Journal of Universal Computer Science, vol. 14, no. 8 (2008), 1314-1343
submitted: 15/11/07, accepted: 25/3/08, appeared: 28/4/08 © J.UCS

CrossMDA: a Model-driven Approach for Aspect Management

Marcelo Pitanga Alves
(Departamento de Ciência da Computação, Núcleo de Computação Eletrônica, Federal University of Rio de Janeiro (UFRJ), P.O.BOX 2324, Rio de Janeiro – RJ - Brazil
mpitanga@gmail.com)

Paulo F. Pires
(DIMAp: Departamento de Informática e Matemática Aplicada - Federal University of Rio Grande do Norte (UFRN), Natal – RN – Brazil
paulo.pires@dimap.ufrn.br)

Flávia C. Delicato
(DIMAp: Departamento de Informática e Matemática Aplicada - Federal University of Rio Grande do Norte (UFRN), Natal – RN – Brazil
flavia.delicato@dimap.ufrn.br)

Maria Luiza M. Campos
(Departamento de Ciência da Computação, Núcleo de Computação Eletrônica, Federal University of Rio de Janeiro (UFRJ), P.O.BOX 2324, Rio de Janeiro – RJ – Brazil
mluiza@ufrj.br)

Abstract: Nowadays, the complexity of software applications has brought new challenges to developers, having to deal with a large number of computational requirements. Among these requirements, those known as crosscutting concerns transpass components boundaries, leading to maintainability and comprehension problems. This paper presents CrossMDA, a framework that encompasses a transformation process to integrate crosscutting concerns in model-oriented systems. It uses the concepts of horizontal separation of concerns from AOP to create independent business and aspect models, integrating those models through MDA transformations (vertical separation of concerns). CrossMDA comprises a development process, a set of services and support tools. The main advantages of this approach are to raise the abstraction level of aspect modeling, to promote the reuse of crosscutting concerns modeled as PIM elements, besides automating the process of mapping the relationship of crosscutting concerns and business models through the process of MDA transformations.

Keywords: Model Driven Architecture, Aspect Oriented Software Development, Crosscutting concerns, MDA Transformations
Categories: D.2.10, D.2.2, D.2.13, H.4.3

1 Introduction

The increasing complexity of current software applications, along with the emergence of new technologies and the demand of final users for a high quality in the delivered systems, require developers to deal with a growing set of software requirements. Among these requirements, computational requirements such as concurrency,

distribution, persistence, and fault recovery, affect a large number of components in a given system, that is, they *crosscut* the boundaries of such components. This crosscutting behavior leads to the *scattering* and *tangling* of software functionalities and, as a consequence, of the code that implements such functionalities. The code *scattering* and *tangling* hinder the comprehension, maintainability and evolution of the generated system [Tekinerdogan, 04].

Requirements that crosscut components, spreading over several different parts of a system instead of being encapsulated in a unique component, are known as *crosscutting concerns*. Such concerns typically crosscut system parts according to two different dimensions: horizontal and vertical. The horizontal dimension refers to concerns that crosscut system components within the same abstraction level of the system life cycle (analyses, design, and implementation). On the other hand, the vertical dimension refers to concerns that crosscut components spread over different levels of abstractions of the system life cycle. Since both dimensions of crosscutting behavior decrease the modularity of the system and compromise the reuse of parts, it is important to adopt principles and techniques to avoid such behavior [AOSD, 07] [MDD, 03].

In order to manage crosscutting behavior issues, thus promoting reusability, adaptability and modularity of the system, a possible approach is to employ the principle of *Separation of concerns*. Two important and complementary approaches to provide advanced separation of concerns are Model Driven Development (MDD) [MDD, 03] and Aspect Oriented System Development (AOSD) [AOSD, 07].

MDD is a software development approach where models are created before source code is written. A primary example of MDD is the Model Driven Architecture (MDA) [OMG-MDA, 06]. The Model Driven Architecture (MDA) is an OMG initiative for model driven development that proposes three different abstraction levels for system modeling: Computational Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). These models are mapped from one abstraction level to the other through the process of successive transformations, during which new elements are included in the model, and the abstraction level is decreased until reaching a level of platform dependency, meaning a model that is coupled to the specific target platform where the application is to be deployed.

MDA initiative naturally provides a way for vertical separation of concerns, since each model encompasses only the elements related to a given abstraction level. For instance, computational requirements are only included in the PSM model. However, the separation of concerns according to the horizontal dimension is not addressed in the MDA approach, that is, it lacks mechanisms for identifying and insulating crosscutting concerns inside each particular model.

Regarding the horizontal dimension, Kiczales et al. [Kiczales, 97] presented the Aspect Oriented Programming (AOP), which complements the Object Oriented Programming by offering a set of techniques that allow the appropriate encapsulation and insulation of crosscutting concerns in a new abstraction named *aspect*. Moreover, they proposed mechanisms for aspect composition (*weaving*) and reuse of the aspect code. The adoption of the aspect oriented approach promotes the horizontal separation of concerns. However, techniques used in the context of AOP concentrate in the system implementation phase. Therefore, such techniques are more suitable for

development processes in which the effort falls in producing software artifacts at the code level.

The horizontal separation of concerns at the modeling level is being tackled in the area of Aspect Oriented Modeling (AOM) [AOM, 06]. Works in AOM focuses on techniques for the identification, analyses, management and representation of crosscutting concerns in the modeling phase, by using UML extensions [Aldawud, 03] [Baniassad and Clarke, 04][Chavez, 04] [Suzuki and Yamamotto, 99] [Stein, 02] [Stein et. al, 02] [Tekinerdogan, 04]. However, the lack of suitable tools for modeling and managing the relationship among business elements and a particular crosscutting concern (weaving process) has been a hindrance in the wide-spread adoption of AOM concepts in the MDA approach. Such gap is being addressed in works that combine the concepts of the AOP area with MDA and propose the integration of crosscutting concerns in models by using MDA transformations [Chaves, 04] [Graziadei, 05][Reina and Torres, 05] [Simmonds et. al., 05] [Solberg et al., 05] [Wampler, 05]. Nevertheless, there are several open issues regarding the full combination of AOP and MDA.

Relevant open issues in the area of AOSD concern aspect reuse and composition. Several works report that the development based on aspects suffers from three drawbacks: limited reuse [Gybels and Brichau, 03], hard to predict behavior [McEachen and Alexander, 05], and difficult modular reasoning [Clifton and Leavens, 03] [Aldrich, 05]. These issues hinder the full adoption of an Aspect Oriented approach for software development.

The main motivation behind our work consists of finding a balance point between both AOSD and MDA approaches, aiming to fully exploit their advantages, as well as the synergy resulting from their integration. With this goal in mind, we propose a framework, named CrossMDA, which encompasses a transformation process as well as a set of services and associated support tools. Our approach aims at: (i) raising the abstraction level of aspect modeling through the use of PIM models representing crosscutting concerns independent on the business models; (ii) promoting the reuse of crosscutting concerns modeled as PIM elements; (iii) automating the process of mapping the relationship of crosscutting concerns and business models through the process of MDA transformation; (iv) promoting the reuse of artifacts of MDA transformations, and (v) promoting the reuse of PIM business models.

CrossMDA allows handling aspects at the modeling level and it provides mechanisms that enable the separation of concerns both over the horizontal dimension, among models of a same abstraction level, as well as the separation over the vertical dimension, among models from different abstraction levels. The separation of concerns over the horizontal dimension is achieved by adopting a process that models aspects independently from business elements at the PIM level. The PIM aspect model is an abstract representation of a particular crosscutting concern, allowing the hiding of implementation details from the business developer, thus raising the abstraction level of the modeling at the PIM level.

Regarding the vertical dimension, it is addressed by extending the MDA transformation process with an interactive phase, carried out by the developer, responsible for weaving the aspect and business models. The result of this model weaving process is the generation of a MDA transformation program, which corresponds to a formal specification of all the relationships among aspect and

business elements specified by the developer. The MDA transformation program is generated using a transformation language based on MOF-QVT (*Query, View, Transformation*) standard [OMG-QVT, 06].

This paper presents and describes CrossMDA, a framework to deal with the horizontal and vertical separation of concerns. As a proof of concept and aiming at showing the several steps comprising the use of CrossMDA, we also present a complete case study. The target application used on the case study was Health Watcher [Soares et al., 02], a typical Web-based information system, which has been already adopted in several works [Soares et al., 06] [Kulesza et. al., 06].

The remainder of this paper is organized as follows. Section 2 describes the CrossMDA framework, its operation and components. Section 3 presents a case study. Section 4 presents related works. In Section 5, conclusions and future directions of the work are depicted.

2 CrossMDA Framework

CrossMDA encompasses a process, a set of guidelines for modeling, and a set of services supporting the process, which are described in the next subsections.

2.1 CrossMDA Process

Figure 1 depicts the CrossMDA process through a UML Activity Diagram. Such process consists of several activities, organized in 4 phases: Phase 0 – Modeling, Phase 1 – Source Model Selection, Phase 2 – Mapping, and Phase 3–Model Weaving.

