A MDA tool for the development of service-oriented component-based applications

Nestor Riba, Humberto Cervantes
Universidad Autonoma Metropolitana-Iztapalapa (UAM-I),
nestor_arz@yahoo.com, hcm@xanum.uam.mx

Abstract

This paper presents a process for the development of a Model-Driven Architecture (MDA) tool for the construction of service-oriented component-based applications. The process is used in the construction of a tool for one particular domain, but can be easily adapted to other domains. The tool in itself simplifies the development of components using a MDA approach, in which modeling is at the core of the development activities. After components are modeled, the tool validates the correctness of the design of the model based in a specification that is embedded inside the tool. Once the model has been validated, the tool is capable of creating the skeletons of the code for the components which are then executed inside the OSGi platform.

1. Introduction

Component orientation is a software development approach where applications are built through the assembly of reusable software building blocks called components. Generally, the construction of component-based solutions is achieved by assembling components that are 'physically' available at the time the application is built. The availability of components at the time of assembly is, however, not always possible or even desirable. Certain kind of component-based applications, such as plugin-based ones, allow components that are unavailable at the time of application construction to be integrated into the application later into its life-cycle (for example after it has been installed). One particular variety of component models push this situation further, as they allow the components to be introduced or removed from the application dynamically (i.e. at run-time). These component models are useful for constructing applications whose architecture needs to evolve constantly as they execute. Supporting this *dynamic availability* of components requires, however, some kind of architectural adaptation mechanism to be present in the execution environment so that connectors can be modified automatically upon the arrival or removal of components.

One particular variety of component model which supports dynamic availability is described in the Declarative Services (DS) chapter of the OSGi platform specification [7]. The DS component model is a service-oriented component model based on the concepts introduced by its precursor, called the Service Binder, which is discussed in [1]. In both these models, components are bound using a service-oriented interaction pattern, and their structure is described declaratively (thus the name of the Declarative Services). Components are service providers and their services can be required by other components in order to work properly. In these models, the component's structure includes simple adaptation rules which are associated to the component's dependencies. At run-time these adaptation rules are used by an execution environment to manage the connections between the components. In this component model, the support of dynamism can be seen as a non-functional aspect which is managed by the container. This is similar way to the support that exists in other component models to non-functional aspects such as persistence, security or transactions.

This approach to supporting dynamic availability is of great help to component developers who can focus on writing application logic and are relieved from the burden of writing adaptation logic necessary to cope with dynamism. Developers, however, must still be careful when writing the component descriptors along with the adaptation rules. These descriptors can be difficult to write as they must map precisely to classes that implement the components, and they must also respect several constraints which can be difficult to remember. In addition to this, there are ongoing efforts to extend these component models and to port them to platforms other than OSGi [6], which is the platform in which they are currently implemented. This situation will make it more difficult for developers to write more complex descriptors and to port their components to other platforms in the future.

This paper presents ongoing work that solves these problems by proposing a development tool built following an Model-Driven Architecture (MDA) approach. In MDA, the construction of applications focuses on the definition of platform-independent models which are further used to generate platform-specific models and later, code automatically. The proposed tool, which currently focuses on the Platform Specific Model (PSM) to code transformation, provides a way to develop components visually as models. The tool also verifies the semantics of the models and furthermore it generates component skeleton code for a specific platform automatically. Using an MDA approach in the construction of this tool allows it to be adjusted easily with respect to modifications in the component model. Finally, the tool provides a layer of isolation from the particular technology in which the components are implemented.

Another important aspect of this paper is to present the lightweight process defined for the development of this MDA tool, this process was designed to be easily adapted in by any software development organization in order to create customized MDA tools for particular domains.

The structure of the paper is as follows: section 2 discusses the concepts behind service-oriented component models and gives more detail about the problems that motivate this work, section 3 describes the tool construction process, section 4 presents current results, section 5 presents related work and finally, section 6 concludes the paper.

2. Service-oriented component based application development

This section discusses the process of developing service-oriented component based applications and details the issues faced by some existing models.

