Modeling and Generating Tailored Distribution Middleware for Embedded Real-Time Systems

Thomas Vergnaud, Irfan Hamid, Khaled Barbaria, Elie Najm, Laurent Pautet, Sylvie Vignes

Modeling and Generating Tailored Distribution Middleware for Embedded Real-Time Systems

A. Thomas Vergnaud[1], B. Irfan Hamid[1], C. Khaled Barbaria1,
D. Elie Najm[1], E. Laurent Pautet[1], F. Sylvie Vignes[1]

1: GET-Télécom Paris — LTCI-UMR 5141 CNRS
École nationale supérieure des télécommunications
46, rue Barrault, F–75634 Paris Cedex, France

Abstract: Distributed real-time embedded (DRE) systems are becoming increasingly complex. They have to meet more and more stringent requirements, either functional or non-functional. Because of this, DRE systems development makes use of formal methods for verification; and, in some cases, generation of proven code. The distribution aspects are typically handled by a middleware, which must meet the system constraints. In this article, we describe our approach to model and generate middleware-based distributed systems for DRE applications. Our methodology is a three-step approach. First, we model the high-level inter-component interactions using connectors. We then use the Architecture Analysis and Design Language (AADL) as a pre-implementation description language to capture all the non-functional aspects of the system. Finally, we generate actual application code and the appropriate middleware from the AADL description. In order to demonstrate the feasibility of our approach, we created an application generator, Gaia. It is part of the Ocarina AADL tool suite and generates application source code for use with the PolyORB middleware.

Keywords: connectors, AADL, Ocarina, middleware, PolyORB, real-time systems

1 Introduction

The design of distributed real-time embedded (DRE) systems is a very difficult task. The designing process must capture many requirements and constraints. Some are functional, such as algorithms to implement; others are non-functional, such as constraints on memory footprint for each node of the distributed application, or transmission times between nodes. All these parameters are specific to each system.

Most distributed systems rely on a middleware to mediate communications. As it is the keystone of the application, the middleware has to satisfy all the requirements regarding both functional and non-functional properties of the application.

The best way to achieve these performance and reliability objectives is to build a specifically designed middleware for each application. A general purpose middleware would drag numerous components that are not needed to perform the specific functions of a given application. Conversely, a dedicated middleware would only embed the needed mechanisms and components. However, it is impossible for cost and maintenance reasons to maintain one middleware per application. Therefore, a *tailorable* middleware [6] is required to ensure flexibility at a reasonable cost.

In order to configure the middleware, the designer has to capture the communication and non-functional application specifications. Semi-formal methods such as architecture description languages (ADLs) [10] typically address the issues regarding the capture process. The entire distributed system could hence be described using abstract semi-formal methods, thus gathering all information regarding the application requirements. The functional aspects of the systems are a result of the capture of the application requirements; they do not imply any assumption on the actual implementation. On the contrary, non-functional aspects are tightly related to the technical solution used to implement the system. Hence, even if these two aspects are related, they can be addressed at different stages of the design process.

The concept of connectors provides a formal support to describe the interactions between the different entities of an application. Since it is a very abstract, high-level notion, it is perfectly suited to capture the mediation functions required by the application. We present a method to model connectors using UML.

The abstract model must be injected into a more concrete semi-formal description; such a lower-level description integrates the deployment parameters and non-functional constraints (location of the nodes and network connections, memory footprint, etc.).

The concrete description constitutes a pre-implementation model: it integrates all the information describing the application. Thus, various

tools can process it, to perform verification, simulation on the architecture etc. ADLs such as the AADL are well-suited for these manipulations.

The concrete description can then be used to generate the actual system, with respect to the requirements primarily described in the connector definition phase. The generated code needs a runtime that provides communication and execution features (threading and tasking). This runtime has to fulfill the requirements captured in the connector and AADL modeling phases. Therefore, it must rely on a highly tailorable and verifiable middleware. The schizophrenic middleware architecture [6] provides a good solution for such a runtime.

