Towards an MDA-Oriented UML Profile for Distribution

Raul Silaghi, Frédéric Fondement, Alfred Strohmeier
Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland
{Raul.Silaghi, Frederic.Fondement, Alfred.Strohmeier}@epfl.ch

Abstract

The era of distributed systems is upon us. Middleware-specific concerns, and especially the distribution concern, which is the core of any middleware-mediated application, are addressed every day in all sorts of enterprise systems. However, object-oriented UML designs offer a very limited perspective on what exactly is distributed, how exactly the distribution is achieved, and where exactly distributed services are located. In order to answer these questions, the MDA-compliant Enterprise Fondue method proposes a hierarchy of UML profiles as a means for addressing the distribution concern at three different MDA-levels of abstraction. Model transformations are provided to incrementally refine existing design models according to the proposed profiles. For the last phase of the Enterprise Fondue process, code generation for specific middleware infrastructures is supported through the Parallax framework. The CORBA technology is used for illustrating the entire approach on a concrete example.

1. Introduction

With the rapid growth of the Internet and the associated web services revolution, distributed systems become more and more pervasive setting up new standards for modern industries. Current enterprise applications consist of heterogeneous components written in different programming languages and distributed in heterogeneous environments that comprise different hardware platforms, operating systems, data bases, and network protocols. The only way of masking these differences within an enterprise, or between enterprises, is by relying on middleware infrastructures, which can integrate diverse software components and allow them to interoperate effectively.

Model Driven Architecture (MDA) [1][2], an Object Management Group (OMG) [3] initiative, raises the integration and interoperability barrier at higher levels of abstraction, moving it from a mainly syntactic interface level (realized through agreed upon IDL [4] interfaces, for instance) to a more expressive behavioral level (realized through agreed upon *models*), promoting models to the status of first-class citizens. A key characteristic of MDA is the separation of concerns between two modeling dimensions: one focusing on the business functionality (resulting in *Platform Independent Models – PIMs*), and another one focusing on the implementation of that functionality on a specific middleware platform, such as COM/DCOM/COM+ [5], RMI [6], CCM/CORBA [7][4], Jini [8], EJB/J2EE [9][10], .NET [11], Web Services [12], or other message-oriented middleware platforms (resulting in *Platform Specific Models – PSMs*). While model transformations should be used to move between PIMs and PSMs, code generators are supposed to map PSMs to concrete middleware-based implementations. Since MDA is just a visionary approach to software development, without a concrete specification behind it, literature on the subject is not difficult to find [13][14][15], but many different questions still have to be answered [16].

Before going any further, referring to the "myth of absolute platform independence" and "platform relativism" [17], and in order not to leave any doubts or to risk any misinterpretations, we would like to make clear that, in the context of this paper, we consider the *middleware* to be our MDA platform, and not the operating system, or anything else. Moreover, even though MDA is completely independent of any modeling language, the Unified Modeling Language (UML) [18][19] established itself as the de-facto standard. As a consequence, we only focus on the UML support for MDA.

From a pragmatic point of view, in order to be able to realize the code generation step of the MDA vision in the context of distributed enterprise systems, MDA needs to provide support for understanding, describing, and implementing different middleware-specific concerns, such as distribution, interoperability, concurrency, transactions, security, and so on, also referred to as *pervasive services* in MDA's PIM terminology [2]. However, the current UML does not provide any specific or standard support for mod-

Proceedings of the 8th IEEE Intl Enterprise Distributed Object Computing Conf (EDOC 2004)
1541-7719/04 $20.00 © 2004 IEEE

eling pervasive services. What it does offer, is the possibility to "extend" the UML metamodel through, and only through, *profiling*, which specifies how specific UML model elements are customized and extended with new semantics as if they were instances of new "virtual" metamodel constructs. This unique position of UML profiles makes them play a key role in MDA, since developers must know about, or define, the metamodels of their input and output models before implementing any model transformation.

The Enterprise Fondue software development method [20] proposes a systematic approach to addressing pervasive services in an MDA-compliant manner, at different levels of abstraction, through incremental refinement steps along middleware-specific concern-dimensions. In order to prove the feasibility of the method, we consider in this paper a concrete middleware concern, namely *distribution*, and we use the CORBA technology to illustrate how the refinement process is applied to a concrete example. The main contribution of this paper is a hierarchy of UML profiles that address the distribution concern in an MDA-oriented fashion at different levels of abstraction, together with model transformations that incrementally refine existing design models (within the same or between different MDA-levels) along distribution-related concern-dimensions and in conformance to the proposed UML profiles. We also show how concrete implementation code can be generated for a specific middleware infrastructure.

The outline of the rest of this paper is as follows: Section 2 provides the motivation of this work by highlighting some insufficiencies in the current UML diagrams for representing distribution-related information; Section 3, based on the Enterprise Fondue software development method, shows how design models are incrementally refined along the distribution concern-dimension; Section 4 introduces the hierarchy of UML profiles; Section 5 presents the MTL model transformations that realize the refinement process; Section 6 describes the Parallax support for generating code targeted at specific middleware infrastructures, and Section 7 draws some conclusions and presents future work directions.

2. From Object-Oriented Designs to Distributed Systems

From a rather pragmatic point of view, we argue in this section that UML diagrams lack precision when it comes to providing specific distribution information that is typically required for generating distribution code out of UML models.

Let's consider the object-oriented design of a simple Bank system, like the one illustrated in Figure 1. We consider the example to be enough self-explaining for not entering into more details. It is important to mention nevertheless that not all designs follow the design by contract principles [21][22]. Typically, the interfaces in Figure 1 do not exist and clients act directly on the `Bank`, or on the `Account`, respectively. In that case, an intermediate "extract interface refactoring" [23] step is required in order to refactor the design and enforce good design principles.

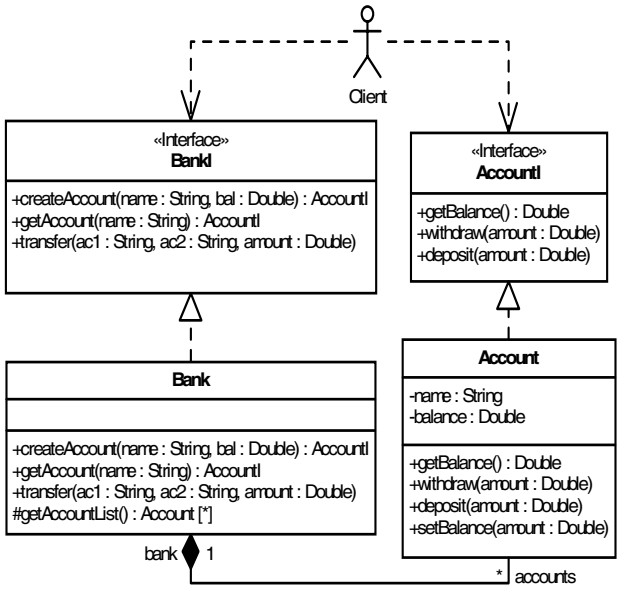

Figure 1. The Bank example

Once we have a "good" design, based on interfaces, we would like to automate the distribution of such object-oriented designs on different middleware infrastructures, and we would like to achieve this as transparently as possible for the developer.

