Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Information and Software Technology 48 (2006) 498–516

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Applying Model-Driven Architecture to achieve distribution transparencies

Apostolos Zarras*

Computer Science Department, University of Ioannina, P.O. Box 1186, GR 45110 Ioannina, Greece

Received 6 September 2004; revised 14 May 2005; accepted 31 May 2005
Available online 12 July 2005

Abstract

This paper proposes a principled methodology for the realization of distribution transparencies. The proposed methodology is placed within the general context of Model-Driven Architecture (MDA) development. Specifically, it consists of a UML-based representation for the specification of platform independent models of a system. Moreover, it comprises an automated aspect-oriented method for the refinement of platform independent models into platform specific ones (i.e. models describing the realization of the system's distribution transparency requirements, based on a standard middleware platform like CORBA, J2EE, COM+, etc.). Finally, the proposed methodology includes an aspect-oriented method for the generation of platform specific code from platform specific models.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Aspect-oriented modeling; Aspect-oriented programming; Refinement; Code generation; Middleware

1. Introduction

Middleware is the current practice in the development of today's software systems [1]. It provides reusable solutions to pervasive software development problems like heterogeneity, interoperability, security, dependability, etc. These solutions are offered either by the core of a middleware platform (i.e. the middleware broker), or by complementary services. The broker mediates the interaction between the elements of a system and masks differences in data representations and communication mechanisms to enable their interoperation. In other words, it provides *access transparency* [2]. The complementary middleware services enable several other *distribution transparencies* like the ones for *location, persistence, failure, transaction*, etc. [2]. Lately, there have been efforts to come up with standards describing the semantics and the structure of middleware platforms, capable of supporting a wide range of distribution transparencies. The Common Object Request Broker Architecture (CORBA) specification [3] is among the well-known results of these efforts. J2EE [4] and COM+ [5] are further widely used infrastructures in both industry and academia.

Given this wide variety of solutions, what is still missing, from an engineering point of view, is a principled methodology that facilitates selecting and using the one that better tackles the particular requirements of the system.

Recently, the OMG architecture board made a statement concerning the coordinated use of existing standards towards Model-Driven Architecture (MDA) development [6]. MDA relies on early ideas, proposed by the software architecture community [7]. More specifically, structural and behavioral models of the system are specified in terms of a standard modeling notation like UML [8]. These models direct the overall development process. In a first step, the models are platform independent, i.e. they describe the Platform Independent Elements (PIEs) of the system, while abstracting away technological details that do not relate with the fundamental functionality of these elements. The Platform Independent Models (PIMs) may be of the following kinds:

- *Enterprise models*, describing the business domain and the business processes of the system.
- *Computational models*, specifying the decomposition of the system into basic computational elements.

* Tel.: +30 26510 98862.
E-mail address: zarras@cs.uoi.gr
URL: http://www.cs.uoi.gr/zarras.

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.05.010

- *Information models*, prescribing the semantics of the information, managed by the computational elements.
- *Engineering models*, specifying the distribution transparency requirements of the system.

The step that follows the specification of PIMs amounts in selecting a middleware platform that provides means for achieving the distribution transparency requirements of the system. Given the selected platform, we have to refine the engineering models of the system into *technology models*, describing *how* the platform is used to achieve these requirements. In the context of MDA, technology models are also called Platform Specific Models (PSMs). In general, the technology models are quite complex compared to the engineering models of the system and their specification requires expertise on the selected middleware platform. The PSMs include Platform Specific Elements (PSEs), corresponding to the PIEs that constitute the refined engineering models. Moreover, the PSMs comprise additional PSEs that are part of the middleware services used for the realization of the distribution transparency requirements. Finally, the PSMs specify relationships between the PSEs, prescribing the achievement of the distribution transparency requirements. The specification of PSMs is necessary as they serve as a blueprint targeted to the developers who take in charge of the system's implementation. The previous is actually the last step in the MDA development process. The system's implementation can be divided into source code that realizes the fundamental behavior of the system's PIEs and Platform Specific Code (PSC), which integrates the corresponding system's PSEs with the additional PSEs of the middleware services used.

The provision of a disciplined development process that relies on standards is by itself a guarantee for building software while successfully balancing the trade-off between the quality of the product and the time-to-market. Architects, designers and developers are constrained in favor of the clear separation of concerns, which promotes design/software reuse and simplifies evolution. However, MDA becomes even more beneficial with the support of automated methods that facilitate the individual steps of the development process. Into this context, in [9] we already proposed a systematic framework that enables the selection of middleware platforms, given the distribution transparency requirements of the system over the middleware.

In this paper, we focus on a principled methodology for the refinement of engineering models into technology models and the automated generation of corresponding platform specific code. Specifically, the main contributions of this paper are:

- A UML representation for the specification of engineering models.
- An automated method for the refinement of engineering models into technology models.
- An automated method for the generation of platform specific code.

Both the refinement and the code generation methods are completely independent from the selected middleware platform. They rely, respectively, on Aspect-Oriented Modeling (AOM) [10,11] and Aspect-Oriented Programming (AOP) [12,13]. Aspect-oriented methods are based on languages that enable the abstract specification of modeling or programming constructs, which must be incorporated in specific points within a model or a program, respectively, towards achieving pervasive concerns like concurrency control, access control, etc. Such kind of specifications are called *aspects*. Aspect-oriented methods further rely on automated tools, called *weavers*. The input of a weaver is a set of aspects and a model or a program. Its output is also a model or a program, enhanced with the additional modeling or programming constructs that are specified within the input aspect.

The remainder of this paper is structured as follows. Section 2 presents the necessary background on Aspect-Oriented development, middleware and distribution transparencies. Section 2, further introduces a motivating example, used throughout the paper to demonstrate the proposed methodology. Section 3 presents the main steps of the methodology. Section 4 details the UML representation for the specification of engineering models. Sections 5 and 6 detail the automated refinement and code generation methods, respectively. Section 7 provides an assessment of the proposed methodology. Section 8 presents related work. Finally, Section 9 summarizes our contribution and points out the future directions of this work.

2. Background and motivating example

Aspect-Oriented development, middleware and distribution transparencies are the foundation concepts employed in the proposed methodology. These concepts are briefly discussed in this section together with a case study followed throughout the paper towards exemplifying the use of the methodology.

2.1. Aspect-oriented programming and modeling

Aspect-Oriented development is an emerging software development paradigm that originates from the need to achieve a high degree of separation of concerns [12,13]. The different concerns involved in the realization of a software system can be divided into problem-specific and cross-cutting ones.

Problem-specific concerns relate to the main functionality that should be provided by the system. Their realization is tackled in traditional software development methodologies by the functional decomposition of the system into abstractions like classes or modules, which represent

Table 1
Basic forms of transparency provided by middleware platforms

Transparency	Semantics	CORBA		J2EE		COM+
Access	Masks differences in data representations and communication mechanisms to enable interoperation	CORBA GIOP Object model	 Component Model (CCM)	JAVA RMI Object Model	 EJBs	DCE RPC DCOM model
Location	Allows accessing elements of the system without knowledge of their physical location	CORBA Naming Service (NS) CORBA Trading Service (TS)		RMI Registry JNDI		GUIDs Monikers
Concurrency	Allows concurrent processing on resources without interference	Concurrency Control Service (CCS)		Java Synchronization Mechanisms		COM Synchronization Mechanisms
Migration	Hides from the system's elements the ability of the system to change their physical location	CORBA Lifecycle Passing objects by value		Passing objects by value		–
Failure	Enables masking from an element the failure and recovery of other elements	Fault tolerant CORBA		–		–
Persistence	Allows masking from an element the deactivation and reactivation of other elements	CORBA Persistent State Service (PSS)		JDBC SQL/J		ADO OLEDB
Transaction	Hides the coordination of certain activities performed towards the atomic and isolated execution of transaction	CORBA Object Transaction Service (OTS)		Java Transaction Service (JTS)		Microsoft Transaction Service (MTS)

problem-specific concerns conceptually or physically. The conceptual abstractions form the system's model, while the physical ones constitute the system's implementation.

Distribution transparencies are typical examples of crosscutting concerns. Their realization imposes the need for additional functionality, which should be spread across the system's problem-specific abstractions. Aspect-Oriented development is based on languages (e.g. AspectJ [14], Hyper/J [15]) that allow specifying abstractions, which modularize this additional functionality. In AspectJ, these abstractions are called aspects. An aspect consists of point-cuts, i.e. sets of related points (called *join-points*) in the source code of the system. Point-cuts are associated with functionality that should execute before, or right after them. This functionality is automatically introduced in the system by the AspectJ weaver. The abstractions used in Hyper/J are called *hyper-slices*. A hyper-slice is a fragment of a class hierarchy. The classes in the fragment contain only methods and attributes that relate to the particular concern, modularized by the fragment. Hyper-slices are automatically composed by merging corresponding join-points, specified within them. Specifying correspondence relationships between the join-points of different hyper-slices is a responsibility of the developer.

2.2. Middleware and distribution transparencies

The different transparencies we consider in this paper are summarized in Table 1. The realization of these transparencies in CORBA, J2EE and COM+ is discussed in detail in [9]. It is important to note here that the realization of some transparencies requires certain others. For example, any distribution transparency involves using middleware services that require a broker, which provides access transparency. Achieving transaction transparency amounts in using services that implement a distributed atomic commitment protocol (e.g. OTS, JTS, MTS in Table 1). These services further require others that enable synchronization and persistence (e.g. PSS, CCS in Table 1). The *realization dependencies* for the transparencies considered in this paper are depicted in Fig. 1 and they play a rather important role in the overall methodology proposed in this paper. Further details are given in the rest of the paper.

2.3. Motivating example

The example we adopt in this paper to highlight the use of the proposed methodology consists of a Climate Control

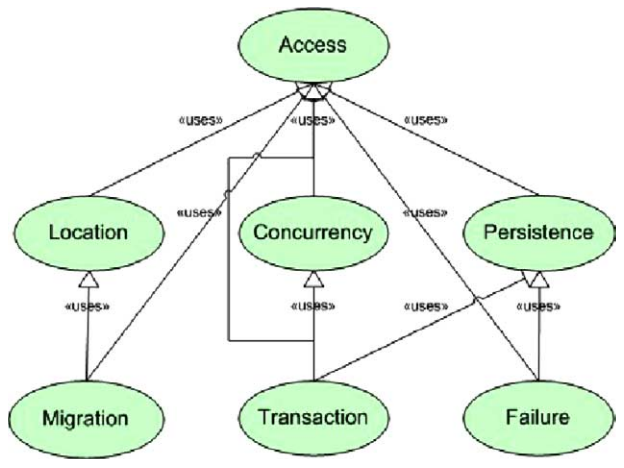


Fig. 1. Realization dependencies among distribution transparencies.

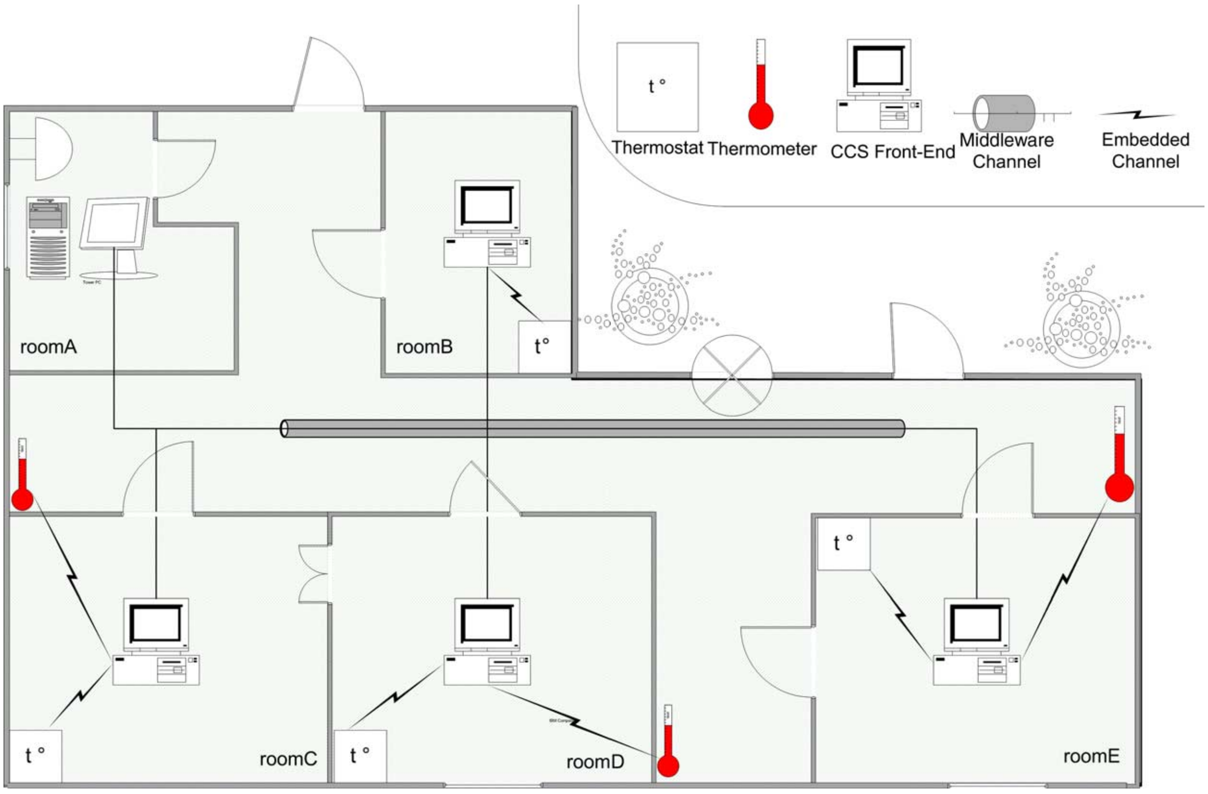


Fig. 2. Overview of the CCS case study.