In this article we describe our methodology to achieve such a design and generation process. We especially focus on the middleware, as it is the key component of the distributed application. We first describe high-level, abstract modeling techniques that rely on the connector concept. Second, we provide an overview of the AADL, which we use for a pre-implementation, concrete description. We then describe the schizophrenic middleware architecture. This architecture allows the generation of middleware instances as a function of the target application needs; on the other hand, it also allows the reuse of proven components and eases formal verification thereof. We describe PolyORB, our implementation of the schizophrenic architecture. Finally, we explain our approach to automatically generate application source code from AADL descriptions. We describe our application generator, Gaia, which is part of the Ocarina tool suite. It uses PolyORB as a basis for the distribution runtime.

2 Formalizing inter-component interaction

During the '90s, the software engineering research community introduced the concept of *connectors* as elements of the architecture description of a system. Shaw defines a connector as follows [1]:

"Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanisms required"

This is, understandably, an abstract definition; concrete examples of connectors that most software practitioners are familiar with are

1. Remote procedure calls
2. Shared variables
3. Pipes

In the world of ADLs, the main artifact has been and continues to be the component. But, in the last decade, the architecture research community has been emphasizing the need to make connectors *first-class citizens* of the architecture description modeling phase. In [2], Mehta et. al. provide a taxonomy of connectors, classifying them on the basis of the service provided and the type. In [3], the authors introduce the method of using process algebrae to specify connectors. In [4], this approach is extended to allow protocol transformations and composability of connectors.

The literature generally identifies the end-points of the connector that interact with the components as *roles*, the corresponding end-points on components are usually called *interfaces*.

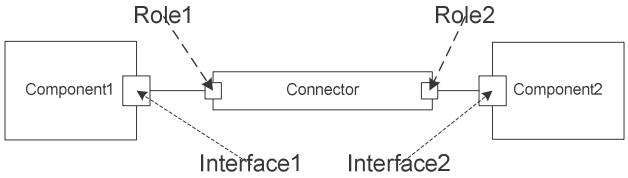

Figure 1: A connector links components together

From figure 1, it is obvious that a connector encapsulates a certain piece of the functionality of a *distribution middleware*. The roles, and their definitions, can serve to specify formally the legal *alphabet* of a connector, i.e.: the operations allowed on a connector and their sequence of invocation.

We define four different *views* of connectors

1. Problem statement view
2. Service view
3. Computational view
4. Deployment view

Each view corresponds to a different abstraction level of the problem. The problem statement, service and computational views are defined using UML, whereas the deployment view is defined using a dedicated ADL. Our primary objective in using the UML to define connectors is to generate code for them, with an eventual transformation towards an ADL. We take the view of assuming that the underlying distribution middleware provides a set of primitive connectors, on top of which we can build more complex ones. In fact, the ultimate aim is to be able to rapidly build complex real-time services (like consensus) on top of an existing middleware.

2.1 Problem Statement View

In this view we are not concerned with how the connector performs its intended function, only what that intended function is. In other words, we are concerned neither with the algorithms involved, nor any artifacts such as interfaces, messages or ports. This view is defined using the UML. We are interested only in describing the effects of the global coordination function of the connector. We can describe these using the OCL (Object Constraint Language), which is a part of the UML.

In figure 2, we represent the class diagram for the problem statement view of a message queue connector (represented as a class). The two roles

(the sender and the receiver) are also represented as classes. The classes of the connector are the two roles and the connector itself, these are adorned with stereotypes of <<role>> and <<connector>> respectively. We will use stereotypes such as these to enhance the UML meta-model [5] with meta-attributes to express properties specific to connectors. Generally, roles will be specified as being an aggregation of the connector in the problem statement view (as shown in figure 2). This implies that the lifetime of the roles may be independent of each other. Note that the message queue connector is simply an example, almost all middleware implementations do provide it as a basic service.

