From Acme to CORBA: Bridging the Gap

Márcia J. N. Rodrigues[1], Leonardo Lucena[2], and Thaís Batista[1]

[1] Informatics and Applied Mathematics Departament,
Federal University of Rio Grande do Norte, Brazil
marciaj@dimap.ufrn.br, thais@ufrnet.br

[2] Federal Center of Technological Education of Rio Grande do Norte, Brazil
leonardo@cefet-rn.br

Abstract. Software architecture and middleware platforms are different abstraction levels of component-based development that have been evolved separately. In order to address the gap between these two areas, in this paper we discuss the integration of a generic and extensible architecture description language, Acme, with a standard middleware platform - CORBA. We propose mapping rules to transform an ACME description into a CORBA IDL specification. To make it possible, we define some extensions to Acme to include some features according to the CORBA IDL specification. These extensions explore the facilities provided by Acme for expressing additional information. We use a case study to illustrate the mapping proposed.

1 Introduction

Software architecture [1] is an important field of software development that focus on the early design phase of component-based development. It concerns the design and specification of the high-level structure of an application. This is especially important to solve design problems in the initial stages of development. Architectural description languages (ADLs) are used to describe software architectures in terms of components and the relationship among them.

Although various architectural languages are available at the moment [2], each of them has its own particular notation and some are designed for specific application domains. This makes them inappropriate for expressing a broad range of architectural design and also for sharing and reusing architectural descriptions. In order to address this problem, the Acme Architecture Description Language [3] provides a common language for the support of the interchange of architectural descriptions. It provides a generic and extensible infrastructure for describing software architectures.

At the implementation level of component-based development, middleware platforms are playing an important role as an underlying infrastructure that offers transparent communication between distributed and

heterogeneous components. In this context, CORBA has been successful because it is a language and platform independent model.

Software architecture and middleware platforms deal with different levels of abstraction of component-based development and share some common characteristics. Both offer support for the management of large, complex and distributed applications as well as to reduce the costs of applications development by promoting components reuse. Besides, both focus on composing systems by assembling components. They are complementary approaches to a component-based development. However, there are few interactions between the two research areas. In [4] is showed that it is necessary to integrate such areas in order to use existing component middleware technologies to implement systems modeled with architectural languages.

An important challenge for software developers today is the ability to translate a software architecture description into a corresponding description for a target implementation platform. They have to know details about the two models in order to identify the mapping between the concepts. In general, this task is done in an ad-hoc way because there is a lack of reference models and tools to identify and relate the concepts of the two research areas. Thus, the mapping is an error-prone task that can lead to inconsistencies between the architectural description and the corresponding description in the target implementation platform.

In this paper we address the integration between this two research areas, discussing how the concepts of the Acme architecture description language can be translated into corresponding concepts of CORBA IDL. Our goal is to provide mapping rules in order to reduce the gap between the Acme architecture description and the CORBA IDL specification. Besides, the rules can be used in the development of automatic transformation tools.

In order to evaluate our proposal we present a case study of a distributed application: a multiagents system for buying and selling goods[5, 6].

This paper is structured as follows. Section 2 presents the background of this work: Acme and CORBA. Section 3 discusses the mapping from Acme to CORBA. Section 4 presents the case study that illustrates the application of the mapping. Section 5 regards about the related works. Finally, Section 6 contains the final remarks.

2 Background

2.1 Acme

Acme [3, 7] is a software architectural description language whose main goal is being an interchange language among different ADLs. Acme was projected to consider the essential elements of the different ADLs and to allow extensions to describe the most complex aspects of others ADLs.

An Acme architecture is structured by using the following aspects: structure, properties, constraints, types and styles [7]. In this section we will present each aspect and then we will illustrate their use with an example.

Structure Acme has seven entity types to architectural representation: components, connectors, systems, ports, roles, representations, and rep-maps.

The **components** are the basic elements of an Acme description. They represent primary elements of a system. The Acme component can model hardware and software elements or both. The Acme component has interfaces, named **ports**, that represent interaction points with the computational environment. Each port represents an interface that is offered or required by the component. However, the ports do not distinguish between neither what is offered nor what is required by a component.