System (CCS), detailed in [16]. The purpose of this system is to control the air-conditioning in various locations of a building, based on a number of thermometers that report the current temperature in these locations (Fig. 2). Furthermore, CCS aims at controlling the temperature of various manufacturing devices, through the use of device-specific thermostats. The thermometers and the thermostats can be accessed through proprietary communication protocols. Hence, the CCS system provides a front-end to the aforementioned peripherals so as to integrate them with the rest of the IT infrastructure used by the enterprise, installed in the building. The main CCS engineering elements are:

- Front-end elements of type *Thermometer* that provide operations for accessing various technical features of thermometers (e.g. their model, asset number, location). Thermometer elements further provide an operation that reports the current temperature in the location where the devices are installed.
- Front-end elements of type *Thermostat* that inherit the basic functionality of Thermometer elements (i.e. the Thermostat class inherits from the Thermometer class) and additionally provide operations for accessing and changing the temperature of thermostats.
- A *Controller* element that keeps track of the Thermometer and Thermostat elements installed and provides operations for:
  - Listing the Thermometer and the Thermostat elements.
  - Locating a set of Thermometer or Thermostat elements given an asset number, a location, or a model.
  - Updating a set of Thermostat elements as a group by increasing or decreasing their temperature-setting, relatively to the current temperature-setting. Some thermostats may not be able to increase or decrease their temperature as requested because they are already close to their temperature limit. In these cases, the update operation sets those thermostats to their temperature limit.

The requirements for the CCS front-end elements include the ones for access and persistence transparency. Access transparency is important given the heterogeneous nature of the sensing devices. Persistence is also significant, given that the front-end CCS elements may be deactivated and reactivated for several reasons (e.g. power loss, upgrade, etc.). For the Controller element, we further require location transparency to be able to communicate with it indirectly, using a client program that possibly executes on mobile devices.

3. A principled methodology for distribution transparencies

The main objectives of the proposed methodology are the gradual refinement of engineering models into technology models and the generation of platform specific code, given

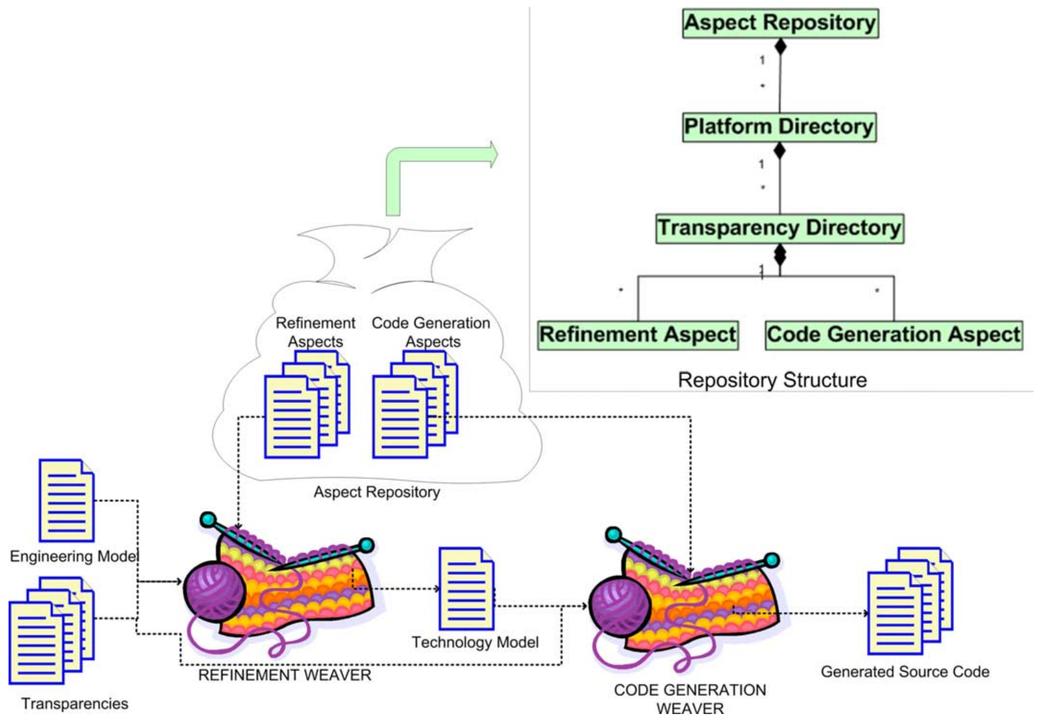


Fig. 3. Overview of the methodology.

a set of distribution transparency requirements. To achieve these objectives we follow the basic steps given in Fig. 3, which are further detailed in the rest of this section.

3.1. Specification of the system's engineering models

This first step of the methodology involves using UML [8] for modeling engineering models. This choice originates from the fact that UML is an emerging industrial standard, providing a rich vocabulary of modeling constructs that enable the specification of both structural and behavioral models of a system. However, the semantics of UML are quite generic. The previous is reasonable, considering that UML aims at becoming a base for the development of a family of representations, called UML profiles, which serve different modeling purposes (e.g. UML profiles for realtime systems [17], CORBA systems [18], etc.). One of these profiles is particularly targeted on modeling Enterprise Distributed Object Computing (EDOC) systems [19]. EDOC provides a foundation for the specification of business and computational PIMs (Section 1). However, EDOC does not support the specification of information, engineering, and technology models and suggests referring to the Reference Model for Open Distributed Processing (RM-ODP) [2] towards dealing with these issues.

RM-ODP proposes a generic architectural style, consisting of different types of engineering elements that should be provided by middleware platforms to facilitate the realization of engineering models. For this architectural style we define a corresponding UML Platform Independent Representation (PIR) for the specification of engineering models. In [19], we resulted in several significant similarities in the architectural styles assumed by CORBA, J2EE and COM+. These similarities mainly originate from the fact that all three platforms are influenced by the RM-ODP architectural style. Hence, the proposed representation is generic enough to tackle the refinement of engineering models into technology models that rely on the three platforms that we consider. The proposed PIR comprises the definition of a number of stereotypes. A *stereotype* consists of a set of constraints and properties, enhancing the definition of a standard class of UML model elements (i.e. a meta-model element, called the *base class* of the stereotype). Applying the stereotype on a particular UML model element of the class (i.e. an instance of the meta-model element) implies that the element conforms to the enhanced definition instead of the standard one.

3.2. Refinement of engineering models into corresponding technology models

The refinement of engineering models into technology models is the second step of the methodology. Specifically, the overall refinement method accepts as input an engineering model, a set of required transparencies and a selected middleware platform that is going to be used to achieve them. Its output is a technology model, which describes how we use the selected platform to achieve the required set of transparencies. Building this output model amounts in performing the following steps:

(1) Map the PIEs and the relationships of the engineering model into corresponding PSEs and relationships in

the technology model. These PSEs are called hereafter the *direct mappings* of PIEs.

(2) Introduce in the technology model additional PSEs for the realization of the required set of transparencies.

(3) Establish the relationships (associations, aggregations, generalizations, dependencies) between the direct mappings of PIEs and the additional PSEs, introduced in the previous step.

As already mentioned in Section 1, one of our basic concerns is to keep the refinement method independent from the middleware platforms that may be selected for the construction of the target technology model. To satisfy this concern, we use an AOM approach. Specifically, the refinement method is an AOM weaver, which retrieves from a *repository* a set of *refinement aspects*, corresponding to the set of distribution transparency requirements. Each aspect consists of a number of *point-cuts*. A point-cut specifies generically a group of PIEs or a group of relationships. We may group PIEs or relationships based on their types, their behavioral features (i.e. operations), their structural features (i.e. attributes), or the stereotypes that characterize them. Each point-cut contains *advice* statements, specifying the mapping of the grouped elements and relationships in the resulting technology model. The mapping relies on a UML-based Platform Specific Representation (PSR) that corresponds to the selected middleware platform. In particular, we consider PSRs for CORBA, J2EE and COM+. The CORBA PSR relies on the standard UML profile for CORBA [18]. The J2EE and the COM+ representations are based on existing non-standard representations that come along with widely used UML modeling tools (e.g. the Rational Rose modeling tool[1]). The advice statements further prescribe additional PSEs, relationships, behavioral and structural features that should be introduced in the technology model for each one of the grouped elements.

3.3. Generation of platform specific code

The last step of the methodology comprises the generation of platform-specific code. The input to the code generation method is a technology model that results from the application of the refinement method and the set of required transparencies that were used to produce it. Its output comprises a number of source code files corresponding to the elements of the input model. Specifically, the code generation method results in:

(1) *Skeleton code* for the direct mappings of PIEs.

(2) *Implementation code*, needed for the integration of the direct mappings of PIEs with the additional PSEs, introduced in the technology model towards the realization of the required set of transparencies.

Alike the refinement method, the code generation one is kept platform-independent. To satisfy this issue we use an AOP approach. In particular, the code generation method is realized as an AOP weaver, which uses a set of code generation aspects, corresponding to the set of distribution transparency requirements. A code generation aspect consists of the following kinds of point-cuts:

(1) *Point-cuts for skeleton code:* specifying groups of direct mappings of PIEs. A point-cut for skeleton code may further group behavioral or structural features of PIEs. Each point-cut contains a number of *advice* statements, specifying the skeleton code that must be generated for each one of the grouped entities.

(2) *Point-cuts for implementation code:* specifying groups of points within the skeleton code that is generated for the direct mappings of PIEs. Each point-cut contains *advice* statements, specifying the code to be generated at these points towards implementing a relationship between the direct mappings of PIEs and the additional PSEs that were introduced in the technology model during the refinement method.

Fig. 3 depicts the overall organization of the repository that supports the refinement and the code generation methods. In principle, different combinations of PSEs can be employed towards achieving a set of required transparencies [9]. This is immediately visible in Table 1. In CORBA, for instance, we can achieve location transparency using either the CORBA Naming Service, or the CORBA Trading Service [20]. Consequently, the aspect repository consists of platform-specific directories, divided into transparency-specific sub-directories. A sub-directory contains a set of refinement aspects, describing *alternative* ways of refining an engineering model, to achieve the particular form of transparency. Each refinement-aspect is further associated with a corresponding code generation aspect.

Further details on the constructs we use for the specification of refinement and code-generation aspects are provided in Sections 5 and 6.

4. UML representation for engineering models

The stereotypes we define for the specification of engineering models rely on UML v2.0 [8], which has been recently finalized. This particular version enhances previous UML versions by introducing the concepts of components and connectors. Historically, components and connectors were the basic modeling concepts of Architecture Description Languages (ADLs) [21]. In UML v2.0, a *Component* is a modular part of a system that defines its

[1] http://www.rational.com.

behavior in terms of provided and required interfaces. A component may be of an arbitrary granularity, consisting of other components. UML v2.0 allows distinguishing between components that can be *directly instantiated* and components that only exist at *design-time*. The latter are *indirectly instantiated* through the explicit instantiation of their constituent elements. A UML *Connector* specifies a link that enables the communication between component instances. It represents a communication protocol between the instances and it is used for binding parts of components' functionality that realize a provided interface, with corresponding parts of components' functionality that require using this interface. The connector may be an instance of an association, specified between the components, whose instances are linked.

