Chapter 1

Metamodeling in Software Architectures

To manage the complexity of systems and enhance their understanding, different modeling techniques (e.g. object-oriented, component-based and services-oriented software architectures) often define two or more modeling levels using the definition of a metamodel, or even a meta-metamodel to explain, comment, document, simplify, create and compare models.

In this chapter, we intend to emphasize the importance of metamodeling in the context of software architecture compared to the metamodeling in the object paradigm. This concept helps to abstract architectural concepts (components, connectors and configurations) and facilitate handling, using and reusing, analyzing and developing software architectures.

1.1. Introduction

During the last two decades, several languages for describing software architectures have been proposed to promote the development of architecture-focused applications. These languages provide, in general, formal or semi-formal notations for describing and analyzing software systems. They are usually accompanied by tools to analyze, simulate and, sometimes, to generate the code of the modeled systems. The consensus on the fundamental concepts that architecture description languages (ADLs) must include was achieved relatively late; however, through the efforts of

---

Chapter written by Adel SMEDA and Mourad Chabane OUSSALAH.

standardization and interoperability, new ADLs (known as second generation ADLs) were introduced.

Moreover, it is proved that the specification of an architecture may go through several levels of modeling, thus reflecting different categories of users (application architects, application developers, builders of software infrastructure, etc., see Chapter 3). One of the most appropriate techniques to highlight these different levels of modeling is metamodeling and, more precisely, the software architecture modeling frame that implements the four levels of modeling proposed by the Object Management Group (OMG) [OMG 03]. The highest level is represented by the meta-object facility (MOF) meta-metamodel [MOF 02]. The goal of MOF is to define a single standard language for describing metamodels. It consists of a relatively small set (although not minimal) of "object" concepts for modeling this type of information. For example, unified modeling language (UML) is one of the metamodels described using MOF. Other notations for object metamodeling also exist, including KM3 [JOU 06], ECORE [BUD 08] and Kermeta [MUL 05].

In software architecture field, very few works have adopted the technique of metamodeling. Architecture Meta-Language (AML) [WIL 99] is the first attempt to offer a base that provides ADLs with a solid semantic base. AML only defines the statements of three basic constructions: components, types and relationships.

In this chapter, we propose an approach to model software architectures at different levels of abstraction using a meta-metamodel called Meta Architecture Description Language (MADL) [SME 05a].

In this context, MADL, like MOF, works as a unified solution for architectural representations. In addition, it allows easy handling, using, reusing and developing of software architectures. It reduces the complexity of software architectures and facilitates the transformation and transition of architectures from one to another. In addition, this meta-metamodel has a minimal core whose purpose is to define the concepts and basic relations, such as metacomponents, metaconnectors and metainterfaces. Based on this meta-metamodel, we describe a strategy for mapping of ADL concepts to UML concepts (particularly UML 2.0). We used UML as an example of mapping because of its popularity in the industrial world. This strategy is carried out in four steps: instantiating MADL by the selected ADL, mapping

MADL to MOF, instantiating MOF by UML and, finally, the selection of the most appropriate UML concepts for the selected ADL. This strategy reduces the number of concepts obtained. As an example, we will show how to map the ADL Acme [GAR 00] to UML 2.0.

1.2. Metamodeling, why?

An act of metamodeling has the same objective as an act of modeling with the only difference in the modeling purpose. In all cases, it involves supporting all or part of the lifecycle of a model: (formal or informal) specification, design and implementation. In the case of reflexive models, metamodeling makes it able for models to describe themselves.

We distinguish three different views of metamodeling [BOU 97]:

– Metamodeling as a reflexive technique: metamodeling is an act of modeling applied to a model. It can allow a model to self-represent itself: we refer to this as reflective models.

– Metamodeling as a technical dialogue: a picture is worth a thousand words. This technique is increasingly used to explain, comment upon, document and compare models (in particular semi-formal models used in design methods). It is concerned with describing a model by its conceptual concepts, the result of this is a specification step using a semi-formal (most of the time) metamodel. This mapping therefore constitutes an explanatory document and/or documentation of the model. It can also serve as a means of comparison and unification of models.

– Metamodeling as technical engineering: this is concerned with documenting, explaining and unifying "semi-formal" models. Natural languages, if not so ambiguous, could be used as a metamodel. This is not the case in an objective to support model engineering. It means applying our own engineering techniques to our models. We can model a model for the same reason as application systems are modeled.

1.3. Software architecture metamodeling

Metamodeling techniques have now reached maturity and have been widely used to address real problems in programming languages, databases, engineering models or distributed systems. In this chapter, we will show how these techniques can be applied to the field of software architecture. We will

also highlight the need to propose mechanisms of reflexivity in the context of software architectures metamodeling.

In knowledge representation, we talk about meta-knowledge to evoke knowledge about knowledge, metamodel for a model representing a model, etc. [OUS 02]. In the context of software architectures, metamodeling is an act of modeling applied to the architecture. The result of an act of modeling, i.e. using an architecture to establish an abstraction of an application architecture, is called architecture (level A1 in Figure 1.1) the A0 application. Similarly, the meta-architecture of an architecture itself is an architecture that models an architecture. Since the act of modeling applies to software architectures, the process is called meta-architecting. A meta-architecture is therefore a formal or semi-formal ADL that allows the modeling of particular systems, namely architectures. The meta-architecture (level A2) is an architecture itself and therefore, in a more general way, a system can thus be modeled. We then obtain the architecture of the meta-architecture: meta-meta-architecture (level A3). As with any recurring modeling, this process should stop on a reflexive architecture, that is self-description. The number of levels makes little difference, but it seems that three levels of modeling are sufficient in an architecture engineering framework where level A3 will be used to self-model and model ADLs.

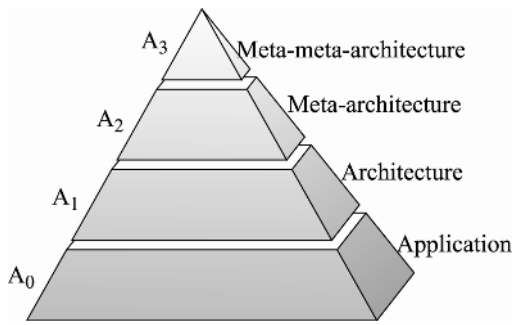

Figure 1.1. *Four levels of modeling in software architectures*

In the context of object-oriented models, this self-modeling is generally implemented by the concept of a metaclass that guarantees the uniformity of concepts and extensibility of the system. This involves designing an

architecture by itself or using an architecture to describe or design another architecture.

Metamodeling can also be a means of formalizing semi-formal architectures. Formal ADLs based on mathematical forms ensure unambiguous specification. However, these formal ADLs may seem repulsive to application designers, and they are not easily understood by an untrained person. It is therefore useful to be capable of carrying out transcription of a semi-formal specification using a formal specification. The approach is to specify, once and for all, the concepts of semi-formal architecture in a formal specification. We then obtain a meta-architecture of the semi-formal architecture, which can be used directly. The formal architecture is used as a meta-architecture and produces the meta-architecture of the informal architecture.

To summarize, we can say that the meta-architecture is a good way to:

– Standardize: architectures that are based on well-defined semantics. These semantics are provided through meta-architectures. Each architecture must conform to a meta-architecture, which shows how to define architectures. The description of different architectures using the same meta-architecture gives the meta-architecture a standardization role.

– Use and reuse: the same meta-architecture can be used and reused several times to define new architectures.

– Compare: the meta-architecture is a good tool to compare several architectures. In fact, the description of several architectures with the same formalism facilitates comparison and analysis.

– Define and integrate multiple architectures: the meta-architecture facilitates and supports exchange of architectures between the ADLs.