2.1. Service-oriented component models

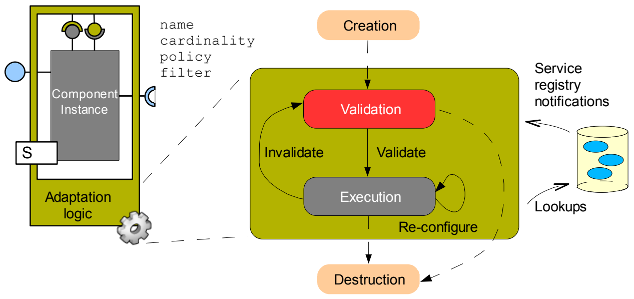


Figure 1: Service-oriented component model

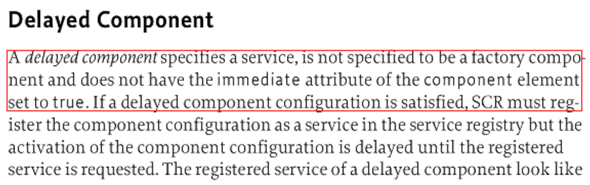
Service-oriented component models, such as the Service Binder [1] or the Declarative Services [7], support the introduction and removal of components from the application at run-time (i.e. dynamic availability). To do so, they introduce on one hand concepts from service-orientation, and on the other, adaptation rules and mechanisms into the component model and its execution environment. In these models, components play the role of both service providers and requesters, and the interfaces they implement are used as service descriptions which are published in a service registry at runtime. When components are introduced or removed from the execution environment, their services are also introduced or removed from the service registry respectively. The execution environment monitors changes in the service registry and adapts components that have dependencies on particular services when changes occur (see figure 1). These changes include the introduction, change, or removal of services in the service registry. The way that components are adapted varies according to simple rules that are defined at the component dependency level in the component descriptor. These rules define several behaviors including: how many connections can be created at runtime with respect to components that provide a particular service (cardinality), whether the component with the dependency supports dynamic changes in the connections (policy), and also which particular components that provide the required service can be connected (filter). Figure 2 shows an example from a Declarative Services component descriptor, which is written in XML. This particular component provides an English spell check service via an interface. The component can connected to one or more components that provide the dictionary service that it requires (cardinality = 1..n), and it supports dynamic modification of the connections towards the dictionary services (policy = dynamic). Furthermore, the dictionary services from any provider can be used by the spell checker (provider=*), and once they are located, their references are bound to the component using the bind and unbind methods defined in its implementation class.