As discussed in the RM-ODP standard, engineering models are refinements of corresponding computational models, which describe the functional decomposition of a system into *basic engineering elements*. Engineering models additionally specify the way engineering elements are organized in a distributed execution environment and their distribution transparency requirements. At this point, we could argue that engineering models violate the overall idea of separating different concerns as they mix-up information regarding the functional decomposition of the system with information related to the system's distribution. However, this recognized limitation is imposed by the RM-ODP standard as it is necessary towards the refinement of engineering models into technology ones and the generation of platform-specific code.

The basic engineering elements of a system are specified using the EngElem stereotype, given in Table 2. This stereotype is derived by the generic PIE stereotype which represents all the different kinds of modeling elements used in the specification of engineering models. By definition, the basic engineering elements are UML Components that can be directly instantiated. They are primitive in that their constituent elements do not have any further distribution transparency requirements. Provided and required interfaces are specified using the EngInterf stereotype, whose base class is the standard UML Interface element. A basic engineering element *X* that requires an interface *Z*, communicates with another one *Y* that provides *Z* though the use of an *interface reference*. Interface references are specified using the EngRef stereotype.

Table 2
UML stereotypes for the specification of engineering models

Stereotype	Base class	Parent	Constraints
PIE	Component	NA	NA
EngElem	Component	PIE	isIndirectlyInstantiated = false
EngInterf	Interface	PIE	NA
EngRef	Component	PIE	NA
EngCluster	Component	PIE	isIndirectlyInstantiated = true
ClusterMgr	Component	EngRef	NA
EngCapsule	Component	PIE	isIndirectlyInstantiated = false
CapsuleMgr	Component	EngRef	NA
EngChannel	Association	PIE	self.connector-⟩forAll(c|c. kind = assemply)

The basic engineering elements are organized into *clusters*, specified using the EngCluster stereotype; all the elements that belong to a particular cluster form a single unit for the purpose of activation and deactivation. Moreover, they share common transparency-related properties. Clusters are design-time elements; hence they cannot be directly instantiated. Each cluster is associated with a cluster manager, i.e. the cluster holds a reference to a basic engineering element, which coordinates the activation and deactivation of the elements contained in the cluster and participates in the realization of their distribution transparency requirements. To specify references to cluster managers we define the ClusterMgr stereotype, which is derived from the EngRef stereotype.

Clusters are organized into *capsules* for the purpose of encapsulation of processing, storage, and request flow. In other words, capsules model different types of runtime processes, consisting of basic engineering elements. Capsules can be directly instantiated and we specify them using the EngCapsule stereotype. As with the case of clusters, each capsule is associated with a capsule manager, i.e. the capsule holds a reference to a basic engineering element that coordinates the managers of the capsule's constituent clusters. The references to capsule managers are specified using the CapsuleMgr stereotype. Basic engineering elements that belong to different capsules communicate through *channels*. A channel is an association between an interface reference, held by an engineering element that requires the referenced interface, and an engineering element that provides the referenced interface. An instance of this association is a UML Connector. To specify channels in UML we use the EngChannel stereotype.

To demonstrate the use of the PIR we defined, we use the motivating example of Section 2. Figs. 4 and 5 give, respectively, the runtime and the static views of CCS for the building showed in Fig. 2. In the remainder of this paper, we concentrate on the refinement of the structural models of CCS. However, the proposed methodology further concerns behavioral models, which are treated in a similar way. Fig. 5(a) gives the interfaces, the structural and the behavioral features of the Thermometer, the Thermostat and the Controller elements, which constitute the basic engineering elements of CCS. In each one of rooms B–E of the building there is an instance of the Location capsule, whose structure is given in Fig. 5(b). Specifically, in room E the Location capsule instance encapsulates a Thermometer and a Thermostat instance, managed by an instance of the ThClMgr cluster manager. The overall capsule is managed by an instance of the LCpsMgr capsule manager. Room A contains an instance of the ControlLocation capsule, whose structure is given in Fig. 5(c). In particular, this capsule contains an instance of the Controller element, which encapsulates interface references to

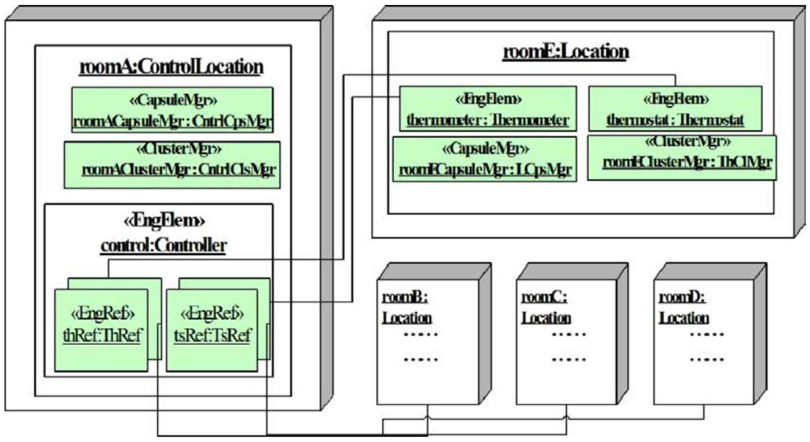

roomA:ControlLocation
«CapsuleMgr» roomACapsuleMgr : CntrlCpsMgr
«ClusterMgr» roomAClusterMgr : CntrlClsMgr
«EngElem» control:Controller
«EngRef» thRef:ThRef
«EngRef» tsRef:TsRef
roomE:Location
«EngElem» thermometer : Thermometer
«EngElem» thermostat : Thermostat
«CapsuleMgr» roomECapsuleMgr : LCpsMgr
«ClusterMgr» roomEClusterMgr : ThClMgr
roomB: Location
roomC: Location
roomD: Location

Fig. 4. The runtime view of CCS.

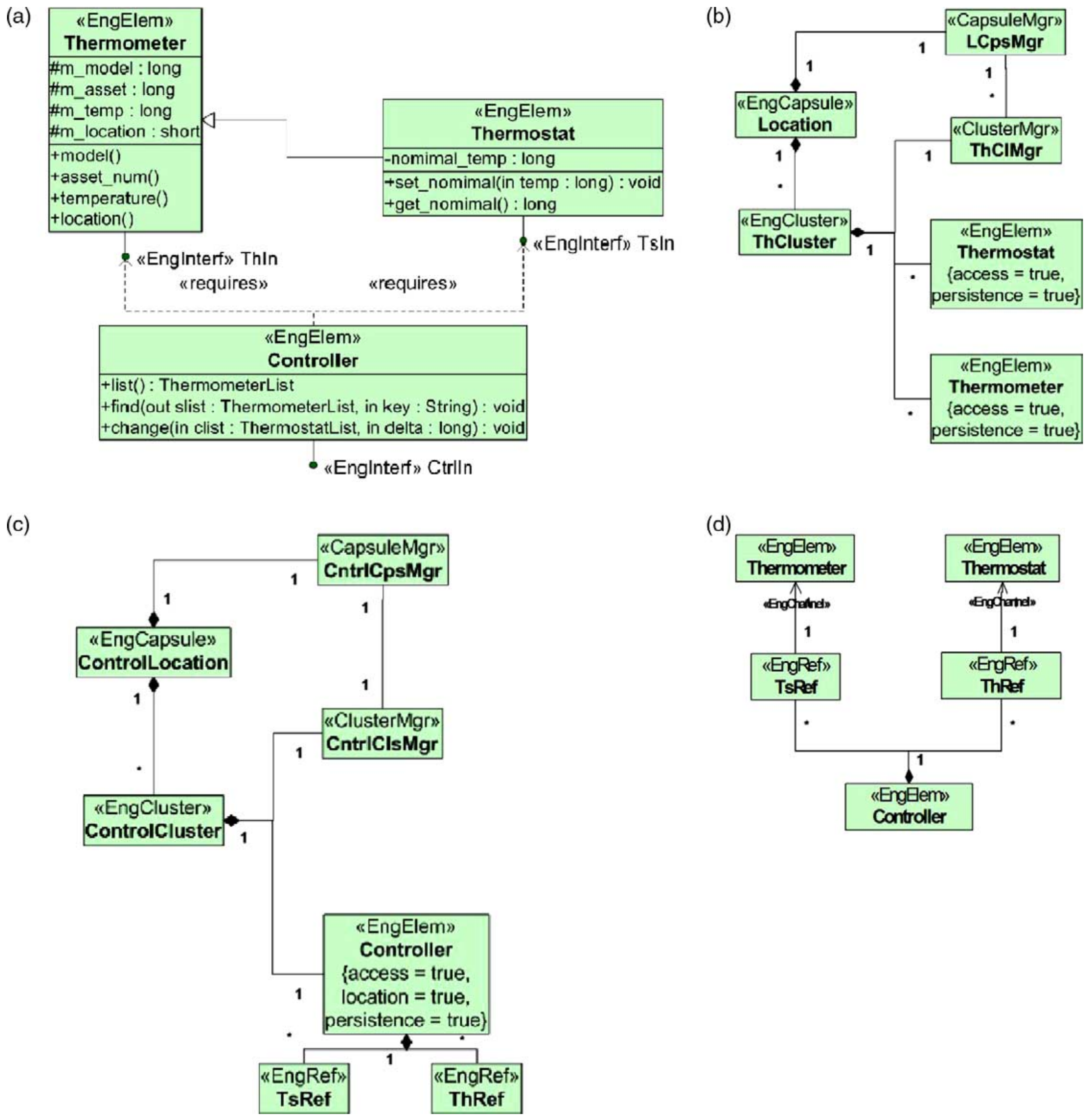


Fig. 5. The engineering model of CCS. (a) Basic elements and interfaces. (b) The structure of the Location capsule. (c) The structure of the ControlLocation capsule. (d) Basic communication channels.

the Thermometer and the Thermostat elements installed in rooms B–E. These references are linked with the corresponding basic engineering elements through instances of engineering channels, as prescribed by the structural diagram given in Fig. 5(d).

5. Refinement method

This section provides details regarding the specification of refinement aspects and the functioning of the refinement weaver.

5.1. Refinement aspects

Defining point-cuts that group different kinds of PIEs and structural or behavioral features provided by these PIEs, involves using constructs like the ones given in Table 3. The constructs of this table specifically focus on components and interfaces. Similar constructs are used for the case of relationships (i.e. associations, generalizations, or dependencies). All the different point-cut constructs accept as input a number of arguments that correspond to properties characterizing the grouped PIEs and relationships. Such properties are their type, the stereotype that characterizes them, etc. The arguments serve for constraining the grouped elements and relationships, determining thus the contents of the resulting group. All of the arguments are optional. In case that we choose to leave an argument unspecified we use a named variable in its place, specified using the ? symbol.