1.4. MADL: a meta-architecture description language

1.4.1. *Four levels of modeling in software architectures*

The four levels of OMG metamodeling [MOF 02] (see Table 1.1) can be applied to software architecture. The result is a conceptual structure of four levels: meta-meta-architecture level, meta-architecture level, architecture level and application level, as shown in Table 1.1:

– The meta-meta-architecture level (M^2A, denoted as level A3): this provides the minimum components of modeling architecture. The basic concepts of an ADL are represented at this level (e.g. metacomponent and a metaconnector in Figure 1.2).

– The meta-architecture level (MA, denoted as level A2): this provides the basic modeling components for an ADL – component, connector, architecture, ports, roles, etc. These basic concepts are used to define different architectures. Meta-architectures conform to the meta-meta-architectures. As part of a conformity relation, each element of MA is associated with an element of M^2A. For example, in Figure 1.2, component is associated with metacomponent.

– The architecture level (known as level A or A1): at this level, several types of components, connectors and architectures are described. Architectures comply with meta-architectures (ADL); therefore, each element of A is associated with an element of MA. For example, in Figure 1.2, the client and server are components, the RPC is a connector and the client-server is a system.

– The application level (A_0): A_0 is the place where the executive bodies are located. An application is seen as a set of instances of types of components, connectors and architectures. Applications are consistent with an architecture. Each element A_0 is associated with an element of A. For example, in Figure 1.2, CL1 is an instance of client, S1 is an instance of server, RPC1 is an instance of RPC and C-S is an instance of client-server.

	Modeling by objects	Modeling by components
The meta-metamodel at M3 level	MOF	MADL
The meta-metamodel at M2 level	UML, CWM, SPEM, etc.	Acme, COSA, UniCon, C2, Fractal, etc.
The model at M1 level	Models	Architectures
The instance at M0 level	Information	Applications

Table 1.1. *The four conceptual levels in object-oriented modeling and component-oriented modeling*

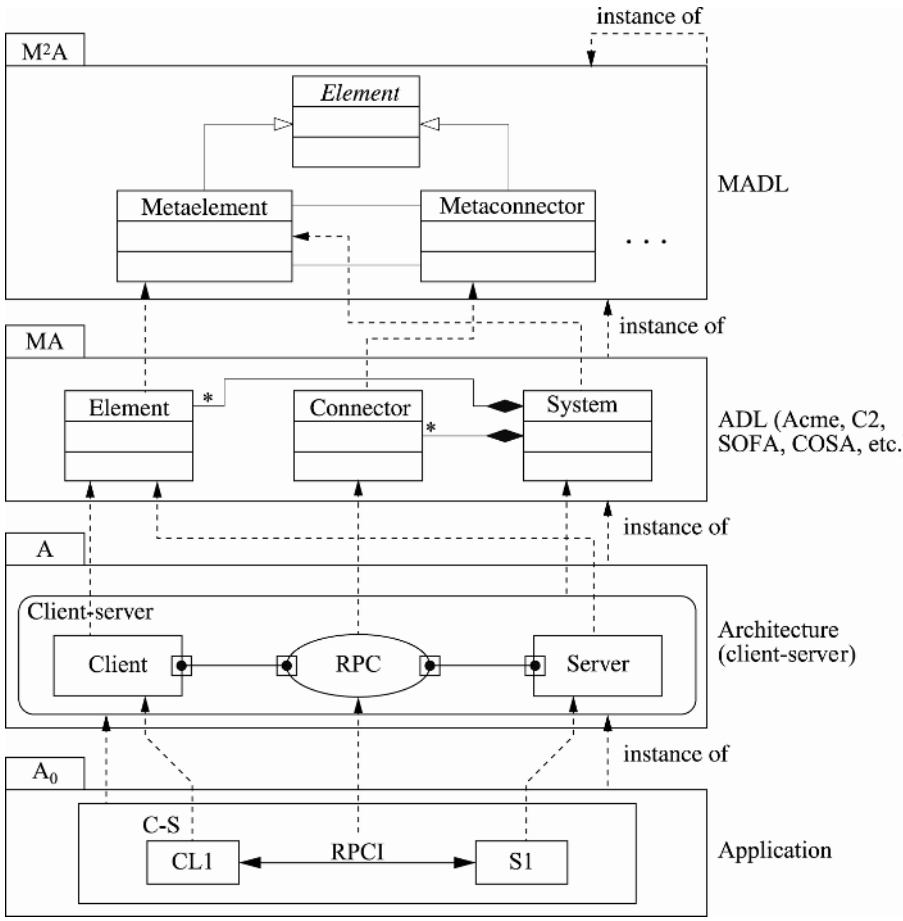

Figure 1.2. *The application of the four levels of OMG in software architecture*

1.4.2. *MADL: reflexive core dedicated to the meta-meta-architecture*

In this section, we describe the entities that are essential to precisely define a reflexive meta-meta-architecture dedicated to metamodeling. This meta-meta-architecture, like the MOF, functions as a unified solution of representing architectures. It also needs to define a minimum core whose purpose is to define the concepts and the following basic relations:

– meta-element to define components and compotation units;

– metaconnector to define connectors and associations;

– meta-architecture to define architectures and configurations;

– metainterface to define interfaces.

MADL is a “reflexive” model, i.e. it models itself. This provision is intended to end the “stacking” of architecture models, i.e. the use of an architectural model to model the considered architecture model. In theory, such a process can be infinite.

1.4.3. *MADL structure*

The meta-meta-architecture must be a minimum core whose purpose is to define the components of meta-meta-architecture, which in its part defines the element types for meta-architectures. It introduces the concepts of metacomponents, metaconnectors, metainterfaces and meta-architectures that are required to process and define the architectural concepts (structural and behavioral). These metaconcepts are organized in a meta-meta-architecture called MADL, shown in Figure 1.3 [SME 05a, SME 05b].

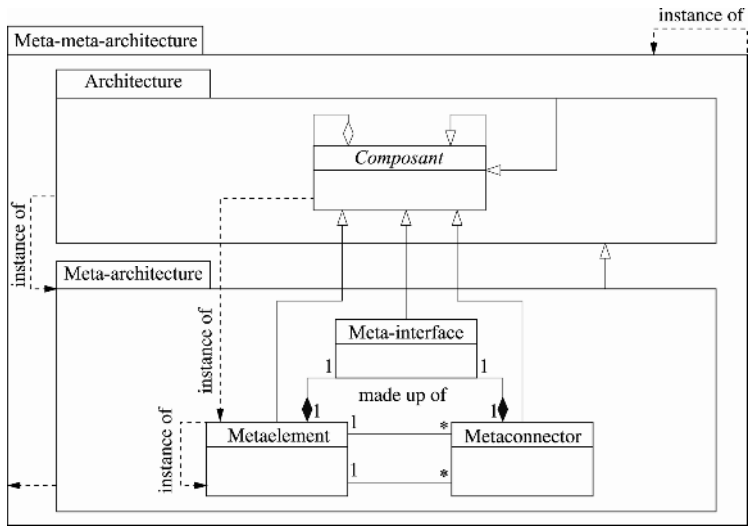

Figure 1.3. *MADL structure*

Some basic principles about the structure of MADL are the following:

– MADL is component oriented in the sense that everything is a component (all components are subtypes of the abstract class “Component”), as in the MOF where everything is subclass of the abstract class “modelElement”. We consider that the component is the basic architectural entity of the model.

– Each architecture must be explicitly derived from a meta-architecture; thus, all components of an architecture component are derived from the meta-architecture.

– To remain compatible with the four conceptual levels of the OMG, each architecture is an instance of its superior architecture (its meta) including the meta-meta-architecture, which is an instance of itself.

– Dependencies between architectures and components are based on the dependencies used in the model of the OMG, including instantiation, inheritance and composition. Thus, we do not introduce new relationships.

MADL is organized into three packages:

– The meta-meta-architecture package: to define an architecture we need a meta-architecture, and to define a meta-architecture we need a meta-meta-architecture. Hence, the meta-meta-architecture package reflects the fact that any architecture is derived from a meta-architecture. The meta-meta-architecture package has all the concepts needed to define meta-architectures and architectures. Meta-meta-architecture is not consistent with another architecture, but it acts as its own meta-architecture. Similarly, each element of a meta-meta-architecture must be combined with another element of a meta-meta-architecture, to meet the self-consistent relationship, except the meta-meta-architecture package, which is an instance of itself.