One key aspect of distributed systems is their location transparency. Typically, *registries* are used to store the location of distributed objects. Clients `find` and use services (i.e., the interfaces of distributed objects that were already `bound` into registries) and do not care where they are located. Flexibility is very much increased, since distributed objects can be moved around and run on different machines, without any impact on the client side. It is only the information published in registries, and the registries themselves, that clients and distributed objects must agree upon.

Besides binding distributed objects into registries, there is also the problem of distributed interfaces. If we have a closer look at the `Bank::createAccount(...)` or `Bank::getAccount(...)` methods, we notice they both return `AccountI` interfaces to the client. If we consider, for example, the CORBA technology for implementing the distributed system, code generators must generate a CORBA IDL for the `AccountI` interface as well, otherwise the `AccountI` interface cannot be passed around in a CORBA distributed setting. As a consequence, this interface must

have been previously marked as distributed inside the design model, so that code generators know to deal with it properly.

As a conclusion, in order to be able to generate distribution code for a specific middleware infrastructure, design models must be refined and enhanced with distribution related information. But the question is how to model such distribution-related information in UML? How to specify that an interface should be distributed? How to specify that an object instance of class `Bank` should be the entry point of the distributed system? How to differentiate that object from other objects in the model in order to be able to bind it into a naming registry? How to infer that, because the `BankI` interface should be distributed, the `AccountI` interface should be distributed as well?

Of course, we are aware that component-oriented models address some of these issues up to a certain level through component and deployment diagrams. However, from the implementation point of view, when it comes to generating distribution code, we still have to rely on object-oriented programming constructs and specific middleware support. Therefore, the work presented in this paper focuses only on the distribution of object-oriented designs, presenting new UML modeling elements for addressing distribution, and stressing the amount of distribution code that can be automatically generated from the enhanced object-oriented models. Moreover, we believe that it is a very good preliminary step before analyzing the actual support for generating a similar amount of distribution code out of component-oriented designs based on information that can be retrieved or inferred from component and deployment diagrams.

3. Enterprise Fondue and the Distribution Concern

In this section, after a concise overview of Enterprise Fondue's terminology, we show how the questions raised in the previous section can be answered in Enterprise Fondue by incrementally refining existing models along distribution-related concern-dimensions.

The *Enterprise Fondue* software development method [20] brings together four important paradigms in software engineering, namely Component-Based Software Engineering (CBSE), Separation of Concerns (SoC), Model Driven Architecture (MDA), and Aspect-Oriented Programming (AOP), and shows how they can complement each other at different stages in the development life-cycle of enterprise, middleware-mediated applications. The method identifies five layers corresponding to different levels of abstraction, each layer addressing specific concerns that pertain to enterprise applications in general. Model transformations are used to refine design models inside the same layer, or between different layers, along specific concern-dimensions.

For consistency reasons, we tend to use the terms middleware-specific *concern-dimensions* in relation with the *refining* activity ("refining along a dimension"), and middleware-specific *concerns* in all other contexts. Nevertheless, both terminologies refer to the same concepts, i.e., distribution, concurrency, transactions, security, and so on.

Figure 2 presents how the refinement process in Enterprise Fondue evolves from one abstraction level to the next one by incremental refinements along different concern-dimensions. We use a mixed notation for representing the process flow, on one hand, and for representing UML 2.0 dependencies or relationships between modules, on the other hand.

`MTL1` refines along a middleware-specific *concern*-dimension (`Cx`) according to an associated UML profile for that concern. This transformation is performed inside the Concern-Driven Object-Oriented Models Layer (L_2) as defined in Enterprise Fondue. In the context of this paper, `MTL1` will refine along the distribution concern-dimension. However, several `MTL1s` can be applied at this layer, addressing different middleware-specific concerns.

The refinement along the *technology*-dimension (`Ty`) is performed by `MTL2`. As we tried to show in Figure 2, `MTL2` is a sequence of model transformations (not necessarily two). One such transformation (`MTL21`) will always refine the model along the technology-dimension according to the UML profile for that technology. All the other transformations in the sequence correspond to refinements along middleware-specific concern-dimensions according to UML profiles for those concerns on the specific technology (`Cx` on `Ty`). All these transformations are performed inside the Technology-Dependent Layer (L_1) as defined in Enterprise Fondue. For example, by refining along the CORBA technology-dimension, we will first apply the UML Profile for CORBA [24], and then we will apply the profile addressing the distribution concern on the CORBA technology (the `CORBADistributionRealizationProfile`, as it will be introduced in section 4.3).

The last two refinements, along the *platform*-dimension (`Pz`) and *language*-dimension (`Lw`), are performed in a single code generation step inside Parallax [25], which covers both the Platform-Dependent and Language-Dependent Layers of Enterprise Fondue (L_0). In order to achieve this step, as we will see in more details in section 6, Parallax must be enabled with a plug-in that knows how to generate code for the concern `Cx`, with the technology `Ty`, on the platform `Pz`, and using the programming language `Lw`.

As this paper only focuses on the distribution concern and less on the CORBA technology, which was merely considered for the sake of providing a concrete refinement, we

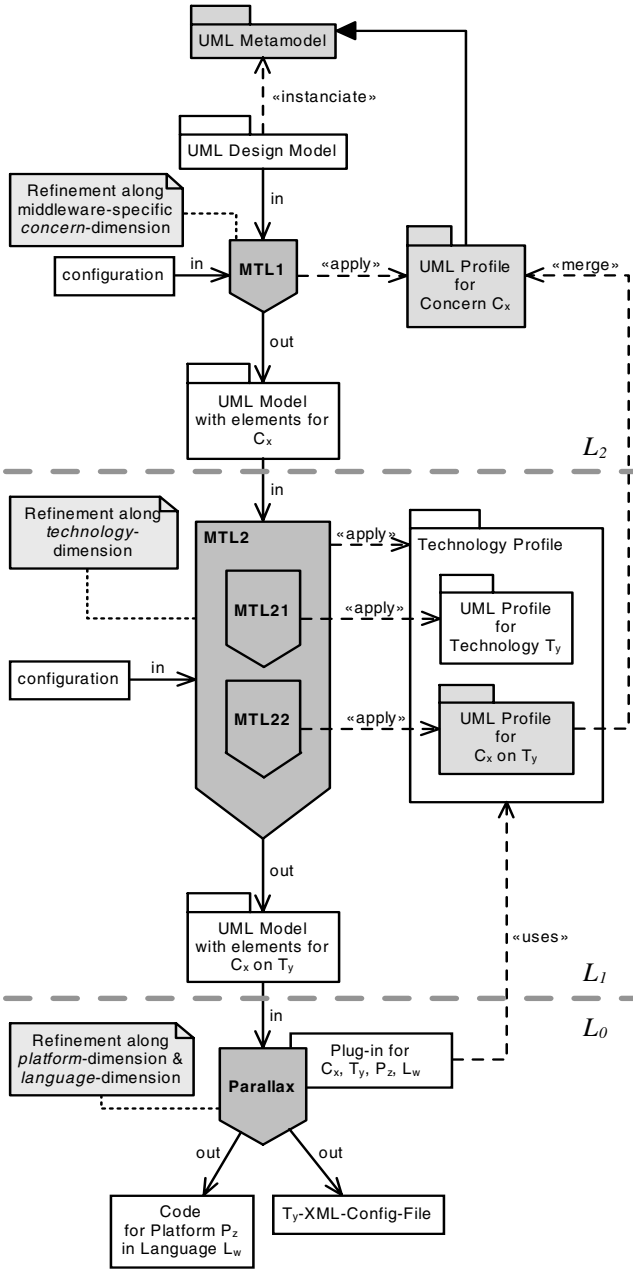

Figure 2. The refinement process in Enterprise Fondue

will only present profiles and transformations that involve the distribution concern. As a consequence, the `MTL21` transformation will be omitted, and only the `MTL22` will be presented, along with `MTL1` and the Parallax support.