To specify advice statements within the point-cuts of Table 3 we rely on the constructs given in Table 4. Similar advice statements are used for point-cuts that group relationships. Each advice implies that the refinement weaver must create a PSE, or a relationship *X* for every PIE or relationship *Y* that belongs in the group, specified by the point-cut, which encapsulates the advice. Each advice takes as input a number of arguments, which determine the properties of the PSE or the relationship that is to be created. Such arguments may be the type of the element that is to be created, the stereotype that characterizes the element, etc. As with the case of point-cuts, all of the arguments are optional. An unspecified argument in an advice statement is

Table 3
Point-Cuts for grouping UML model elements

Point-Cut:
`MatchElem(Stereotype, Type)`: Groups stereotyped components or interfaces. These elements may be constrained by their type or by the stereotype that characterizes them
Example:
`MatchElem(EngElem, ?x)`: Groups components, characterized by the EngElem stereotype

Point-Cut:
`MatchInstance(Stereotype, Instance, Type)`: Groups component instances. These instances may be constrained by their name and type. Moreover, they may be constrained by the stereotype that characterizes their type
Example:
`MatchInstance(EngElem, ?x, Thermometer)`: Groups Thermometer instances, characterized by the EngElem stereotype

Point-Cut:
`MatchStructuralFeature(Stereotype, Type, Feature, FeatureType, Visibility)`: Groups structural features. The features are constrained by the type and the stereotype of the components or the interfaces that contain them. Moreover, the features are constrained by their type, their name and their visibility. The latter may be set to public, private, or protected
Example:
`MatchStructuralFeature(EngElem, Thermometer, ?x, ?y, private)`: Groups all the private structural features of the Thermometer element

Point-Cut:
`MatchBehavioralFeature(Stereotype, Type, Feature, FeatureRetType, Visibilty, Arg1, ArgType1, ArgKind1…, ArgN, ArgTypeN, ArgKindN)`: Groups behavioral features. The features may be constrained by the type and the stereotype of the components or the interfaces that contain them. Moreover, the features are constrained by their name, their return type, their visibility and their arguments. An arbitrary number of arguments may be specified within the MatchBehavioralFeature construct. For each argument we specify its name, type and kind. The latter may be set to in, out, or inout
Example:
`MatchBehavioralFeature(EngElem, Thermometer, ?x, void, private, ?y, ?z, ?w,…, ?o, ?p, ?q)`: Groups all the private behavioral features of the Thermometer element. These features may have an arbitrary number of arguments but they are constrained to have a void return type

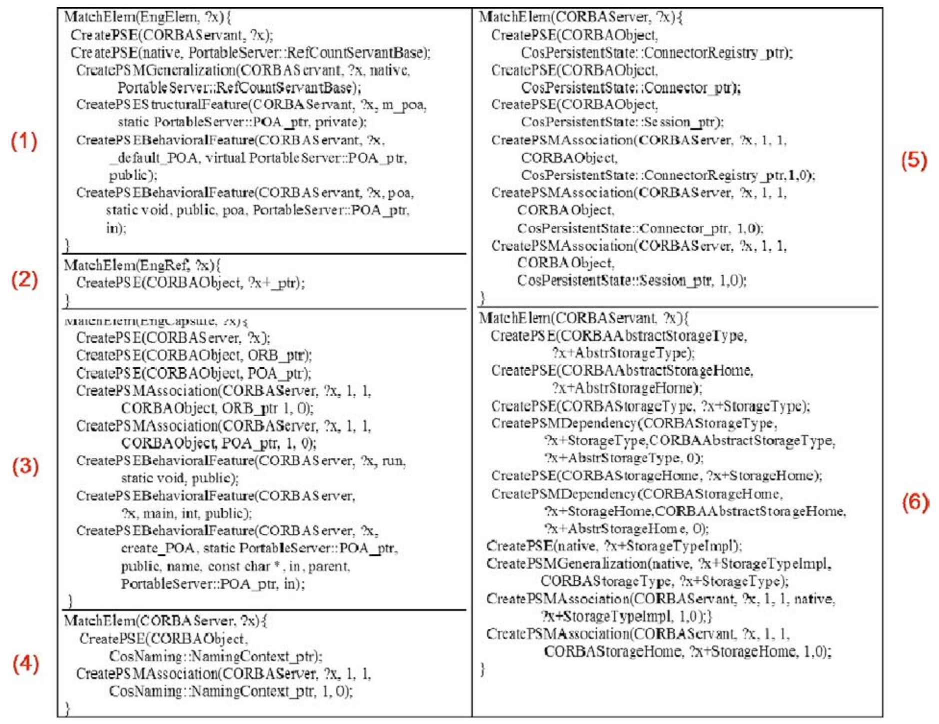
Table 4
Advice statements for UML model elements

Advice:
`CreatePSEElem(Stereotype, Type)`: creates a stereotyped component or interface
Example:
CreatePSEElem(CORBAServant, ?x): creates a CORBAServant element of type ?x

Advice:
`CreatePSEInstance(Stereotype, Instance, Type)`: creates a component instance
Example:
`CreatePSEInstance(CORBAServant, ?x, Thermometer)`: creates a CORBAServant instance ?x of type Thermometer

Advice:
`CreatePSEStructuralFeature(Stereotype, Type, Feature, FeatureType, Visibility)`: creates a structural feature within a stereotyped component
Example:
`CreatePSEStructuralFeature(CORBAServant, Thermometer, ?x, ?y, private)`: creates a private structural feature ?x of type ?y within a CORBAServant element of type Thermometer

Advice:
`CreatePSEBehavioralFeature(Stereotype, Type, Feature, FeatureRetType, Visibilty, Arg1, ArgType1, ArgKind1…, ArgN, ArgTypeN, ArgKindN)`: creates a behavioral feature within a stereotyped component or interface
Example:
`CreatePSEBehavioralFeature(CORBAServant, ?x, _default_POA, void, public)`: creates an public operation, named _default_POA, within a CORBAServant element of type ?x. The feature takes no arguments and returns no value



```
(1)
MatchElem(EngElem, ?x){
 CreatePSE(CORBAServant, ?x);
 CreatePSE(native, PortableServer::RefCountServantBase);
  CreatePSMGeneralization(CORBAServant, ?x, native,
        PortableServer::RefCountServantBase);
  CreatePSEStructuralFeature(CORBAServant, ?x, m_poa,
        static PortableServer::POA_ptr, private);
  CreatePSEBehavioralFeature(CORBAServant, ?x,
       _default_POA, virtual PortableServer::POA_ptr,
       public);
  CreatePSEBehavioralFeature(CORBAServant, ?x, poa,
       static void, public, poa, PortableServer::POA_ptr,
       in);
}

(2)
MatchElem(EngRef, ?x){
  CreatePSE(CORBAObject, ?x+_ptr);
}

(3)
MatchElem(EngCapsule, ?x){
  CreatePSE(CORBAServer, ?x);
  CreatePSE(CORBAObject, ORB_ptr);
  CreatePSE(CORBAObject, POA_ptr);
  CreatePSMAssociation(CORBAServer, ?x, 1, 1,
        CORBAObject, ORB_ptr 1, 0);
  CreatePSMAssociation(CORBAServer, ?x, 1, 1,
        CORBAObject, POA_ptr, 1, 0);
  CreatePSEBehavioralFeature(CORBAServer, ?x, run,
        static void, public);
  CreatePSEBehavioralFeature(CORBAServer,
        ?x, main, int, public);
  CreatePSEBehavioralFeature(CORBAServer, ?x,
        create_POA, static PortableServer::POA_ptr,
        public, name, const char *, in, parent,
        PortableServer::POA_ptr, in);
}

(4)
MatchElem(CORBAServer, ?x){
  CreatePSE(CORBAObject,
       CosNaming::NamingContext_ptr);
  CreatePSMAssociation(CORBAServer, ?x, 1, 1,
       CosNaming::NamingContext_ptr, 1, 0);
}

(5)
MatchElem(CORBAServer, ?x){
  CreatePSE(CORBAObject,
       CosPersistentState::ConnectorRegistry_ptr);
  CreatePSE(CORBAObject,
       CosPersistentState::Connector_ptr);
  CreatePSE(CORBAObject,
       CosPersistentState::Session_ptr);
  CreatePSMAssociation(CORBAServer, ?x, 1, 1,
       CORBAObject,
       CosPersistentState::ConnectorRegistry_ptr,1,0);
  CreatePSMAssociation(CORBAServer, ?x, 1, 1,
      CORBAObject,
      CosPersistentState::Connector_ptr, 1,0);
  CreatePSMAssociation(CORBAServer, ?x, 1, 1,
      CORBAObject,
      CosPersistentState::Session_ptr, 1,0);
}

(6)
MatchElem(CORBAServant, ?x){
  CreatePSE(CORBAAbstractStorageType,
          ?x+AbstrStorageType);
  CreatePSE(CORBAAbstractStorageHome,
          ?x+AbstrStorageHome);
  CreatePSE(CORBAStorageType, ?x+StorageType);
  CreatePSMDependency(CORBAStorageType,
         ?x+StorageType,CORBAAbstractStorageType,
         ?x+AbstrStorageType, 0);
  CreatePSE(CORBAStorageHome, ?x+StorageHome);
  CreatePSMDependency(CORBAStorageHome,
         ?x+StorageHome,CORBAAbstractStorageHome,
         ?x+AbstrStorageHome, 0);
 CreatePSE(native, ?x+StorageTypeImpl);
 CreatePSMGeneralization(native, ?x+StorageTypeImpl,
        CORBAStorageType, ?x+StorageType);
 CreatePSMAssociation(CORBAServant, ?x, 1, 1, native,
        ?x+StorageTypeImpl, 1,0);}
 CreatePSMAssociation(CORBAServant, ?x, 1, 1,
         CORBAStorageHome, ?x+StorageHome, 1,0);
}
```

Fig. 6. CORBA-based refinement aspects for CCS.

given in terms of a named variable, which should match with a named variable specified in the point-cut that encapsulates the advice statement. This signifies that in both X, Y, the corresponding property is the same.

As discussed in Section 2, there exist certain realization dependencies between different distribution transparencies. Consequently, there also exist dependencies between the specification of their corresponding refinement aspects. In particular, the refinement aspect for access transparency is completely independent and specifies the mapping of the different kinds of PIEs, used in an input engineering model, into corresponding PSEs that are going to be used in the output technology model. The refinement aspects for location, concurrency and persistence prescribe additional PSEs that must be used to achieve these transparencies. Moreover, they specify the relationships between the additional PSEs and the PSEs, specified in the access transparency aspect. The aspect that realizes migration incorporates additional PSEs in the resulting technology model and further relates these elements with the PSEs involved in the aspects that realize the access and the location transparencies. Similarly, the aspect for transaction transparency relates its PSEs with the ones involved in the aspects that realize access, concurrency and persistence. Finally, the aspect for failure transparency relates its PSEs with the PSEs of the aspects that realize access and persistence.

To further exemplify the use of refinement aspects we revisit the CCS case study. Specifically, assume that we select CORBA as the target platform for the realization of the transparencies required by CCS. Access transparency in CORBA relies on the standard CORBA Object Model (Table 1). According to this model, the basic engineering elements of a system are called servants and they are built using conventional programming languages like C++, Java, etc. A servant realizes a number of CORBA interfaces. CORBA interfaces define different types of CORBA object references that can be held by clients who wish to interact with the servant. Based on the previous, to refine the CCS engineering model into a CORBA technology model we map EngElem elements into CORBAServant elements. Moreover, we map EngRef elements into CORBAObject elements. The aforementioned mappings are specified using the point-cut statements described in Fig. 6(1) and (2), respectively. Fig. 6(1) further specifies that CORBAServant elements must inherit from certain CORBA specific classes and encapsulate additional structural and behavioral features, required for the elements' realization.

CORBA servants that share common transparency-related properties are conceptually organized into clusters managed by CORBA objects that implement the standard POA (Portable Object Adapter) interface.[2] Hence, the ClusterMgr elements used in the CCS engineering model are mapped into POA elements. One or more clusters, sharing the same processing resources constitute a CORBA server. Consequently, the EngCapsule elements in CCS are mapped into CORBAServer elements. This is achieved

[2] A more sophisticated approach for clustering objects consists of building CORBA components instead of typical CORBA objects. Components extend the semantics of simple objects in that they can register to containers, which implicitly manage the components' activation/deactivation, transactions, security, etc.

using the point-cut statement described in Fig. 6(3). According to this point-cut, each server holds a reference to a CORBA object that implements the standard ORB interface. The ORB element plays the role of the capsule manager. Moreover, each server provides a number of additional behavioral features, serving for the server's initialization. CORBAObject references are linked with the CORBAServant elements that provide them though typical RPC channels, consisting of proxies, skeletons, binders and GIOP protocol objects.

To realize location transparency we use the standard CORBA Naming service [20] (Table 1), which allows servants to publish the CORBA object references that they provide under publicly known names. Clients can then retrieve those references and communicate with the servants without any knowledge of their physical location. Fig. 6(4) gives a point-cut used to incorporate the Naming service in the CCS case study. This point-cut groups all the CORBAServer elements, created according to the point-cut specified in Fig. 6(3). For each CORBAServer element, a reference to the Naming service is created.

The case of persistence transparency is slightly more complicated. Specifically, for every CORBA server we have to create a connection to a data-store. The creation of this connection is realized according to the point-cut given in Fig. 6(5). For every servant encapsulated by the server we have to create a corresponding storage object (i.e. a set of typed attributes that constitute the state of the servant) in the data-store. The creation of storage objects involves several development activities including the specification of their type (in PSDL [20]) and the implementation of the functionality that they provide. The aforementioned activities are codified in the point-cut specification given in Fig. 6(6). Finally, we must appropriately configure certain properties of the POA object that manages the life-cycle of each servant so as to create object references that persist to the servant's deactivation and reactivation [16].