– The meta-architecture package: to define architectures, we need a meta-architecture; thus, the meta-architecture package classifies and defines architectures. Architectures contain components and connectors. Therefore, meta-element and metaconnector are components of the meta-architecture package. Accordingly, each component and each connector at the MA level must be part of an architecture at the MA level. Meta-architecture itself is an architecture; therefore, meta-architecture inherits the architecture package. To allow architectures at the MA level to have interfaces, meta-architecture also includes metainterface within its components. Meta-architecture is consistent with the definition of meta-meta-architecture,

that is, it is an instance of meta-meta-architecture. The meta-architecture package contains the following metacomponents:

- Meta-element: this is an architectural metaelement that classifies and defines constructors for the computing and support units at the MA level. Meta-element is a component, so it inherits from the component class of the architecture package. Meta-element is a part of the meta-architecture package. To respect the principle of reflexivity upon which the MADL is based, meta-element is an instance of itself;

- Metaconnector: this is an architectural meta-element that classifies and defines constructors for the interactions at the MA level. Metaconnector is part of the meta-architecture package;

- Metainterface: this is an architectural meta-element that classifies and defines interfaces. Meta-element, metaconnector and meta-architecture may have metainterfaces to allow the definition of interfaces for components, connectors and architectures at the MA level. We assign a metainterface to architecture for enabling architectures at the MA level to communicate with other architectures or components. Moreover, metainterface can be composed of another architecture and can inherit other architectures.

– The architecture package: software architecture components are part of architectures; also MADL components are part of the MADL architecture package. The principle on which the MADL is built, “everything is a component”, is applied to the architecture package. Thus, the architecture package inherits components. Note that this relationship is a conceptual relationship. Therefore, architectures at the MA level behave like components, i.e. they can communicate and have a relationship of composition, and inheritance architecture is an instance of meta-architecture. Architecture consists of the abstract class component that classifies and defines all components of MADL. Therefore, all components of MADL are inherited, directly or indirectly (the principle of “everything is a component”) from the component class. Components and connectors at the MA level may be generalizable and specializable, and they can also be composed of other components. This justifies the existence of the inheritance relationship and the relationship of the composition between the component class and itself, and thus allows components at the MA level to have these relationships. The component class is a part of the architecture package, and each element at the MA level must be part of an architecture.

1.4.4. *MADL instantiation: example of the ADL Acme*

The definition of a meta-architecture can be approached in two ways:

– In the first definition: “a meta-architecture is considered as an architecture whose instances are architectures”. This definition fits well within the paradigm of object-oriented languages. According to this definition, MADL can be considered as a MADL allowing the creation of concepts of languages such as Acme and C2. Based on this definition, a meta-architecture can help create architectures by instantiation like in object-oriented languages.

– In the second definition: “a meta-architecture is a representation of an architecture made with an architecture model”. This definition is applied to designate the representation of an Acme architecture made with an extension of a MADL meta-architecture. Thus, for example, the “Meta-element” Acme modeled with MADL represents all components that can be created with Acme such as client components and servers. Based on this definition, a meta-architecture is derived from an architecture by representation operation.

The above two definitions of a meta-architecture are based on two operations:

– instantiation: operation that helps create architectures from a meta-architecture;

– representation of an architecture: operation that helps create a meta-architecture from an architecture.

To define a new meta-architecture (new ADL), MADL is instantiated, and a new model consistent with the definition of MADL is obtained. Each element of the meta-architecture is an instance of an element MADL. For example, the components and notations related to computation are instances of meta-element; components and notations related to interaction and communication are instances of metaconnector. Thus, for example, components, properties and constraints are instances of meta-element, while connectors, bindings and attachments are instances of metaconnector.

Figure 1.4 shows how MADL can be instantiated to obtain Acme. As shown in this figure, each Acme element must conform to an element of MADL. Components and systems are instances of meta-element; connectors

are instances of metaconnector; ports and roles are instances of metainterface; attachments (which connect a port to a role) and bindings (which connect two ports or two roles) are instances of metaconnector. Similarly, styles (which define families of systems) are instances of meta-architecture.

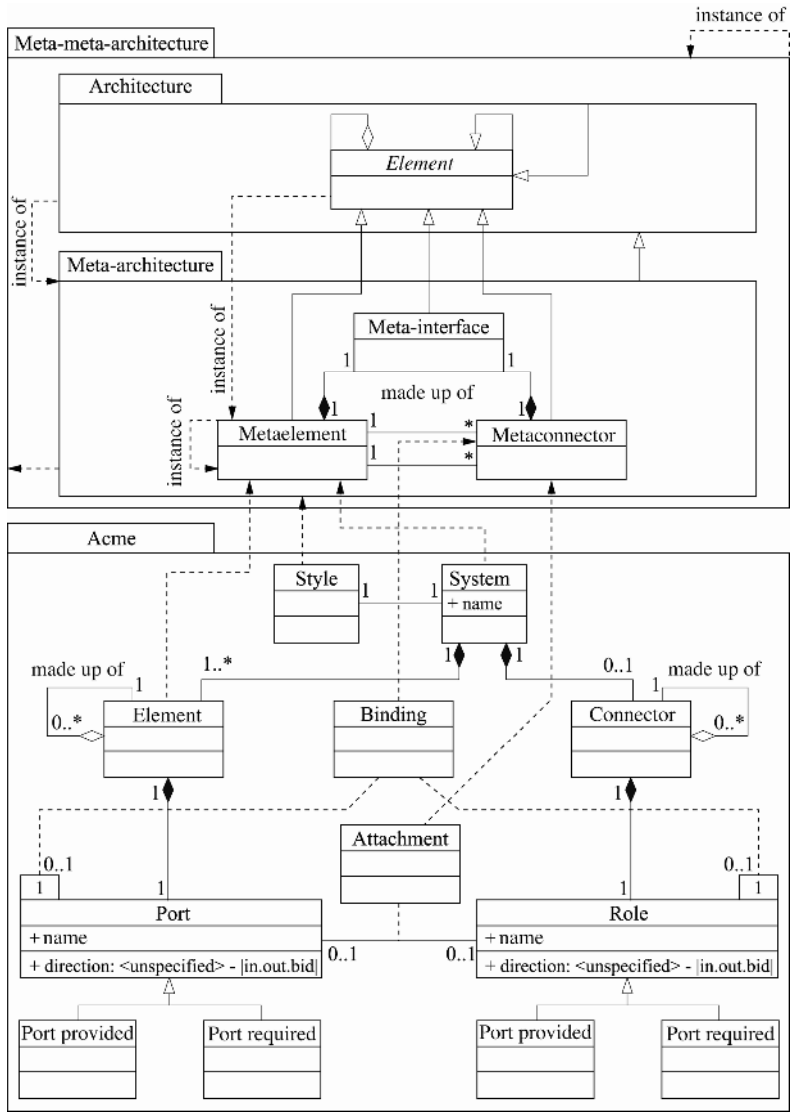

Figure 1.4. *MADL instantiation to obtain Acme*

1.4.5. *Comparison of MADL and MDA/MOF*

In this section, we will try to position MADL compared to the MDA (model-driven architecture [FRA 03]) approach and to MOF.

1.4.5.1. *Model-driven architecture (MDA) approach*

The MDA approach was proposed by the OMG. It is presented as a product in line with the OMG's concerns of integration and interoperability ("ensure that you can incorporate what you've built, with what you are building, with what you will build" [FRA 03]).

MDA provides a response to the complexity of middleware. These middleware can be "standard" such as CORBA, DCOM or "somewhere in between" (JMI or HTTP/XML/SOAP) [TRA 05]. The idea of MDA is to provide a "stable", independent model for middleware, from which it is possible to derive different tools. The objective is to build the model and use/reuse it instead of constantly migrating one middleware to another. The MDA is based on the OMG modeling standards: UML, CWM and MOF.