The Acme **connectors** represent interactions among components. A typical connector may define a communication synchronization model, a communication protocol or features of a communication canal.

Acme provides a way to explicitly document the system communication, thus, it is necessary to provide the concept of connector. This is an important feature of the architectural modeling: the interactions are considered first class concepts. In contrast, in object-oriented project approaches, the interactions are implicit within diagrams that describe classes and objects. The connectors have a set of interfaces represented by **roles**. Each role defines a participant in the interaction defined by a connector. A role is seen as an interface, in a communication canal, defining an interface to the connector just as a port provides an interface to a component.

Acme **systems** are defined as graphs in which nodes represent components and edges represent connectors. Therefore, a graph of a computational system is defined by a set of attachments. Each attachment represents an interaction between a port and a role.

The Acme language uses **representations** that allows the components and connectors to encapsulate subsystems. Each subsystem may be seen as the most concrete description of the element that it represents. This allows the analysis of the system in various abstraction levels.

When a component or connector has an architectural representation, it should have a way of showing correspondence between internal system representation and external interfaces of components or connectors that are being represented. The **rep-maps** define this correspondence. They associate internal ports/roles to external ports/roles.

Figure 1 shows an example of an architectural description in Acme. The System has two components, `X` and `Y`, joined by a connector. The connector has its `roleA` attached to `portM` of component `X` while `roleB` is attached to `portN` of component `Y`. The textual description is shown in Figure 2.

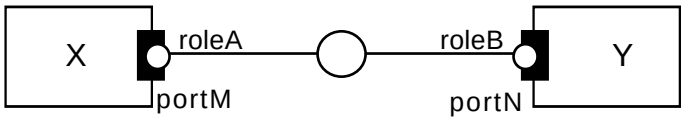

Fig. 1. Example of an architectural description in Acme

```
System System1 {
  Component X{
    Ports {portM;}
  };
  Component Y{
    Ports {portN;}
  };
  Connector C{
    Roles {roleA; roleB;}
  };
  Attachment X.portM to C.roleA;
  Attachment Y.portN to C.roleB;
};
```

Fig. 2. Textual description of the architectural description of Figure 1

Properties The components, as well as other Acme elements, have **properties** that are used to describe their structural and functional aspects. Each property has a name, an optional type and a value. The properties do not define semantics in Acme, but their values have meaning in tools that analyze, translate, show or manipulate Acme descriptions.

Constraints Acme may define the constraints that should be used by the computational system. These constraints are a special type of properties that are associated with any Acme description element. This association determines the scope of the constraint. This means that if one constraint is associated to a system, then every element comprised within the element also is associated to this constraint.

The constraints may be associated to design elements in two ways: using invariants or heuristics. The violation of invariants makes the system invalid, while the violation of heuristics is treated as a warning.

Types and Styles The ability to define system styles (families) is an important feature for an architecture description. **Styles** allow the definition of a domain-specific or application-specific design vocabulary.

The basic block for defining styles in Acme is a **type** system that is used to encapsulate recurring structures and relationships. In Acme there are three ways to define these structures: property types, structural types, and styles. Property types have been previously showed.

Structural types enable the definition of types of components, connectors, ports, and roles. Each type provides a type name and a list of necessary substructure, properties and constraints.

The other existent type in Acme is style, also named family. Just as structural types represent sets of structural elements, a family represent a set of systems.

2.2 CORBA

CORBA (Common Object Request Broker Architecture) is a standard proposed by OMG that allows interoperability between applications in heterogeneous and distributed environment. CORBA determines the separation between object interface and object implementation. An object interface is described using the *Interface Definition Language (IDL)*. Object implementation can be done using a programming language with a binding to CORBA.

A CORBA architecture is composed by a set of functional blocks that use the communication support of ORB (Object Request Broker) - the element that coordinates the interaction between objects intercepting the client invocations and directing them to the appropriate server.

