets

The AADL defines a set of standard properties. These are used to specify execution deadlines for *threads,* bindings between software and execution platform components, protocols for *connections*, transmission times for *buses*, etc. They can describe all the information required to check the validity of the system, or to complete the description of its architecture.

Property types can be integers, floats, strings or booleans, *component* references or enumerations. Complex data structures such as Ada records or C structures do not exist.

Each *property* name is meant to be applied to some (or all) elements of a description: processors, connections, ports, etc.

Properties can be specified inside elements (e.g. component *types* or *implementations*). They can also be associated to instances of subcomponents. This allows for great flexibility, as a given component implementation can be characterized when instantiated; it is not necessary to specify another implementation.

If a given characteristic does not correspond to a property of the standard set of properties, it is possible to define specific properties, using *property sets*. A *property set* defines a namespace that contains *property types* and *property names*.

2.4 Packages

By default, all elements of an AADL description are declared in a global namespace. To avoid possible name conflicts in the case of a large description, it is possible to gather components within *packages*.

A *package* can have a public part and a private part; the private part is only visible to elements of the same package.

Packages can contain *component* declarations. So, they can be used to structure the description from a logical point of view – unlike systems, they do not impact the architecture.

3 Architecture concerns for distributed applications

Middleware is a fundamental element of a distributed application, as it addresses several distribution issues. Some of them are related to the distributed nature of the application, like the location of the physical nodes. Others are related to each local node, like the execution of the whole application. Some other considerations are related to both local configuration and deployment, like the communication protocol used between the nodes. All these issues can be separated, as shown on figure 1. In this paper, we focus on the local node concerns.

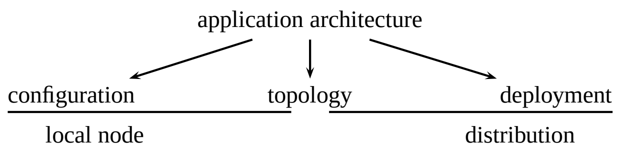

Fig. 1. Principles of distributed application description

3.1 Tailorable middleware architectures

There are two main reasons to design a highly tailorable middleware. First, such a middleware would fit exactly with the application requirements with a reasonable development cost. Second, it could meet the requirements of several system families by being configured for a specific distribution model. Some middleware architectures have been proposed to provide tailorability; for example configurable and generic middlewares.

The main limitation of configurable architectures (e.g. TAO [11]) is that they focus on a given distribution model (CORBA in the case of TAO). They are not efficient enough with applications that do not fit well into this model: An application designed in a Message Oriented Middleware (MOM) approach will not be as efficient if implemented with a Distributed Object Computing (DOC) middleware such as TAO.

The main drawback of generic architectures (e.g. Jonathan [12]) is that the development of a new personality implies the engineering of a significant amount of code. For instance, since Jonathan is mostly based on abstract interfaces, personalities like David (for CORBA applications) and Jeremie (for RMI applications) reuse only 10% of the generic code. Despite the fact that such an architecture could be a good solution to adapt the middleware to application needs, the cost of this adaptation is too high in most cases.

3.2 The schizophrenic architecture

Configurable and generic architectures ease middleware adaptation; they are one step towards middleware modularity. However, they do not provide complete solutions, as they are either restricted to a distribution model, or too expensive. A middleware architecture combining configurability, genericity but also addressing interoperability with other middlewares is needed to support a distribution infrastructure that can be fully tailorable and built from reused components.

This requires an architecture that provides a synthesis of different middleware architectures, and emphasizes the separation of concerns. Such an architecture should be compared to the one adopted in classical compilers: compiler theory describes a flexible architecture, separating machine code generation from source code analysis: a front-end module analyzes source code; a back-end assembles machine code; both of them interact using different neutral representations. Projects like GCC[4] clearly demonstrates component reuse capabilities while providing support for multiple languages and targets.

Similarly, we proposed an original middleware architecture which separates concerns between distribution model API and protocol, and their implementation related mechanisms.

Decoupling middleware functionalities A schizophrenic middleware refines the definition and role of personalities. It introduces *application-level* and *protocol-level* personalities and a *neutral* core layer which are to middleware what front-ends, back-ends and an intermediate layer are to compilers.

Application personalities constitute the adaptation layer between application components and middleware through a dedicated API or code generator; they provide services similar to those provided by a compiler front-end: translation of high-level constructs into simpler ones. They provide APIs to plug application components with the core middleware; they interact with the core layer in order to allow the exchange of requests between entities.