In fact, MDA offers several core models, corresponding to the M2 level of the four levels of metamodeling architecture. According to the terminology of the OMG, these models are also called "UML Profiles". They are dedicated to a class of problems, such as Enterprise computing or Real-Time computing. These models are independent of any platform. Their number is expected to grow to cover other needs. However, it is intended to be relatively unimportant because everyone wants to be "general" and includes features common to all problems in its category.

The UML profile is the foundation of the MDA approach, which explains the use of the equivalent term "core model". Below, we give an overview of the concept of the UML profile. At present, there is no standard definition of the UML profile. However, it is widely accepted that a UML profile is a specification consistent with one or more of the following points:

– Identification of a subset of the UML metamodel: this subset can be the entire UML metamodel;

– Definition of well-formalized rules, in addition to those contained in the subset of the UML metamodel: a *well-formalized rule* is a term used in the standardized specification of the UML metamodel. Such rules help describe

a set of constraints in natural language (or using the Object Constraint Language (OCL)) to define an element of the metamodel;

– Definition of standard components (standard components) in addition to those contained in the subset of the UML metamodel: a *standard element* is a term used in the specification of the UML metamodel to describe a standard instance of a UML stereotype or a constraint;

– Definition of new semantics expressed in natural languages, in addition to those contained in the subset of the UML metamodel;

– Definition of common components of the model, i.e. instances of UML components, expressed in terms of a profile.

Whatever the target platform (component corba model (CCM), entreprise java bean (EJB), etc.), the first step in developing an MDA-based application is the creation of a Platform Independent Model (PIM) by instantiating the metamodel.

Second, specialists of the target platform are responsible for the conversion of this general application model to CCM, EJB or another platform. Standard mappings, based on the "core model", help consider a partial automation of the conversion.

Not only artifacts specific to the platform are generated (in interface description langage (IDL) and other languages), but also a specific model (Platform-Specific Model (PSM)) is generated in this way. This model is also described using a UML profile. This is called "jargon". Due to the mapping, UML reveals components related to the implementation on the target platform. This gives a richer sense to the semantics of the solution than with IDL or XML.

The next step is the generation of the code itself. The higher the UML of the PSM reflects the target platform, the better the semantics and behavior of the application can be integrated into the PSM, and the code can be generated.

Among the advantages of the MDA approach described by the OMG, we can report the simplicity of managing interoperability between developed applications using the same "core model".

Although the MDA approach is very often associated with profiles, the OMG describes the MDA as a general framework for metamodeling in

which the model is considered a first-class entity. Applicable operations to these models (mainly PIM and PSM) are of two kinds:

– Manipulation: this term covers various operations such as storage, exchange, display, derivation, fusion and alignment;

– Mapping (between models): conventionally, we count "PIM2PIM", "PIM2PSM", "PSM2PSM" mappings, etc.

1.4.5.2. *Positioning MADL/MDA/MOF*

The MDA approach has a number of advantages. From the general point of view, it combines the advantages of metamodeling and standardization implemented by the OMG. If the number of basic models available is low, it is, however, not fixed. The MDA device is "extensible" through new models, as much as the MDA approach leaves open many challenges. Strictly speaking, the integration activity, the consideration of legacy applications of previous developments and the management of interoperability between middleware are points frequently mentioned by the OMG as a number of avenues to be explored. In addition:

– If the maximum automation of the mapping is a goal, we must consider the use of the "manual" procedure, in the absence of MDA tools.

– At present, the degree of automatic code generation is relatively low.

– The OMG recommends completing the middleware-centered approach by a model-centered approach.

As a result, it is possible to position the MADL as a new MDA base model. More specifically, it involves proposing a new UML profile (PIM) dedicated to software architecture whose originality, besides the fact that it deals with a meta-architecture problem understudied in the ADL, lies in the nature of its specialization. In fact, existing UML profiles are dedicated to an application type (distributed systems, real time, etc.); thus, MADL would be "orthogonal", specialized in a particular activity of modeling ADLs. Being able to represent meta-architectures with MADL with UML notation provides a natural reuse of the graphical UML modeling tools associated with it.

More specifically, the "UML Profile" approach is aimed to extend the UML metamodel by adding new concepts of ADL. This operation is based on the use of stereotypes, tagged values and OCL constraints [WAR 98]. The recognized advantage of this approach is the use of UML tools. The major drawback lies in the difficulty to understand the separation between the meta-meta-architecture (M^2A level), the meta-architecture (MA level) and architecture (level A). The term "dialect UML" is also used to describe this situation.

The OMG proposes another metamodeling approach, which overcomes this problem. This approach is called the "MOF". It involves building a new metamodel binding metaclasses by meta-associations and specifying OCL constraints. It is an alternative to positioning the MADL. This approach is tantamount to depriving de facto the UML tools and in any event has no ADL concepts equipped with independent semantic loads from object-oriented support concepts.

This is usually shown as the main drawback of this approach. At present, the MOF approach is less "popular" than the UML Profile approach. The latter is clearly implemented more frequently.

Currently, some research conducted by the ADL community focuses on the development of "second generation" generic languages. The ADL approach arising from academic circles has not been imposed. It is often criticized for using difficult to implement formal notations and for providing tools that are difficult to process.

Research is thus taking a new direction toward the work of the OMG. These two communities have worked in parallel in recent years; the UML semantics poverty in terms of software architecture concepts probably being the cause (at least partially). Nowadays, the integration of the component concept in UML 2.0 can also be interpreted as a sign of this development and suggests the possibility of defining architectures (based on components) whose passage to the adapted platforms of execution will be facilitated.

We place the MADL in this latest trend. It combines the advantages of the ADL approach and of the MDA/MOF of the OMG approach. From the ADL approach, it retains the concepts and mechanisms inherent to components, connectors and architectures. As for the MDA/MOF approach based on the metamodeling, it has the advantage of being able to consider the models

(in our case, architectures, meta-architectures and meta-meta-architectures) as first-class entities, and being able to apply various operations, in particular, mappings enabling the implementation of the system on different execution platforms [ALT 07].

Standardization of architectural concepts around the MADL allows the provision of common functionalities for handling different architectures arising from these different meta-architectures. MADL has therefore been defined as an expression language of meta-architectures. This is its main function, but it can also be used so as to be independent of a particular ADL and to manipulate the architecture of these ADLs.

Concerning the technical point of view of the MADL, we have opted for a MOF-type approach, but one which explicitly responds to the following shortcomings:

– the absence of meta-entities representing meta-architectures;

– the absence of meta-entities representing architectures;

– the absence of meta-entities representing components;

– the absence of meta-entities representing connectors;

– the lack of an explicit relationship between an architecture and its meta-architecture;

– the lack of an explicit relationship between an element and its meta-element, a connector and its metaconnector.

In summary, we can consider MADL as a “new” MOF intended for modeling ADLs and, more importantly, it can be part of the MDA approach.

1.5. Mapping of ADLs to UML

Some studies have tried to establish a mapping of an ADL to UML such as Acme to UML and C2 to UML. In this section, we show an approach enabling the mapping of a given ADL to UML 2.0. This method is based on the instantiation of the MADL by ADL, then mapping MADL to MOF and, finally, the instantiation of the MOF by the UML.