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="example.component">
   <implementation class="mx.uam.examples.components.ComponentImpl"/>
   <property name="language" value="english"/>
   <service>
      <provide interface="mx.uam.examples.services.SpellCheckService"/>
   </service>
   <reference name="dictionaries"
      interface="mx.uam.examples.services.EnglishDictionaryService"
      cardinality="1..n"
      policy="dynamic"
      target="(provider=*)"
      bind="setDictionary"
      unbind="unsetDictionary"
   />
</component>
```

Figure 2: An example of a component descriptor for the Declarative Services

2.2. Current issues

The declarative approach promoted by the aforementioned service-oriented component models greatly simplifies development tasks. Without this support, developers must face the complexities of writing service management and adaptation logic in addition to application logic. It must be noted than the former code can frequently end up being more complex than the latter. To use these component models, however, developers must still be careful when creating component descriptors, as the descriptors must map correctly to the code that implements the component and, in addition, the descriptor must respect several constraints. Figure 3 shows an excerpt from the Declarative Services specification which gives an idea of the kind of constraints that the developers must follow when writing the component descriptors. This excerpt specifies that a *delayed* component (a component that is only activated when is used for the first time) cannot be *immediate* (a component that is activated at the moment it is installed). The specification contains several similar restrictions which must all be taken into account when developing components.


Delayed Component

A *delayed component* specifies a service, is not specified to be a factory component and does not have the immediate attribute of the component element set to true. If a delayed component configuration is satisfied, SCR must register the component configuration as a service in the service registry but the activation of the component configuration is delayed until the registered service is requested. The registered service of a delayed component look like

Figure 3: Fragment of the Declarative Services specification (Section 112.2.3, p. 281)

The difficulties associated to writing component descriptors are likely to increase if the component model is modified and more characteristics and constraints are added to the component model. This is likely to happen as the OSGi specification is constantly being updated. This could also occur, for example, if additional types of connections, such as event-based, were to be supported by the components (this area of research is currently being explored in a project where one of the authors is participating). Finally, the aforementioned models are also tightly tied to a particular implementation platform, which in this case is OSGi [6]. The concepts behind service-oriented component models are, however, independent of this platform and it would be desirable to reduce the coupling to this specific service platform and to use them in a different one.

3. MDA tool construction process

Model Driven Architecture (MDA) is a software development approach which promotes the use of models as the fundamental artifacts during the software development life-cycle. In MDA, an application is initially modeled independently from the particular platform in which it will be implemented. Platform-independent models (PIMs) are subsequently transformed into other models that are closer to a specific platform. At the end of this transformation process, a platform-specific model (PSM) is obtained [3]. This model can then be used to easily generate the application's code. One goal of MDA is to automate model transformations as much as possible, and to achieve this, this approach promotes an extensive use of tools to support the developers.

An essential advantage of using the MDA approach is that the developers only have to worry about the modeling of essential application elements, typically the ones related with the business logic. The supporting tools then deal with all the technical and architectural aspects that will be modeled and transformed into code. This approach provides many benefits as it accelerates both the development and the maintenance processes. When new functionalities are required, they must simply be introduced at the model level, and the application can be quickly re-generated.

The issues associated to the development of service-oriented components which were discussed in the previous section can be solved through the construction of a tool that follows an MDA approach. Such tool allows developers to build components by modeling them visually. These models can then be validated to verify them respect the constraints of the component model, and furthermore the models can be used to generate the component's code and descriptors automatically.