[4] Free software compiler front-ends and back-ends available at `http://gcc.gnu.org`

– On the client side, they map requests made by client components from their personality-dependent representation to a personality-independent one. This neutral representation is then processed by the neutral core layer; results are translated back from neutral to personality-dependent form.
– On the server side, they receive requests for local objects from the core middleware, assign them to actual application components for evaluation, and return results.

Application personalities can instantiate middleware implementations such as CORBA, the Distributed System Annex of Ada 95 (DSA), the Java Message Service (JMS), etc.

Protocol personalities handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages exchanged using a chosen communication network and protocol; similar to a compiler back-end which transforms intermediate code representation into low level mnemonics. Requests can be received either from application entities (through an application personality and the neutral core layer) or from another node of the distributed application. They can also be received from another protocol personality: in this case the application node acts as a proxy performing protocol translation between third-party nodes. 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. This enables the selection of any combination of application and/or protocol personalities; as the GCC compiler allows the selection of any given front-end/back-end pair.

Fundamental services A schizophrenic middleware requires a flexible implementation and the identification of the functionalities involved in request processing to ease the prototyping of new personalities and their interactions.

Figure 2 describes the main elements of the schizophrenic architecture.

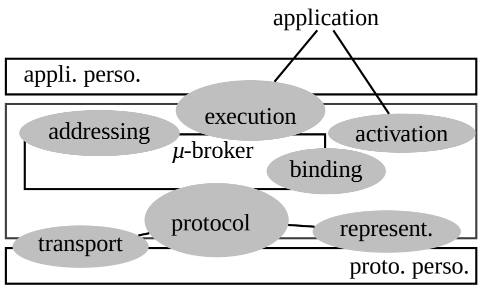

Fig. 2. The schizophrenic architecture

The personalities and the neutral core layer are built on top of seven fundamental services embodying client/server interactions found in the distribution models.

A client personality invokes the **addressing** service to get the reference of the server entity (e.g. an object). The **binding** service then associates a binding object to this reference; a gateway is created between the actual server entity and the surrogate entity on the client side. The **protocol** service calls the **representation** service to format the request data and sends them through the **transport** service. Upon reception on the server side, the **activation** service ensures the targeted entity is available. The **execution** service is then invoked so that the targeted entity actually processes the request. The response is returned using the same mechanism.

The composition of these fundamental services allows for the implementation of different distribution models. The inner part of the middleware core is controlled by a central element named μ-broker, on which the services rely. It is formally described, and supports verification facilities to ensure real-time properties [13].

A distributed application is made of several components, an important one being the middleware. The schizophrenic middleware architecture provides a canonical architecture, made of fundamental services that provide well identified functions. Schizophrenic middleware is versatile enough to instantiate middleware supporting different distribution models. The architecture of the neutral core layer remains unchanged from one instantiation to another. It can ensure various properties regarding real-time requirements. The schizophrenic architecture helps model middleware using a component-based language such as an ADL.

4 Using the AADL to describe a DRE system

We now present elements on the description of the server node of a simple mono-task application. We first describe the middleware architecture. This description is based on PolyORB, our implementation of the schizophrenic architecture, presented in 3.2. We then describe how the middleware part integrates with the other parts of the whole server application: the application itself and the operating system.

4.1 Describing the schizophrenic architecture

Middleware is made of active components (i.e. *threads*) that call reactive components (i.e. *subprograms*). The data exchanged between subprograms or threads are modeled by *data* components.

The middleware architecture mainly consists of the reactive components. Those components model the different middleware parts: personalities and the internals of the neutral core layer. Those parts naturally correspond to AADL *packages*. We cannot use *systems* to structure the architecture into more abstract components: the AADL syntax does not allow *subprograms* to be subcomponents of *systems*.

A middleware configuration consists of a selection of the appropriate component implementations for the neutral layer and the personalities.

The seven services of the neutral layer and the μ-broker are represented by distinct packages. Thus we can isolate the different fundamental functions of the neutral layer. The public part of each package should only contain the subprograms that are required for the interconnection with the other elements of the middleware. The auxiliary subprograms are to be placed in the private part of the packages. So the public parts of the packages will contain the data components and the entry points for the services.