5.2. Refinement weaver

The refinement weaver accepts as input an engineering model E, a set of required transparencies, $T=\{t_i|i=1,\ldots,N\}$, and a selected middleware platform. In order to generate the resulting technology model it performs the following steps:

(1) Given the set of required transparencies T, the weaver checks its *completeness*. Specifically, it checks whether T includes one or more transparencies that depend on certain other transparencies, $\mathrm{DepT}=\{t_k|i=1,\ldots,K\}$, which are not included in T. Performing the previous relies on the transparency realization dependencies, given in Fig. 1. Then, it completes the set of required transparencies, i.e. it performs the following: $T=T\cup\mathrm{DepT}$.

(2) Based on the selected input platform, the weaver retrieves from the aspect repository the contents of a number of transparency-directories, $\mathrm{TD}=\{\mathrm{td}_i|i=1,\ldots,N+K\}$. Each td_i corresponds to a required transparency $t_i\in T$. Recall that td_i is a set of refinement aspects, $\mathrm{td}_i=\{\mathrm{rasp}_{i_j}|j=1,\ldots,M\}$, specifying alternative ways for achieving t_i.

(3) For every t_i, the weaver exposes to the designer the alternative refinement aspects $rasp_{i_j}$ and requires him to select one of them towards the refinement of E. The result from this step is a set of refinement aspects $\mathrm{RA}=\{\mathrm{rasp}_i|i=1,\ldots,N+K\}$, containing at least one that realizes access transparency. The previous holds because all of the distribution transparencies depend on access transparency (Fig. 1).

(4) Following, the weaver applies the refinement aspects in the PIEs of the input engineering model, depending on the particular distribution transparency requirements of each one of them. The aspects are applied in the order imposed by the transparency realization dependencies of Fig. 1. Specifically, the first aspect is always the one that realizes access transparency.

(5) The result of the previous step is a technology model T_1, which is subsequently refined into T_2 based on the aspects of RA that depend *only* on the one for access transparency. The order used for applying these aspects does not affect T_2. In general, T_i is refined based on aspects that depend *only* on the ones that were used for producing $T_1, T_2,\ldots,T_{i-1}$.

At this point, we must highlight that the overall refinement method further facilitates the *upgrade* of a particular technology model. The refinement method consists of subsequent refinement steps, *each one of which is applied on the results of its preceding steps*. Consequently, the refinement weaver may eventually accept as input a technology model TE that realizes a particular set of transparencies T and an additional set of required transparencies T'. Then, it can produce an enhanced technology model TE' that provides $T\cup T'$.

Going back to our example scenario, we have chosen CORBA for the realization of the access, the location and the persistence transparencies. Based on the refinement aspects we discussed in Section 5.1 (the most important parts of which were given in Fig. 6), we refine the engineering model of CCS (Fig. 5). The resulted technology model is given in Fig. 7.

As prescribed in the point-cut of Fig. 6(1), the EngElem elements are mapped to CORBAServant elements (Fig. 7(a)). The latter inherit from CORBA specific classes and provide additional behavioral and structural features, used for their initialization and management. In Fig. 7(b) and (c) the structure of the Location and the ControlLocation elements is enhanced with additional elements (e.g. Session_ptr, Connector_ptr), used for connecting CCS with a data-store. These elements are added with respect to the point-cut of Fig. 6(5). Several elements are further associated with the Thermometer the Thermostat and the Controller elements. As imposed by the point-cut of Fig. 6(6), these additional elements are used for the specification (e.g. ThermometerStorageType, ThermometerAbstrStorageType) and the realization (e.g. ThermometerStorageTypeImpl) of storage objects.

6. Code generation method

This section details the specification of code generation aspects and the code generation weaver.

6.1. Code generation aspects

To generate skeleton code we specify point-cuts that group different kinds of PSEs, used within a PSM that results from the refinement method. This is done using the constructs we introduced in Table 3 for the specification of refinement aspects. The advices given for the point-cuts are collections of code statements. In our case study, for instance, we use an access transparency code generation aspect to generate skeleton implementations for the CORBAServant, the CORBAServer, and the CORBAInterface PSEs, specified in Fig. 7. Specifically, for CORBAServant PSEs we generate C++ classes, containing the *declarations* of attributes and operations that correspond to the structural and behavioral features of these PSEs. For the operations defined in the classes we further generate corresponding C++ *definitions*. Similarly, for the CORBAServer PSEs the aspect generates attribute and operation declarations and definitions. For CORBAInterface PSEs we generate CORBA IDL interfaces. In the case study, we further use an aspect that generates skeleton code for persistence transparency. In particular, this aspect generates PSDL specifications and skeleton C++ implementation classes for the different types of storage objects used. Note that for the case of location transparency there is no need to generate any skeleton code. Obviously, the code statements, concerning the generation of skeleton code depend on properties that characterize the grouped elements. The names of the generated CORBA IDL interfaces, for instance, must match the names of

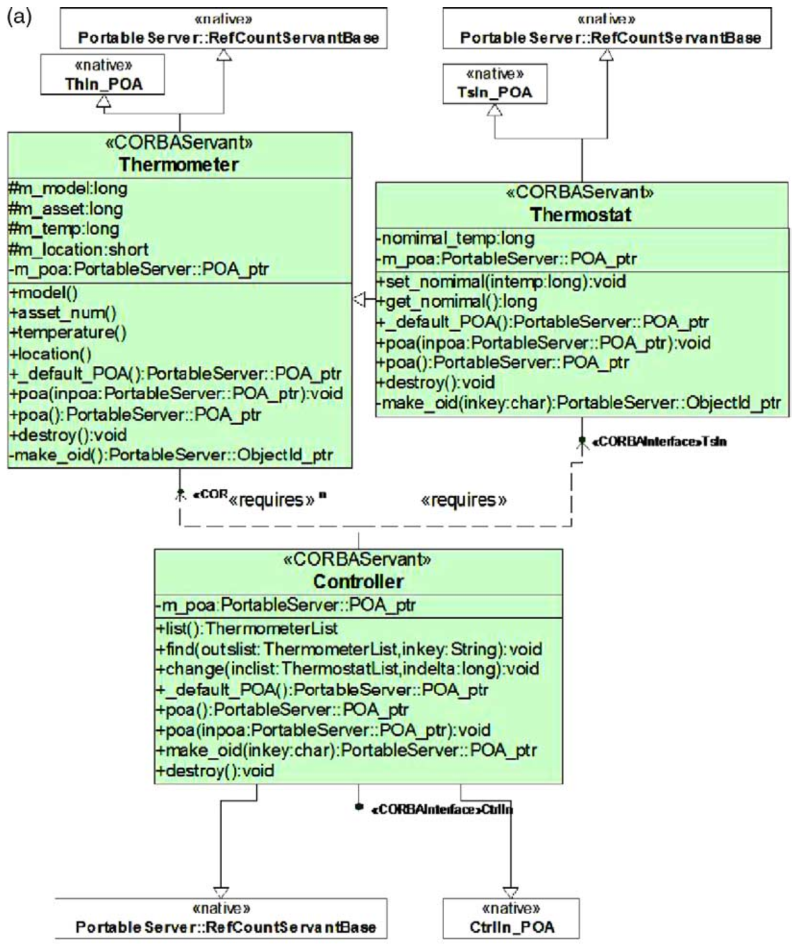

(a)
«native» PortableServer::RefCountServantBase
«native» ThIn_POA
«native» PortableServer::RefCountServantBase
«native» TsIn_POA
«CORBAServant» Thermometer
#m_model:long
#m_asset:long
#m_temp:long
#m_location:short
-m_poa:PortableServer::POA_ptr
+model()
+asset_num()
+temperature()
+location()
+_default_POA():PortableServer::POA_ptr
+poa(inpoa:PortableServer::POA_ptr):void
+poa():PortableServer::POA_ptr
+destroy():void
-make_oid():PortableServer::ObjectId_ptr
«CORBAServant» Thermostat
-nomimal_temp:long
-m_poa:PortableServer::POA_ptr
+set_nomimal(intemp:long):void
+get_nomimal():long
+_default_POA():PortableServer::POA_ptr
+poa(inpoa:PortableServer::POA_ptr):void
+poa():PortableServer::POA_ptr
+destroy():void
-make_oid(inkey:char):PortableServer::ObjectId_ptr
«CORBAInterface»TsIn
«COR «requires»
«requires»
«CORBAServant» Controller
-m_poa:PortableServer::POA_ptr
+list():ThermometerList
+find(outslist:ThermometerList,inkey:String):void
+change(inclist:ThermostatList,indelta:long):void
+_default_POA():PortableServer::POA_ptr
+poa():PortableServer::POA_ptr
+poa(inpoa:PortableServer::POA_ptr):void
+make_oid(inkey:char):PortableServer::POA_ptr
+destroy():void
«CORBAInterface»CtrlIn
«native» PortableServer::RefCountServantBase
«native» CtrlIn_POA

Fig. 7. The technology model of CCS. (a) Basic elements and interfaces. (b) The structure of the Location capsule. (c) The structure of the ControlLocation capsule. (d) Basic communication channels.