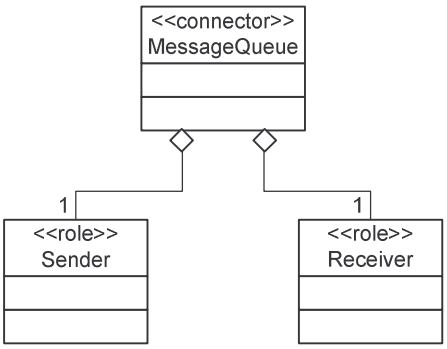

Figure 2: The problem statement view

As an example, the constraints on the message queue connector might specify that messages handed over by the *sender* role must eventually be delivered to the *receiver* role. Or that the receiver role hands messages over to its associated component following a certain priority scheme rather than in simple FIFO order. Generic OCL does not support timing constraints. But temporal extensions to OCL, such as those proposed in [8], could be used to describe real-time properties of the connector in the problem statement view.

2.2 Service View

The service view is where we will describe the connector as a monolithic UML entity. The connector is described as a single UML class in this view, with the roles being represented as *ports* on that class. The behavior of the connector will be described as the behavior of the connector class (in the form of State-charts).

Figure 3: The service view

In figure 3, we give the composite structure diagram of the service view of our message queue connector. The connector consists of two roles; *sender* and *receiver*, here signified by the ports on the MessageQueue class. The algorithm of MessageQueue consists of simply taking elements submitted to the sender by the *producer* component and manifesting them at the receiver for eventual delivery to the *consumer* component.

The presence of a distribution middleware is assumed in the service view, as we simply give the centralized algorithm in this view. This view will be used in the early stages of development of the system, as the distributed algorithm may not have been developed or may not have been implemented at this stage.

2.3 Computational View

The computational view is where we will split the connector up into distributed entities and attach these to the participating components. The distributed algorithm, if there is one (as in the case of a consensus protocol), will be implemented within the roles which will be attached to the components.

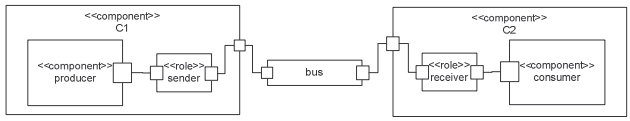

Figure 4: The computational view

In case there is no (or a trivial) algorithm, as is the case with our example message queue connector, then the roles will only call upon the functionality provided by the middleware.

In figure 4, we provide the composite structure diagram representation of the computational view for the message queue connector. The middleware is abstracted out as the *bus* object, which we use for simulations. This object provides functions such as broadcast and message transmission. The final goal is to have a UML model for our middleware which we will be able to plug in place of the *bus* object. The roles, which are now objects themselves, are embedded inside container components (*C1* and *C2*) which contain the application components *producer* and *consumer*. The interfaces of these two application components are linked to the corresponding roles.

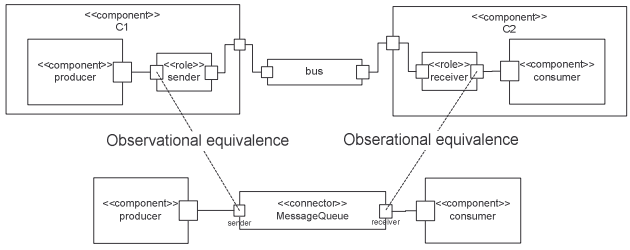

Figure 5: Equivalence among views

For verification purposes, we can easily see that there must be a kind of observational equivalence between the message traces obtained at the role ports of the service view (figure 3, sender and receiver ports) and the component side ports of the role objects in the computational view, as shown in

figure 4. This point is illustrated diagrammatically in figure 5.

2.4 Deployment View

This is the final view for the application architecture. It is described using an implementation ADL. The UML components are transformed to ADL components, and the connectors are embedded into these, with dependencies stipulated on the middleware portion of the architecture.

ADLs provide constructs to represent standard distribution mechanisms such as shared variables, message queues, and remote procedure calls. These will be used to model the connectors that will be transformed from UML to the ADL. The meta-attributes applied to the connectors in UML will become properties of the ADL artefacts such as connections and end-points that will be used to represent them.

3 AADL: a pre-implementation language

As we saw in the previous section, the concept of connectors can be used to describe the communication between the components of a given architecture. Before producing source code, we have to describe the actual architecture from a deployment point of view. Thus, there is an intermediate step, in which we will describe the non-functional aspects, such as memory footprint constraints, execution times etc, and also the actual components to implement within the application. The UML does not quite fit this purpose, since it provides rather fuzzy semantics.