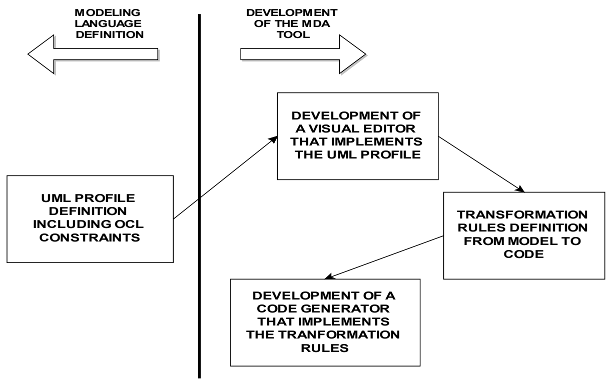

MODELING LANGUAGE DEFINITION
DEVELOPMENT OF THE MDA TOOL
DEVELOPMENT OF A VISUAL EDITOR THAT IMPLEMENTS THE UML PROFILE
UML PROFILE DEFINITION INCLUDING OCL CONSTRAINTS
TRANSFORMATION RULES DEFINITION FROM MODEL TO CODE
DEVELOPMENT OF A CODE GENERATOR THAT IMPLEMENTS THE TRANFORMATION RULES

Figure 4: MDA tools creation process

The process we propose for building such a tool is depicted in figure 4. The tasks depicted in this process can be grouped into two main groups of activities. The first one is the definition of a visual vocabulary to model applications. The definition of the vocabulary also includes the definition of several model validation constraints. The second group of activities concerns the development of the MDA tool based on the previously developed vocabulary. The activities include: (1) the development of a visual editor that implements the vocabulary and its constraints, (2) the definition of transformation rules which are used by the tool to convert the platform-independent models into platform-specific ones and into code, and finally (3), the implementation of the transformation rules in order to create a code generator. It must be noted that the process depicted is generic as it can be followed to built different kinds of MDA tools. The activities of the process are discussed in detail in the following sections.

3.1 Definition of a modeling vocabulary for the component model

The Unified Modeling Language (UML), which is the most extended modeling language in use today, provides a standard way to model many aspects associated to software development. In the case of some particular domains and applications, however, a more specific modeling vocabulary than the one provided by standard UML is needed. This vocabulary can be created by defining a UML profile [2]. A UML profile extends the standard UML notation to a particular domain or platform by defining new elements derived from the basic UML ones (which include classes, associations, attributes, etc). The use of a UML profile is favored over the use of a Domain Specific Language (DSL) since UML is a widely accepted modeling language.

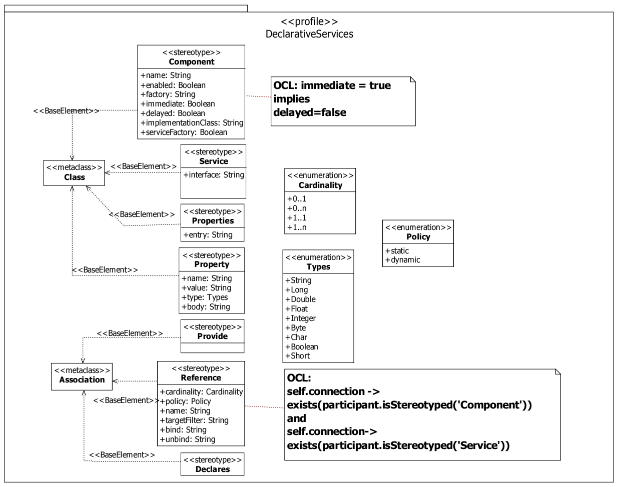
The profile is created using UML's extension mechanisms which are *stereotypes*, *tagged values* and *constraints*. Stereotypes are used to define new basic elements, tagged values are used to define attributes of an element or stereotype, and constraints are used to specify semantics and conditions associated to the elements of the model used to validate it.

Figure 5 shows the elements of the UML profile for the Declarative Services component model, which was obtained from the DS specification. The figure shows the Reference, Service, Component, Provide, Declares and Property stereotypes. These stereotypes are used to represent essential elements of the model, in particular, the figure illustrates the fact that a Component can have a Reference to a Service and at the same time that this Service is Provided by another Component. In addition, a Component can Declare some Properties necessary to configure its behavior or to distinguish its services from the ones provided by other components.

Each stereotype has its own attributes (or tagged values) that represent the properties that the stereotype can have. For example a component has a boolean attribute labeled *immediate* that indicates if the component will be activated at the moment it is installed. Other attributes represent many other component behaviors or characteristics as specified by the Declarative Service Specification [7].

Once the UML profile has been defined, it is necessary to provide a means to validate the semantic correctness of models built using the profile. This is achieved by complementing the UML profile with constraints written in a language called Object Constraint Language (OCL) [8]. This language is part of the UML 2.0 specification. With respect to the service-oriented component model, these constraints are obtained from textual descriptions, such as the specification fragment from DS presented in figure 3.

The definition of the OCL constraints is essential because these constraints are later needed to support the automatic validation of the semantics of the models. This frees the developers of the burden of learning and remembering all the rules defined in the component model specification and allows them to focus their effort in developing the business logic. Figure 5 shows a simple OCL rule used to validate the rule described in the fragment of the specification of figure 3. Due to lack of space, the profile does not show all of the OCL constraints associated to it.