The first phase of the CrossMDA process (Phase 0) encompasses two different views of software artifacts modeling: (i) aspect modeling and (ii) business modeling. The aspect model is an abstract representation, that is, a platform-independent representation, in the MDA sense, of crosscutting concerns. Crosscutting concerns are modeled as classes decorated with the stereotype <<aspect>> [Stein, 02] and organized in packages. In CrossMDA, an aspect package is an entity that aggregates related aspects, that is, aspects that deal with the same category of requirements. For instance, a given package can contain several aspects related to authentication, another one related to logging, and so on. Similarly to the aspect model, the business model is also a platform-independent view, but of business process. However, the CrossMDA process does not impose any constraint in business modeling. Therefore, the business model can be composed of any valid UML element for modeling business entities and their relationships. The aspect and the business models can be developed independently by two different actors: the aspect architect and the business architect, respectively. The models built in this phase are stored in a repository for further use in the next phases.

The next phase (Phase 1) is carried out by a system architect in order to augment a given business model with the necessary crosscutting concerns to address the system requirements. This phase comprises two activities: (i) model selection and (ii) model loading. Model Selection consists of selecting aspect and business PIM source models that will be used in the transformation process. The Model Loading activity is in charge of loading and making the selected models persistent in a metadata repository.

After the source model loading, the system architect starts the mapping phase (Phase 2) which consists in specifying the relationships among aspects and elements of the business model. This phase starts with the selection of crosscutting concerns packages that are relevant for the application domain being modeled. Next, an iterative process of defining the relationships among aspects and business elements begins, in which the system architect selects the aspects that should be applied to a (set) of business elements. When this iterative process ends, CrossMDA generates a set of mappings that represents the pointcut and inter-type definitions generated according to the relationships among aspects and business elements specified by the system architect.

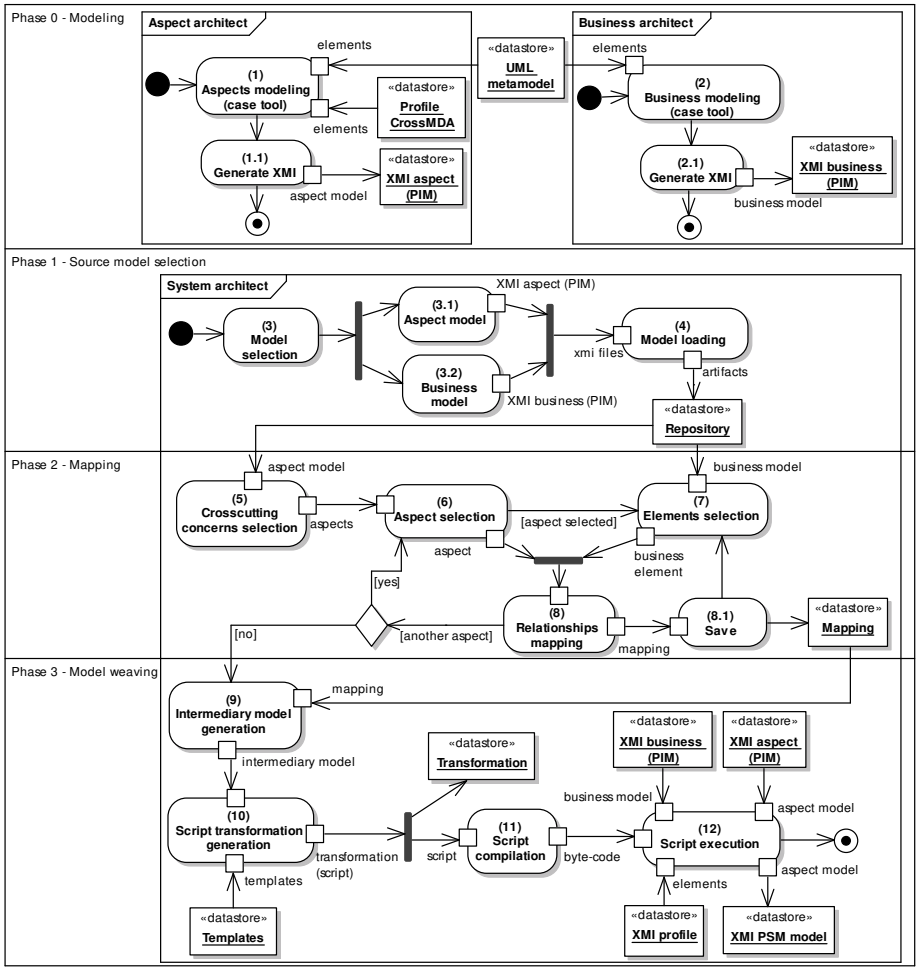

Figure 1: CrossMDA Process

The next phase (Phase 3) is responsible for generating the PSM model, which represents a refinement of the source business model added with aspect elements. The phase is composed of four activities organized within two sub-processes: (i) model weaving and (ii) model generation. The model weaving sub-process starts with the generation of an intermediate model from the set of mappings produced as outcome of Phase 2. This intermediate model is a representation that contains each aspect class instance and its respective dependencies to the business model elements. Next, the intermediate model is transformed into a formal specification through the generation of a transformation program based on OMG MOF QVT specification [OMG-QVT, 06]. This transformation program is then compiled and executed in the model generation sub-process generating as outcome the PSM model.

In the next sections we detail the guidelines that should be followed during Phase 0 of CrossMDA process.

2.2 Aspect Modeling in CrossMDA

This section details the guidelines for aspect modeling at PIM level in CrossMDA. According to these guidelines, in order to build an aspect PIM the system architecture should: (i) organize aspect in UML packages, (ii) follow the CrossMDA aspect categorizations, and (iii) decorate aspect classes according to the CrossMDA profile (Section 2.2.1). Since the organization in packages groups related crosscutting concerns, it facilitates the choice of those aspects to be used in a given application, constraining the amount of aspects presented to the architect during the model weaving process. Regarding the adopted categorization for aspects, CrossMDA supports the representation of abstract and non-abstract aspects. Abstract aspects are meant to allow aspect reuse in different application domains.

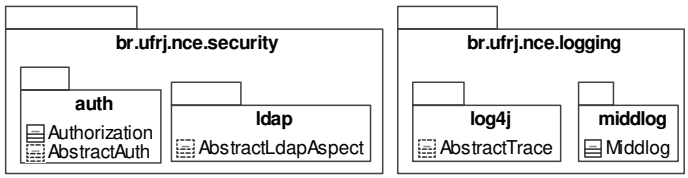

Figure 2: PIM aspect organized in crosscutting concerns packages

Figure 2 illustrates an aspect model following the described guidelines. In this example, the package named *br.ufrj.nce.security.auth* contains authentication related aspects. Figure 3 is a bird eye of a class that represents the *AbstractAuth* abstract aspect inside of the *auth* package. An abstract aspect can have a pointcut defined as abstract and it can either have or not an advice associated with this abstract pointcut. Since abstract pointcuts have no knowledge of the join points that can be affected by them, they are used in CrossMDA as a mechanism that allows aspect reuse in different scenarios. In the example, the depicted abstract aspect has the pointcut *authOperations* defined as abstract, which has an associated advice *adv_auth*. Since the pointcut *authOperations* has no associated join points, it can be (re)used in different applications. In CrossMDA the binding between an abstract pointcut is defined by the system architect during the mapping phase (Section 2).

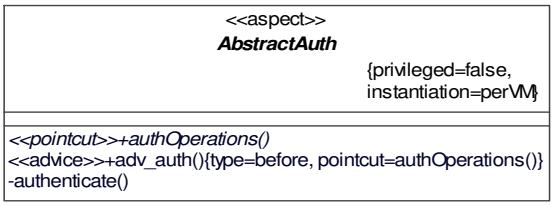

Figure 3: Class representing an authentication abstract aspect of the PIM model (adapted from Stein[Stein, 02] proposal)

The aspect PIM can be developed by the aspect architect from scratch or a previously built model can be reused, assuming that it has been developed following the presented guidelines. The next section describes the last CrossMDA guideline, which refers to the UML profile to be used in aspect modeling.

2.2.1 CrossMDA Profile

This section describes the UML profile created for modeling aspects used in the PIM model as well as aspects generated as output of the PSM transformation process. The CrossMDA UML profile is based on [Stein, 02] and [Camargo and Masiero, 04]. In spite of some works criticizing Stein's notation as being too close to AspectJ syntax and therefore generating an aspect model not fully platform agnostic, we have adopted an approach based on such notation for two reasons. First, currently there is no standard notation for representing aspects at the PIM level. Second, AspectJ is a widely used aspect language nowadays, containing all the most relevant aspect constructs. Moreover, since our approach is based on MDA metamodeling, CrossMDA is able to incorporate a different aspect model without affecting its architecture. It is worthwhile mentioning that the current CrossMDA aspect metamodel is expressive enough to represent all the main constructs encompassed in the existent aspect languages.