In the following three sections we discuss in more details the key elements that support the Enterprise Fondue method when refining along the distribution concern-dimension, namely the UML Profiles for Distribution for describing distribution-specific concepts at different MDA-levels of abstraction, the MTL model transformations for actually applying these profiles to concrete designs, and the Parallax support for code generation.

4. UML Profiles to Address the Distribution Concern (UML-D Profiles)

The hierarchy of UML profiles presented in this section addresses the distribution concern in an MDA-oriented fashion at three different levels of abstraction: at a *platform-independent* level, at an *abstract realization* level, and at a *concrete realization* level for the CORBA technology (see Figure 3). We rely on specialization relationships between distribution profiles: each specialization introduces new modeling elements or simply refines the already existing ones.

UML provides a standard set of extension mechanisms, including stereotypes, tag definitions, tagged values, and constraints. These mechanisms are used to specify how UML model elements can be customized and extended with new semantics. A coherent set of such extensions, defined for a specific domain or purpose, constitutes a UML *profile*. The *stereotype* concept provides a way of branding (classifying) model elements. A stereotype creates a "virtual" UML metaclass by extending an existing UML metaclass with new metaattributes and additional semantics. Metaattributes are specified as *tag definitions*, which introduce new kinds of properties that may be attached to model elements. The actual properties of individual model elements are specified using *tagged values*. Simplifying, we could also say that a tag definition specifies the tagged values that can be attached to a kind of model element. The *constraint* concept allows new semantics to be specified linguistically for a model element. The language used can be a special constraint language (such as the Object Constraint Language, OCL [26]), a programming language, a mathematical notation, or even natural language.

A certain number of UML profiles have already been defined, either for generic purposes, such as the SPEM Profile [27], or to deal with specific middleware technologies, such as the CORBA profile [24] and the EJB Profile [28], or to address enterprise distributed systems, such as the EDOC set of profiles [29], and the list could very well continue.

When defining a UML profile, certain choices have to be made regarding the "dialect" it proposes. From this point of view, we are aware that, in some cases, better terminology might have been chosen to express the introduced concepts and their intended semantics/purpose. Opinions, whether `«Remote»`, or anything else, would have been a better choice than `«Distributed»` (to be presented in

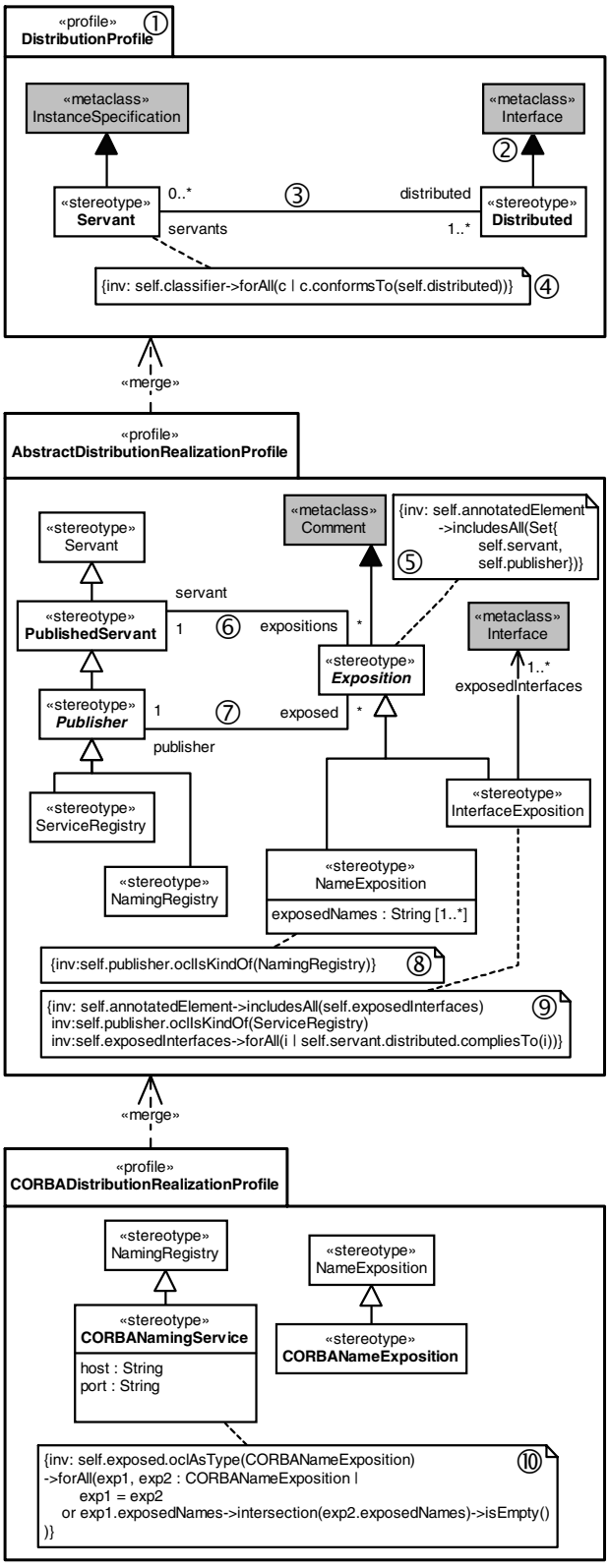

Figure 3. MDA-oriented hierarchy of UML-D Profiles

section 4.1), are divided and go beyond the scope of this paper.

Relying on Figure 3, the rest of this section describes in more details the UML-D Profiles, presenting the new modeling elements that we introduced at each level of abstraction in order to address distribution-related concerns. Well-formedness rules, expressing OCL constraints on model elements, are specified each time the validity of the models needs to be enforced. Natural language is used for describing the semantics of the UML extensions that we propose in our profiles. For the sake of clarity, we described our profiles using the UML 2.0 notation with some slight extensions that are explained in more details in section 4.4.

4.1. UML Distribution Profile

The purpose of distribution is to logically, or even physically separate communicating elements, typically a "core" system from its users. In UML, what these elements know about each other is described in terms of interfaces. UML interfaces are defined as sets of coherent publicly available features and obligations, fulfilled at runtime by instances of classes realizing them. The distribution process should lead to independent subsystems communicating through known interfaces. We qualify these interfaces as distributed. This additional information can be added to the model of the system by applying the *DistributionProfile* presented in Figure 3 ①.