1.5.1. *Why to map an ADL to UML?*

After the introduction of UML as a unified language for all notations and object-oriented modeling techniques, object-oriented modeling has become a *de facto* standard in the development process of software systems. In fact, UML has become a standard language for specifying, visualizing, constructing and documenting the objects of software systems [BOO 98]. However, UML lacks the semantic support for capturing and exploiting certain architectural aspects whose importance has been demonstrated by software architecture research and practice. In particular, UML lacks direct support for modeling and exploiting connectors, interfaces and architectural styles. However, with the introduction of UML 2.0 [OMG 03], new notations have been proposed and existing ones were modified to meet the needs of software architecture, including:

– the definition of components as a kind of classifier, thus the components may have instances and have access to some mechanisms, such as subtyping by the generalization relationship, behavioral description, internal structure, interfaces and ports;

– the redefinition of interfaces that can include not only interfaces but also the required interfaces;

– the introduction of ports as points of interaction for classifiers;

– the introduction of structured classifiers to represent the internal structure (decomposition) of classifiers;

– the introduction of connectors to represent a connection between two or more instances. However, connectors in UML 2.0 are defined by a simple relationship between two components and thus cannot be associated with a behavioral description or attributes that characterize the connection.

The idea of mapping ADL concepts to UML concepts enables UML to include semantics related to ADLs and also to get the benefit of:

– the variety of advantages of UML such as multiple views, a semiformal semantics expressed in a metamodel, a powerful associated language to express constraints (e.g. OCL [WAR 98]) and a code generation rules;

– a variety of tools that are implemented for UML, such as Rational Rose, Microsoft Visual Studio, Poseidon for UML. Most of these tools provide code generation services in different languages such as C++, Java and C#;

– the widespread presence of UML in the field of modeling and the description of software systems, where UML is a dominant standard for the analysis and design of software systems;

– the MDA approach, which aims to provide an accurate and efficient framework for the production and maintenance of software [FRA 03].

1.5.2. *ADL mapping to UML*

Mapping an ADL to UML can be done in two ways: either manually by examining all UML notations and then selecting the appropriate concepts, or semi-automatically by defining a connection between their respective metamodels. In the second strategy, we map the notations of the meta ADL to MOF and, by instantiation, we obtain the correspondence of their respective instances. Figure 1.5 shows the two mapping strategies.

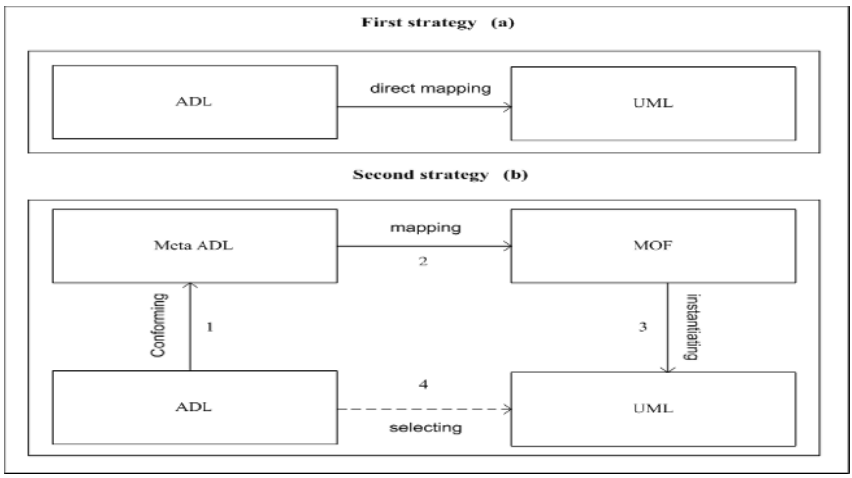

Figure 1.5. *The two strategies for mapping an ADL to UML*

1.5.2.1. *First mapping strategy (at the meta level)*

This strategy should be considered first because it has the merit of having paved the way to work regarding mapping of ADL to UML. It is based on the review of all connections between the notations and semantics of UML and a given ADL. If appropriate notations are not found, the UML extension mechanisms (stereotypes, tagged values, constraints, etc.) are used (see Figure 1.5(a)). However, this strategy has a number of disadvantages:

– The mapping is done manually. A well-established methodology that guides the process does not currently exist, but some researchers have proposed a number of criteria in order to justify their choices [GAR 02].

– This strategy requires that the entire UML metamodel should be considered for each ADL notation.

– The same long research process takes place every time a new ADL is considered for mapping to UML.

This strategy has been used in several studies. In [MED 02], the authors presented two approaches to support components and architectural notations in UML 1.4. The first approach uses UML notations without changes, "UML as is", while the second approach uses the UML extension mechanisms (stereotypes, tagged values, constraints, etc.) to incorporate the concepts of three ADLs (C2, Wright Fast). The mapping of C2 is shown in Table 1.2 and the mapping of Wright is given in Table 1.3.

In [GAR 02], the authors have tried to select a number of UML 1.4 notations that best represent architectural notations. They stressed the advantages and limitations of each notation and showed aspects of architectural description that are intrinsically difficult to model by UML 1.4. However, UML 1.4 and earlier versions are not adequate enough to represent architectural concepts such as connectors, configurations, interfaces (ports and roles), styles or even components as components in UML 1.4 are representations of a deployable and replaceable system part which encapsulates execution.

UML 2.0 [OMG 03] has been enriched by new architecture concepts such as connectors, ports and structural classifiers and has redefined the concept of components to be a subtype of classes in UML metamodel. Thus, components have an expressive power similar to the classes as they may have interfaces, they can contain other components or classes, etc.

In [GOU 03], the authors considered connectors as stereotyped components without other interfaces than those defined by their roles and properties. However, describing the connectors as components can clutter the design and make it difficult to understand its overall structure and the roles of connector and component ports difficult to distinguish. Mapping Acme to UML 2.0 using this strategy is summarized in Table 1.4.

In [IVE 04], Ivers *et al.* studied the relevance of these new notations to describe the view of components and connectors (C&C) of software architecture, in particular for the Acme language. They examined the mapping of Acme notations that are related to the C&C view (i.e. components, connectors, ports, roles and attachments) to UML 2.0. They chose the semantic connection, visual clarity and support by tools as basic criteria for selecting UML notations, which may represent the architectural description. They offered two choices for each notation (except for attachments that have not been considered) as shown in Table 1.5. They concluded that even if the new notations have improved the description of software architecture using UML, they still have major drawbacks. Moreover, some aspects of architectural description continue to be problematic, for example, the lack of ability of UML 2.0 to associate semantic information with a connector to describe its behavior. Also, UML 2.0 does not distinguish the roles, which are interface connectors, and ports, which are component interfaces.

C2	UML 1.4	OCL/Tagged values
Component (Type)	<<C2 Component>> (Class metaclass instances)	C2 Component must implement exactly two interfaces.
Connector (Type)	<<C2 Connector>> (Class metaclass instances) <<C2 Attach Over Comp>> (Association metaclass instances) <<C2 Attach Under Comp>> (Association metaclass instances) <<C2 Attach Conn Conn>> (Association metaclass instances)	C2 Connector must implement exactly two interfaces. C2 Attachments are binary associations, one end of attachment must be a C2 Component and the other end must be a C2 Connector.
Port	<<C2 Interface>> (Interface metaclass instances)	
Role	<<C2 Interface>>(Interface metaclass instances)	
System	<<C2 Architecture>> (Model metaclass instances <<C2 Attach>>	Architecture is a network of C2 concepts.
Message	<<C2 Operation>> (Operation metaclass instances)	C2 Operations are labeled as notifications or requests and as incoming or outgoing.
Interface	<<C2 Interface>>	

Table 1.2. *Mapping C2 to UML 1.4 [MED 02]*

Wright	UML 1.4	OCL/Tagged values
Component (Type)	<<WrightComponent>> (Class metaclass instances)	WrightComponent must implement at least one WrightInterface.
Connector (Type)	<<WrightConnector>> (Class metaclass instances) <<WrightGlue>> (Operation metaclass instances)	WrightComponent must implement at least one WrightInterface. WrightGlue contains a WrightStateMachine.
Port	<<WrightInterface>> (Interface metaclass instances)	
Role	<<WrightInterface>> (Interface metaclass instances)	
System (Architecture)	<<WrightArchitecture>> (Modelmetaclass instances) <<WrightAttachment>> (Association metaclass instances) <<WrightComponentInstance>> (WrightComponent instances) <<WrightConnectorInstance>> (WrighConnector instances)	Architecture is composed of instances of components and connectors.
CSP protocol (state machine)	<<WSMTransition>> (Transitionmetaclass instances) <<WrightState>> (Statemetaclass instances) <<WrightStateMachine>> (StateMachinemetaclass instances)	WSMTransition is labeled as an event or action. All transitions made in a WrightState must be WSMTransitions.
Operation	<<WrightOperation>> (Operation metaclass instances)	WrightOperations have no parameters. All operations in a WrightInterface are WrightOperations.
Interface	<<WrightInterface>> (Interface metaclass instances)	WrightInterface is labeled as ports or roles.