Table 1 shows the stereotypes defined in the CrossMDA profile along with the corresponding UML base classes and the respective tags.

Stereotype	**UML Base Class**	**TagDefinition**
aspect[1]	Class	instantiation, privileged
pointcut[1]	Operation	base
advice[1]	Operation	type, pointcut
crosscut[1]	Dependency	-
introduction[2]	Association	attribute, method
parents_extends[2]	Operation	pattern, type
parents_implements[2]	Operation	pattern, type

Table 1: CrossMDA profile stereotypes definition

[1] The stereotype semantics is based on Stein work [Stein, 02].

[2] The stereotype semantics is based on Camargo and Masieiro work [Camargo and Masiero, 04].

The stereotype *aspect* is used to identify a class as an aspect. It requires the presence of 2 tags: (i) *instantiation* and (ii) *privileged*. The tag *instantiation* specifies how an aspect is to be instantiated in the aspect scope. Aspects may be instantiated on a per object basis, a per control flow basis, or once for the global environment. Having a distinct aspect instance for each object or for each control flow, the aspect state may differ for each object or control flow, respectively. The possible values for this tag are: per object (perTHIS or perTARGET); per control flow (perCFLOW or perCFLOWBELOW); per virtual machine (perVM). The tag *privileged* defines if the aspect can access members of its base class and its possible values are of type Boolean.

The stereotype *pointcut* is used to identify aspect methods with pointcut semantics and it requires the tag *base*, which defines the rules for executing advices, i.e., the set of join points for which an aspect is to be instantiated. The possible values are of type *LinkSetExpression* [Stein, 02]. The stereotype *advice* is used to identify aspect methods with advice semantics. It requires 2 tags: (i) *type*, to identify the advice type (after, before, after returning, after throwing, around); and (ii) *pointcut*, to assign a method defined as pointcut to an advice. The stereotype *crosscut* defines dependencies among aspects as well as among classes and aspects in the class diagram. The stereotype *introduction* is used to inject attributes and/or methods in the target entity. It requires 2 tags: (i) *attribute*, to identify the attributes to be injected; and (ii) *methods*, to identify the methods to be injected.

Both stereotypes *parents_extends* and *parents_implements* are declarations that cut across classes and their hierarchies. The stereotype *parents_extends* is used to define an inheritance relationship between two classes or interfaces. It requires two tags: (i) *pattern*, to identify the name of children classes or interfaces; and (ii) *type*, to identify the name of the class or interface to be extended. The stereotype *parents_implements* is used to define an *implements* operation of an interface. It also requires two tags: (i) *pattern*, to identify the names of classes that implement an interface; and (ii) *type*, to define the name of the interface that is to be implemented.

2.3 CrossMDA Services

This section describes CrossMDA services, which provide the foundation to perform the activities encompassing the process offered by the framework. The provided services are: (i) model persistence; (ii) element mapping; (iii) model weaving and; (iv) model transformation.

2.3.1 Model Persistence Service

This service is responsible for implementing the basic operations to load and store UML models as well as operations for browsing, fetching and creating new elements in the current model. This service uses the NetBeans Metadata Repository [NetBeans-MDR, 07] for managing model elements. Such choice was based mainly on the fact that this repository is a popular and open implementation of the OMG MOF (Meta Object Facility) pattern [OMG-MOF, 06]. This repository is handled by the Model Persistence Service through one class that implements the *IRepository* interface (Figure 4).

```
public interface IRepository  {
      public org.omg.uml.UmlPackage getUmlPackage();
      public Object getRepository();
      public  void  loadModel(String[]  fileXmi,  String
searchRef) throws Exception;
      …
```

Figure 4: Interface for a class for repository handling

2.3.2 Element Mapping Service

The Element Mapping Service supports two types of mapping, the pointcuts and the inter-types mappings.

Pointcuts Mapping

This type of mapping provides mechanisms for managing the mapping of the relationships among aspects and business elements. In CrossMDA, this type of mapping follows the pointcut specification pattern of AOP approach and AspecJ language [AspectJ, 06][Laddad, 03] (Figure 5). This approach was chosen based on the wide use of such specification pattern by many aspect-oriented languages and frameworks [JBossAOP, 06].

```
[visibility] [abstract] key-word name([args]) : pointcut designator (join point )
```

Figure 5: Pointcut definition

A pointcut specification is accomplished by using a primitive pointcut designator and a join point signature. A primitive pointcut designator or PCD provides a definition around join points, which designate pre-defined sets of join points from the join point model. For instance, the PCD *call* pick out all join points that correspond to a call to an existent method or constructor. The CrossMDA mapping process supports several types of PCDs [Laddad, 03, 77pp]. PCDs can also be combined through logic operators, which allow generating more complex pointcut specifications.

Aiming to facilitate mapping among business elements and their related crosscutting concerns, CrossMDA provides the system architect with both a process and a service to store the mapping elements. The process encompasses the following steps: (i) selecting aspects; (ii) selecting a predefined pointcut, which can be either abstract or non-abstract; (iii) selecting one or more elements from the business model (classes, interfaces, methods, attributes, or packages); (iv) indicating the type of the pointcut; and (v) indicating the type of the advice (after, before, after returning, after throwing or around) [Laddad, 03, 81pp]. The steps of the process are quite repetitive and steps (iii) and (iv) have a higher degree of complexity since they can be combined in different ways by using logic operators.

```
select aspect from PIM model,   ASPECT_SELECTED <- aspect
if ASPECT_SELECTED is abstract then
  ASPECT_SELECTED.name<- Question("Aspect name for implementation:")
Endif
RULES.initialize()
select pointcut (PC) from ASPECT_SELECTED
    PC <- selected pointcut
    Repeat
      if Question("Would like to include precedence operator?") = YES then
          select Precedence operator (OPD),   OPD <- ( "(" || ")" )
          if  OPD = ")"  then
              if (hasBracketOpen() = YES) and
                (RULES.prior = PCD or RULES.prior = ")") then
                    RULES.add(OPD)
              else  Message("Inválid Operator")  endif
          else  RULES.add(OPD)  endif
      endif
      if ( RULES.prior = PCD or RULES. Prior = ")" ) then
          Add Logical Operator (LO),  LO <- (OU || E)
          RULES.add(LO)
      Endif
      select joinpoint (JP) from Business PIM model
         JP <- (package || interface || class || method || attribute)
         if (JP = class) or (JP = interface)  then
              if Question("Would like to specialize?") = YES then
                 SPECIALIZE <- YES    else    SPECIALIZE <- NOT  endif
         Endif
      select pointcut designator (PCD)
         PCD <- (execution || call || initialization || get || set || this || within ||
                   withincode || target || args || cflow || handler )
        if Question("Would like a negation operator "(!)" for the PCD?") = YES
        then  PCD.operatorNot <- "!"   else   PCD.operatorNot <- ""   endif
      M<-Create_Mapping (ASPECT_SELECTED, PC, PCD, JP, SPECIALIZE)
      RULES.add(M)
      if Question("Would like to continue?") = YES then
           V <- To verify precedence operators
           if V = OK then   break    else   Message("Has brackets open")  endif
      endif
    end repeat
```

Figure 6: Pseudocode of the pointcut mapping process.

The pseudo code in Figure 6 details the execution steps of the mapping process. Initially, the system architect selects an aspect from the PIM model and, whenever an abstract aspect is selected, the architect should provide a name to be used in the implementation of the concrete aspect. Next, the pointcut mapping begins, which

follows a rule composed of pointcut designators (PCD) that can be combined by using logical operator and brackets. The PCD is associated with a joinpoint selected by the architect from the business model. After the selection of all element of the rule (pcd, joinpoint, operator logical and/or brackets), the final mapping rule is generated.

The outcome of the mapping process is a set of mapping rules, which are made persistent by the mapping service, following a simplified aspect metamodel (Figure 7), which represents a pointcut specification tailored to be managed by a MDA transformation.

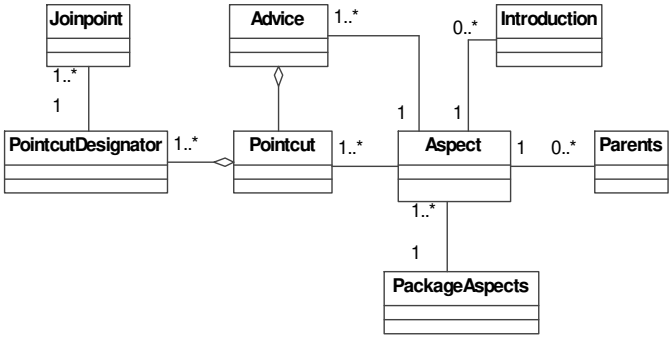

Figure 7: Simplified Aspect Metamodel

Inter-Type Mapping

Intertype declarations provide a way to express crosscutting concerns affecting the structure of modules in a program. They are declarations related to a program structure, allowing declaring in one place members or parents of another class, typically to combine all the code related to a concern in one single aspect. For instance, one aspect may be used to add new attributes and/or methods to a given class. Another type of declaration allows that a class/interface extends a new super-class/interface or implements a new interface.