The entities that compose the syntax of IDL are: modules, interfaces, operations, attributes and exceptions.

Module is the element that groups other elements. An interface defines a set of operations provided by an object and its attributes. The declaration of attributes initiates with the keyword `attribute`. Attributes types can be: basic, built, templates or interfaces.

Figure 3 shows a simple IDL interface definition with attributes and operations.

```
module People{
   interface Student{

     //Attributes
       attribute string name;
       attribute string phone;
       readonly attribute long identification;

     //Operations
       void RegistrationInDisc(in long ident);
   };
```

Fig. 3. IDL Description

3 Mapping Acme to CORBA

This section shows our proposed mapping strategy of Acme architecture descriptions to IDL specifications.

Acme is a generic language for architectural description and, thus, it has a small number of elements. Therefore, architectural descriptions using only Acme's basic elements are, semantically, very poor. For this reason, some extensions are proposed in this paper in order to make architectural descriptions more meaningful for transformation into IDL specifications. In this way, in addition to the mapping, we show the structures that must be part of the Acme descriptions to make them suitable for mapping to IDL.

3.1 Systems

Both Acme and IDL contain structures that aggregate other elements. Acme uses Systems and Families (Styles), while IDL uses Modules. The mapping preserves this grouping by transforming systems and families in IDL modules.

3.2 Components

Components are the main elements of an architectural description. They are mapped directly to IDL interfaces by a one-to-one relationship. The structural types that define components are also mapped into interfaces. The internal details of the interfaces are obtained through ports and connectors.

3.3 Ports

Ports define the points of interaction of each component with the environment. The details of the interaction are described through the properties of the ports. These properties do not have semantics in Acme but they are interpreted at the moment of the transformation in IDL specifications.

In the mapping strategy used in this article, the ports that compose each component are combined to comprise a single IDL interface. Figure 4 shows a specification in Acme that corresponds to the IDL specification of Figure 3.

The types `idl_attribute` and `idl_operation` indicate that the properties of these types have significance in IDL. The properties `name`, `phone` and `id` represent attributes, while `registration` represents an operation.

Another aspect that must be considered is that ports can offer and request services. This leads to the classification of the ports as input ports (offers services), output ports (requires services), and input and output ports (offers and requires services)[3]. The example of Figure 4 shows input ports (`InputPort`). The other options are `OutputPort` and `InputOutputPort`.

The output ports are represented in IDL through attributes. Each output port (or input and output port) is mapped into an attribute. The name and the type of the attribute depend on the connector that is attached to the port. For the example in Figure 1, if `portM` is an output port then the corresponding IDL interface of component `X` will have an

[3] The original semantics of Acme does not make distinction among these types of ports.

```
System People {
  Component Student{
    Port personal : InputPort = {
      Properties {
        // Attributes
        name : idl_attribute = "attribute string name";
        phone: idl_attribute = "attribute string phone";
        // Operation
        registration: idl_operation =
                      "void RegistrationInDisc(in long ident)";
      };
    };
    Port school : InputPort = {
      Properties {
        // Attribute
        id : idl_attribute = "readonly attribute long identification";
       };
    };
  };
};
```

Fig. 4. Acme Specification

```
interface X{
   //require
   attribute Y roleB;
   ...
};
```

Fig. 5. Mapping Output Ports

attribute called `roleB` of type `Y`. Figure 5 shows the result of this transformation.

The use of constraints allows better specification of the interface required by Output ports. Invariants can specify which roles can be attached to a port. In the example of Figure 1, constraints can be used to state that `portM` can only be attached to `roleA`.

3.4 Connectors and Roles

The connectors specify how components are combined into a system. They do not have a corresponding representation in IDL. Instead, the connectors are used to determine the type of interface that output ports require,

as seen in Section 3.3. In the same way, the roles contribute to determine the names of the attributes that are related to output ports.