```
OCL: immediate = true
implies
delayed=false
```

```
OCL:
self.connection ->
exists(participant.isStereotyped('Component'))
and
self.connection->
exists(participant.isStereotyped('Service'))
```

Figure 5: UML profile for Declarative Services (not all OCL rules are shown)

3.2 Implementation of the UML profile in a visual editor.

After defining the visual modeling language, the next step in the process is the construction of a visual editor. This editor uses the profile to support the creation of models of components and is capable of validating the models using the constraints written in OCL. In order to accomplish the objective of creating a lightweight process that can be easily followed to create MDA tools by any software development organization, a framework that accelerates the development process of the visual editor was used.

The framework that was used for the creation of the editor is GMF (www.eclipse.org/gmf). It was chosen because it is open source, based on standards of the OMG, and integrated with the Eclipse platform tool. GMF can be seen as an MDA tool for the creation of visual editors. GMF works with various models, each one representing distinct aspects of the editor. One important model of

GMF is the called domain model, which is in fact the meta-model of the visual language, which corresponds directly to the UML profile. Other models of GMF represent the graphical and tooling aspects of the visual editor. The developer who wants to construct a visual editor has to create the different models required by GMF, and GMF generates the code for the editor. The time required develop a visual editor depends of the complexity of the meta-model, but it is generally much shorter than the development of a specific visual editor from scratch.

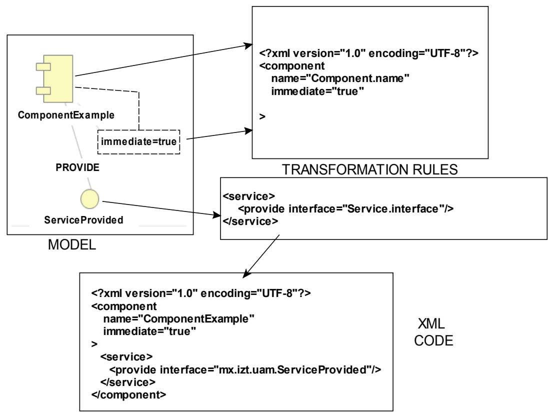
One particular characteristic of GMF which is helpful for the development of a complete MDA solution is that it accepts, inside one of his internal models, the definition of the OCL constraints. As a result, the visual editor can validate the model against the constraints defined in the profile. The construction of the visual editor represents the first step in the construction of the MDA tool. The visual editor allows developers to model components visually and furthermore it validates the models automatically against the constraints defined in the component model's specification.

3.3 Establishing and implementing the transformation rules

Once the model is created and validated using the visual editor, the next step is to transform it into another more detailed platform specific model or directly into code. To do this, it is necessary to indicate how the transformation will take place by defining which elements of one model correspond to which elements of the code.

The transformation rules specify which part of the descriptor or piece of code will be generated from a particular element of the model. Figure 6 shows an example of a transformation rule which is used to generate the component descriptor from a visual model of the component. In the model, the Component and Service stereotypes are displayed using an iconic representation. Each element of the model has some attributes that will be used in the transformation process. For this example, the name of the component is used to generate a fragment of code in the XML descriptor of the component. To express which property of the element will be used in the code, a dot syntax is used. In this example, Component.name, which indicates that the value given by the user to the property name of the element Component, will be used in the generated code. The Component has an attribute called immediate, that in the case of being true results in a particular fragment of code to be included in the generated code of the XML component descriptor. The association labeled PROVIDE between a Service an a Component specifies that a Component will provide the Service, this element will also be included in the XML descriptor. Again due to lack of space only these transformation rules are shown, but many more are necessary to successfully transform all the elements of the model.

The XML schema of the descriptor and the code elements are described in the DS specification. However, as in the case of the constraints defined by the specification, it is not necessary for the developer to need to know them in detail because they will be generated by the tool.



```
<?xml version="1.0" encoding="UTF-8"?>
<component
   name="Component.name"
   immediate="true"