The AADL [9] has been defined and standardized by the Society of Automotive Engineers (SAE). It is an architecture description language targeted to describing DRE systems. Thus, it focuses on the definition of clear block interfaces, and separates the implementations from those interfaces. The AADL allows for the description of both software and hardware parts of architectures. It can be expressed using graphical or textual syntaxes; a UML meta-model of AADL is defined which provides the XMI serialization, a UML profile is also available to allow practitioners to define their models using familiar UML tools and notations.

3.1 Overview of the AADL

An AADL description is made up of *components*. The AADL standard defines software components (data, threads, subprograms, processes…), execution platform components (memories, buses, processors…) and hybrid components (systems).

Components model clearly identified elements of the actual architecture. *Subprograms* model procedures such as those in C or Ada. *Threads* model the active part of an application, i.e.: the units of execution; *processes* are memory spaces that host executing *threads*. *Processors* represent microprocessors and a minimal operating system (mainly a scheduler). *Memories* model hard disks, RAMs, etc. *Buses* model all kinds of networks, wires, etc. Unlike other components, *systems* do not represent anything concrete: they actually create building blocks to help structure the description.

Most components can have subcomponents; thus, an AADL description is hierarchical. Component declarations have to be instantiated as subcomponents of other components in order to model architectures. At the top-level, a *system* component contains all the component instances that make up the application.

Each component has an interface (called *component type*) that provides *features* (e.g.: communication ports). Components communicate with each other by *connecting* their features.

To a given component type correspond zero or more implementations. Each of them describes the internals of the component: subcomponents, connections between those subcomponents, etc. Besides that, implementations of threads and subprograms can specify *call sequences* to other subprograms. Thus it is possible to describe the totality of execution flows in the architecture model.

The AADL defines a set of standard *properties* that are applied to most entities (components, connections, features, etc.). Standard properties are used to specify things such as the clock frequency of a processor, the execution time of a thread, the bandwidth of a bus etc. In addition, it is possible to add user-defined properties to express application-specific constraints.

The AADL does not allow the description of completely dynamic architectures: all the component instances that are present in the architecture must be described; there are no implicit elements. Yet, it is possible to define *modes* within each component implementation. AADL modes allow for the description of different component configurations: properties, connections, subcomponents, etc can depend on modes. The AADL syntax also describes the switching conditions between modes. Thus, the AADL provides support for a limited amount of dynamism: architectures can describe sets of known configurations that can be verified; the switching operation between modes is described in the AADL standard.

By default, all elements of an AADL description are declared in a global namespace. To avoid possible naming 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; only the elements of the package can access the private part. Packages can contain component

declarations. Thus, they can be used to structure the description from a logical point of view. Unlike systems, they do not impact the architecture.

3.2 Using the AADL to model distributed systems

Architecture models described with the AADL are semantically precise and concrete; the AADL is meant to model components very accurately. Hence, AADL descriptions represent the actual system structure: deployment information is provided by the execution platform and software components; constraints on the architecture and component characteristics are expressed by properties.

Since all the elements of an AADL description are explicitly described, it is possible to perform analysis on the described architecture, such as thread schedulability; or check whether the memory footprint of the application elements are consistent with the constraints of the execution platform.

The AADL syntax provides the ability to describe message passing, remote procedure calls, distributed objects, and distributed memory. Hence, the language can be used to describe most distributed architectures. The AADL is thus able to describe distributed architectures that are subjacent to connector-based functional descriptions.

Figure 6: The deployment view

Figure 6 is a graphical representation of one of the possible AADL models that could correspond to the UML model of figure 4. The *producer* and *consumer* component objects have been transformed to AADL threads; whereas *C1* and *C2* have been transformed to AADL processes that contain the threads. They are linked together by *event data ports* (AADL equivalents of message queues). The message transfer functionality will be provided by a distribution middleware. In case the connector requires algorithmic functionality that is not present in the existing implementation of the middleware, it will be represented as AADL subprograms in our model, the behavior of which will be given by the UML model of the computation view.