Table 1.3. *Mapping Wright to UML 1.4 [MED 02]*

Acme	UML 2.0	OCL/Tagged values
Component (Type)	<<AcmeComponent>>	Components only have interfaces known as ports or properties.
Connector (Type)	<<AcmeConnector>>	Connectors have no other interfaces than those defined by their roles.
Port	Port	Ports can only be used with Acme components and they have a provided interface and a required interface.
Role	<<AcmeRole>>	Roles are related to Acme connectors and have a provided interface and a required interface.
System	<<AcmeSystem>>	Systems represent a graph of components that communicate with each other.
Rep-maps	Delegation connector	All delegation connectors binding ports and roles
Properties	<<AcmeProperties>>	<<AcmeProperty>> port has a provided interface that must provide the *get* and *set* operations for the value and the type of property.
Constraints	<<AcmeConstraints>>	This stereotype must have an enumerated attribute with two allowed values: invariant and heuristic.
Style (Family)	Package	All connectors used in a pipe filter system must comply with PipeT.

Table 1.4. *Mapping Acme to UML 2.0 [GOU 03]*

Acme	Choice 1	Choice 2
Element (Type)	Object (Class)	Element instance (Element)
Connector (Type)	Object binding (Association class)	Object (Class)
Port	Port	Port
Role	Port	Port
Attachment	–	Assembly connector

Table 1.5. *Mapping Acme to UML 2.0 [IVE 04]*

1.5.2.2. *Second mapping strategy (at the meta-meta level)*

We believe that the best way to benefit from UML in the field of software architecture is to map the concepts of architectural description to the UML metamodel (certainly without changing the metamodel itself and taking advantage of all its tools and environment). To do this, we propose to establish the connection of architectural concepts and objects at the meta level (MOF) to reduce the number of concepts to be translated and help improve the readability and visibility of different notations at stake. Therefore, we rely on the MADL as meta ADL for software architecture and then map the MADL components to the MOF. In this strategy, the mapping is carried out in four steps [SME 05a]:

– instantiating MADL to obtain the desired ADL (i.e. we want to map to UML);

– mapping each MADL element to an MOF element;

– instantiating each element of the MOF for one or more UML components. At the end of this step, we can obtain more than one choice for each ADL component, but the number of choices is very limited;

– reviewing and selecting the UML concept, which is the most suitable for our ADL, by using appropriate criteria.

The advantages of this strategy are:

– Instead of working with the UML metamodel, we work with the UML meta-metamodel (MOF); therefore, the number of notations and elements that we deal with is considerably less.

– The passage from an ADL to the meta-ADL is natural and inherent, hence quite easy, where the two use coherent architectural elements.

– The passage from MOF to UML already exists and is well defined.

– The selection of the corresponding UML concept is easier when the number of choices is reduced.

However, this strategy requires the definition of a higher level of abstraction for the software architecture, i.e. the metacomponents, the metaconnectors and the metaconfigurations. That is why we use the MADL meta-metamodel.

In the following sections, we will describe the process of mapping an ADL to UML (Figure 1.5(b)).

1.5.2.2.1. Instantiating MADL by an ADL

The first step (Figure 1.5(b)) is instantiating MADL by an ADL; each component of the ADL must comply with a component of the MADL. For example, we will try to map Acme to UML 2.0. Figure 1.4 shows how the MADL was instantiated by Acme. Components and systems (which represent configurations of components and connectors), connectors, attachments (which connect a port and role), connections (which connect two ports or two roles), ports and roles (which are the interfaces of components and connectors), styles (which define families of systems) are obtained through successive instantiations of the meta-element, metaconnector, metainterface and meta-architecture. Table 1.6 summarizes the instantiation relationship of the MADL by Acme.

Acme	MADL
Element	Meta-element
Connector	Metaconnector
System	Meta-element
Attachment	Metaconnector
Binding (*Rep-Map*)	Metaconnector
Port	Metainterface
Role	Metainterface
Architectural style	Meta-architecture

Table 1.6. *MADL instantiation by Acme*

1.5.2.2.2. Mapping of MADL concepts to MOF

MOF is a meta-metamodel, which is used to define metamodels (e.g. UML). In principle, MOF can be used to define a metamodel for software architecture. However, as we noted earlier, MOF has a number of limitations on the architectural description.

In fact, MOF contains basic concepts to define meta-entities (MOF Class), the metarelations (MOF Association) and packages acting as containers for these meta-entities and metarelations. A meta-architecture described with the MOF for example, will consist of a main package that contains all the MOF definition of the meta-architecture. Therefore, although there is no concept for representing the MOF meta-architectures, a package can be used for this purpose.

A package can therefore be used and generalized to represent a meta-architecture, but nothing can be used to represent an architecture. Therefore, the MOF relationship, which enables binding of an architecture to its meta-architecture, does not exist. In addition, although the concept of the MOF Class may represent the concept of meta element, no concept can be used to represent a component or connector. Therefore, no MOF relationship can be defined between a component and its meta-element or a connector and its metaconnector.

It may be noted that no relationship in the MOF enables the specification that an architecture is defined by a meta-architecture (i.e. all the architectural components of an architecture cannot find their type in that meta-architecture).

For all these reasons, we turned to the definition of MADL, which is equivalent to the MOF for software architecture. In fact, MADL, by construction, includes the concepts and mechanisms of ADLs.

When considering integration strategies, it is important to define a number of criteria to select a notation from several notations. These criteria help determine whether a particular notation is likely to be appropriate or not. The criteria that we selected for our study are:

– the semantic connection: the interpretation chosen by the MOF notations must comply with the homogeneous interpretation of the original description of the MADL;

– expressiveness: all architectural concepts that are defined in the MADL must have the capacity to be represented by the MOF;

– structural connection: mapping (of the topology) of the MADL on the MOF must comply not only to the MADL but also to the MOF;

– compliance with the MOF specification: mapping of the MADL must correspond to specific components of the MOF complying with their definitions.

Each component of the MADL is represented as a subclass component of the MOF, which has a similar semantic. The reason for which we define MADL notations as subclasses of MOF notations is that we do not want to violate the semantic of the MOF by changing its structure with new associations and new notations.

However, if no semantic connection is found, a new component is introduced by stereotyping a subclass. In the following, we map each component of the MADL to its connection in the MOF (see Figure 1.6):

– A component of the MADL is mapped to ModelElement MOF. In MOF, ModelElement is an abstract class that classifies the basic manufacturer's models. It is the root element in the MOF model. Thus, all components of MOF are subclasses of ModelElement. The same principle is applied to the MADL. Component classifies and defines all components and entities of the MADL (MADL is based on the principle that "everything is a component").

– The MADL Architecture is mapped to Namespace MOF. In MOF, Namespace is a subclass of ModelElement. It classifies and characterizes ModelComponents, which may contain other ModelComponents. Namespace contains ModelComponents. Note that in the MADL, the Architecture contains one or more components.

– The meta-element of the MADL can be represented by the MOF class. A class in the MOF defines a classification for a set of instances defining the state and behavior. It is a subclass of class Namespace. A class can be associated with another class. In MADL, meta-element classifies and defines the components that may associate themselves.