3.5 Representations

Representations enable the existence of architecture descriptions with different levels of abstraction. Representations allow elements to enclose internal subsystems. However, IDL descriptions cannot be encapsulated. Only the most concrete (internal) elements are mapped to IDL. The mapping process creates an auxiliary version of the system with only one level of abstraction. The complex elements are *blown up* displaying their internal representations[4].

The mapping rules of Acme to IDL are summarized in the Table 1. The style `IDLFamily` (Figure 6) aggregates the extensions that makes Acme descriptions suitable for mapping to IDL. The family also has some constraints, not shown here, that checks if the descriptions are valid.

```
Family IDLFamily = {
  Port Type InputPort = {}
  Port Type OutputPort = {}
  Port Type InputOutputPort = {}
  Property Type idl_attribute = String;
  Property Type idl_operation = String;
}
```

Fig. 6. Style IDLFamily

4 Case Study

To illustrate the mapping from Acme to IDL CORBA we use an e-commerce multi-agents system[6].

A Multi-agents system is a society of agents that cooperates with each other to solve a specific problem. Thus, a problem is divided in more specific problems that are attributed to agents, according to their individual capability.

[4] When there are two or more representations for the same element, one of them must be chosen to be mapped.

Table 1. Summary of the Mapping Rules of Acme to IDL

Element	*Acme*	*IDL*
System	**System** { ... }	**module** { ... }
Component	**Component** X = { ... }	**interface** X { ... }
Component Type	**Component Type** X = { ... }	**interface** X { ... }
Input Port	**Component** X = { ... **Port** p : InputPort = **new** InputPort **extended with** { **Properties** { attr : idl_attribute = "[idl attribute]"; oper : idl_operation = "[idl operation]"; } } ... }	**interface** X { ... *// Port p* [idl attribute] [idl operation] ... }
Output Port	**Component** X = { ... **Port** px : OutputPort = **new** OutputPort **extended with** { ... } ... } **Component** Y = {**ports** {pY;}} **Connector** C = {**roles** {roleA;roleB;}} **Attachment** X.px **to** C.roleA; **Attachment** Y.pY **to** C.roleB;	**interface** X { ... *// Require (output port)* **attribute** Y roleB ... } ... **interface** Y { ... } ...

In the case study the agents are distributed through a net and need to interact with each other to negotiate goods. The system has three types of agents. The buying and selling agents are negotiating agents that buy or sell goods.

The market agent plays as a facilitator that presents an agent to other negotiators.

These agents are modeled by software components. The features that agents needed such as autonomy and communication are founded in the components.

The distribution of the components must allow communication among components implemented in different platforms without worrying about the communication details.

Figure 7 shows how a fragment of the Acme architectural description is mapped into IDL. The `Buyer` component are transformed into `Buyer` interface. The two ports of the component are combined and their `idl_operation` properties are transported to the component interface. The ports are output ports therefore produce `seller` and `market` attributes. The other components are mapped in the same way.

5 Related Works

The integration of ADLs and middleware platforms is a current trend in component-based development. Following this trend, OMG has published a specification of a standard to support all the systems lifecycles: MDA [8]. MDA is a vendor and middleware independent approach language that uses UML to build system models[9]. MDA does not specify mapping models between UML and platform-specific models. Some works [10] are addressing this issue by defining mapping rules to transform UML descriptions into CORBA IDL specifications.

The ABC environment [11] does a gradual mapping from an ADL to a middleware platform. It offers an ADL, named JBCDL, whose descriptions are mapped to an OO design model described in UML and then mapping rules are applied to convert the OO model to a CORBA IDL description. The authors mention that an OO model adds more flesh to perceive components and connectors specified in the architecture description. We argue that using a generic ADL, such as Acme, that is flexible and provides annotation facilities, it is not necessary to have an intermediate model to enhance the expressiveness of the architecture description.