The AADL can be seen as a “pre-implementation” language, which can be used to describe all the architectural parameters. The description of the algorithms is not in the scope of the AADL, since the language only focuses on architectural concerns. Yet, it is possible to use the AADL properties to host the behavioral descriptions of the components (e.g.: by identifying the piece of source code that implements the algorithm). Therefore, the AADL can be used as a backbone to host all the aspects of the system description, both functional and non-functional.

3.3 Requirements for an AADL runtime

AADL is precise enough to allow the description of the application structure (using software components) and the deployment information (using the execution platform components). Given a precise enough description, we can then aim to automatically generate and deploy a complete application from its AADL description.

Such an application is likely to run on top of a runtime that provides the functionalities required to ensure correct execution. An AADL runtime should at least support the execution of threads. Since we design distributed applications, the AADL runtime should also provide support for the different distribution models that can be described in the AADL.

Thus, the AADL runtime is an execution middleware which provides two main services:

- The management of the AADL threads that support the execution of the AADL application on each node
- The management of the communication between the application nodes

This underlying middleware has to provide guaranties regarding its reliability and its ability to meet the constraints described in the AADL description (typically memory footprint).

4 Framework for an adaptable AADL runtime

An AADL runtime has to provide traditional middleware functionalities such as distribution as well as thread management. Different middleware architectures have been proposed to allow the middleware to be configured as a function of application requirements. In [11], the authors proposed the means to configure the middleware. In [12], the authors proposed the concept of generic middleware that is configurable and which allows the selection of a distribution model to suit the application. Schizophrenic middleware extends the idea of configurable and generic middleware by defining an adaptable architecture that allows for the verification of middleware properties.

The schizophrenic architecture allows the adaptation of the middleware to meet the needs of the target application. The schizophrenic architecture enforces the principle of separation of concerns, and allows for proven code and the reuse of components. Thanks to its clear structure, it lends itself well to formal verification of properties.

4.1 A three-layer architecture

The schizophrenic architecture comes with the concept of personalities and with the notion of canonical middleware services. In this architecture, middleware functions are decoupled into application and protocol level personalities connected to the neutral core, on which all personalities rely.

Application personalities constitute the adaptation layer between the application components and the middleware. They provide APIs to register application entities with the neutral core layer. Application entities can interoperate with the core to allow the exchange of requests with remote entities (like CORBA or DDS APIs).

Protocol personalities handle mapping of requests (representing interactions between application entities) onto messages exchanged through a communication network, according to a specific protocol. The requests are received either from application entities (through an application personality and middleware core), another protocol personality or a node in the middleware instance.

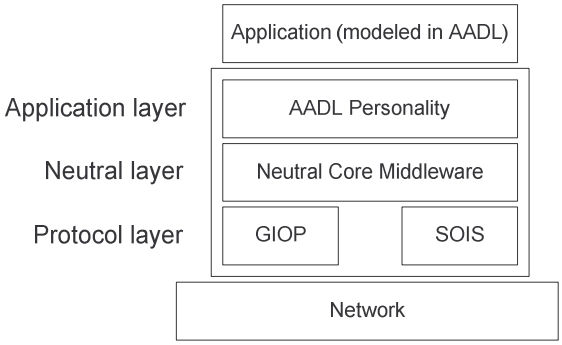

Figure 7: the three-layer schizophrenic architecture

The Neutral Core acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides necessary abstractions to transparently pass requests between protocol and/or application personalities in a neutral way. It is completely independent from both application and protocol personalities, enabling the selection of any combination of them. Figure 7 depicts this first view of the schizophrenic architecture.

Building an AADL runtime based on the schizophrenic architecture actually consists of building an application personality. The exact structure of this personality depends on the distribution features used in the AADL architecture model of the application.

4.2 Configurability

The neutral layer provides common services that are typically part of all middleware implementations. Seven canonical services are isolated, each of which implements one fundamental aspect of a middleware:

- The *addressing service* gives a unique identifier within the distributed system to each entity
- The *binding service* provides mechanisms to associate the interacting objects with the resources supporting this interaction.
- The *representation service* allows the translation of data into a representation suitable for transmission over the network
- The *protocol service* allows entities present in different nodes to communicate
- The *activation service* associates implementing objects to incoming requests
- The *execution service* assigns execution resources to process incoming requests

These services are shown in figure 8.