To make a difference between *what* interfaces are used for communication within a system and between systems, the profile defines the *«Distributed»* stereotype as an extension of the `Interface` metaclass as shown in Figure 3 ②. All features defined within a `«Distributed»` interface are remotely available. In the case a `«Distributed»` interface extends other interfaces, available remote features are only those defined within `«Distributed»` interfaces. In this way, we clearly separate between distributed and non-distributed interfaces even within the same hierarchy of interfaces. For instance, the following OCL query may be defined within the scope of the `«Distributed»` stereotype to find all its remotely available operations:

```
context Distributed
def: allRemoteOperations : Set(Operation) =
  self.ownedOperation->union(
    self.allParents()
     ->select(oclIsKindOf(Distributed))
     .ownedOperation
  )
```

We have assumed here that the predefined OCL `oclIsKindOf` operation holds true if a model element has the stereotype passed as parameter. A similar query may also be defined to know about all remotely available attributes.

Moreover, it is also necessary to indicate key instances that the environment needs to know in order to start an interaction with the system as an entity. They are identified by applying the *«Servant»* stereotype. As these instances receive invocations from the environment, their class must realize a "well-known" «Distributed» interface, that is why «Servant» and «Distributed» stereotypes are associated as shown in Figure 3 ③. This allows one, once these stereotypes are applied to the corresponding instances and interfaces, to indicate at the model level what are the «Servant» instances of a «Distributed» interface, and what are the «Distributed» interfaces of a «Servant» instance. As indicated by the multiplicities of the association ③, a «Servant» instance must realize at least one «Distributed» interface, otherwise a client application does not know how to communicate with it. On the other side, a «Distributed» interface may be connected to any number of «Servant» objects. If none, it means the given interface does not participate in any interaction set-up. If a «Distributed» interface defines many «Servant» objects, this means the system has many entry points with this interface.

For the profiled model to be consistent, it is important to add a constraint stating that one of the classes of each «Servant» object must realize the «Distributed» interface it is related to through the distributed association end ③. This constraint is given in Figure 3 ④.

Note that, even though UML 2.0 does not allow the use of associations between stereotypes, we used this kind of association here only for the sake of legibility and we explain its intended semantics in section 4.4.

4.2. UML Abstract Distribution Realization Profile

The purpose of the *AbstractDistributionRealizationProfile* is to provide a framework to describe *which* «Servant» instances are made available to the outside world, *how*, and *where*. We enter here the technology-dependent layer (L_1) described in Figure 2. As this profile specializes entities described in the DistributionProfile, it merges this latter profile and as a consequence integrates into it all its stereotypes and tag definitions.

The *«PublishedServant»* stereotype is a specialization of «Servant», in order to show that «Servant» instances may be exposed to the outside world. Such «Servant» instances will be further referred as *«PublishedServant» instances* (PSI). Due to the specialization, the «PublishedServant» stereotype inherits the InstanceSpecification base class, the distributed tag definition, and the constraint ④, which must now hold for all elements conforming to this stereotype as well.

A *«Publisher»* is a «PublishedServant» instance specialized in making available a given «PublishedServant» to the outside world. This means that in order to interact with a PSI, an actor from the environment should first locate it by sending a request to a «Publisher» instance. If the Bank example in Figure 1, for instance, is made distributed, a Client should first retrieve the right «Servant» Bank instance by sending a request to a «Publisher» instance. How an external actor localizes a «Publisher» is voluntarily left unresolved and should be defined by specializing the «Publisher» stereotype according to the concrete technology to be used. This is the reason why the «Publisher» stereotype is abstract. Moreover, since the «Publisher» stereotype inherits the «PublishedServant» stereotype, an instance stereotyped «Publisher» is also a PSI, and may be published by another «Publisher» instance.

The relationship between a «PublishedServant» and a «Publisher» is expressed by the *«Exposition»* stereotype. Unfortunately, there is not really an ideal relationship between instances in the UML metamodel that this stereotype could extend. Therefore, we decided to make the Comment metaclass the base class of this stereotype. Information of *what* is published and *who* is the publisher is gathered through the associations ⑥ and ⑦ respectively. The constraint ⑤ requires all the «Exposition» comments to be attached to both the «Servant» and the «Publisher» instances. A given PSI may be published several times within several «Publisher» instances, and conversely, a «Publisher» may expose several PSIs, all these relationships being modeled through «Exposition» comments.

«Publisher» and «Exposition» are abstract stereotypes. Therefore, they cannot be applied to a model element as such, because some *registration information* needs to be provided. We include therefore in the profile a kind of "reference implementation", although it would be possible to define an additional independent profile that extends the AbstractDistributionRealizationProfile and describes additional information required by another implementation mechanism.

As a first reference implementation mechanism, we propose to register a PSI by *names*, which are character strings, within a «Publisher» instance. We therefore define *«NamingRegistry»* as an extension of the «Publisher» stereotype, together with the *«NameExposition»* as an extension of the «Exposition» stereotype. The registration names are stored in the exposedNames tag definition, that is defined in the «NameExposition» stereotype. This tag definition has a 1..* multiplicity meaning that the PSI can be exposed by at least one name. A «NamingRegistry» may only publish PSIs through a «NameExposition», and a «NameExposition» may only refer to a «NamingRegis-

try». This is enforced by the constraint ⑧. This kind of exposition mechanism looks like the one of the phone directory of residents ("white pages").

As a second mechanism, we propose to register a PSI with the *services* it can offer. For instance, one can ask the environment for a printer, provided that there is a printer that knows how to *print* PostScript documents. This mechanism is described by the couple of stereotypes *«ServiceRegistry»* for the publishing part and *«InterfaceExposition»* for the exposition part. This time the registration is performed by means of interfaces, referenced by the tag definition exposedInterfaces within the «InterfaceExposition» stereotype. Once again, a constraint (⑨) enforces that these two stereotypes are used together and only together. This kind of exposition mechanism looks like the one of the business phone directory ("yellow pages").

4.3. UML CORBA Distribution Realization Profile

The *CORBADistributionRealizationProfile* addresses the realization of the distribution concern when the implementation is supposed to use the CORBA technology. It takes advantage of the AbstractDistributionRealizationProfile by adapting its abstract concepts to the CORBA technology, so that a code generator has enough information to generate the necessary distribution code.

The *«CORBANameExposition»* stereotype represents a «NameExposition» using the CORBA technology. It extends the «NameExposition» but does not add particular information to it. The idea is to state clearly that a PSI is registered by name using the CORBA technology. The same holds for the *«CORBANamingService»* stereotype that extends the «NamingRegistry». In order for an actor of the environment to find the «CORBANamingService», we add the host and port tag definitions. As previously, an OCL constraint (⑩) enforces that these two stereotypes work together and only together. This constraint also enforces that exposition names are all different within the context of a «Publisher».

We chose to describe here only the CORBA technology, but the same principle may be applied to any kind of middleware technology, possibly with additional intermediate profile specialization steps.