Darwin [12] is an ADL that has been a pioneer in the integration of an ADL with CORBA. In the mapping from Darwin to CORBA, the Darwin

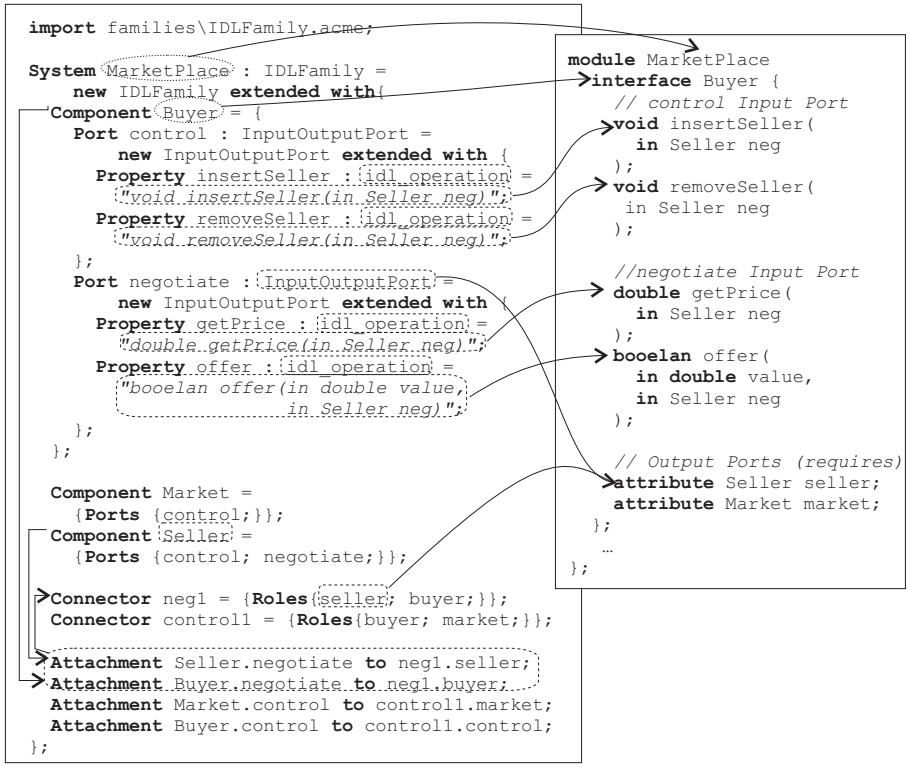

Fig. 7. Market Place mapping from Acme to IDL

compiler translates a Darwin component to the IDL interface. Each provision in the Darwin specification is translated into a read only attribute of the object reference type. Each requirement is similarly mapped into an attribute which is not read only because it is set externally to reflect the binding of the component instance. While this work uses a particular ADL, we choose to use a generic ADL in order to allow the integration between other ADLs and CORBA via Acme. Since Acme provides a means to integrate the features of existing ADLs and to share architectural descriptions between these ADLs, it is possible to transform specifications of other ADLs into Acme and then into CORBA.

A mapping from an ADL, named ZCL, to a component-based environment that uses CORBA components is proposed in [13]. This work defines structural mapping from the ADL to the environment. However, this mapping does not only address the CORBA description but also the features of the scripting language used in the environment. Although this work aims to shorten the gap between design and implementation, they rely on the same problem of many works: to use a particular ADL. Besides, the mapping is restrictive to the specific environment. In contrast, in our work we join an ADL that allows interoperability of ADLs with a standard middleware platform. We consider that this combination will be appropriate for different classes of applications.

6 Final Remarks

In this paper we investigated the feasibility of combining the use of two different technologies in order to reduce the gap between different phases of component-based development: design and implementation. Software architecture description languages (ADLs) and middleware platforms deal with composing systems from compiled parts. However, ADLs do not focus on component development and middleware platforms do not cope with the high-level model of a system. An architecture description should be implemented in a specific development platform, thus bringing these research areas together is essential to the composition of large systems.

We identified the common features of a generic architecture description language - Acme - and a component-based development platform - CORBA. We proposed a mapping from Acme to CORBA. In order to make it possible, we improve the expressiveness of Acme specifying the concept of input and output ports and properties that will be transformed into attributes and operations. This extension clarifies the mapping to CORBA IDL. Since Acme is flexible and provides facilities for additional

ADL-specific information, we explore these facilities to specialize the concept of ports.