>
```

```
<service>
    <provide interface="Service.interface"/>
</service>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<component
   name="ComponentExample"
   immediate="true"
>
  <service>
    <provide interface="mx.izt.uam.ServiceProvided"/>
  </service>
</component>
```

Figure 6: From model to code

3.3 Implementation of the transformation rules.

To accelerate the process of development of a complete solution that can be easily connected to the previously described visual editor, an open source tool was also selected for the last activity: the development of the code generator.

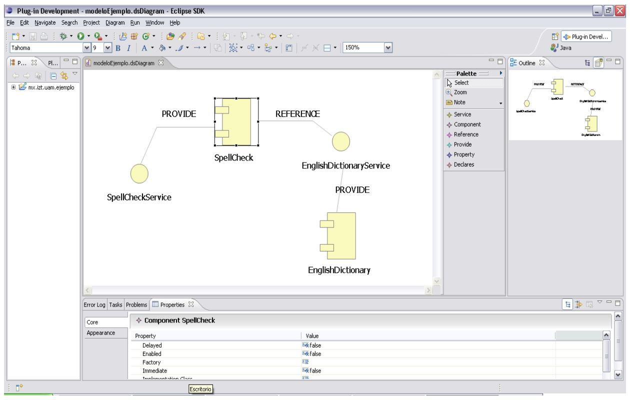
The chosen tool is Acceleo (www.acceleo.org), an easy to use open source tool that takes a model in XMI format and transforms it into code or other models by defining a series of templates that contain the transformation rules previously defined.

Coupling the visual editor to the code generation subsystem results in a complete MDA tool that supports the component development process from modeling to code generation.

4. Current results.

This section presents the resulting tool and discusses how the same tool construction process was used to generate a different tool.

4.1 Component development MDA tool.

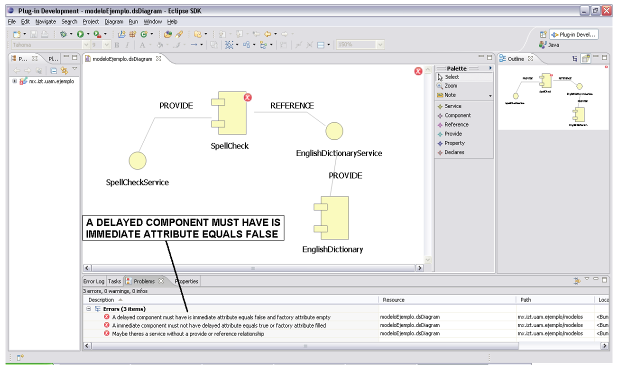


Component SpellCheck

Property	Value
Delayed	false
Enabled	false
Factory	
Immediate	false

Figure 7: The resulting service-oriented component development tool.

The resulting MDA component development tool is delivered as an Eclipse plugin (see figure 7). Once launched, the component development tool displays a graphical editor where components are modeled visually using the familiar component representation. The properties of the graphical elements can be configured using property sheets. Once a component is modeled, the tool verifies the correctness of the model and warns the developer of any constraint that is not respected. After the model of a component has been validated, the developer can request the tool to generate a DS component automatically. The tool generates the component descriptor along with skeletons of support classes (such as the component implementation and service interfaces).



A DELAYED COMPONENT MUST HAVE IS IMMEDIATE ATTRIBUTE EQUALS FALSE

3 errors, 0 warnings, 0 infos

Description	Resource	Path	Loca
Errors (3 items)			
A delayed component must have is immediate attribute equals false and factory attribute empty	modeloEjemplo.dsDiagram	mx.izt.uam.ejemplo/modelos	<Bun
A immediate component must not have delayed attribute equals true or factory attribute filled	modeloEjemplo.dsDiagram	mx.izt.uam.ejemplo/modelos	<Bun
Maybe theres a service without a provide or reference relationship	modeloEjemplo.dsDiagram	mx.izt.uam.ejemplo/modelos	<Bun

Figure 8: An example where a constraint is violated to show how the tool highlights the problem.

Through the use of this tool, the component developer's only tasks are to model the components and to write the business logic associated to the component implementations. This approach, which can be seen as a PSM to code transformation in the context of MDA, significantly accelerates component development when compared to the traditional approach where developers must write the component descriptor and support classes manually. Furthermore it also reduces the occurrence of errors due to omissions or mistakes when writing the component descriptors.

Figure 8 shows a model that violates the constraint of the fragment of DS specification shown in figure 3, a value of 'true' is assigned to the immediate property of the component and a value of 'true' is also assigned to the delayed property of the component. The figure shows how the tool highlights the problem of the model and indicates where the problem resides.

4.2 Testing of our proposed MDA tool process development.

To evaluate the flexibility of the approach, two more MDA tools were developed to support other different types of component connections, in particular connections that support a producer-consumer approach as defined by the WireAdmin service section of the OSGi specification [7]. The creation of this new MDA tools was performed in a few days following the complete process described in figure 4 and using GMF and Acceleo as supporting tools.

5. Related work.

There is currently an enormous amount of work related to MDA. In the particular field of MDA and component-based software development, several profiles for different component models have been developed. A UML profile for the Corba Component Model is described in [5]. The research described in [2] describes PervML, which is a language to model pervasive applications. This work is closely related to the work presented in this article as the authors define a UML profile for the OSGi platform. Their profile, however, does not cover the Declarative Services component model specification and it does not detail the construction of tools based on the profile.

There is a considerable amount of research work related to the construction of MDA tools. The work described in [9] presents the development of an MDA tool that facilitates the modeling of distributed real-time embedded (DRE) systems, validating the PSM models and generating the code for them. However, this work focuses on a specific domain and does not present the process for the construction of the tool or how to create tools for others domains.