4.4. Notation, Semantics, and UML Profile Compatibility

We chose to present the profiles in Figure 3 using the UML 2.0 notation because of its readability. However, for the sake of clarity, we have used associations between stereotypes despite the fact that the UML 2.0 specification forbids it, except for subsetting a metaassociation belonging to

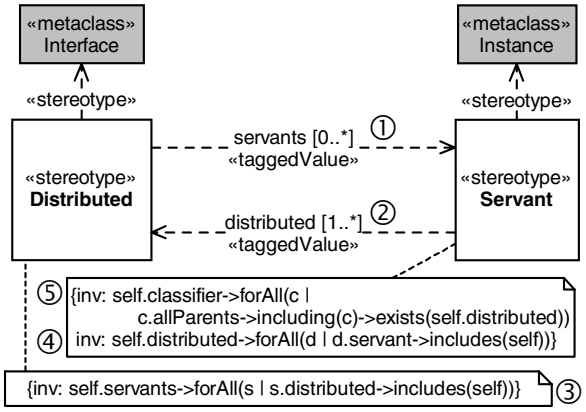

Figure 4. The DistributionProfile in UML 1.5

the metamodel. E.g., for the associations ⑥ and ⑦ in Figure 3 a metaassociation exists between the Comment and Element metaclasses with a many-to-many multiplicity. In fact, this metaassociation is navigated by the constraint ⑤. But the rule fails for the association ③ in Figure 3. To give a precise semantics to the stereotype association construct, we rely on UML 1.5: an association between two stereotypes acts as two crossed tag definitions together with constraints enforcing that they are indeed bidirectional, like classical association end values. As an example, Figure 4 shows the DistributionProfile as a standard UML 1.5 profile. The association ③ in Figure 3 is encoded in Figure 4 in the tag definitions ① and ② together with the constraints ③ and ④ enforcing that they are indeed crossed. Because of differences between the UML 1.5 and 2.0 metamodels, the UML 2.0 InstanceSpecification metaclass becomes the UML 1.5 Instance metaclass and the constraint ④ in Figure 3 is encoded by the constraint ⑤ in Figure 4. Table 1 provides some guidelines for mapping UML 2.0 to UML 1.5 profiles.

Table 1. UML backward compatibility

UML 2.0	UML 1.5
«merge» dependency	package generalization
InstanceSpecification metaclass	*Instance* metaclass
base class arrow	«stereotype» dependency arrow
stereotype attribute	stereotype tag definition
association between stereotypes (within the scope of this work)	crossed tag definitions

5. MTL Model Transformations for Applying the UML-D Profiles

After providing some background on existing model transformation languages, and motivating some advantages of the Model Transformation Language (MTL), we present in this section the model transformations that incrementally refine existing design models according to the UML-D Profiles introduced in the previous section. The Bank example (Figure 1) is used for illustrating the refinement process.

Model transformations and language support for implementing them are the much needed keystones for OMG's MDA vision [30]. To fill this gap, OMG has posted a Request for Proposal called MOF 2.0 Query/Views/Transformations RFP [31], which has been answered by eight different initial submissions, only five revised submissions, and only two second revised submissions (up to this date). Most of the last ones already propose tool support for their submissions, like DSTC/IBM/CBOP [32], OpenQVT [33], or QVT-Partners [34].

However, outside the QVT submission process, other solutions to model transformations exist. Some are specific to a CASE tool, such as Objecteering J [35], others are specific to a repository, like ModFact [36], while still others are providing an API for specific programming languages, such as JMI [37], and the list could continue. Each solution has its own advantages and disadvantages, making the selection process rather difficult. We have chosen the INRIA Model Transformation Language (MTL) [38] according to the following requirements:

- it can transform XMI-serialized UML models,
- it provides support for the UML profiling mechanism,
- the compiler is maintained and easily available,
- it is independent of CASE tools and model repositories,
- it is an imperative language, and thus more readable,
- it is supported by an active community with strong intentions to evolve towards the upcoming OMG MOF QVT standard.

5.1. Refining Along the Distribution Concern-Dimension (`MTL1-D`)

The distribution transformation (`MTL1-D`) is refining centralized design models along the distribution concern-dimension according to the `DistributionProfile` presented in section 4.1. Its name indicates, on one hand, that it belongs to the `MTL1` family of transformations (as shown in Figure 2), and on the other hand, that it is related to the distribution concern.

In order to achieve the distribution, the transformation requires the developer to provide information about the interfaces that the «Servant» object should realize. These special interfaces will be further referred as *«Servant» interfaces* (`SIs`). If several «Servant» objects are needed, then the transformation may be called several times. As already mentioned in section 2, we rely on the premise that all interactions with the environment occur through well-defined interfaces.

In the first step, the `MTL1-D` transformation "imports" the `DistributionProfile` into the model, making available the UML extensions it defines. The second step is to find the right classifier for the «Servant» object. Note that if more than one classifier realizes all the interfaces the developer has specified, then the `MTL1-D` transformation may choose an arbitrary one, or ask the developer to choose among the possible realizations; if no class is found, then the transformation ends in error, without modifying the model. Once the right classifier is found, the corresponding `SIs` are marked with the «Distributed» stereotype and an object instance of the found classifier is created and marked with the «Servant» stereotype. As these stereotypes are associated, it is still necessary to provide crossed tagged values as specified in section 4.4. This means that the «Servant» object references its `SIs` by means of the `distributed` tagged value, and each `SI` references its «Servant» objects by means of the `servants` tagged value, according to the `DistributionProfile`.

```
//Within the MTL class Distributor
//- toDistribute are the interfaces to be distributed
treatOperationDependencies() {

  foreach (op : m::Core::Feature)
       in (toDistribute.feature)
    where (op.oclIsKindOf(!m::Core::Operation!)) {

    foreach (pa : m::Core::Parameter)
         in (op.parameter) {

       new Distributor().init(self,pa.type).run();
    }
  }
}
```

Figure 5. The `MTL1-D` transformation: exploring operations part

The last step of the `MTL1-D` transformation is to infer all interfaces that participate in interactions with the environment and to stereotype them «Distributed» as well. To this end, the transformation explores, starting from the provided `SIs`, the types of parameters of all operations, and the types of all attributes of each interface. During exploration, all encountered interface types are stereotyped «Distributed» (if not yet the case), and recursive explorations are started for each such interface. The MTL code for this exploration is shown in Figure 5.

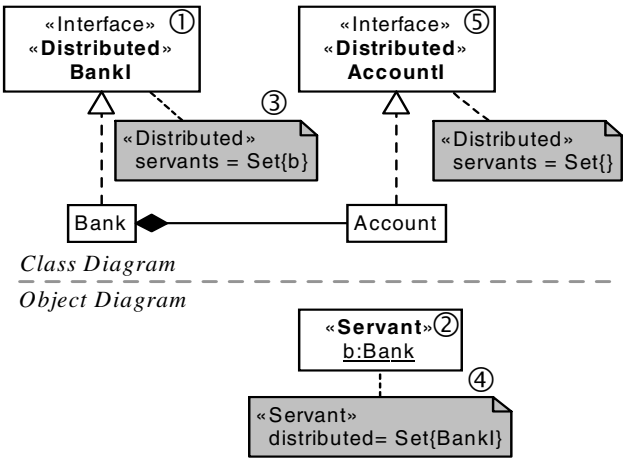

Figure 6. The `MTL1-D` outcome for the Bank example

As an example, we show in Figure 6 the outcome of the `MTL1-D` transformation on the Bank example when `BankI` is the only `SI` requested by the developer. As one can see, `BankI` gets stereotyped as `«Distributed»` (①). An object of class `Bank`, the only classifier realizing the `BankI` interface, is created and stereotyped `«Servant»` (②). The tagged values are shown in grey colored notes. Due to the association between `«Distributed»` and `«Servant»` stereotypes (as defined in the profile), the `«Distributed» BankI` interface references its `SIs` by means of the `servants` tagged value (③), and the `«Servant» b` references the `«Distributed» BankI` interface by means of the `distributed` tagged value (④). As the `«Distributed» BankI` interface contains operations, such as `createAccount` and `getAccount`, involving the `AccountI` interface, this last interface is stereotyped `«Distributed»` as well, but with an empty `servants` tagged value (⑤).