Possible types of intertype declarations are: (i) inclusion of members (methods, constructors, attributes) for types, including other aspects; (ii) inclusion of concrete implementation for interfaces; (iii) declaration of new extensions or implementations for types; (iv) declaration of aspect precedence order; (v) declaration of customized errors or warnings messages; and (vi) conversion of checked exceptions to unchecked exceptions [Winck and Junior, 06, 109-110pp] [Gradecki and Lesiecki, 03, 187pp]. Intertype declarations are declared as static crosscutting concerns since they affect the program structure as a whole.

The CrossMDA process provides three different ways for the system architect to accomplish intertype mappings: (i) inclusion of members for particular types; (ii) inclusion of concrete implementations for interfaces; (iii) declaration of new extensions for types. Aiming to facilitate such mappings, CrossMDA provides the architect with a process and a service to store mappings elements. The pseudo code in Figure 8 describes the provided process.

```
select aspect from PIM model,  ASPECT_SELECTED <- aspect
if ASPECT_SELECTED is abstract then
  ASPECT_SELECTED.name <- Question("Aspect name for implementation:")
endif
RULES.initialize()
select inter-type (IT),  IT <- ( introduction || declare parents )

if IT = introduction then
  STEREOTYPE <- <<introduction>>
  select member from  aspect class,   MEMBER<-(attribute || method || constructor)
  select target class, TARGET <- selected class
M<-Create_Introduction_Mapping(ASPECT_SELECTED,STEREOTYPE,MEMBER,TARGET)
  RULES.add(M)
endif
if IT = declare parents then
  select predefined declare_parents method from ASPECT_SELECTED
  METHOD_PARENTS <- declare_parents selected
  if METHOD_PARENTS = NULL then
     select implementation type (TIMPL),  TIMPL <- (implements || extends)
     if TIMPL = implements  then
        STEREOTYPE <- <<parents_implements>>
        select  interface(business || aspects) base of  implementation
        TYPE <- selected interface
        select classes from business model that will implement TYPE
        PATTERN <- selected elements
     else   STEREOTYPE <- <<parents_extends>>
            select  element (class || interface) base of extension
            TYPE <-  selected element
            select elements from business model that will extend TYPE
            PATTERN <- selected elements
     endif
  else
     select elements from METHOD_PARENTS
     STEREOTYPE <- METHOD_PARENTS.stereotype
     TYPE <- METHOD_PARENTS.taggedValues("type")
     if STEREOTYPE = <<parents_implements>> then
        select classes from business model,   PATTERN <- selected elements
     else  select elements(classes||interface) from business model that will extend
              TYPE
           PATTERN <- selected elements
     endif
  endif
 M<-Create_Parents_Mapping(ASPECT_SELECTED,METHOD_PARENTS,STEREOTYPE,
TYPE,PATTERN)
  RULES.add(M)
endif
```

Figure 8: Pseudocode of the inter-type mapping process.

The pseudo code in Figure 8 details the execution steps of the inter-type mapping process. Initially, the system architect selects an aspect from the PIM model and, whenever an abstract aspect is selected, the architect should provide a name to be used in the implementation of the concrete aspect. Next, the architect selects the desired inter-type mapping (*introduction* and/or *declare parents*). The *introduction* mapping is generated based on the selection of a target class and of a member (attribute, method or constructor) from the aspect class. The *declare parents* mapping has the following steps. First, the algorithm checks if a predefined *declare parents* method exists in the selected aspect; if this is true the architect selects the elements from the business model for mapping. Otherwise, the architect selects an implementation of inter-type and the elements for mapping.

The outcome of the inter-type process is stored according to the mapping model presented in Figure 7. In this model, classes *Introduction* and *Parents* are used to store each instance of an inter-type mapping.

2.3.3 Model Weaving Service

The model weaving (composition) mechanism consists of generating the instances of the selected aspects and their associations with the respective business elements, thus integrating both the aspect and the business models. The CrossMDA weaving service is provided by a class, named *weaver*, which is in charge of generating the transformation program. The transformation program is a formal specification that implements a model weaving of business elements and their related aspects according to the outcomes of the element mapping service (Section 2.3.2).

The model weaving starts when the *weaver* receives a set of mapping instances and generates the intermediary model, which is an internal representation, used only inside the CrossMDA framework. Relying on this intermediary model, the generation of the transformation program is initiated. The transformation program is generated through the use of code template files (Figure 9), which are joined together, meaning that several templates are integrated in a single template. In the resulting template file, tags are replaced by information on aspects originating from the mapping set, in order to generate the final code of the transformation program. For instance, during the program generation, the *tag* <ASPECT_NAME> is replaced by the name of an aspect. *Templates* are coded in the ATLAS Transformation Language (ATL) [Jouault and Kurtev, 05], a transformation language proposed by ATLAS group (INRIA & LINA, Nantes University) that is aligned to the OMG MOF QVT specification [OMG-QVT, 06].

The reuse of transformation artifacts is an important feature implemented in CrossMDA. The underpinning of such reuse is the use of template files, which allows that a same template is used to generate different transformation programs according to the application requirements.

The transformation program generated by the Weaver has as its target metamodel a PSM aspect model based on the aspect model proposed in [Stein, 02], in which UML classes marked with the *aspect* stereotype represent aspects. Pointcuts are represented by methods of an *aspect* class marked with the *pointcut* stereotype while advices are represented by methods with the *advice* stereotype and tags (tagged values) *type* and *pointcut,* identifying the type of the advice (after, before, after returning, after throwing or around) and the pointcuts, respectively. Specifications of

pointcut, PCDs and the join point signatures are mapped as tags of methods marked as pointcut in the PSM.

```
lazy rule newClass {
from className : String, namespace : String
to t : UML!Class (
  name <-className,
  namespace <- thisModule.getPackage(namespace),
  stereotype <- thisModule.getStereotype('aspect')),
  ...}
lazy rule newOperation {
from           c:UML!Class,           s:UML!Operation,
stereotypeName:String
to t : UML!Operation (
  owner <- c, visibility <- #vk_public, name <- s.name,
  stereotype<-thisModule.getStereotype(stereotypeName),
 ... }...
thisModule.umlClass<-
if thisModule.classExists('<ASPECT_NAME_IMPL>','aspect')
then thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')
else thisModule.newClass('<ASPECT_NAME_IMPL>',
 '<ASPECT_OWNER>') endif;
thisModule.umlOperation <-
  if thisModule.operationExists('<ASPECT_NAME_IMPL>',
      '<POINTCUT_NAME>','pointcut')
  then  thisModule.getOperation('<ASPECT_NAME_IMPL>',
      '<POINTCUT_NAME>','pointcut')
  else thisModule.newOperation( thisModule.umlClass,
      thisModule.getOperation('<ASPECT_NAME>',
      '<POINTCUT_NAME>','pointcut'), 'pointcut')
  endif;
if thisModule.taggedValueExists(thisModule.umlOperation
 ,'base') then true
else thisModule.newTaggedValue(thisModule.umlOperation,
'base',thisModule.toString('',
   Sequence{<POINTCUT_VALUE>})) endif;
```

Figure 9: Example of ATL template code for class and method creation

2.3.4 Model transformation

The activity of model transformation starts when a transformation program, generated by the weaver, is to be compiled and executed. CrossMDA provides a service (Figure 10) to compile and execute the transformation program, generating the PSM model that combines the source business PIM with the aspects specified by the system architect.

```
public interface IScriptCompiler {
      public void compile (String fileName);
}
public interface IScriptExecute {
      public int parseArgs(String[] args);
      public   String[]   setParameters(String   script,
String in, String out, String libs);
      public void run(); }
```

Figure 10: Interface for services of compilation and execution of the transformation motor

The Model Transformation service encapsulates an open source ATL transformation engine (*ATL engine*) [ATL, 07]. The transformation engine is a framework that includes a virtual machine (*ATLvm*) and a compiler. It also provides a set of classes written in Java programming language that offers, among other services: (i) *parser*, to perform syntax analysis of the transformation program; (ii) *compiler*, to generate the byte-code; and (iii) *loader*, responsible for loading and executing the transformation program. Since the Model Transformation service acts as a wrapper, any other transformation engine can be used. However, in this case it is necessary to rewrite the templates using the syntax of such engine (e.g. OMG Q.V.T).

3 Case Study

This section presents a case study of the application of CrossMDA process and services to an information system. The case study was carried out using Health Watcher (HW) [Soares e. al., 02], a typical Web-based application that manages health-related complaints in order to improve the quality of services provided by Health Institutions. HW was chosen since it is a testbed for the AOSD community. Furthermore, its requirements are easy to understand, encompassing both crosscutting as well as non-crosscutting concerns [Soares et al., 06][Kulesza et. al., 06].