Protocol and applicative personalities are not modeled the same way. An application personality can be modeled as a subprogram. This subprogram is to be called by the execution service, which transmits the neutral request. This neutral request has to be translated into the particular data format used by the application. This translation is typically handled by auxiliary subprograms of the personality. The main subprogram of the personality is to be placed in the public part of the personality package, while the translation subprograms should be in the private part.

A protocol personality is actually a combination of three services: protocol, transport and representation. Consequently, a protocol personality may just correspond to a selection of given service implementations. However, in practice, protocol personalities often require specific service implementations. So a protocol personality is typically modeled by a package which contains the required service implementations.

The active part of the middleware is an execution *thread*. The thread receives requests and returns responses using *sockets*. *sockets* are modeled as event data ports, since at the lowest network level, data frames can be actually compared to messages.

Upon the reception of a request, this thread calls the subprograms of the μ-broker. Then the μ-broker will invoke the appropriate services to process the request. The response returned by the μ-broker will be sent back by the thread.

4.2 Describing a complete node

We gave the outline of the description of a schizophrenic middleware architecture using the AADL. In order to be able to perform analysis related to memory footprint or schedulability, we have to completely describe each node of the distributed system. A node is constituted by the application executed on the node, the middleware and the operating system components (cf. figure 3). The hardware part of the system could also be of some interest, but we will not discuss this here, as we focus on the software architecture.

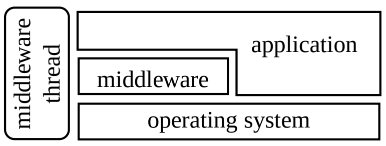

Fig. 3. A server application

The application relies on the middleware and operating system components. Since the application consists of purely software components (i.e. mainly subprograms), it should be described as a package, like the middleware application personalities.

The operating system can be modeled by a set of subprograms. Since a processor component is meant to model both a hardware micro-processor and a minimalist operating system, the entry point subprograms of the operating system may be integrated in a processor component, while the auxiliary subprograms could be located in a package.

The middleware and application subprograms and the threads are instantiated as subcomponents of a process. This process must be bound to the processor which contains the operating system.

5 Discussion

We gave the main lines of the modeling of the server part of a PolyORB-based DRE system using the AADL.

We isolated three main parts: the operating system, the middleware and the application itself. These three parts are independent enough to be treated as separated issues, provided that the interfaces are clearly defined. This allows the separate development of the different parts of the system. For example, in the ASSERT project, PolyORB is to be used on the real-time kernel ORK [14], separately developed. The middleware itself is not represented as a component, but as a set of components defined in packages; this illustrates the fact that the middleware is part of the application, not an independent component. The services of the schizophrenic architecture remain easily identified.

A noticeable aspect of this description is that all AADL packages and components have clearly identifiable Ada counterparts: AADL packages correspond to Ada packages, same thing for subprograms; threads can be compared to Ada tasks. The AADL allows the specification of additional properties, such as execution time, etc. In addition, the AADL allows for the description of hardware components; it provides a unified notation to describe the whole system.

We can see that a software description made with the AADL leads mainly to define *subprograms*. Therefore, a too much detailed description would nearly lead to a direct mapping between Ada procedures and AADL subprograms, which would be useless. The AADL *subprograms* should rather correspond to sets of Ada procedures.

Component implementation should not necessarily be described using programming language: as it is the control part of the middleware, the μ-broker is likely to be modeled using formal methods in order to ensure the reliability of the application.

So, the generation of the whole system shall require intermediate code generation: some components are described using formal methods; others are purely related to the middleware configuration (e.g. the execution service). Other parts of the middleware, such as the personalities, are likely to be written in plain Ada (or any language chosen to implement the subprograms).

Relevant properties, such as the required amount of memory or the processing time could be associated to each component of the description. This would facilitate the verification or the simulation of the whole system.

Besides node generation or analysis, the AADL could also be used to describe the deployment of the whole distributed system. GLADE, an implementation of the Distributed System Annex (DSA), provides a configuration language to partition a distributed system. We are currently working on an AADL model to generalize such an approach for any distribution model and to express the deployment of a distributed system using AADL properties.

Tools are required to process AADL descriptions: perform analysis on the description (schedulability of the whole system, compliance to system constraints…); or to instantiate an executable system, by generating the code for the components, and then linking them to an AADL execution runtime; or to configure and deploy a system, according to its description; or simply to check the syntax and the completeness of the description.