5.2. Refining Distribution Along the CORBA Technology-Dimension (`MTL22-D`)

The CORBA distribution realization transformation (`MTL22-D`) is refining distributed models along the CORBA technology-dimension. More precisely, it refines models, to which the `DistributionProfile` was already applied, to ones that are more specific about how the distribution concern is actually implemented on the CORBA technology. As previously, the name indicates that it belongs to the `MTL22` family of transformations (as shown in Figure 2), on one hand, and that it is related to the distribution concern, on the other hand.

In order to be able to apply the `CORBADistributionRealizationProfile`, the `MTL22-D` transformation requires the developer to provide all the new information that this profile adds with respect to the `DistributionProfile`. Indeed, `MTL22-D` needs to be able to provide each CORBA publisher (i.e., CORBA Naming Service [39], `«CORBANamingService»`) with its `name`, `host` and `port`, and each `PSI` with its expositions (`«CORBANameExposition»`) containing the names to be exposed (`exposedNames`) and the publisher to be used (`publisher`). The transformation does not integrate any `«InterfaceExposition»`, which is more related to the Trading Service [40] of CORBA that is not discussed in this paper.

Like for `MTL1-D`, the first step of the `MTL22-D` transformation is to "import" the `CORBADistributionRealizationProfile` into the model. The `MTL22-D` also "imports" standard CORBA libraries, like the interface of the CORBA publisher, which is `org::omg::CosNaming::NamingServiceExt`. Then, the transformation creates `«CORBANamingService»` publisher objects and sets their `host` and `port` tagged values according to the provided parameters. Since `«CORBANamingService»` inherits from the `«Servant»` stereotype the `distributed` tag definition, and because of its multiplicity `1..*`, it is necessary to provide it with a value. The other inherited tagged values, namely `exposed` and `expositions`, are not mandatory as their lower bound multiplicity is zero and because they would have no meaning for the `«CORBANamingService»`

```
//The exposition
//- publisher is the publisher object
//- servant is the published servant object
//- profile is an MTL proxy for the
//          CORBADistributionRealizationProfile
//- expositionName is the provided name
//                 of the exposition
//- publishedNames are the names the servant
//                 is registered with
ex := new m::Core::Comment();
ex.name := expositionName;
associate (
   comment := ex : m::Core::Comment,
   annotatedElement := servant
                          : m::Core::ModelElement);
associate (
   comment := ex : m::Core::Comment,
   annotatedElement := publisher
                          : m::Core::ModelElement);

profile.applyStereotype(ex,
   profile.cORBANameExposition);

profile.setTaggedValueData(servant,
   profile.publishedServantExpositionsTag,ex);

profile.setTaggedValueData(publisher,
   profile.publisherExposedTag, ex);

profile.setTaggedValueData(ex,
   profile.expositionServantTag, servant);

profile.setTaggedValueData(ex,
   profile.expositionPublisherTag, publisher);

profile.setTaggedValueData(ex,
   profile.nameExpositionExposedNamesTag,
   publishedNames);
```

Figure 7. The `MTL22-D` transformation: creating exposition part

publisher. Moreover, as the «CORBANamingService» publisher instance provides behaviors described in the NamingServiceExt interface, this interface must be stereotyped as «Distributed». The distributed and servants tagged values are used to relate the instance and the interface together.

At the end, the MTL22-D transformation creates the expositions as specified by the developer. As defined in the profile, each «CORBANameExposition» is a Comment on the PSI and the «Publisher» instance, which are referenced through the servant and publisher tagged values respectively. Both the PSI and the «Publisher» («CORBANamingService» in our case) know about the «Exposition» comment by means of the expositions and exposed tagged values respectively. The exposed names of a «CORBANameExposition» are stored in the exposedNames tagged value. Figure 7 shows the MTL code for this last step.

The outcome of the MTL22-D transformation for the Bank example is shown in Figure 8. The source model for the transformation is the one shown in Figure 6. The MTL22-D provides one «CORBANamingService» publisher object cns (①) and sets its distributed tagged value to reference the NamingServiceExt interface. The «Distributed» stereotype was applied to this last interface and its servants tagged value was set to reference the cns, but this is not depicted here. The diagram also shows the b object, to which the «PublishedServant» stereotype (②) was applied. A new comment, named BankExposition, with stereotype «CORBANameExposition», represents the expositions (③), and it is referenced by the expositions and exposed tagged values in the b and cns instances respectively. This comment annotates these instances and references them with the servant and publisher tagged values respectively. For the sake of clarity, all tagged values have been provided, even if empty, and classified according to the stereotype in which they were declared.

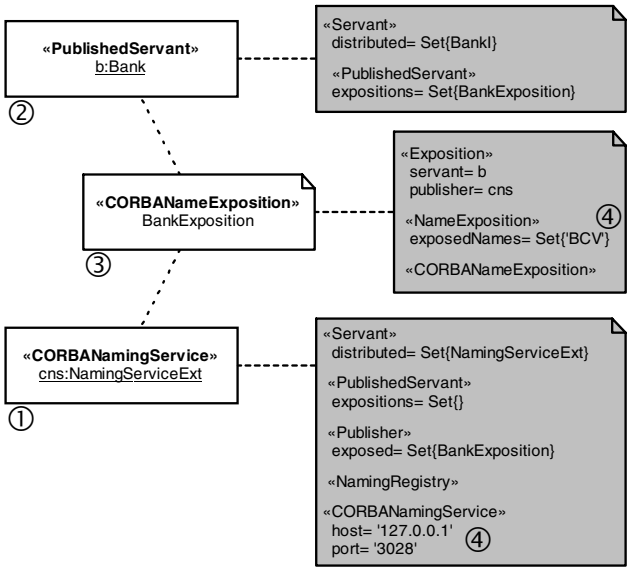

Figure 8. The MTL22-D outcome for the Bank example

6. The Parallax Support

After describing the UML profiles for distribution and presenting how model transformations refine existing design models according to these profiles, it is now time to move to the last phase of the Enterprise Fondue process and address how code generation for a specific CORBA platform, namely OpenORB [41], is supported through the Parallax framework.

The main purpose of the XML Metadata Interchange (XMI) [42][43] is to enable easy interchange of metadata between modeling tools (based on UML) and between tools and metadata repositories (based on MOF [44]) in distributed heterogeneous environments. We rely on XMI when moving from different CASE tools to model transformation languages, and then to Parallax, in the MDA tool-chain of the Enterprise Fondue process.

Parallax [25] is a framework for defining different architectural *views* for component-based middleware-mediated applications. Implemented as an Eclipse plug-in [45][46], Parallax accepts as input application designs exported as XMI files from different UML modeling tools, such as Poseidon, IBM Rational Rose, Borland Together, etc., and allows developers to look at these designs from different perspectives by providing an extensible system of views.