– The MOF association is the metaconnector of the MADL. The Association in the MOF defines a classification for a set of bindings through classes. Each binding, which is an instance of the association, denotes a connection between instances of Class. In the MADL, the metaconnector defines connectors that handle relationships between components. The Association is redefined by the composition relationship with the Feature class in order to have connectors with interfaces.

– The metainterface of the MADL is mapped to the MOF Feature class. In the MOF, Feature defines static and dynamic characteristics of ModelElement. Thus, it defines the services offered by ModelElement. The metainterface in the MADL classifies and defines interfaces for components and connectors. These interfaces represent services provided or required by a component or connector.

– The package of the MOF represents the MOF meta-architecture. In the MOF, a package is a container for a collection of ModelComponents that forms a logical metamodel. Packages may consist of and inherit other packages. The meta-architecture classifies and defines the architectures, which are containers of components and connectors, and can compose and inherit other architectures. The definition of the package is redefined; we then have architectures with interfaces and therefore they may be combined with other architectures.

The MADL meta-meta-architecture as a meta-architecture is stereotyped with the following constraints:

– A meta-meta-architecture only contains meta-architectures and architectures.

– A meta-meta-architecture cannot inherit from other meta-meta-architectures.

The relationship of composition between the meta-meta-architecture and Namespace and Package is defined to allow the meta-meta-architecture to contain meta-architectures and architectures.

Figure 1.6 shows the mapping of the MADL concepts and those of the MOF. Even with the introduction of new relationships, the criteria mentioned above are taken into account and respected. Table 1.7 summarizes this mapping.

In this model, only a stereotype and three relationships (relationship of composition between Feature and Class, Feature and Association, Feature and Package) are added to the original model. Note that other studies have stereotyped almost all concepts (see [GOU 03, MED 02]).

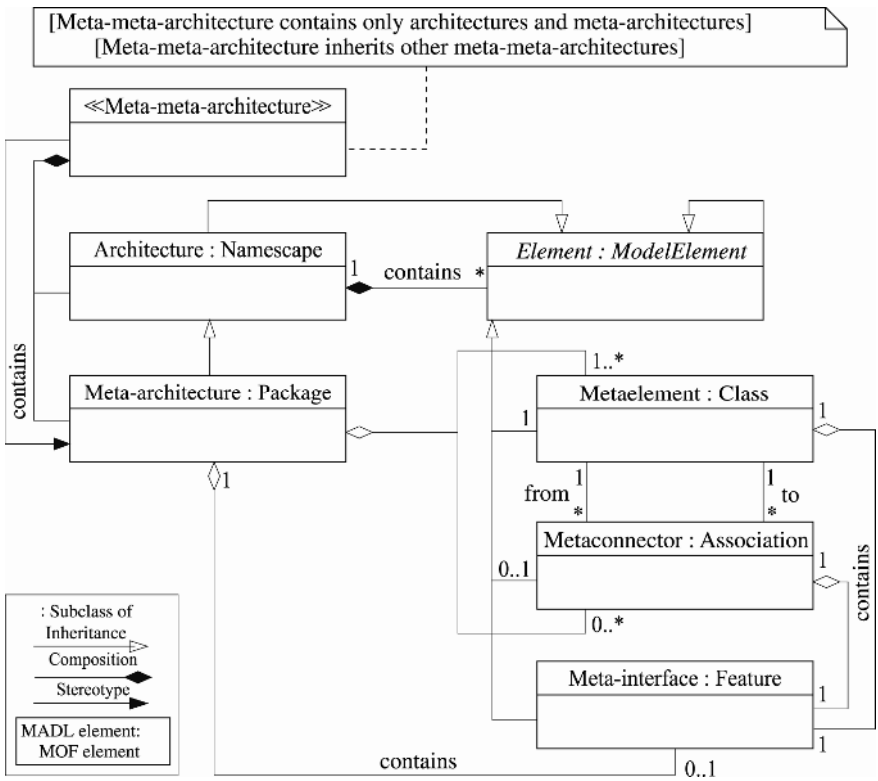

Figure 1.6. *Mapping MADL to MOF*

MADL	MOF
Element	ModelElement
Meta element	Class
Metaconnector	Association
Metainterface	Feature
Architecture	Namespace
Meta-architecture	Package
Meta-meta-architecture	Stereotype package

Table 1.7. *Mapping MADL to MOF*

1.5.2.2.3. Instantiation of MOF by UML

The instantiation of the MOF by UML is done automatically using the specifications of the OMG [OMG 03], which describe the instantiation of the MOF by UML. Each element of a UML is an instance of the MOF component including the UML itself. Table 1.8 shows the possible UML instances for each element of MOF. We deduce the following comments regarding the instantiation of the MOF by UML:

– Components and systems can be represented as UML classes, or UML components or stereotyped UML classes.

– Connectors are mapped to UML classes, UML associations, UM association classes or UML communication paths.

MOF	UML 2.0
Class	Class, Component, Stereotype
Association	Class, Association, AssociationClass, CommunicationPath
Feature	Interface, Port, Operation, Property, Connector UML
Package	Package

Table 1.8. *MOF instantiation*

Associations can be used for connectors defined implicitly (as in Darwin, Fractal, etc.) and classes or association classes, which are statements of semantic relationships, for connectors defined explicitly (as in Wright, Acme, etc.). However, UML communication paths cannot represent ADL connectors because a UML communication path is a physical association that can only be defined between nodes. Therefore, the choice of ADL connectors as UML communication paths is discarded.

ADL interfaces are mapped to UML interfaces, ports, operations, properties or connectors. However, only interfaces, ports and operations may represent the interfaces of software architecture. Other concepts cannot represent interfaces insofar as properties are values denoting a characteristic of a component and the connectors are bindings that allow communication between two or more instances; therefore, this choice is

discarded. In addition, ports may represent points of interaction and operations may represent services for the ADLs which separate the services of points of interaction; architectural styles can be represented as UML packages.

1.5.2.2.4. Selection of UML 2.0 concepts

The last stage of mapping is the selection of the final concepts. At the end of the instantiation step of the MOF by UML, we obtain a reduced number of choices for each ADL concept and we must choose the concept which is the most representative and which is semantically closest to the concept of the ADL mapped. To do this, the user can define and apply some criteria that will help and guide him in his choices.

1.6. A mapping example: the case of the Acme language

Let us take Acme as an example and try to map its concepts to UML 2.0. This example has been studied by Ivers *et al.* [IVE 04] (Table 1.5). Table 1.6 summarizes the mapping of Acme to MADAL. From the table, we can see that Acme components and Acme systems are instances of the MADL meta-element, Acme connectors, bindings and attachments are instances of the MADL metaconnector, Acme ports and roles are instances of the MADL metainterface, and Acme styles are instances of the MADL meta-architecture. Table 1.7 shows that the MADL meta-element is mapped to the MOF class, the MADL metaconnector is mapped to the MOF association, the MADL metainterface is mapped to the MOF Feature class and the MADL meta-architecture is mapped to the MOF package. From Table 1.8, we find that UML classes, components and stereotypes are instances of the MOF class, associations and UML association classes are instances of the MOF association, UML packages are instances of the MOF package and UML ports, operations, properties and connectors are instances of the MOF Feature class.

Therefore, we can deduce that the Acme concepts can be mapped as follows: components and systems to classes, UML components or stereotypes; connectors, bindings and attachments to associations or to UML association classes; Acme styles to UML packages and roles and ports to UML ports or operations. Table 1.9 summarizes the mapping of Acme concepts to UML 2.0.

Acme	UML 2.0	Comment
Element and systems	Class, component, stereotype	The UML components are the most appropriate components to represent the components and systems of Acme. However, stereotyped classes may also represent the Acme components.
Connector	Class, Association, AssociationClass	The association classes (AssociationClasses) are the most appropriate components to represent the Acme connectors. Using AssociationClasses, we can define the behavior of connectors and interfaces.
Style	Package	Systems and styles can be considered as packages.
Port	Interface, Port, Operation	As Acme does not separate the points of interaction of services, UML ports are the most appropriate components to represent the ports and roles of Acme.
Role	Interface, Port, Operation	
Attachment	Association, AssociationClass	The UML associations represent bindings between classes; they represent the most appropriate components to describe attachments and bindings.
Binding (Rep-Map)	Association, AssociationClass	

Table 1.9. *Final mapping step of Acme to UML 2.0*

1.7. Some remarks on the mapping of ADL concepts to UML

The mapping of Acme concepts to UML has raised a number of comments and issues concerning the use of UML to describe software architecture.