The development process offered by CrossMDA allows the designer to build his/her own aspect models and integrate aspects with elements of business models. In this case study, we intend to demonstrate CrossMDA features by implementing the persistence crosscutting concern in HW classes as an aspect. We have organized the case study according to the phases of CrossMDA process (Section 2.1).

3.1 Phase 0 – Modeling

The tasks of building and maintaining the business and aspect models can be performed using any existent UML modeling tool. The resultant models must be exported as XMI files in order to be imported in CrossMDA framework. Currently, CrossMDA supports only XMI 1.4 format. This constraint is due to the use of NetBeans-MDR in the implementation of CrossMDA Model Persistence Service.

3.1.1 The HW PIM

The model presented in Figure 11 is a partial implementation of the class model for the HW case study that specifically addresses the use cases related to the register of complaints. Such model will be used as one of the source models (business PIM) in the first phase of the process.

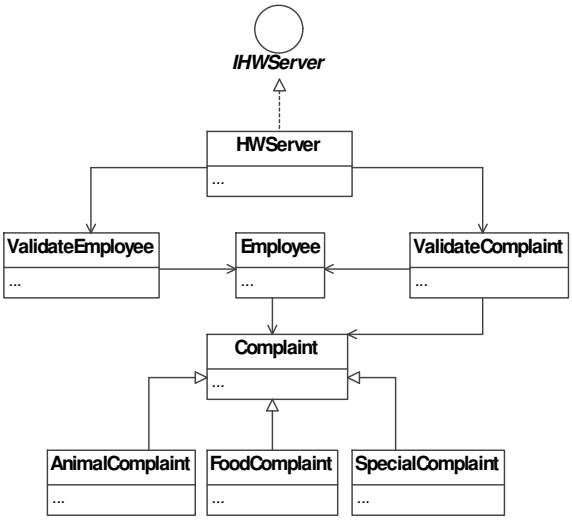

Figure 11: Fragment of the HW class diagram

The class *HWServer* provides methods that perform operations for registering, updating and querying complaints for a system user. The methods of this class are used by the HW presentation layer to process the requests issued by users. Each operation is validated by classes *ValidateEmployee* and *ValidateComplaint* that communicate with classes *Employee* and *Complaint*. It is worthwhile noting that this model is a business model free from any crosscutting behavior. Whenever using CrossMDA with legacy models containing crosscutting concerns entangled with business elements, a refactoring is needed in order to clean the business models from these concerns.

3.1.2 The Aspect PIM

The crosscutting concerns that comprise the aspect PIM model are modeled as UML abstract classes using the CrossMDA profile (see Section 2.2.1). These artifacts are abstract representations of crosscutting concerns and they are independent from any aspect-oriented platform or language. Moreover, since the information represented in the aspect PIM is not tied to any business model, it can be reused in different application scenarios. The building of the aspect model will be demonstrated through the modeling of a persistence aspect, responsible for performing data persistence in the database.

The persistence aspect (Figure 12) is an abstract class with stereotype <<aspect>> that represents the persistence crosscutting concern. It implements two abstract pointcuts that will be configured in the implementation of the aspect, during its weaving with the business model. The pointcuts are: (i) *startMechanismPC()* and; (ii) *persistElementsPC()*.

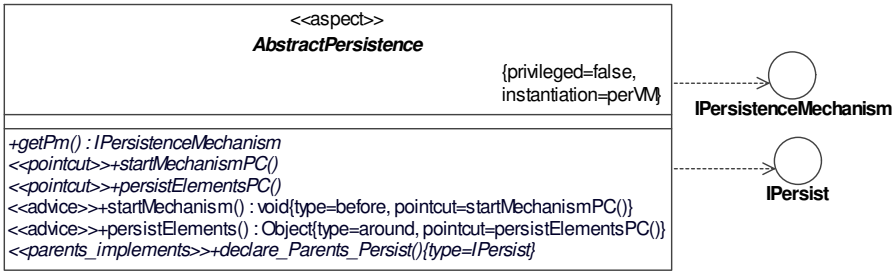

Figure 12: Abstract aspect for persistence.

The pointcut *startMechanismPC()* indicates the point in the application execution where the aspect should initialize the persistence mechanism. The pointcut *persistElementsPC()* is used to get a reference to the business elements that are to be managed by the persistence mechanism. In order to implement this feature, the aspect modifies the business classes to implement the interface *IPersist*. This modification is indicated by the inter-type declaration defined by the abstract method *declare_Parents_Persist*(). It is important to notice that abstract inter-type declarations are not part of AOP semantics. However, we have added such construct in order to increase the degree of reuse of aspect declarations at PIM level. Moreover, this declaration is to be discarded during the PSM transformation process.

The aspect *AbstractPersistence* also includes the abstract method *getPM()*. The *getPM()* is an utility method used to get a reference of the persistence mechanism. This method provides flexibility once the aspect can implement different persistence mechanisms.

In order to illustrate the next phases, we choose the HW use cases related to complaint register. For such use cases, a data persistence mechanism is an important requirement that crosscut several parts of the system. Thus, a persistence aspect can be used to avoid the scattering and tangling of persistence related code in the HW system and to ensure the transparency of the persistence layer in the event of changing the adopted strategy for persistence.

3.2 Phase 1 – Source model selection

As described previously, the system architect is responsible for combining the business and the aspect models generating the PSM model. To implement such composition process (aspect + business), the system architect initially performs the selection and load of both business and aspect models in the repository. In the case study, the system architect should select the aspect PIM model containing the

persistence aspect definition as well as the HW model. Once these models are loaded, the system architect starts Phase 2, which comprises the task of creating the relationships among the elements of the aspect model and the business model.

3.3 Phase 2 - Mapping

The Phase 2 (Figure 1) is responsible for specifying the relationships among the aspects and the elements of the business model. This phase enables the reuse of artifacts of the aspect model since elements of the aspect PIM can be reused in different mappings with different business models depending on the application specific requirements.

The persistence aspect defined in Section 3.1.2 will be used to link the business code to the persistence mechanism. Once the persistence aspect was selected from the set of available aspects in the aspect model loaded during Phase 1, the point in the application code where the aspect is to be activated must be identified. This information will be used in the definition of the pointcut in the aspect. In the description of HW requirements (Section 3.1.1) the class server (*HWServer*) is the first to be accessed by the application control layer and it is responsible for the creation of the class instances that validate the information (*ValidateEmployee* and *ValidateComplaint*) and that interact with the business classes representing persistent objects (*Employee* and *Complaint*). Therefore, the aspect should be activated when an instance of class *HWServer* is created. The next sections present the activities of pointcut and inter-type mappings for the HW system.

3.3.1 Pointcut mapping

When specifying the persistence aspect (Section 3.1.2), two abstract pointcuts were specified: (i) *startMechanismPC()* and; (ii) *persistElementsPC()*. In order to map these pointcuts the process provided by the Element Mapping service (Section 2.3.2) should be used. Table 2 shows the information that should be provided by the system architect when performing these mappings.

Pointcut (PC)	**PCD**	**Joinpoint (JP)**	**Logical Operator**
startMechanismPC()	call	HWServer constructor	
persistElementsPC()	call	Specialized Complaint constructor	
			OR
	call	Employee constructor	
			AND
	withincode	all method of insertion of ValidateComplaint	
			OR
	withincode	all method of insertion of ValidateEmployee	

Table 2: Mapping of pointcut for AbstractPersistence

The outcome of this mapping process can be visualized as follows:

- pointcut startMechanismPC(): call(HWServer.new(..))
- pointcut persistElementsPC():

```
(call(Complaint+.new(..)) || call(Employee.new(..))) &&
 (withincode(* ValidateComplaint.addAnimalComplaint(..)) ||
 withincode(* ValidateComplaint.addFoodComplaint(..)) ||
 withincode(* ValidateComplaint.addSpecialComplaint(..)) ||
 withincode(* ValidateEmployee.addEmployee(..)))
```

3.3.2 Inter-Type mapping

The persistence aspect in this case study uses the inter-type mapping to access the instances of the persistence elements of the business model. In the aspect specification (Section 3.1.2) the abstract method (*declare_Parents_Persist*) identifies an inter-type mapping. This definition indicates that the concrete persistence aspect should set this inter-type. To map this inter-type the process provided by the Element Mapping service (section 2.3.2) should be used. Table 3 shows the information that should be provided by the system architect when performing such mapping.

Type	Method	Stereotype	TYPE	PATTERN
Dp	*declare_Parents_Persist()*	parents_implements	IPersist	Complaint, Employee

Legend: dp – declare parents

Table 3: Inter-Type mapping for AbstractPersistence

The outcome of this mapping process can be visualized as follows:

- declare_Parents_Persist : Complaint, Employee implements IPersist

3.3.3 Phase 3 – Model Weaving

The first step of Phase 3 (Figure 1) is the generation of the intermediary model based on the set of mapping resulting from Phase 2, followed by the generation of the transformation program. During the generation of the transformation program, the weaver loads the templates and starts a parser in the template code searching for tags. Each *tag* is then replaced by the appropriate value according to the outcome of Phase 2.