Middleware-specific views are dedicated to middleware-specific concerns, such as distribution, transactions, security, etc., provided that the imported XMI file already contains elements describing the middleware-specific concerns as defined in their corresponding UML profiles. Parallax, based on *aspect-oriented* support and through a well-defined system of *plug-ins*, enables developers to view their designs through a prism of middleware platforms and to see how middleware-specific concerns are actually implemented at code level. Each such view is supported via a separate middleware-specific plug-in that can be loaded/installed into Parallax. Each middleware-specific plug-in has four dependency dimensions (*4-DD*); it is at the same time *middleware-concern* dependent, *technology* dependent, *platform* dependent, and *language* dependent, e.g., a plug-in for distribution, with EJB, on BEA WebLogic, using Java.

For more information about Parallax and about other design-specific views that it can provide, please refer to our web site at [25].

Since we are completely relying on UML's extension mechanisms, and since the structure of XMI is based on the UML metamodel, which incorporates extension mechanisms as well, there is no problem in exporting design models that were refined according to the profiles introduced in section 4, i.e., models that contain new stereotypes, tag definitions, or tagged values. Figure 9 shows an XMI1.2[UML1.5] snippet corresponding to the «PublishedServant» stereotype exactly as it was defined in the AbstractDistributionRealizationProfile in Figure 3. The idref a107 (Figure 9) corresponds to the generalization between «PublishedServant» and «Servant». The stereotype's base class and the tag definitions it participates in, can also be seen in the exported XMI.

```
<UML:Stereotype xmi.id = 'a106'
  name = 'PublishedServant'
  isSpecification = 'false' isRoot = 'false'
  isLeaf = 'false' isAbstract = 'false'>
  <UML:Stereotype.baseClass>Instance
  </UML:Stereotype.baseClass>
  <UML:GeneralizableElement.generalization>
    <UML:Generalization xmi.idref = 'a107'/>
  </UML:GeneralizableElement.generalization>
  <UML:Stereotype.definedTag>
    <UML:TagDefinition xmi.id = 'a108'
    name = 'expositions' isSpecification = 'false'
    tagType = 'Exposition'>
      <UML:TagDefinition.multiplicity>
        <UML:Multiplicity xmi.id = 'a109'>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id = 'a110'
              lower = '0' upper = '-1'/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:TagDefinition.multiplicity>
    </UML:TagDefinition>
  </UML:Stereotype.definedTag>
</UML:Stereotype>
```

Figure 9. PublishedServant stereotype

Importing into Parallax such an enhanced XMI design model, like the one presented in Figure 8, and installing the distribution plug-in for CORBA, on OpenORB, using Java (see layer L_0 in Figure 2), we are able to generate IDLs for the «Distributed» interfaces along with special code snippets for initializing and starting the ORB, and for binding the «Servant» object according to the deployment configuration stored in a separate CORBA-XML-Config-File, which is described later in this section. Figure 10 illustrates generated code snippets on the «Servant» class side. Please notice that for the sake of space and legibility, we did not show how certain configuration information, like the name "BCV" given to our «Servant» object, is in fact read from the CORBA-XML-Config-File.

The CORBA-XML-Config-File contains what we call the *deployment configuration information* (dci), as shown in Figure 8 ④. Parallax exports this information into a specialized file in order to allow for easy deployment customization. It is this CORBA-XML-Config-File that both the client and the server applications read at startup time in order to know how to find each other.

```
org.omg.CORBA.ORB orb = null;
org.omg.PortableServer.POA poa = null;
org.omg.CosNaming.NamingContextExt nc = null;
org.omg.CORBA.Object so = null;

java.util.Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass",
          "org.openorb.CORBA.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
          "org.openorb.CORBA.ORBSingleton");

orb = org.omg.CORBA.ORB.init(
  "-ORBProfile=default", props);
poa = org.omg.PortableServer.POAHelper.narrow(
  orb.resolve_initial_references("RootPOA"));
poa.the_POAManager().activate();
Bank b = new Bank();
so = poa.servant_to_reference(b);

nc = org.omg.CosNaming.NamingContextExtHelper.narrow(
  orb. resolve_initial_references("NameService"));
nc.rebind(nc.to_name("BCV"), so);

orb.run();
```

Figure 10. Generated code in the «Servant» class

One should notice that the dci is specific to the *technology* and not to the platform. For instance, all distribution plug-ins for CORBA, even though supporting different implementation platforms, such as OpenORB, VisiBroker, ORBacus, etc., should use the same unique CORBA deployment configuration information stored in the CORBA-XML-Config-File. Nevertheless, when changing the technology, the AbstractDistributionRealizationProfile might not be extended in the same way, and therefore the information that will have to be stored in the corresponding XML-Config-File might not be the same, which shows once again why such XML-Config-Files are technology dependent.

In order to illustrate the benefit of having a stand alone CORBA-XML-Config-File, let's consider a concrete example. Suppose for a second that the same Bank system that we have just developed fits the entire organizational structure and fulfills the entire business needs of two different banks, and that both of them would like to buy it. We already have the code generated from Parallax that, besides implementing the requested functionality, also reads the dci from the CORBA-XML-Config-File. In this case, it is very convenient to just change the dci, e.g., the name of the bank, the location and the port of the naming service, and immediately be able to start the second Bank system. A similar problem arises if a bank needs to migrate a specific service, such as the naming service, from one machine to another.

In future work on the Parallax framework, we intend to follow the Eclipse contribution circle and define extension

points for Parallax. Once we have published these extension points, we enable other developers and middleware vendors to contribute and enrich Parallax by implementing and providing the community with new Parallax plug-ins for their favorite middleware infrastructures.

Moreover, as soon as UML modeling tools will implement the UML 2.0 specification [19] and will provide XMI export facilities for the enhanced interactions that can appear in sequence diagrams (`Fragments::InteractionOperators` such as `seq`, `alt`, `opt`, `break`, `loop`, etc.), we will enhance the code generation support inside Parallax accordingly. Currently, the generated code corresponds to the static structure (from class diagrams) and to the behavior –to some extent– (from interactions in collaboration/sequence diagrams).

7. Conclusions and Future Work

For the MDA approach to software development to become a reality for distributed enterprise systems, UML (as a de-facto standard) must provide support for modeling different middleware-specific concerns, such as distribution, concurrency, transactions, security, also referred to as pervasive services in MDA's PIM terminology, at both platform-independent and platform-specific levels. Only once these concerns are an integral part of PSMs, code generators will be able to generate appropriate code for specific middleware platforms.

The Enterprise Fondue method proposes a systematic approach to addressing pervasive services in an MDA-compliant manner, at different levels of abstraction, through incremental refinement steps along middleware-specific concern-dimensions. In this paper, we introduced the key elements that support the Enterprise Fondue method when refining along the *distribution* concern-dimension, namely: (1) the UML Profiles for Distribution (*UML-D Profiles*) that address the distribution concern in an MDA-oriented fashion at three different levels of abstraction (*platform-independent* level, *abstract realization* level, and *concrete realization* level), (2) the model transformations that incrementally refine existing design models (within the same or between different MDA-levels) along distribution-related concern-dimensions and in conformance to the proposed UML profiles, and (3) the Parallax support for generating code targeted at specific middleware infrastructures. The CORBA technology was used to illustrate how the refinement process is applied to a concrete example.

Please notice, however, that the Enterprise Fondue method and the Parallax tool support are relatively young. Both are undergoing refinement and improvement, but they are already being applied. Besides the CORBA example that was presented in this paper, refinement along the RMI [6] technology-dimension and down to RMI code generation using Parallax did not raise any problems either.