1.7.1. *UML 2.0 as an ADL*

UML 2.0 cannot be considered as an ADL even with new notations and redefinitions. For example, the introduction of connectors to represent a binding between two instances is not sufficient to represent connectors of software architecture. A connector in UML 2.0 remains a simple binding between two components and, therefore, cannot be associated with a behavioral description or attributes that characterize the connection. In general, some software architecture notations are absent in UML 2.0. For

example, we can note the absence of entities to represent architectures (configurations) and architectural styles.

1.7.2. *Mapping strategies*

The first strategy:

– This strategy has served as a starting point for the mapping of the ADLs to UML. It is based on an evaluation of the entire UML metamodel for each component of the ADL. Also a long research process takes place, every time a new ADL needs to be mapped.

– The mapping is done manually.

The second strategy:

– This strategy can be seen as a mapping methodology dedicated to software architecture.

– Although this strategy leads to a finite number and reduces choices for each component, it has the merit of systematizing the connection and providing better readability and, therefore, a better understanding of concepts. In fact, the first strategy uses the entire UML metamodel, which has a significant number of notations and definitions. In addition, some of these are not related to software architecture.

This strategy proposes a semi-automatic method for mapping any ADL to UML. All components and notations of software architecture are considered in the meta-meta-architecture. Therefore, the model is reusable and can be applied to most existing ADLs. In fact, steps 2 and 3 of this strategy are completely reusable for other ADLs.

The fact that the largest part of the process is carried out at the meta-meta level allows us to work with a reduced number of notations (more than 100 notations for the meta level and 10 notations for the meta-meta level).

In contrast to existing work [IVE 04] on the mapping of ADLs to UML, the second strategy has the advantage of being reusable on different ADLs and of automatically justifying the selected connections.

The process can be used for any version of the UML as it is based on the same version of the MOF.

1.8. Conclusion

In this chapter, we have discussed the meta-architecture concept. The meta-architecture of an architecture is an architecture that models an architecture. The meta-architecture being itself an architecture can therefore be modeled. We then obtain the architecture of the meta-architecture, i.e. the meta-meta-architecture. We have also presented a meta-metamodel for software architecture called MADL. This meta-metamodel works as a unified solution for representing software architectures. It makes it possible to structure these architectures, reduces their complexity, facilitates the mapping of architectures and, finally, promotes the passage from one to another.

By using this meta-metamodel, we described a strategy to map ADL concepts to UML 2.0. The mapping of architectural notations to UML is strongly encouraged due to the extensive use of UML in the industrial world. This strategy consists of four steps: instantiating MADL by ADL, mapping MADL to MOF, instantiating MOF by UML and, finally, the selection of the most appropriate UML concepts for the ADL concerned. This strategy reduces the number of concepts obtained. To illustrate this strategy, we have shown how to map the concepts of Acme to UML 2.0.

1.9. Bibliography

[ALL 04] ALLOUI I., OQUENDO F., "Describing software-intensive process architectures using a UML-based ADL", *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS '04)*, Porto, Portugal, April 2004.

[ALT 07] ALTI A., KHAMMACI T., SMEDA A., "Integer notation software architecture concepts into the MDA platform with UML profile", *Journal of Computer Science*, vol. 3, no. 10, pp. 793–802, 2007.

[BOO 98] BOOCH G., RUMBAUGH J., JACOBSON I., *The Unified Modeling Language User Guide*, Addison-Wesley Publishing, Reading, 1998.

[BOU 97] BOUNAAS F., CHABRE-PECCOUD M., CUNIN P.Y., *et al.*, "Objets et méta-modélisation", in OUSSALAH M. (ed.), *Ingénierie objet – Concepts et techniques*, Interéditions, Paris, 1997.

[BUD 08] BUDINSKY F., MERKS E., STEINBERG D., *Eclipse Modeling Framework*, Addison-Wesley Professional, Reading, 2008.

[FRA 03] FRANKEL D., *Model Driven Architecture Applying MDA to Enterprise Computing*, Wiley, Indianapolis, 2003.

[GAR 00] GARLAN D., MONROE R., WILE D., "Acme: architectural description of component-based systems", in LEAVENS T., SITARAMAN M. (eds), *Foundations of Component-Based Systems*, Cambridge University Press, Camridge, 2008.

[GAR 02] GARLAN D., CHENG S., KOMPANEK J., "Reconciling the needs of architectural description with object-modeling notations", *Science of Computer Programming Journal*, Special UML, no. 44, pp. 23–49, 2002.

[GOU 03] GOULÃO M., ABREU F., Bridging the gap between Acme and UML 2.0 for CBD, Specification and Verification of Component-Based Systems Workshop, Lapland, Finland, 2003.

[IVE 04] IVERS J., CLEMENTS P., GARLAN D., *et al*., Documenting component and connector views with UML 2.0, Technical Report CMU/SEI-TR-008, School of Computer Science, Carnegie Mellon University, April 2004.

[JOU 06] JOUAULT F., BÉZIVIN J., "KM3: a DSL for metamodel specification", *8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 14–16 June 2006.

[MED 00] MEDVIDOVIC N., TAYLOR R., "A classification and comparison framework for software architecture description languages", *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pp. 70–93, 2000.

[MED 02] MEDVIDOVIC N., ROSENBLUM D.S., ROBBINS J.E., *et al.*, "Modeling software architecture in the unified modeling language", *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 1, pp. 2–57, 2002.

[MOF 02] META-OBJECT FACILITY – MOF, version 1.4., Object Management Group, Document Formal/2002-04-03, April 2002.

[MUL 05] MULLER P.A., FLEUREY F., JÉZÉQUEL J.M., "Weaving executability into object-oriented meta-languages", *Model Driven Engineering Languages and Systems*, pp. 264278, 2005.

[OMG 03] OBJECT MANAGEMENT GROUP, UML 2.0 Superstructure specification: final adopted specification, available at www.omg.org/docs/ptc/03-08-02.pdf, August 2003.

[OQU 06] OQUENDO F., "Formally modelling software architectures with the UML 2.0 profile for π-ADL", *ACM SIGSOFT Engineering Notes*, vol. 31, no. 1, January 2006.

[OUS 02] OUSSALAH M., "Component-oriented KBS", *14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, Ischia, Italy, 2002.

[SME 05a] SMEDA A., OUSSALAH M., KHAMMACI T., "MADL: Meta Architecture Description Language", *Third ACIS International Conference on Software Engineering, Research, Management and Applications (SERA '05)*, Pleasant, MI, 2005.

[SME 05b] SMEDA A., KHAMMACI T., OUSSALAH M., "Meta architecting: towards a new generation of architecture description languages", *Journal of Computer Science*, vol. 1, no. 4, pp. 454–460, 2005.

[SME 05c] SMEDA A., OUSSALAH M., KHAMMACI T., "Mapping ADLs into UML 2.0 using a meta ADL", *5th Working IEEE/IFIP Conference on Software Architecture (WICSA '05)*, Pittsburg, PA, 2005.

[TRA 05] TRAVERSON B., "Les Modèles de Composants Industriels", in OUSSALAH M. (ed.), *Ingénierie des Composants: Concepts, Techniques et Outils*, Vuibert Informatique, Paris, 2005.

[WAR 98] WARMER J., KLEPPE A., *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley Publishing, Reading, 1998.

[WIL 99] WILE D., "AML: an architecture meta language", *14th International Conference on Automated Software Engineering*, Cocoa Beach, FL, October 1999.