In our example, the first information retrieved from the intermediary model is the persistence aspect. Therefore, the first artifact to be generated is responsible for creating an instance of an aspect class. The generation of this artifact is done by loading the code template depicted in Figure 13.

```
thisModule.umlClass<- if
 thisModule.classExists('<ASPECT_NAME_IMPL>','aspect')
then thisModule.getClass('<ASPECT_NAME_IMPL>','aspect')
else thisModule.newClass('<ASPECT_NAME_IMPL>',
        '<ASPECT_OWNER>') endif;
```

Figure 13: Template for generating an aspect class instance

Once this template is loaded the weaver starts parsing the *tags* and replacing each *tag* by the corresponding values from the mapping. Table 4 presents the resulting replacement according to our example.

TagName	**Description**	**Value**
<ASPECT_NAME>	Aspect instance name	AbstractPersistence
<ASPECT_NAME_IMPL>	Aspect implementation name	HWPersistence
<ASPECT_OWNER>	Aspect namespace	br.ufrj.nce.persistence

Table 4: Tags and their corresponding values for the aspect HWPersistence

Upon processing the template, the weaver generates a transformation artifact (Figure 14) for creating the instance of the persistence aspect class to be added to the main program.

```
thisModule.umlClass<-
 if thisModule.classExists('HWPersistence','aspect')
 then thisModule.getClass('HWPersistence','aspect')
 else thisModule.newClass('HWPersistence',
           'br.ufrj.nce.persistence') endif;
```

Figure 14: Transformation code for generating an instance of aspect HWPersistence

Since the persistence aspect is abstract, the weaver must use other template (Figure 15) to generate an instance of the generalization relationship.

```
if thisModule.generalizationExists(thisModule.getClass(
    '<ASPECT_NAME>','aspect'),
    thisModule.getClass('<ASPECT_NAME_IMPL>','aspect'))
then ''
else thisModule.newGeneralization(thisModule.getClass(
      '<ASPECT_NAME>','aspect'),
thisModule.getClass('<ASPECT_NAME_IMPL>','aspect'))
endif;
```

Figure 15: Template for generating an instance of generalization

Once the template is loaded, the weaver again performs the operations of parsing and tags replacement. After processing the template, the weaver generates the artifacts to create an instance of UML generalization element between the concrete aspect and the abstract aspect (Figure 16).

```
if thisModule.generalizationExists(thisModule.getClass(
    'AbstractPersistence','aspect'),
   thisModule.getClass('HWPersistence','aspect'))
then ''
else thisModule.newGeneralization(thisModule.getClass(
      'AbstractPersistence','aspect'),
      thisModule.getClass('HWPersistence','aspect'))
endif;
```

Figure 16: Artifact for generating an instance of UML generalization

The next information retrieved from the intermediary model is the pointcut mapping. To map each pointcut, the weaver loads the pointcut template (Figure 17) and performs a new parser and *tag* replacement.

```
thisModule.umlOperation <-
  if thisModule.operationExists('<ASPECT_NAME_IMPL>',
      '<POINTCUT_NAME>','pointcut')
  then thisModule.getOperation('<ASPECT_NAME_IMPL>',
            '<POINTCUT_NAME>','pointcut')
  else thisModule.newOperation( thisModule.umlClass,
       thisModule.getOperation('<ASPECT_NAME>',
          '<POINTCUT_NAME>','pointcut'), 'pointcut')
  endif;

if thisModule.taggedValueExists(
       thisModule.umlOperation,'base')
then true
else
 thisModule.newTaggedValue(thisModule.umlOperation,
       'base',thisModule.toString('',
           Sequence{<POINTCUT_VALUE>}))
endif;
```

Figure 17: Template for generating pointcut instance

For the persistence aspect two pointcut were mapped: *startMechanismPC* and; (ii) *persistElementsPC*. The pointcut *startMechanismPC* (Table 5) was chosen to illustrate the steps for generating the transformation artifacts for pointcuts.

TagName	Description	Value
<ASPECT_NAME>	Aspect instance name	AbstractPersistence
<ASPECT_NAME_IMPL>	Aspect implementation name	HWPersistence
<POINTCUT_VALUE_ID>	Pointcut identifier	PointcutValueID_1
<POINTCUT_NAME>	Pointcut instance name	startMechanismPC
<ADVICE_TYPE>	Advice type	-
<POINTCUT_VALUE>	Pointcut designator (PCD) and joinpoint	call(HWServer.new (..))

Table 5: Values selected for mapping the pointcut startMechanismPC

After processing the template shown in Figure 17, the weaver generates a transformation artifact for creating an instance of a method marked as pointcut that will be added in the main program. Figure 18 presents the resulting transformation program.

```
thisModule.umlOperation <-
  if thisModule.operationExists('HWPersistence',
      'startMechanismPC','pointcut')
  then thisModule.getOperation('HWPersistence',
           'startMechanismPC','pointcut')
  else thisModule.newOperation( thisModule.umlClass,
      thisModule.getOperation('AbstractPersistence',
         'startMechanismPC','pointcut'), 'pointcut')
  endif;
if
thisModule.taggedValueExists(thisModule.umlOperation,
     'base') then true
else thisModule.newTaggedValue(thisModule.umlOperation,
       'base',thisModule.toString('',
         Sequence{call(HWServer.new(..))}))
endif;
```

Figure 18: Transformation artifact for generating pointcut startMechanismPC

The next information retrieved by the weaver from the intermediary model is the inter-type mapping of *declare_parents* type. The weaver loads the template of inter-type (Figure 19) and performs a new parse and *tags* replacement.

```
thisModule.umlOperationDeclare <-
 if thisModule.operationExists('<ASPECT_NAME_IMPL>',
    '<PARENT_VALUE_ID>','<PARENTS_STEREOTYPE>') then
       thisModule.getOperation('<ASPECT_NAME_IMPL>',
       '<PARENT_VALUE_ID>', '<PARENTS_STEREOTYPE>')
 else
thisModule.newOperationDeclare(thisModule.umlClass,
        '<PARENT_VALUE_ID>','<PARENTS_STEREOTYPE>')
 endif;
thisModule.declareType  <- Sequence{<PARENT_TYPE>};
thisModule.declarePattern <- Sequence{<PARENT_PATTERN>};
if thisModule.taggedValueExists(
   thisModule.umlOperationDeclare, 'type')
then true
else thisModule.newTaggedValue(
      thisModule.umlOperationDeclare,'type',
         thisModule.declareType) endif;
if thisModule.taggedValueExists(
   thisModule.umlOperationDeclare, 'pattern') then true
else thisModule.newTaggedValue(
        thisModule.umlOperationDeclare,'pattern',
            thisModule.declarePattern)
endif;
```

Figure 19: Template for generating an instance of inter-type method

Table 7 presents the information used for *tag* replacement related to the inter-type mapping.

TagName	**Description**	**Value**
<PARENT_VALUE_ID>	Inter-type identifier	declare_Parents_Persist
<ASPECT_NAME_IMPL>	Implementation name	HWPersistence
<PARENTS_STEREOTYPE>	Inter-Type stereotype for method definition	parents_implements
PARENT_TYPE	Interface name that will be implements.	IPersist
PARENT_PATTERN	Class name to implements the interface	Complaint, Employee

Table 7: Values of inter-type for HWPersistence

After processing the template of Figure 19, the weaver generates the transformation artifact for creating an instance for a method (*declare_Parents_Persist*) that represents the inter-type operation that will be added in the main program. Figure 20 presents resulting transformation artifact.

```
thisModule.umlOperationDeclare <-
 if thisModule.operationExists('HWPersistence',
    'declare_Parents_Persist','parents_implements')
then thisModule.getOperation('HWPersistence',
        'declare_Parents_Persist','parents_implements')
else
thisModule.newOperationDeclare(thisModule.umlClass,
      'declare_Parents_Persist','parents_implements')
endif;
thisModule.declareType <- Sequence{'IPersist'};
thisModule.declarePattern<-Sequence{'Complaint','Employee'};
if thisModule.taggedValueExists(
   thisModule.umlOperationDeclare, 'type')
then true
else thisModule.newTaggedValue(
       thisModule.umlOperationDeclare,'type',
         thisModule.declareType) endif;
if thisModule.taggedValueExists(
    thisModule.umlOperationDeclare,'pattern') then true
else thisModule.newTaggedValue(
     thisModule.umlOperationDeclare,'pattern',
       thisModule.declarePattern) endif;
```

Figure 20: Artifact for generating instance of inter-type method declare_Parents_Persist

The next information to be processed is the dependency mapping. There are two types of dependencies to be processed: crosscut and inter-type dependencies. The same steps for generating the previous transformation artifacts are performed for generating dependency artifacts. Figure 21 shows the used template.

```
if thisModule.dependencyExists(thisModule.getClass(
   '<ASPECT_NAME_IMPL>','aspect'),thisModule.getClass(
   '<DEPENDENCY_NAME>','<DEPENDENCY_STEREOTYPE>'),'',
        'crosscut')
then ''
else thisModule.newDependency(thisModule.getClass(
     '<ASPECT_NAME_IMPL>','aspect'), thisModule.getClass(
 '<DEPENDENCY_NAME>','<DEPENDENCY_STEREOTYPE>'),'crosscut')
endif;
```