Using the mapping proposed in this work, it is possible to generate interface definitions described in CORBA IDL. The IDL description is an important part of the CORBA-based development and it is the basis for programmers to produce the implementation code. Thus, once the interface has been defined, the programmer will be able to reuse existing components or coding components according to the architectural description. Besides, IDL description can be automatically mapped into client and server languages by using an IDL compiler.

A tool, named ACMID, that performs an automatic transformation from Acme to CORBA IDL using the mapping proposed by this work is under development. ACMID implements a conversion algorithm that does such transformation. The transformation is based on XML (eXtensible Markup Language) [14]. ACMID receives as input an XMI (XML Metadata Interchange Format) [15] file that contains the meta-model description of the Acme architecture model. A modified version of Acme Studio [16] is used to produce the Acme model. The conversion rules are described in XSLT (eXtensible StyleSheet Language Transformations) [17] and they produce a specific model to the CORBA platform represented in IDL (Interface Definition Language).

As a future work we intend to observe the enhancement provided by the ACMID in the development of a number of practical cases of component-based development.

References

1. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall (1996)
2. Medvidovic, N.: A classification and comparison framework for software architecture description languages. Technical Report UCI-ICS-97-02, Department of Information and Computer Science, University of California, Irvine (1997)
3. Garlan, D., Monroe, R., Wile, D.: ACME: An architecture description interchange language. In: Proceedings of CASCON'97, Toronto, Ontario (1997) 169–183
4. Oreizy, P., Medvidovic, N., Taylor, R., Rosenblum, D.: Software architecture and component technologies: Bridging the gap. In: Digest of the OMG-DARPA-MCC Workshop on Compositional Software Architectures, Monterey, CA (1998)
5. Chavez, A., Maes, P.: Kasbah: An agent marketplace for buying and selling goods. In: First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM'96), London, UK, Practical Application Company (1996) 75–90
6. Maes, P., Guttman, R., Moukas, A.: Agents that buy and sell: Transforming commerce as we know it. Communications of the ACM **42** (1999)

7. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In Leavens, G.T., Sitaraman, M., eds.: Foundations of Component-Based Systems. Cambridge University Press (2000) 47–68
8. Miller, J., Mukerji, J.: Model-driven architecture - mda. Technical report, OMG (2001) ormsc/2001-07-01. www.omg.org/mda.
9. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1999)
10. Nascimento, T., Batista, T.: Tupi - transformation from pim to idl. In: Proceedings of the International Symposium on Distributed Objects and Applications - DOA2003, Catania, Sicily, Italy (2003)
11. Mei, H., Chen, F., Wang, Q., Feng, Y.: Abc/adl: An adl supporting component composition. In George, C., Miao, H., eds.: Proceedings of the 4th International Conference on Formal Engineering Methods, ICFEM2002, LNCS 2495. (2002) 38–47
12. Magee, J., Tseng, A., Kramer, J.: Composing distributed objects in CORBA. In: Proceedings of the Third International Symposium on Autonomous Decentralized Systems, Berlin, Germany, IEEE (1997) 257–63
13. de Paula, V., Batista, T.: Mapping an adl to a component-based application development environment. In: Proceedings of FASE2002 - Lecture Notes in Theorical Computer Science (LNCS) - 2306. (2002) 128–142
14. Birbek, M.: Professional XML. Wrox Press Inc. (2001)
15. OMG: Xml model interchange (xmi). Technical report, OMG (1998) OMG Document ad/98-10-05.
16. AcmeStudio: (AcmeStudio: Supporting architectural design, analysis and interchange) available at http://www-2.cs.cmu.edu/ acme/acme_documentation.html.
17. W3C: Xsl transformations specification. Technical report, W3C (1999) www.w3.org/TR/xslt.
18. Garlan, D., Cheng, S.W., Kompanek, A.J.: Reconciling the needs of architectural description with object-modeling notations. Science of Computer Programming **44** (2002) 23–49