The services defined in the schizophrenic architecture can be reduced to well-known abstractions (mainly pipes and filters). The µBroker coordinates these services. It is in charge of resource allocation, and of data propagation through the middleware. It is the most critical component in the middleware since it manipulates tasks and I/O. All the behavioral properties of the middleware are satisfied in this central component; the services are reactive.

The combination of the selected implementations of the services allows for the instantiation of a middleware tailored to the needs of the application. As an example, the activation service can manage incoming requests according to priorities rather than with a simple FIFO protocol. This selection of service implementations impacts non-functional properties such as memory footprint and execution time.

4.3 Reliable execution middleware

The schizophrenic middleware architecture allows for proven components and reuse of basic services, and aims to conserve proofs. These components can be separately modeled and generated; the models can be verified using formals methods (e.g.: colored Petri nets). This helps produce a verified, reliable middleware instance.

Besides, as the µBroker handles all the behavioral aspects of the middleware, it manages the scheduling and dispatching of threads. Thus, a schizophrenic middleware satisfies the two requirements stipulated for an AADL runtime; namely, that it be a distribution middleware *and* manage threads.

PolyORB[1] is an implementation of the schizophrenic middleware architecture. Thanks to this architecture,

[1] http://polyorb.objectweb.org/

PolyORB has been modeled and formally verified using colored Petri nets. They were used to model the µBroker. The work done in [7] proved some essential properties such as absence of deadlocks, absence of buffer overflows, and fairness.

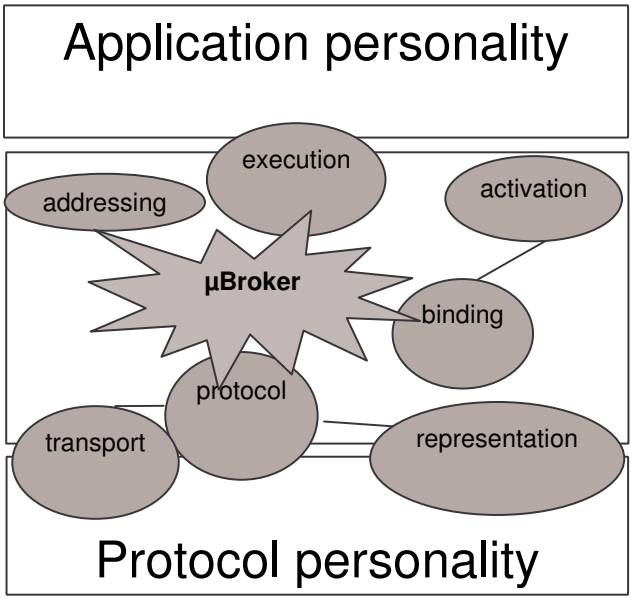

Figure 8: PolyORB services

Thus, PolyORB is a highly tailorable middleware that proposes a canonical architecture and a methodological guide to building a specific distribution platform. CORBA, and also DDS, MOM, DSA and advanced specifications like FT-CORBA and RT-CORBA are already available.

5 From AADL description to application

PolyORB constitutes an appropriate basis to implement an AADL runtime. It provides all the required functionalities regarding thread scheduling and communication management. Since it is tailorable, it can thus be instantiated to meet the exact application requirements. Its clear architecture facilitates the verification against different properties such as no deadlocks, no buffer overflows etc.

In this section, we explain how to generate an executable application from an AADL description.

5.1 Separation of concerns

In the generation process, we must handle two different things:

- The generation of the application code
- The generation of a configured AADL runtime to support the execution of the AADL application on each node

The AADL subprograms model the application itself, while the AADL threads and processes represent the runtime. Indeed, all communications are described by the thread features; and the thread implementations contain the call sequences that drive the application subprograms. The processes define the application nodes in which the threads are executed. The execution platform components describe the deployment information for the runtime: location of the nodes, constraints on the communications between the nodes, etc. Thus we have a clear separation of concerns in the architecture description.