OSATE[5], an open source tool, has been developed for this purpose. OSATE is written in Java and is bound to the Eclipse platform. OSATE is meant to receive *plug-ins* that perform analysis, code generation, etc.

Since we are developing PolyORB in Ada, the fact that OSATE is a Java oriented tool is a drawback for us. As we are experimenting with the AADL, we need complete control on the tools, so that we can study some extensions to the AADL syntax, etc. Thus we are developing our own multi-purpose free software tool in Ada 95: Ocarina[6].

Ocarina is a set of libraries built around a central core. The core provides an API to manipulate and check the semantics of AADL models. We developed a parser/printer for the AADL syntax as described in the revision 1.0 of the standard. Other modules are under development, such as an XML parser/printer to ease the interoperability with other tools (e.g. OSATE). Ocarina will be used for the configuration and deployment tools associated with PolyORB.

6 Conclusion

In this paper, we focused on the modeling of DRE applications. Building DRE applications requires verifications on the architecture. Such verifications are related to quantitative properties like timeliness or memory footprint, as well as properties of reliability (no deadlocks, no starvation, etc.).

We first presented the Architecture Analysis & Design Language. The AADL aims at describing systems as an integration of separate components. Information can be associated to the architectural description, using properties.

We outlined the fact that distributed applications have different and specific requirements. As designing specific middleware to a specific application would cost too much, therefore, adaptable middleware is required that can meet many different requirements. There is a need for fully tailorable middleware which can be verified.

We introduced the schizophrenic architecture as a good solution to middleware tailorability. It relies on a clear separation of middleware functions and can then be structured into different modules; thus it eases modeling using languages such as the AADL. As a large part of a schizophrenic middleware implementation remains unchanged, verification can be performed.

We showed how to describe the architecture of a node of a distributed application. We first described the middleware part and then its integration into the application

[5] available at `http://www.aadl.info`
[6] available at `http://eve.enst.fr/ocarina`

node. The AADL allows for a clear modeling structure. Architectural description and properties provide all the required information to configure a local application node. In addition, the AADL can integrate behavioral descriptions of the components, using either programming languages or formal methods. As additional properties can be defined, the AADL can also be used to describe the deployment of the whole distributed system. Consequently, the AADL can be used as a unification language to aggregate all that is required to entirely describe a DRE system.

References

1. Bernstein, P.A.: Middleware: An archictecture: for distributed system services. Technical Report CRL 93/6, Cambridge MA (USA) (1993)
2. OMG: The Common Object Request Broker: Architecture and Specification, revision 2.2. OMG (1998) OMG Technical Document formal/98-07-01.
3. Pautet, L., Tardieu, S.: GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. In: Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00), Newport Beach, California, USA, IEEE Computer Society Press (2000)
4. W3C: Simple Object Access Protocol (SOAP) 1.1 . (2000) `http://www.w3.org/TR/SOAP/`.
5. Vergnaud, T., Hugues, J., Pautet, L., Kordon, F.: PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In: Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04). Volume LNCS 3063., Palma de Mallorca, Spain, Springer Verlag (2004) 106 – 119
6. Vestal, S.: Technical and historical overview of MetaH. Technical report, Honeywell (2000) available at `http://la.sei.cmu.edu/aadlinfosite/MetaHPublications.html`.
7. Lewis, B.: architecture based model driven software and system development for real-time embedded systems (2003) avilable at `http://la.sei.cmu.edu/aadlinfosite/AADLPublications.html`.
8. SAE: Architecture Analysis & Design Language (AS5506). (2004) available at `http://www.sae.org`.
9. Feiler, P.: Annex A: AADL Model interchange formats. (2004) Part of the AADL standard, available from SAE.
10. Tokar, J.: Annex D: Language compliance and application program interface. (2004) Part of the AADL standard, available from SAE.
11. Schmidt, D., Cleeland, C.: Applying patterns to develop extensible and maintainable ORB middleware. Communications of the ACM, CACM **40** (1997)
12. Dumant, B., Horn, F., Tran, F.D., Stefani, J.B.: Jonathan: an open distributed processing environment in java. In: Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Londres, Springer Verlag (1998) 175–190
13. Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S., Vergnaud, T.: 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 (2004) To be published.
14. de la Puente, J.A., Zamorano, J., Ruiz, J., Fernández, R., García, R.: The design and implementation of the Open Ravenscar Kernel. In: Proceedings of the 10th international workshop on Real-time Ada workshop, ACM Press (2001) 85–90