Figure 21: Template for generating an instance of dependency relationship

In our example, the persistence aspect depends on the following elements: (i) *HWServer*; (ii) *Complaint;* (iii) *Employee;* (iv) *ValidateEmployee;* and (v) *ValidateComplaint*. For each element, the weaver performs the generation of a dependency in the model. Figure 22 presents the transformation artifact for generating the dependency between the class *HWServer* and the persistence aspect.

```
if thisModule.dependencyExists(thisModule.getClass(
'HWPersistence','aspect'),thisModule.getClass(
'HWServer',''),'','crosscut')
then ''
else thisModule.newDependency(thisModule.getClass(
'HWPersistence','aspect'), thisModule.getClass(
'HWServer',''),'crosscut') endif;
```

Figure 22: Artifact for generating an instance of dependency relationship

Upon generating the artifact of dependency the weaver tries to retrieve a new information from the intermediary model and, according to the mapping realized in Phase 2, there is no more mapping to be processed. Thus, the weaver starts the process of merging the several transformation artifacts in a single transformation program. After saving the final transformation program, it is compiled and executed by the model transformation service. This is the final step of the CrossMDA process, which generates the PSM shown in Figure 23.

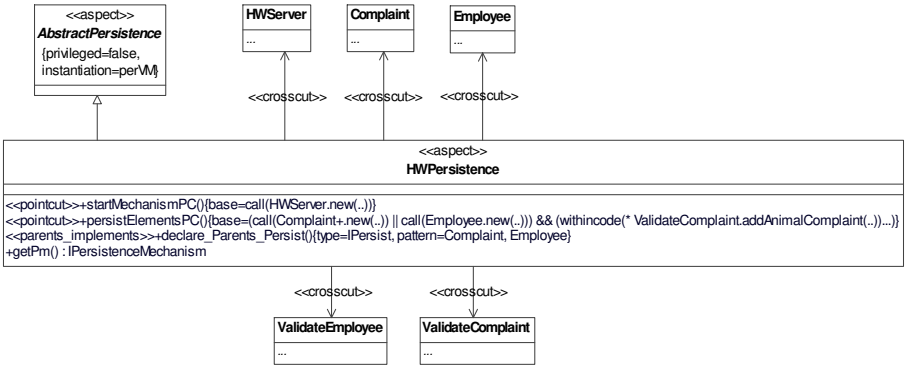

Figure 23: PSM model for persistence aspect

4 Related Work

There is ongoing research in the area of Aspect Oriented Software Development (AOSD) aiming to integrate AOP concepts in the modeling of object oriented systems. Such works use UML extension mechanisms to include new modeling elements that represent AOP concepts. In [Aldawud, 03] the authors propose the creation of an UML profile that provides developers a way for visually representing AOP artifacts, thus creating an aspectual metamodel. Other similar approach, named

Aspect-Oriented Design Model (AODM), is presented [Stein, 02], in which UML elements are extended to represent semantics of the AspectJ language elements [AspectJ, 06]. Research on both AOSD and AODM areas have the advantage of using only the UML extensibility mechanisms, thus being easily integrated to existent UML modeling tools. CrossMDA relates to such approaches since it adopts UML profiles for defining aspectual elements to compose the generated PSM models.

Chavez [Chavez, 04] proposes integrating AOP concepts in a unified framework, thus creating an *Aspect Theory*. A language, *aSideML*, and a metamodel, *aSide*, were built on such Aspect Theory. ASideML is a language for modeling aspect oriented systems while aSide defines the semantics of both structural and behavioral models represented in aSideML. The work in [Chavez, 04] represents a significant advance since it defines a broad model that formalizes the semantics of elements related to the aspect oriented modeling by using UML. As *aSideML* is based on a specific metamodel, the built of specialized tools to support this new metamodel is needed. Once such tools become available, CrossMDA could include transformers to generate models that comply with *aSideML.* Therefore, our approach is complementary to the work in [Chavez, 04].

A crucial property addressed in AODM is obliviousness [Filman and Friedman, 05], which states that base code should not to be explicitly prepared in order to be affected by aspects. However, assuring such property rises problems regarding software evolvability. Since pointcut definitions strongly rely on the structure of the base program, whenever the base program evolves a maintenance effort is needed to update all pointcuts of each aspect related to the evolved base code. This problem has been coined the fragile pointcut problem [Koppen and Stoerzer, 04][Kellens et al., 06]. Sullivan [Sullivan et al., 05] proposes constraining the obliviousness property in order to improve software maintainability and evolvability in AOSD by applying an approach based on design rules [Balswin and Clark, 00]. The idea in this work is to decouple base and aspect design by defining interfaces between them thus constraining their subsequent development. According to the authors [Sullivan et al., 05]: "*Design rules dictates how base code creates join points and how aspects use them to ensure that specific join points are exposed in a way that enables the integration of separately implemented aspect modules*". Since the process proposed in CrossMDA does not enforce any specific rule for business modeling (it only requires this model to be free of crosscutting concerns), the degree of obliviousness to be achieved is a system architect decision. Therefore, approaches such as the one proposed by [Sullivan et al., 05] can be integrated in the CrossMDA process to model business objects without any further modification.

In the MDA research area works concentrate on building transformation models to facilitate the integration of aspects with business models (known as *primary models*). In [Chaves, 04] the authors present a set of aspect oriented UML extensions named *Libra* which enables the specification of both structural and behavioral models. The original class model is enhanced with the capability of representing aspects along with their respective relationship to the primary model. In order to define behavior, an action language is provided which is based on both XML and UML action semantics, and includes reflexive capabilities. *Libra* uses the MDA transformation approach for combining the aspect model with the primary model elements. In spite of fact that this work indicates the feasibility of combining AOP

and MDA approaches to augment the task of aspect integration, it neither proposes a systematic and formal way of realizing such weaving nor addresses important issues as the specification and management of composition models. Both of these issues are addressed in CrossMDA and the formalization of the weaving process is accomplished by a transformation program written in ATL [ATL, 07].

The work in [Reina and Torres, 05][Simmonds et al., 05] uses QVT language [OMG-QVT, 06] to accomplish model transformations. In [Reina and Torres, 05] the authors use MDA transformations to weave AspectJ aspects and basic elements in the PSM level before code generation. CrossMDA adopts a similar approach but works with models at a higher level of abstraction (PIM level), providing a more powerful solution in terms of aspect reuse and alignment to the MDA approach. In [Simmonds et al., 05] a framework is presented that performs transformations of aspect oriented models from PIM to PSM models. Such framework takes as input a primary model and a set of generic aspect models which are specified as UML interaction diagrams. The composition of the new model as well as the bindings between primary and aspect models are realized through QVT transformations based on metamodels. Such QVT transformations are specified by the system designer. CrossMDA approach follows the same pattern of separating the input models (primary or business related model and aspect model). However, this work neither provides a full process for aspect modeling and integration nor tools to support such process. Moreover, since in CrossMDA individual aspects are modeled as UML classes [Stein, 02] its process is aligning with existent transformation tools that adopt a model-text approach and, as a consequence, the process can seamlessly proceed until the code generation.

5 Conclusion

In this paper, we presented CrossMDA, a framework that leverages the management of crosscutting concerns in system development through a model centric approach. A key idea of our approach is the employment of model-based transformations to perform the weaving among aspects and their related business elements. Performing the weaving process through model-based transformations allows the development of completely independent business and aspect models. Three important advantages arose from this approach. First, since crosscutting concerns are considered as first-class elements represented at a high level of abstraction (PIM level), they can be easily reused across different application scenarios. Second, since business PIM models are completely free of computational requirements details (actually, they do not even need to be decorated with stereotypes indicating the need of such requirements), the development of these models is simplified. Moreover, a same business model can be reused in different scenarios by simply applying to it a transformation containing the aspects and related mappings according to the requirements of the target scenario. Third, by providing a structured way of representing and storing the mapping between aspect and business elements, CrossMDA facilitates the system maintenance thus being a step forward towards a solution to allow the evolution of both models without mutual interference.

CrossMDA further improves the reuse by using code templates based on ATL. This approach, besides improving the degree of transformation reuse, facilitates the evolution of existent transformations as well as allows the generation of

transformation programs for different language syntaxes, like QVT or MWDL [Milewski and Roberts, 05]. Since CrossMDA generates PSMs according to current MDA standards (XMI), such models can be integrated in any modeling or MDA tool for further processing, such as model-to-text transformation for source code generation.

In order to validate the ideas behind CrossMDA, we have developed a Java prototype that includes the tools for automating all the activities encompassed by the CrossMDA process. This prototype is public available in [CrossMDA, 07].