Even though the UML-D Profiles are just a preliminary step towards a final MDA-oriented UML profile for distribution, they still have the merit of addressing distribution-related concerns at different MDA-levels of abstraction. By introducing the *abstract distribution realization profile*, and by showing a concrete realization for the CORBA technology, we set up the basis for future extensions and refinements for different middleware technologies, such as Jini, EJB/J2EE, .NET, or Web Services. It is only in this way that limitations will appear leading to improvement.

Further investigations will be carried out to check whether other middleware-specific concerns lend themselves to such an MDA-oriented profiling approach. Addressing concurrency (*UML-C Profiles*), transactions (*UML-T Profiles*), security (*UML-S Profiles*), global time (*UML-GT Profiles*), etc., will be intermediate steps towards an MDA-Oriented UML Profile for Middleware Services, or more precisely Middleware-Specific Concerns (*UML-MS Profiles*).

References

[1] Object Management Group, Inc.: *Model Driven Architecture*. `http://www.omg.org/mda/`, June 2004.

[2] Miller, J.; Mukerji, J.: *Model Driven Architecture (MDA)*. Object Management Group, Document ormsc/2001-07-01, July 2001.

[3] Object Management Group, Inc., `http://www.omg.org/`, June 2004.

[4] Object Management Group, Inc.: *Common Object Request Broker Architecture: Core Specification*, v3.0.3, March 2004.

[5] Microsoft, Inc.: *COM (Component Object Model)*, *DCOM (Distributed COM)*, *COM+*. `http://www.microsoft.com/com/`, June 2004.

[6] Sun Microsystems, Inc.: *Java Remote Method Invocation Specification*. Revision 1.7, Java 2 SDK, Standard Edition, v1.3.0, December 1999. `http://java.sun.com/j2se/1.3/docs/guide/rmi/`, June 2004.

[7] Object Management Group, Inc.: *CORBA Components Specification*, v3.0, June 2002.

[8] Sun Microsystems, Inc.: *Jini Network Technology*. `http://www.sun.com/jini/`, June 2004.

[9] Sun Microsystems, Inc.: *Enterprise JavaBeans Specification*, v2.1, November 2003.

[10] Sun Microsystems, Inc.: *Java 2 Platform, Enterprise Edition Specification*, v1.4, November 2003.

[11] Microsoft, Inc.: *.NET*. `http://www.microsoft.com/net/`, June 2004.

[12] World Wide Web Consortium: *Web Services*. `http://www.w3.org/2002/ws/`, June 2004.

[13] Kleppe, A.; Warmer, J.; Bast, W.: *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[14] Frankel, D. S.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[15] Mellor, S. J.; Balcer, M. J.: *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[16] Bézivin, J.; Gérard, S.; Muller, P.-A.; Rioux, L.: *MDA Components: Challenges and Opportunities*. International Workshop on Metamodelling for MDA, Kings Manor, York, England, November 24-25, 2003. http://www.cs.york.ac.uk/metamodel4mda/index.html.

[17] Frankel, D. S.: *The MDA Marketing Message and the MDA Reality*. MDA Journal, a Business Process Trends Column, March 2004. http://www.bptrends.com/.

[18] Object Management Group, Inc.: *Unified Modeling Language Specification*, v1.5, March 2003.

[19] Object Management Group, Inc.: *Unified Modeling Language Superstructure Specification*, v2.0, August 2003.

[20] Silaghi, R.; Strohmeier, A.: *Integrating CBSE, SoC, MDA, and AOP in a Software Development Method*. Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference, EDOC, Brisbane, Queensland, Australia, September 16-19, 2003. IEEE Computer Society, 2003, pp. 136 – 146. Also available as Technical Report, N° IC/2003/57, Swiss Federal Institute of Technology in Lausanne, Switzerland, September 2003.

[21] Meyer, B.: *Applying Design by Contract*. IEEE Computer, 1992, pp. 40 – 51.

[22] Meyer, B.: *Design by Contract*. Prentice Hall, 2002.

[23] Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[24] Object Management Group, Inc.: *UML Profile for CORBA Specification*, v1.0, April 2002.

[25] Software Engineering Laboratory at the Swiss Federal Institute of Technology in Lausanne: *The Parallax Project*. http://parallax-lgl.epfl.ch/, June 2004.

[26] Warmer, J.; Kleppe, A.: *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[27] Object Management Group, Inc.: *Software Process Engineering Metamodel Specification (SPEM)*, v1.0, November 2002.

[28] Rational Software Corporation: *UML Profile for EJB, JSR-000026*, Public Draft, May 2001.

[29] Object Management Group, Inc.: *UML Profile for Enterprise Distributed Object Computing Specification*, v1.0, February 2002.

[30] Sendall, S.; Kozaczynski, W.: *Model Transformation – the Heart and Soul of Model-Driven Software Development*. IEEE Software, **20**(5), Special Issue on Model-Driven Development, 2003, pp. 42 – 45. An extended version is available as Technical Report, EPFL-IC-LGL N° IC/2003/052, July 2003.

[31] Object Management Group, Inc.: *MOF 2.0 Query/Views/Transformations RFP*. http://www.omg.org/cgi-bin/doc?ad/02-04-10, 2002.

[32] DSTC/IBM/CBOP: *MOF 2.0 Query/Views/Transformations*, Second Revised Submission, January 2004. http://www.omg.org/cgi-bin/doc?ad/2004-01-06.

[33] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Technologies Corp: *OpenQVT: MOF 2.0 Query/Views/Transformations*, First Revised Submission, August 2003. http://www.omg.org/cgi-bin/doc?ad/2003-08-05.

[34] Tata Consultancy Services: *QVT-Partners: MOF 2.0 Query/Views/Transformations*, First Revised Submission, August 2003. http://www.omg.org/cgi-bin/doc?ad/2003-08-08. http://qvtp.org/.

[35] Softeam, Inc.: *UML Profile and the J Language: Totally Control Your Application Development using UML*. Whitepaper, November 1999. http://www.objecteering.com/pdf/whitepapers/us/uml_profiles.pdf.

[36] Laboratoire d'Informatique Paris 6 (LIP6): *MofFact QVT Engine*. http://modfact.lip6.fr/, June 2004.

[37] Java Community Process: *Java Metadata Interface (JMI) Specification*, v1.0, June 2002. Java Specification Request, JSR#040, http://java.sun.com/products/jmi/, June 2004.

[38] French National Institute for Research in Computer Science and Control (INRIA): *Model Transformation Language (MTL)*. http://modelware.inria.fr/, June 2004.

[39] Object Management Group, Inc.: *Naming Service Specification*, v1.2, September 2002.

[40] Object Management Group, Inc.: *Trading Object Service Specification*, v1.0, May 2000.

[41] The Community OpenORB Project: *OpenORB*. http://openorb.sourceforge.net/, June 2004.

[42] Object Management Group, Inc.: *XML Metadata Interchange (XMI) Specification*, v1.2, January 2002.

[43] Object Management Group, Inc.: *XML Metadata Interchange (XMI) Specification*, v2.0, May 2003.

[44] Object Management Group, Inc.: *Meta Object Facility (MOF) Specification*, v1.4, April 2002.

[45] Eclipse Project: *Eclipse*. http://www.eclipse.org/eclipse/, June 2004.

[46] Gamma, E.; Beck, K.: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003.