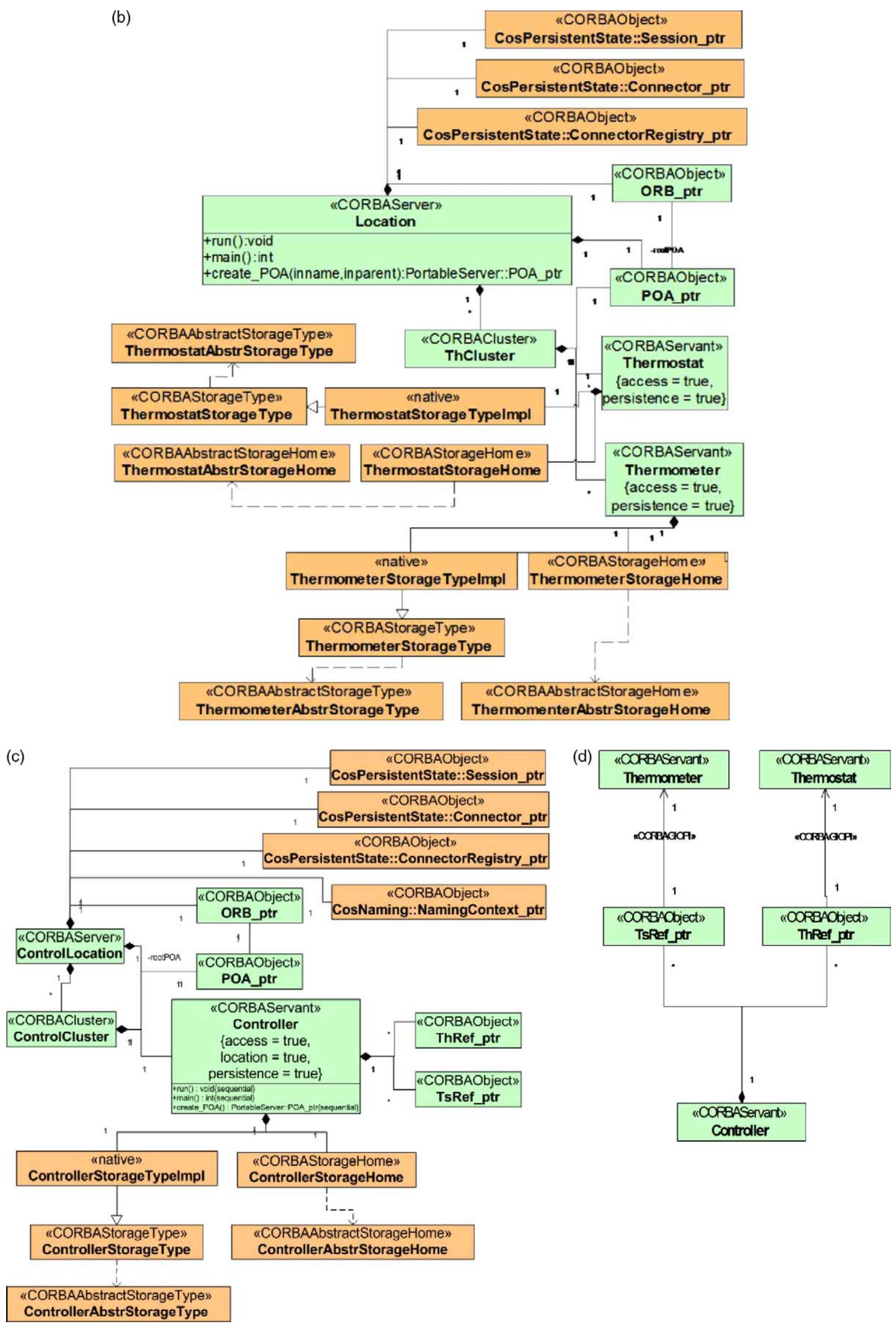

(b)
«CORBAObject» CosPersistentState::Session_ptr
«CORBAObject» CosPersistentState::Connector_ptr
«CORBAObject» CosPersistentState::ConnectorRegistry_ptr
«CORBAObject» ORB_ptr
«CORBAServer» Location
+run():void
+main():int
+create_POA(inname,inparent):PortableServer::POA_ptr
-rootPOA
«CORBAObject» POA_ptr
«CORBAAbstractStorageType» ThermostatAbstrStorageType
«CORBACluster» ThCluster
«CORBAServant» Thermostat {access = true, persistence = true}
«CORBAStorageType» ThermostatStorageType
«native» ThermostatStorageTypeImpl
«CORBAAbstractStorageHome» ThermostatAbstrStorageHome
«CORBAStorageHome» ThermostatStorageHome
«CORBAServant» Thermometer {access = true, persistence = true}
«native» ThermometerStorageTypeImpl
«CORBAStorageHome» ThermometerStorageHome
«CORBAStorageType» ThermometerStorageType
«CORBAAbstractStorageType» ThermometerAbstrStorageType
«CORBAAbstractStorageHome» ThermomenterAbstrStorageHome
(c)
«CORBAObject» CosPersistentState::Session_ptr
«CORBAObject» CosPersistentState::Connector_ptr
«CORBAObject» CosPersistentState::ConnectorRegistry_ptr
«CORBAObject» ORB_ptr
«CORBAObject» CosNaming::NamingContext_ptr
«CORBAServer» ControlLocation
-rootPOA
«CORBAObject» POA_ptr
«CORBACluster» ControlCluster
«CORBAServant» Controller {access = true, location = true, persistence = true}
+run() : void{sequential}
+main() : int{sequential}
+create_POA() : PortableServer::POA_ptr{sequential}
«CORBAObject» ThRef_ptr
«CORBAObject» TsRef_ptr
«native» ControllerStorageTypeImpl
«CORBAStorageHome» ControllerStorageHome
«CORBAStorageType» ControllerStorageType
«CORBAAbstractStorageHome» ControllerAbstrStorageHome
«CORBAAbstractStorageType» ControllerAbstrStorageType
(d)
«CORBAServant» Thermometer
«CORBAServant» Thermostat
«CORBAGIOP»
«CORBAGIOP»
«CORBAObject» TsRef_ptr
«CORBAObject» ThRef_ptr
«CORBAServant» Controller

Fig. 7 (*continued*)

the grouped CORBAInterface elements. Similarly, the names of the generated C++ classes must match the names of the grouped CORBAServant elements. Consequently, the code statements contain variables that correspond to the properties of the grouped PSEs. Recall that these variables are given as point-cut arguments.

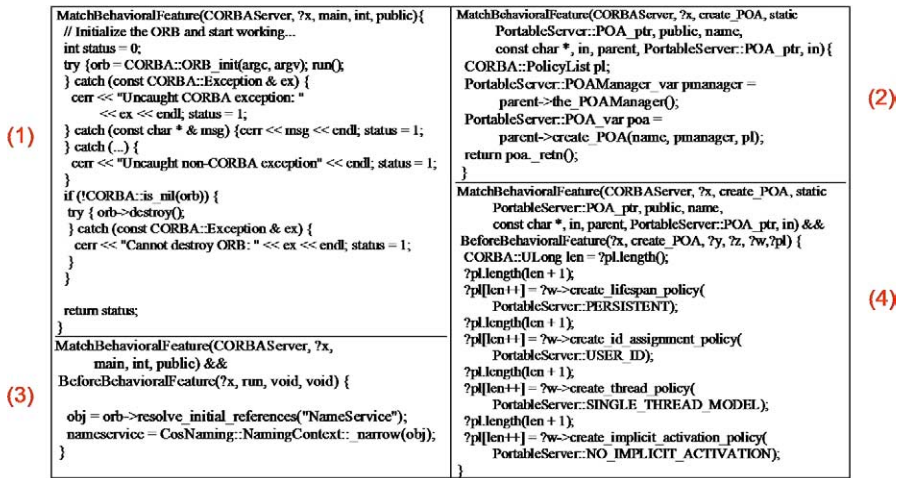
To generate a partial implementation of the skeleton code, we specify code statements that must be placed within the definitions of the generated skeleton code operations. The previous is achieved using the MatchBehavioralFeature construct, given in Table 3. We may, for instance, group all the run operations (Fig. 6(3)), introduced in the different CORBAServer elements, and specify a collection of code statements that should be placed within the definition of each one of them.

The existence of several realization dependencies between the different distribution transparencies that may be required has certain implications on the specification of code generation aspects. Specifically, the fact that a transparency *A* depends on a transparency *B* implies that the implementation code, specified in the code generation aspect of *A*, may need to be placed in specific points within the implementation code, specified in the code generation aspect of *B*. In particular, the code statements of *A* may need to be placed before or after a code statement of *B*. To group such kinds of points within a collection of code statements, we use the point-cut constructs that are given in Table 5. Note that it is obligatory to use these constructs in conjunction with the MatchBehavioralFeature construct, which determines the scope within which we place the code statements of *A* and *B* (see the examples given in Table 5).

Regarding our case study, in Fig. 8(1) and (2) we give two point-cuts, specified in the access transparency code generation aspect, for the generation of implementation code. The two point-cuts group respectively the main( ) and the create_POA( ) operations introduced in the CORBA-Server elements of the system according to the refinement point-cut that is given in Fig. 6(3). Then, they specify code that should be included in each one of the grouped operations. This particular code initializes several parts of the underlying CORBA platform (e.g. the ORB and the POA objects). The point-cut given in Fig. 8(3) is part of the location transparency code generation aspect. It groups all the main( ) operations of CORBAServer elements. Within these operations and right after the call to the operation that initializes the ORB object, it inserts a call to an operation that obtains a reference to the CORBA Naming Service, provided by the platform. Fig. 8(4) gives a point-cut included in the persistence transparency code generation aspect. Its purpose comprises grouping all the create_ POA( ) operations defined for the CORBAServer elements and creating within them a set of properties, which

Table 5
Point-Cuts for grouping points within a collection of code statements

Point-Cut: `BeforeStructuralFeature(Feature, FeatureType)`: specifies that a collection of code statements should be placed right before the declaration of a given structural feature Example: `MatchBehavioralFeature(CORBAServer, ?x, run, static void, public) && BeforeStructuralFeature(fooA, int){float fooB;}`: groups the run operations, defined for all the CORBAServer elements of a PSM model. Within the implementation of these operations and right before the declaration of the fooA variable another variable, fooB, is introduced
Point-Cut: `AfterStructuralFeature(Feature, FeatureType)`: specifies that a collection of code statements should be placed right after the declaration of a given structural feature Example: `MatchBehavioralFeature(CORBAServer, ?x, run, static void, public) && AfterStructuralFeature(fooA, int){fooB=f(fooA);}`: groups the run operations, defined for all the CORBAServer elements of a PSM model. Within the implementation of these operations and right after the declaration of the fooA variable an operation *f*( ) is called with fooA as an argument. The result of this operation is placed in fooB
Point-Cut: `AfterBehavioralFeature(Instance, Feature, FeatureRet, Arg1,..., ArgN)`: specifies that a collection of code statements should be placed right after a call to a behavioral feature, performed on a given PSE instance. FeatureRet is the variable name where the result of the call is stored (void should be used if there is no result). Arg1,..., ArgN are the actual parameters of the call. In place of FeatureRet and Arg1,..., ArgN we may use variables to communicate values between the called behavioral feature and the code statements that are introduced before it Example: `MatchBehavioralFeature(CORBAServer, ?x, run, static void, public) && AfterBehavioralFeature(?u, f, ?y, ?z) {g(?y);}`: groups the run operations, defined for all the CORBAServer elements of a PSM model. Within the implementation of these operations and right after a call to an operation *f*( ), it introduces a call to an operation *g*( ), which takes as input the return value of *f*( ) that was stored in ?y
Point-Cut: `BeforeBehavioralFeature(Instance, Feature, FeatureRet, Arg1,..., ArgN)`: specifies that a collection of code statements should be placed right before a call to a given behavioral feature Example: `MatchBehavioralFeature(CORBAServer, ?x, run, static void, public) && BeforeBehavioralFeature(?u, f, ?y, ?z) {?z=g();}`: groups the run operations, defined for all the CORBAServer elements of a PSM model. Within the implementation of these operations and right before a call to an operation *f*( ) that takes an argument ?z, it introduces a call to an operation *g*( ), whose return value is stored in ?z



(1)

```
MatchBehavioralFeature(CORBAServer, ?x, main, int, public){
 // Initialize the ORB and start working...
 int status = 0;
 try {orb = CORBA::ORB_init(argc, argv); run();
 } catch (const CORBA::Exception & ex) {
  cerr << "Uncaught CORBA exception: "
        << ex << endl; status = 1;
 } catch (const char * & msg) {cerr << msg << endl; status = 1;
 } catch (...) {
  cerr << "Uncaught non-CORBA exception" << endl; status = 1;
 }
 if (!CORBA::is_nil(orb)) {
  try { orb->destroy();
  } catch (const CORBA::Exception & ex) {
   cerr << "Cannot destroy ORB: " << ex << endl; status = 1;
  }
 }

 return status;
}
```

(2)

```
MatchBehavioralFeature(CORBAServer, ?x, create_POA, static
        PortableServer::POA_ptr, public, name,
        const char *, in, parent, PortableServer::POA_ptr, in){
 CORBA::PolicyList pl;
 PortableServer::POAManager_var pmanager =
        parent->the_POAManager();
 PortableServer::POA_var poa =
        parent->create_POA(name, pmanager, pl);
 return poa._retn();
}
```

(3)

```
MatchBehavioralFeature(CORBAServer, ?x,
        main, int, public) &&
BeforeBehavioralFeature(?x, run, void, void) {

 obj = orb->resolve_initial_references("NameService");
 nameservice = CosNaming::NamingContext::_narrow(obj);
}
```

(4)

```
MatchBehavioralFeature(CORBAServer, ?x, create_POA, static
        PortableServer::POA_ptr, public, name,
        const char *, in, parent, PortableServer::POA_ptr, in) &&
BeforeBehavioralFeature(?x, create_POA, ?y, ?z, ?w,?pl) {
 CORBA::ULong len = ?pl.length();
 ?pl.length(len + 1);
 ?pl[len++] = ?w->create_lifespan_policy(
        PortableServer::PERSISTENT);
 ?pl.length(len + 1);
 ?pl[len++] = ?w->create_id_assignment_policy(
        PortableServer::USER_ID);
 ?pl.length(len + 1);
 ?pl[len++] = ?w->create_thread_policy(
        PortableServer::SINGLE_THREAD_MODEL);
 ?pl.length(len + 1);
 ?pl[len++] = ?w->create_implicit_activation_policy(
        PortableServer::NO_IMPLICIT_ACTIVATION);
}
```

Fig. 8. CORBA-based code generation aspects for CCS.

are then used for the creation of persistent CORBA object references.

6.2. Code generation weaver

The behavior of the code generation weaver is similar to the one of the refinement weaver. More specifically, its input comprises a complete set of transparencies and a technology model, resulted from the application of the refinement method. The code generation weaver further uses the refinement aspects *RA* that were selected during the refinement method (Section 5), to retrieve their associated code generation aspects, $\mathrm{CA}=\{\mathrm{casp}_i|i=1,\ldots,N+K\}$. Then, it performs the following steps:

(1) It applies the code generation aspect for access transparency. As a result, it creates source code files for the direct mappings of EngElem PIEs. Similarly, it creates source code files for the direct mappings of EngCapsule and EngInterf PIEs. Then, it applies the advices of the aspect to generate corresponding skeleton and implementation code.
(2) The generated code is subsequently refined based on the code generation aspects of the other transparencies, which are applied in the order imposed by the transparency realization dependencies of Fig. 1.