As future work we intend to investigate how CrossMDA mechanisms handle system evolution. In order to tackle this issue we are researching in two directions. First, we are investigating how to integrate the design rules approach [Sullivan et al., 05] with the CrossMDA process. Second, we are analyzing how CrossMDA mapping process can contribute to solve problems that rise from the independence between models, such as the pointcut fragility [Kellens et al., 06].

References

[Aldawud, 03] Aldawud, O., Elrad, T. and Bader, A. UML Profile for Aspect-Oriented Software Development. In: Third Workshop on Aspect-Oriented Modeling with UML, AOSD'03. Boston, Massachussets, March, 2003.

[Aldrich, 05] Aldrich, J. Open Modules: Modular Reasoning about Advice. In: Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP), Glasgow, UK, 2005, pp. 144-168.

[AOM, 06] AOM. Aspect-Oriented Modeling Workshop. Available at: http://www.aspect-modeling.org. Last access: March, 2006.

[AOSD, 07] AOSD. Aspect-Oriented Software Development. Available at: http://aosd.net. Last access: January, 2007.

[AspectJ, 06] AspectJ, a Java implementation of AOP. Available at: http://www.eclipse.org/aspectj. Last access: April, 2006.

[ATL, 07] ATL Home Page. Available at: http://www.eclipse.org/m2m/atl/. Last access: January, 2007.

[Balswin and Clark, 00] Baldwin, C. Y. and Clark, K. B. Design Rules: The Power of Modularity. MIT Press, Cambridge, MA, 2000.

[Baniassad and Clarke, 04] Baniassad, E. and Clarke, S. Theme: An Approach for Aspect-Oriented Analysis and Design. In: Proceedings of the 26th International Conference on Software Engineering (ICSE), Edinburgh, Scotland, May, 2004, pp. 158-167.

[Camargo and Masiero, 04] Camargo, V.V., Masiero, P.C. UML-AOD - Um Perfil UML para o Projeto de Sistemas Orientados a Aspectos. Technical Report of ICMC-USP, 21 pp, 2004.

[Chaves, 04] Chaves, R. Aspectos e MDA Criando modelos executáveis baseados em aspectos. 2004. 79p. Master thesis, Federal University of Santa Catarina, Florianópolis, Santa Catarina, Brazil.

[Chavez, 04] Chavez, C. F. G. A Model-Driven Approach for Aspect-Oriented Design. Rio de Janeiro, 2004. 304p. Phd Thesis. Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-RIO), Rio de Janeiro, Brazil.

[Clifton and Leavens, 03] Clifton, C. and Leavens, G.T. Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. In: Software-engineering Properties of Languages for Aspect Technologies (SPLAT), a workshop to be held in conjunction with the 2nd international Conference on Aspect-Oriented Software Development (AOSD'03), Boston, Massachusetts, USA, March, 2003.

[CrossMDA, 07] CrossMDA Home Page. http://labdist.dimap.ufrn.br/projetos/crossmda.

[Filman and Friedman, 05] Filman, R. E. and Friedman, D. P. Aspect-oriented programming is quantification and obliviousness. In: Aspect-Oriented Software Development, Addison-Wesley, 2005, pp. 21-35.

[Gybels and Brichau, 03] Gybels, K. and Brichau, J. Arranging Language Features for More Robust Pattern-based crosscuts. In: Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03), Boston, Massachusetts, March, 2003, pp. 60-69.

[Gradecki and Lesiecki, 03] Gradecki, D. J. and Lesiecki, N. Mastering AspectJ: Aspect-Oriented Programming in Java**.** Wiley Publishing Inc. ISBN: 978-0-471-43104-6, Paperback, 456 pp, March, 2003.

[Graziadei, 05] Graziadei, T. R. Aspect-Oriented Model Weaver, 2005. 127p. Master thesis, Fachhochschule Vorarlberg, Dornbirn, Austria.

[Kellens et al., 06] Kellens, A., Mens, K., Brichau, J. and Gybels, K. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In: Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06), Nantes, France, July, 2006.

[Kiczales, 97] Kiczales, G. et al. Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997. Springer-Verlarg LNCS 1241.

[Koppen and Stoerzer, 04] Koppen, C. and Stoerzer, M. Pcdiff: Attacking the fragile pointcut problem. In: First European Interactive Workshop on Aspects in Software (EIWAS), 2004.

[Kulesza et. al., 06] Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., Staa, A., Lucena, C.: Quantifying the Effects of AOP: A Maitenance Study. In: Proceedings of 9th Intl. Conference on Software Maintenance (ICSM'06), Philadelphia, USA, September, 2006, pp. 223-233.

[JBossAOP, 06] JBossAOP. Framework for Organizing Cross Cutting Concerns. Available at: http://labs.jboss.com/portal/jbossaop/index.html. Last access: May, 2006.

[Jouault and Kurtev, 05] Jouault, F. and Kurtev, I. Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005. Montego Bay, Jamaica, 2005, Springer-Verlarg LNCS, 3844, pp.128-138.

[Laddad, 03] Laddad, R.. AspectJ in Action, Pratical Aspect-Oriented Programming. Manning Publications CO, 2003, ISBN 1930110936.

[McEachen and Alexander, 05] McEachen, M. and Alexander, R.T. Distributing Classes with Woven Concerns–An Exploration of Potential Fault Scenarios. In: Proceedings of the 4th international conference on Aspect-oriented software development (AOSD), Chicago, Illinois, 2005, pp.192-200.

[MDD, 03] IEEE Software. Special issue on Model-Driven Development. Vol. 20, number 5. September/October 2003.

[NetBeans-MDR, 07] NetBeans-MDR. Metadata Repository (2007). Available at: http://mdr.netbeans.org. Last access: January, 2007.

[OMG-MDA, 06] OMG MDA Guide version 1.0.1. Formal Doc.: 03-06-01. Available at: http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf. Last access: March, 2006.

[OMG-MOF, 06]. OMG Meta-Object Facility (MOF). Formal Doc.: 2002-04-03. Available at: http://www.omg.org/technology/documents/formal/mof.htm. Last access: 11/04/2006.

[OMG-QVT, 06] OMG MOF QVT. Available at: http://www.omg.org/cgi-bin/doc?ptc/2005-11-01. Last access: November/2006.

[Tekinerdogan, 04] Tekinerdogan, B., Moreira, A., Araújo, J. and Clements, P. Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design. In: Workshop Proceedings. University of Twente, TR-CTIT-04-44, October, 2004, 119 pp.

[Reina and Torres, 05] Reina, A.M. and Torres, J. Weaving AspectJ Aspects by means of transformations. In: First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the 19th European Conference on Object-Oriented Programming (ECOOP 2005), Glasgow, Scotland, 2005.

[Simmonds et. al., 05] Simmonds D., Solberg A., Reddy R., France R., Ghosh, S. An Aspect Oriented Model Driven Framework. In: Ninth IEEE International Enterprise Computing Conference (EDOC'05), Enschede, Netherlands, 19-23 September, 2005, pp. 119-130.

[Soares et al., 02] Soares, S.; Laureano, E. and Borba, P. Implementing distribution and persistence aspects with AspectJ. In: 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'2002, Seattle, USA, 2002, pp. 174–190.

[Soares et al., 06] Soares, S.; Borba, P. and Laureano, E.: Distribution and Persistence as Aspects. Software: Practice and Exprerience, 2006.

[Solberg et al., 05] Solberg, A.; Simmonds, D.; Reddy, R.; Ghosh, S. and France, R. Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development. In: 29th Annual International Computer Software and Applications Conference (COMPSAC'05), Volume 1, 2005, pp. 121-126.

[Sullivan et al., 05] Sullivan, K., Griswold, W.G., Song, Y., Chai, Y., Shonle, M., Tewari, N., Rajan, H. On the criteria to be used in decomposing systems into aspects. In: Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE 2005), ACM Press, 2005.

[Suzuki and Yamamotto, 99] Suzuki, J. and Yamamotto, Y. Extending UML with Aspects: Aspect Support in the Design Phase. In: Proceedings of the 3rd Aspect-Oriented Programming Workshop at the 13th European Conference on Object-Oriented Programming (ECOOP). Lisbon, Portugal, June, 1999, pp. 299-300.

[Stein, 02] Stein, D. An Aspect-Oriented Design Model Based on AspectJ and UML. 2002. 186p. Master thesis, University of Essen, Germany.

[Stein et. al., 02] Stein, D.; Hanenberg, S.; Unland, R. Designing Aspect-Oriented Crosscutting in UML. In: 1st International Workshop on Aspect-Oriented Modeling with UML, AOSD 2002, Enschede, The Netherlands, April 22, 2002.

[Wampler, 05] Wampler, D. The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture, Aspect Programming, Inc. October, 2005, http://aspectprogramming.com/papers/AOP%20and%20MDA.pdf.

[Winck and Junior, 06] Winck, D.V. e Junior, V.G. AspectJ: Programação Orientada a Aspectos com Java. São Paulo, Novatec Editora, 2006. ISBN: 85-7522-087-X.