In order to ensure consistency between the AADL description and the actual executable system, the application generator should process the AADL components. In order it is the runtime that controls the application and not the other way round, application components should be enclosed in wrappers. The runtime is the key element of the executable system, since it is the active part—the application is the passive part.

5.2 Design of the AADL runtime

In order to set up an appropriate runtime, the generator has to take into account two kinds of parameters:

- The structure of the application on each node, e.g.: the required communications, the number of AADL threads, etc
- The deployment information, e.g.: location of the other nodes, the protocols to use according to the configuration of the other nodes, the thread scheduling policy to use, etc

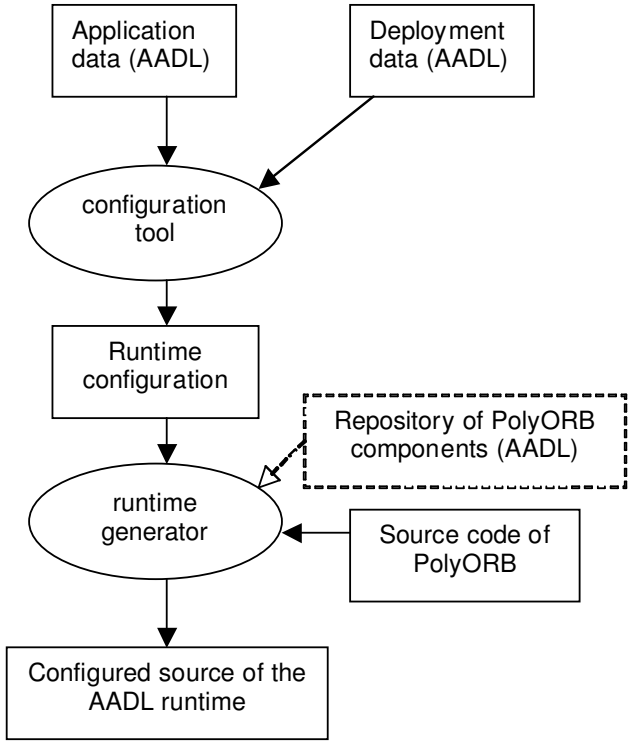

Figure 9: Generation process for the AADL runtime

Thus, the generation of the AADL runtime should be a two-step process (see figure 9):

1. Gather the deployment information in order to select the proper components to use in the execution middleware
2. Generate the middleware by assembling the selected components, and generate an adaptation layer to control the application wrappers

There are different ways to perform the component selection. We can rely on the existing configuration process of PolyORB. This allows for the selection of various implementations for the fundamental services and the configuration of the µBroker structure. This approach provides an efficient way to build a prototype of the middleware. We can also model the different parts of the middleware in the AADL.

Since the schizophrenic architecture provides a clear and modular structure, it greatly facilitates the AADL modeling process. The middleware is then a set of AADL subprograms that are called by the µBroker. The µBroker itself cannot be described in AADL, since it mainly consists of behavioral elements. Following this approach, the major part of the middleware can be seen as being a part of the application. The µBroker is then the actual AADL runtime. Since the middleware components are modeled in AADL, we can integrate them in the verification and simulation process of the application. Thus, each application node is structured in two parts: a minimal runtime (the µBroker) which is modeled in Petri nets to ensure properties regarding the execution of the local node; and the application, which can be processed to compute worst-case execution times, total memory footprint, etc.

5.3 Gaia, an application generator for AADL

In order to experiment on application generation from AADL descriptions, we created an application generator named Gaia. Gaia is part of the Ocarina tool suite, developed at Télécom Paris[2]. It provides a set of lightweight tools to describe distributed architectures in AADL; the descriptions are then transformed to distributed systems driven by a tailored runtime based on PolyORB.

Gaia is structured in two parts: a translator from AADL to programming language and a runtime generator. The translator handles the AADL subprograms and only depends on the target programming language. The runtime generator is to be used in two ways: either to use PolyORB or only the µBroker. In the first case, we generate an AADL application personality and configuration files for PolyORB; this way we use PolyORB as a high level runtime to prototype applications. In the second case, the generator expands AADL threads to create AADL models of the schizophrenic services from the thread properties; thus we have a complete model of the application in AADL. This enables architecture analysis and verification; the architecture can then be transformed to source code, using the µBroker as a runtime.