Alike the refinement method, the code generation one facilitates the *upgrade* of the generated source code. The code generation weaver may accept as input a number of source code files, and a set of additional transparencies. Then, it can apply corresponding code generation aspects directly on the input source code files and produce their revised versions that further realize the additional distribution transparencies.

The application of the access transparency code generation aspect in CCS results in three C++ source code files for the Thermometer, the Thermostat and the Controller elements, respectively. Moreover, we have three CORBA IDL files for the interfaces provided by the aforementioned elements and two C++ files or the Location and the ControlLocation capsules. The effect of the location transparency code generation aspect is the modification of the Controller C++ file. The application of the persistence transparency code generation aspect gives three PSDL files and three C++ files, containing the specification and the realization of storage objects for the Thermometer, the Thermostat and the Controller elements.

7. Assessment

To assess the proposed methodology, we rely on experimental results from our case study scenario. However, CCS is a rather simple system consisting of two types of capsules and three types of engineering elements. To get a more pragmatic view of the proposed methodology, we additionally investigate its behavior in the development of more complex systems, comprising an increased number of different types of capsules and engineering elements. Specifically, we evaluate the *design effort* and the *implementation effort* required, respectively, for the refinement of engineering models and the development of platform specific code. These two issues reflect the gain from applying the proposed methodology, as the designers and the developers do not have to manually perform certain design and development tasks, which are delegated to the refinement and the code generation weavers. Evaluating the design and the implementation effort involves using the following generic framework:

(1) The design effort is measured per different distribution transparency *t*, in terms of the following metrics:
    - The cardinality, $\mathrm{DE}_{\mathrm{Str}_t}$, of the set of structural features and the cardinality, $\mathrm{DE}_{\mathrm{Beh}_t}$, of the set of behavioral features, added in the direct mappings of

capsules, engineering elements and interfaces [22].

- The cardinality, $\mathrm{DE}_{\mathrm{PSE}_{s_t}}$, of the set of additional PSEs, incorporated in the refined model.

  To complete the framework we may further assume metrics [22], concerning the relationships added in the refined model.

(2) The implementation effort is also measured per different distribution transparency t, in terms of the well-know Lines of Code (LOC) metric [23]. Specifically, we consider the amount of skeleton code, $\mathrm{IE}_{\mathrm{Skel}_t}$, and the amount of implementation code, $\mathrm{IE}_{\mathrm{Impl}_t}$, generated for t.

Our evaluation takes place as follows. First, we formalize the design effort and the implementation effort metrics as a function of the *size of the system*, measured in terms of the number of different types of capsules ($N_{\mathrm{EngCapsule}}$), engineering elements (N_{EngElem}) and interfaces ($N_{\mathrm{EngInterf}}$) that constitute it. Then, we calculate the values of the metrics for the specific case of CCS ($N_{\mathrm{EngCapsule}}=2$, $N_{\mathrm{EngElem}}=3$, $N_{\mathrm{EngInterf}}=3$). Finally, we investigate how the values of the metrics behave with respect to the increasing number of capsules and engineering elements.

7.1. Design effort

The realization of access transparency involves adding several structural and behavioral features in the direct mappings of the different types of capsules (three behavioral features), engineering elements (five behavioral and one structural) and interfaces (one behavioral), used in a system (Section 5.1). The direct mappings of engineering elements and interfaces further inherit from CORBA specific PSEs, added in the refined model. The total numbers of additional features and PSEs as a function of the size of the system are given below:

$$\begin{aligned}
\mathrm{DE}_{\mathrm{Beh}_{\mathrm{Access}}} &= 3 * N_{\mathrm{EngCapsule}} + 5 * N_{\mathrm{EngElem}} + N_{\mathrm{EngInterf}}\\
\mathrm{DE}_{\mathrm{Str}_{\mathrm{Access}}} &= N_{\mathrm{EngElem}}\\
\mathrm{DE}_{\mathrm{PSEs}_{\mathrm{Access}}} &= N_{\mathrm{EngElem}} + N_{\mathrm{EngInterf}}
\end{aligned}$$

Regarding location transparency, one additional PSE is added for every direct mapping of a capsule. Hence:

$$\mathrm{DE}_{\mathrm{PSEs}_{\mathrm{Location}}} = N_{\mathrm{EngCapsule}}$$

For persistence transparency, we also use additional PSEs for the connection of capsules with data-stores (three PSEs per capsule) and for the PSDL specification and implementation of storage objects (five PSEs per engineering element). The structural and behavioral features included in the PSDL specification and implementation elements of an engineering element $\mathrm{EngElem}_i$ correspond to the ones provided by the engineering element (denoted by $N_{\mathrm{Str}_{\mathrm{EngElem}_i}}$ and $N_{\mathrm{Beh}_{\mathrm{EngElem}_i}}$, respectively). Hence, the total numbers of additional features and PSEs as a function of the size of a system are:

$$\begin{aligned}
\mathrm{DE}_{\mathrm{Beh}_{\mathrm{Persistence}}} &= 2 * \textstyle\sum_{i=1,\ldots,N_{\mathrm{EngElem}}} N_{\mathrm{Beh}_{\mathrm{EngElem}_i}}\\
\mathrm{DE}_{\mathrm{Str}_{\mathrm{Persistence}}} &= 2 * \textstyle\sum_{i=1,\ldots,N_{\mathrm{EngElem}}} N_{\mathrm{Str}_{\mathrm{EngElem}_i}}\\
\mathrm{DE}_{\mathrm{PSEs}_{\mathrm{Persistence}}} &= 5 * N_{\mathrm{EngElem}} + 3 * N_{\mathrm{EngCapsule}}
\end{aligned}$$

In the CCS case study, the overall system comprises two different types of capsules and three different types of engineering elements and interfaces ($N_{\mathrm{EngCapsule}}=2$, $N_{\mathrm{EngElem}}=3$, $N_{\mathrm{EngInterf}}=3$). Consequently, in the technology model of the system we have 55 additional features and 28 additional PSEs for the realization of the required transparencies.

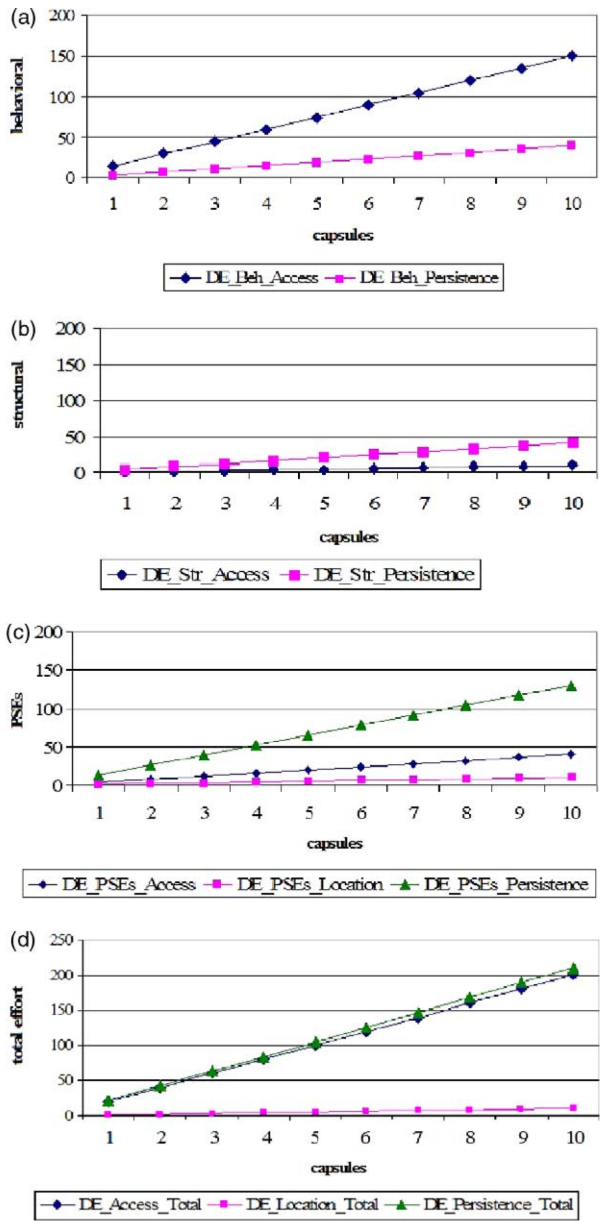


Fig. 9. Design effort. (a) Additional behavioral features. (b) Additional structural features. (c) Additional PSEs. (d) Overall design effort (sum of additional features and PSEs).

Fig. 9 shows the design effort required for the three transparencies in the case of more complex systems. Specifically, we assume that the number of different types of capsules linearly increases. Moreover, we assume that each type of capsule consists of two different types of engineering elements, providing one interface. Each element is characterized by 1 behavioral and 1 structural feature. Amongst the three transparencies, access is the most demanding regarding the additional behavioral features (Fig. 9(a)) that should be added. On the other hand, persistence is more demanding with respect to the number of structural features (Fig. 9(b)) and PSEs (Fig. 9(c)). Finally, the overall design effort required for access and persistence is similar (Fig. 9(d)).

7.2. Implementation effort

The realization of access transparency involves generating skeleton code for the direct mappings of engineering elements, capsules and interfaces (Section 6). Similarly, for persistence transparency, skeleton code is generated for the PSDL specification and implementation of storage objects. Specifically, for the declaration of skeleton C++ classes, IDL interfaces and PSDL specifications we count 2LOC. For the declaration of a structural or a behavioral feature we count 1LOC. Similarly, for the definition of a behavioral feature we count 2LOC. Consequently, the total amount of skeleton code as a function of the size of a system is:

$$
\begin{aligned}
\mathrm{IE}_{\mathrm{Skel}_{\mathrm{Access}}} &= 2 * (N_{\mathrm{EngElem}} + N_{\mathrm{EngInterf}} + N_{\mathrm{EngCapsule}}) \\
&\quad + \sum_{i=1,\ldots,N_{\mathrm{EngElem}}} \left(3 * N_{\mathrm{Beh}_{\mathrm{EngElem}_i}} + N_{\mathrm{Str}_{\mathrm{EngElem}_i}}\right) \\
&\quad + \sum_{i=1,\ldots,N_{\mathrm{EngInterf}}} \left(N_{\mathrm{Beh}_{\mathrm{EngInterf}_i}}\right) \\
&\quad + \sum_{i=1,\ldots,N_{\mathrm{EngCapsule}}} \left(3 * N_{\mathrm{Beh}_{\mathrm{EngCapsule}_i}} + N_{\mathrm{Str}_{\mathrm{EngCapsule}_i}}\right)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{IE}_{\mathrm{Skel}_{\mathrm{Persistence}}} &= 10 * N_{\mathrm{EngElem}} + \sum_{i=1,\ldots,N_{\mathrm{EngElem}}} \Big(N_{\mathrm{Beh}_{\mathrm{EngElem}_i}} \\
&\quad + N_{\mathrm{Str}_{\mathrm{EngElem}_i}}\Big) + \sum_{i=1,\ldots,N_{\mathrm{EngElem}}} \Big(3 * N_{\mathrm{Beh}_{\mathrm{EngElem}_i}} \\
&\quad + N_{\mathrm{Str}_{\mathrm{EngElem}_i}}\Big)
\end{aligned}
$$

Regarding the implementation code generated for access transparency we have 11LOC for the additional behavioral features, added in every engineering element. Moreover, we have 36LOC for the additional behavioral features, added in every engineering capsule (part of the code is given in Fig. 8(1 and 2)). For the case of location transparency, 2LOC of implementation code is generated for each capsule (Fig. 8(3)). Finally, for persistence transparency we have 22LOC of implementation code for every engineering capsule (part of the code is given in Fig. 8(4)). Hence, the total amount of implementation code as a function of the size of a system is:

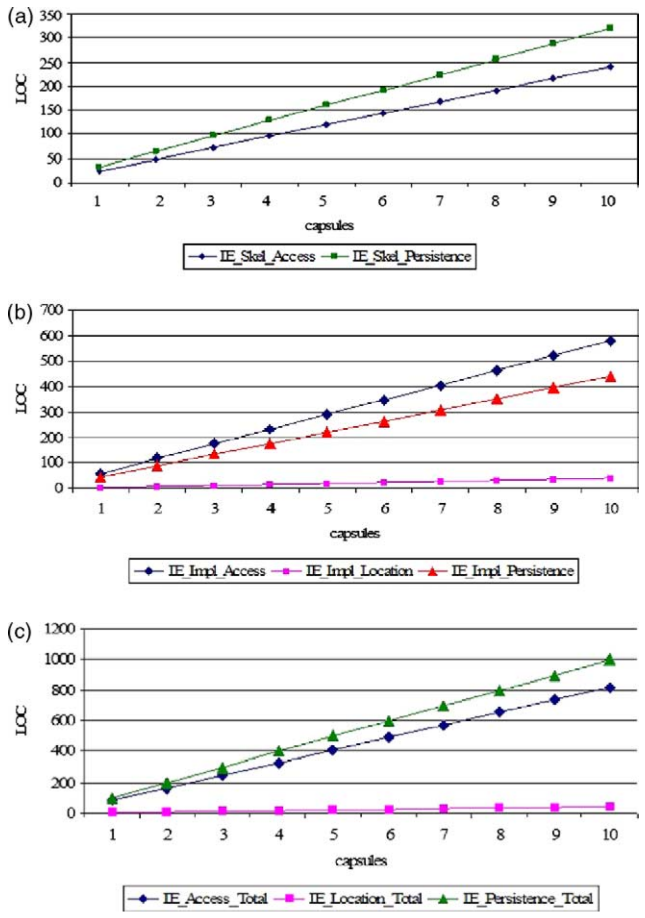


Fig. 10. Implementation effort. (a) Skeleton code. (b) Implementation code. (c) Overall implementation effort.

$$
\begin{aligned}
\mathrm{IE}_{\mathrm{Impl}_{\mathrm{Access}}} &= 11 * N_{\mathrm{EngElem}} + 36 * N_{\mathrm{EngCapsule}} \\
\mathrm{IE}_{\mathrm{Impl}_{\mathrm{Location}}} &= 2 * N_{\mathrm{EngCapsule}} \\
\mathrm{IE}_{\mathrm{Impl}_{\mathrm{Persistence}}} &= 22 * N_{\mathrm{EngCapsule}}
\end{aligned}
$$

In the CCS case study, the overall size of skeleton and implementation code is 425LOC. Fig. 10 shows the implementation effort required for the three transparencies in the case of more complex systems. As with the case of the design effort we assume that the number of different types of capsules linearly increases. The access and persistence transparencies are the most demanding ones with respect to both the skeleton (Fig. 10(a)) and the implementation code (Fig. 10(b)) that should be produced to achieve them. Fig. 10(c) shows how the overall implementation effort increases with the size of the system.

The overall conclusion from the results given in Figs. 9 and 10 is that the benefits from the application of the proposed methodology increase linearly with the size of the system that is under construction. It should be further highlighted that the results from our experiments can become even more encouraging in cases of systems consisting of more complex engineering elements and interfaces, comprising an increased number of structural and behavioral features.

8. Related work

Moriconi et al. [24] addressed the issue of architecture refinement. According to the authors, the architecture specification of a system comprises a description of the different kinds of architectural elements that realize the fundamental functionality of the system and a theory of properties, specifying the semantics of these elements. The properties are specified using formal notations like temporal logic, CSP, pi-calculus, etc. Refining an abstract architecture into a concrete one involves mapping abstract architectural elements into concrete ones. The particular set of mapping rules used is called a refinement pattern. The overall refinement procedure is faithful if: (1) every property in the theory of the abstract architecture is a logical consequence of properties that belong in the theory of the concrete architecture; (2) every property in the theory of the concrete architecture can be expanded into a theory model that proves a property of the abstract theory.

In the context of MDA, there have been several approaches that follow similar methodologies with the one proposed by Moriconi et al. More specifically, in [25] the authors discuss the use of patterns towards the transformation of UML models. Kafka [26], JaMDA [27], AndroMDA [28] and ArcStyler [29] are representative examples of tools that address the issue of model transformation and code generation. In [30], the authors use template forms of UML diagrams to produce CORBA-based PSMs from PIMs. Similarly, in [11,31], the authors use AOM to refine PIMs into PSMs that take into account security concerns. In all these approaches, the resulting PSMs and code mainly realize a single form of transparency. In order to realize more than one forms of transparency the developers should manually tune one or more transformation patterns to result in a single composed transformation pattern. In this particular point, the OptimalJ tool [32] appears more advanced compared to the rest of the tools we discussed. OptimalJ allows to transform PIMs into J2EE PSMs that realize access transparency. Moreover, it allows specifying aspects that can be used to automatically extend the code that is originally generated by the tool. These aspects may concern, for instance, other forms of transparency. However, there is still no systematic support regarding the rules that should govern the aspects' composition. Our approach specifically tackles this problem. Therefore, it can be used in conjunction with existing MDA tools to enhance their provided functionality. On the other hand, the functionality of existing MDA tools may serve for developing refinement and code generation weavers like the ones we propose.

The aspect composition problem is discussed in [33]. The solution proposed by the author consists of specifying aspects in relation to each other. In particular, to compose an aspect A with an aspect B, we have to specify in B point-cuts that group aspect statements of A. Within such a point-cut there exist advice statements that should be placed before or after the aspect statements of A. Then, the author proposes using an aspect preprocessor, which takes as input A and B and produces the composed aspect C. This particular approach is interesting. However, it renders difficult the upgrade of the generated models and code. Each time we need to add a distribution transparency requirement in a system, we have to re-compose all aspects and re-generate the system's enhanced models and code. The proposed methodology efficiently deals with the aforementioned issue. As we detailed in Sections 5 and 6, the idea is to define transparency aspects that can be applied on the *results* of other transparency aspects. Another important issue that is not taken into account in [33] is the scalability problem introduced when defining aspects in relation to each other. Dealing with this problem relates to the identification of realization dependencies between transparency related aspects. In our approach we cover this issue based on results that were further discussed in [9].

9. Conclusions

In this paper, we presented a systematic approach for the realization of distribution transparencies. More specifically:

- We proposed a UML-based representation for the specification of engineering models.
- We developed an AOM-based refinement method that facilitates the refinement of engineering models into technology models.
- We elaborated an AOP-based code generation method for the generation of platform specific code, from technology models, resulting from the refinement method.
- Finally, we assessed the benefits of the proposed methodology, based on a generic framework for the evaluation of the design and the implementation effort, required for the realization of distribution transparencies.

As discussed in Section 8, the proposed methodology can benefit from functionality provided by already existing MDA tools. Most of these tools provide APIs that allow to directly process a graphical UML model. However, using such APIs in the implementation of our approach implies placing strong dependencies between our prototype and the tools that provide these APIs. To avoid such dependencies, we have chosen to build our first prototype from scratch, while keeping it interoperable with existing MDA tools. Most of the existing MDA tools include add-ins enabling the generation of XMI textual specifications of UML graphical models. In consequence, we have also chosen

XMI as the standard input format for our refinement and code generation weavers. Currently, we experiment with the integration of our prototype with the Rational Rose tool. This choice is mainly motivated by our previous experience with this tool. However, we also intend to investigate other tools amongst the ones discussed in Section 8.

The approach we proposed here can be complemented with methods and tools for the automated quality analysis of technology models [34]. Another challenging problem that we investigate consists of reverse engineering platform independent models from platform specific code. Our interest to this problem originates from the fact that middleware platforms keep evolving. Nowadays, for instance, object-oriented middleware platforms start giving their place to component-based middleware platforms. Consequently, the provision of systematic methods and tools for reverse engineering PIMs shall facilitate the migration of systems' implementations that rely on 'legacy' middleware platforms to systems' implementations that are based on 'up-to-date' middleware platforms.

References

[1] P.A. Bernstein, Middleware: a model for distributed system services, Commununications of the ACM (CACM) 39 (2) (1996) 86–98.

[2] ISO/IEC. Open Distributed Processing Reference Model. Part 3: Architecture. Technical Report 10746-3, ISO/IEC, 1995.

[3] OMG. Common Object Request Broker Architecture (CORBA/IIOP) v3.0. Technical Report formal/02-12-02, OMG Document.

[4] Sun Microsystems, The Java 2 Enterprise Edition (J2EE) Specification v.1.4, Technical Report.

[5] Microsoft Corporation, COM+v1.5, Technical Report.

[6] OMG, Model Driven Architecture, Technical Report ormc/2001-07-01, OMG Document.

[7] D. Perry, A. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.

[8] OMG, UML Superstructure v.2.0, Technical Report ptc/03-08-02, OMG Document.

[9] A. Zarras, A comparison framework for middleware infrastructures, Journal of the Object Technology 3 (5) (2004) 100–123.

[10] J. Gray, T. Bapty, S. Neema, J. Tuck, Handling crosscutting constraints in domain-specific modeling, Communications of the ACM 44 (10) (2001) 87–93.

[11] I. Ray, R. France, N. Li, G. Georg, An aspect-based approach to modeling access control concerns, Information and Software Technology 46 (9) (2004) 575–587.

[12] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. Lopes, C. Maeda, A. Mendhekar, Aspect-oriented programming, ACM Computing Surveys 28 (4) (1996).

[13] T. Elrad, R.E. Filman, A. Bader, Aspect-oriented programming: introduction, Communications of the ACM 44 (10) (2001) 29–32.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, Getting started with ASPECTJ, Communications of the ACM 44 (10) (2001) 59–65.

[15] H. Ossher, P. Tarr, Hyper/J: multi-dimensional separation of concerns for Java Proceedings of the 22nd IEEE-ACM-SIGSOFT International Conference on Software Engineering (ICSE'00) 2000 pp. 734–737.

[16] M. Henning, S. Vinoski, Advanced CORBA Programming with C++, Addison Wesley, Reading, MA, USA, 1999.

[17] OMG, UML Profile for Schedulability Performance and Time, Technical Report ptc/03-03-02, OMG Document, 2002.

[18] OMG, UML Profile for CORBA v.1.0, Technical Report formal/02-04-01, OMG Document.

[19] OMG, UML Profile for Enterprise Distributed Object Computing (EDOC), Technical Report ptc/02-02-05, OMG Document.

[20] OMG, The CORBA Services Specification, Technical Report, OMG Document.

[21] N. Medvidovic, R. Taylor, A classification and comparison framework for software architecture description languages, IEEE Transactions on Software Engineering 26 (1) (2000) 70–93.

[22] S.R. Chindamber, C.F. Keremer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.

[23] B. Henderson-Sellers, Object-Oriented Metrics-Measures of Complexity, Prentice-Hall, New Jersey, USA, 1996.

[24] M. Moriconi, X. Quian, R.A. Riemenschneider, Correct architecture refinement, IEEE Transactions on Software Engineering 21 (4) (1995) 356–370.

[25] S. Sendal, W. Kozaczynski, Model transformation: the heart and soul of model-driven software development, IEEE Software 20 (5) (2004) 42–45.

[26] T. Weis, A. Ulbrich, K. Geihs, Model metamorphosis, IEEE Software 20 (5) (2004) 46–51.

[27] The JaMDA Project, Technical Report, http://jamda.sourceforge.net/.

[28] AndroMDA from UML to Deployable Components, Technical Report, http://www.andromda.org/pages/whatisit.html.

[29] ArcStyler 4.0: Product Background Information, Technical Report, Interactive Objects. http://www.iO-Software.com.

[30] Kamalakar, B., Ghosh, S., A middleware transparent approach for developing CORBA-based distributed applications, Technical Report CS04-104, Colorado State University, 2004.

[31] R. France, D-K. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, IEEE Transactions on Software Engineering 30 (3) (2004) 193–206.

[32] OptimalJ: How transformation patterns transform UML models into -high-quality J2EE applications, Technical Report, Compuware, http://www.compuware.com/products/optimalj/1794_ENG_HTML.htm.

[33] L. Bussard, Towards a pragrmatic composition model of CORBA services based on ASPECT/J Proceedings of the ECOOP Workshop on Aspects and Dimension of Concerns 2000.

[34] V. Issarny, C. Kloukinas, A. Zarras, Systematic aid for developing middleware architectures, Communications of the ACM (CACM) 45 (6) (2002) 53–58.