[2] http://ocarina.enst.fr/

6 Conclusion

Building DRE systems requires the capture of a wide range of requirements, either functional (algorithms to use etc.) or non-functional (execution time constraints etc.). We therefore need a methodology based on a formal approach to assist the design process.

We proposed a three-step methodology. We first design the system from a very abstract point of view, to identify the functional properties of the system. Architectural constructs such as connectors provide good support to model interactions between the nodes of a distributed application. They can be described using semi-formal syntaxes such as the UML.

We then have to describe a concrete model of the distributed application in order to capture the non-functional aspects of the system design. One of the objectives of our research is to be able to provide rules to automate the transformation of models from UML to AADL. The AADL perfectly matches the requirements of such descriptions: it allows the representation of both software and hardware parts of a system, all entities of the language have clear semantics and properties can be associated to every element of the architecture to capture the system characteristics. Thus, the AADL can be used as a pre-implementation language to have a very concrete representation of the whole system. It is then possible to generate source code from the AADL description. This source code is to be executed by a runtime that handles the communications and the scheduling of the application threads.

In order to support the execution of the application, we extended the concept of middleware to consider *execution* middleware, which can be compared to a virtual machine. The execution middleware schedules application threads and manages communications.

In order to build an efficient system, the middleware structure (i.e.: its actual configuration) must fit the application requirements. We introduced the schizophrenic middleware architecture as a solution to tackle the middleware specialization problems.

We tested our approach using Gaia and PolyORB to generate distributed applications. Our results have been encouraging in that we have been able to

generate applications in a short time-frame. All Ocarina tools and PolyORB are free software.

7. References

[1] M. Shaw: "*Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status*", Workshop on Studies of Software Design, 1993.

[2] N. R. Mehta, N. Medvidovic and S. Phadke: "*Towards a taxonomy of software connectors*", International Conference on Software Engineering, 2000.

[3] R. J. Allen and D. Garlan: "*A Formal Basis for Architectural Connection*", ACM Transactions on Software Engineering and Methodology, 1997.

[4] B. Spitznagel and D. Garlan: "*A Compositional Formalization of Connector Wrappers*", International Conference on Software Engineering, 2003.

[5] C. Atkinson and T. Kühne: "*Rearchitecting the UML Insfrastructure*", ACM Transactions on Modeling and Computer Simulation, 2002.

[6] T. Vergnaud, J. Hugues, L. Pautet and F. Kordon: "*PolyORB: a schizophrenic middleware to build versatile reliable distributed applications*". Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04), volume LNCS 3063, pages 106 - 119 June 2004.

[7] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud: "*On the Formal Verification of Middleware Behavioral Properties*". Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04), volume ENTCS 133, pages 139 - 157, Linz, Austria, September 2004.

[8] S. Flake: "*Temporal OCL Extensions for Specification of Real-Time Constraints*", Specification and Validation of UML Models for Real-Time and Embedded Systems, 2003.

[9] SAE: "*Architecture Analysis & Design Language (AADL)*". Available at http://www.sae.org, 2004

[10] N. Medvidovic and R. N. Taylor: "*A Framework for Classifying and Comparing Architecture Description Languages*", Proceedings of the Sixth European Software Engineering Conference ESEC/FSE 97, Springer–Verlag, 1997.

[11] D. Schmidt and C. Cleeland: *"Applying Patterns to Develop Extensible and Maintainable ORB Middleware"*. Communications of the ACM, CACM, 1997.

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

8. Glossary

AADL: Architecture Analysis & Design Language

CORBA: Common Object Request Broker Architecture

DDS: Data Distribution Service

DRE: Distributed Real-Time Embedded

DSA: Distributed Systems Annex (for Ada95)

FT-CORBA: Fault Tolerant CORBA

GIOP: General Inter-ORB Protocol

MOM: Message Oriented Middleware

OCL: Object Constraint Language

PolyORB: An implementation of the schizophrenic middleware architecture

RT-CORBA: Real-time CORBA

SAE: Society of Automotive Engineers

SOIS: Spacecraft Onboard Interface Services

UML: Unified Modeling Language