There are many tools in the market that allow developers to visually model their component-based applications, although most of them are platform specific. Some examples of these tools include JDeveloper from Oracle (www.oracle.com), ComponentOne Studio from ComponentOne (www.componentone.com), Visual Cafe from Symantec (www.symantec.com) and VisualAge from IBM (www.ibm.com). In all of them an application can be built following an MDA approach since the application is modeled and the tools perform the PSM to code transformation automatically. However, most of these tools are expensive and are not easily extensible to other domains or platforms. There are also open source tools which implement some aspects of MDA. These tools include Kermeta (www.kermeta.org), which is an open source project for Eclipse that is still under development and AndroMDA (www.andromda.org).

In the particular case of the OSGi platform, at the moment of this writing, there are no similar tools available. The project presented in this paper intends to fill this gap, however, the tool that has been developed is still limited with respect to the functionalities offered by the aforementioned tools as these tools generally allow developers to model certain functional aspects of the application.

6. Conclusions and future work

This paper has presented the development process of a service-oriented component model development tool based on the MDA approach. The resulting tool is of great help to developers who can model the components visually. The tool verifies that the models are semantically correct with respect to the restrictions imposed by the component model's specification and it is also capable of generating the component's skeleton which includes a declarative descriptor and implementation code. At the moment, the tool is limited to modeling component's structure and non-functional characteristics relative to supporting dynamism.

This work led us to conclude that MDA is a very useful and complementary approach to the current trend of extraction of non-functional logic and its expression in a declarative way outside of the component's logic (as in EJB -http://java.sun.com/products/ejb-, Spring -http ://www.springframework.org- or Declarative Services). The declarative approach has been of great help to developers who can now focus on the development of functional aspects of their applications. However, this approach is also error prone and it often becomes difficult to locate and debug errors as they generally appear only at runtime, when the descriptors are interpreted. Tools such as the one presented in this paper prevent these errors from occurring as they verify the semantic correctness at the modeling level. Even though the declarative approach has been of great help in the reduction of development time, tools such as the one presented in this paper further reduce development time as they allow developers to work at a higher level of abstraction.

In addition, the lightweight process used for the development of the tools can be used in the construction of other tools for different domains. Experimentation has shown that the process is easy to follow and can be implemented quickly. The MDA tools in the market today tend to be complex and expensive, hard to customize and may lose their value very quickly as a consequence of the constant evolution of technology. The approach presented in this paper is useful to build a variety of MDA tools suited to different needs, furthermore, if the market or technology changes, the tool can be easily modified or substituted by a new one.

Future work includes the creation of a complete MDA tool by introducing the possibility of modeling independently from a particular platform and also of introducing business logic at the PIM level. Other tasks include the creation of transformation rules for different platforms and the extension of the development tool so that not only components can be modeled, but also compositions. Finally, it is planned that the tool presented in this article will be released as open source.

7. References.

[1] H. Cervantes and R. S. Hall: "*A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences*," CBSE 7, Edinburgh, Scotland, May 2004.

[2] L. Fuentes and A. Vallecillo, *"An introduction to UML Profiles,"* European Journal for the Informatics Professional, Vol. V, No. 2, pp. 6-13, April 2004

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

[4] J. Muñoz, V. Pelechano and J. Fons, *"Model Driven Development of Pervasive Systems"*, ERCIM News, July 2004, vol. 58, pp. 50-51, ISSN: 0926-4981

[5] OMG, *"UML Profile for CCM, V1.0"*. Online document available at: http://www.omg.org/technology/documents/formal/profile_ccm.htm

[6] OSGi Alliance, *"OSGi Service Platform, Core Specification, Release 4"*, August 2005, available online at: http://www.osgi.org

[7] OSGi Alliance, *"OSGi Service Platform, Service Compendium, Release 4"*, August 2005, available online at: http://www.osgi.org

[8] UML 2.0 "*OCL Specification"*. OMG. 2003, available online at: http://www.omg.org

[9] Douglas Schmidt, Jeff Gray, Nanbor Wang, Aniruddha Gokhale, "*CoSMIC: An MDA generative tool for distributed real-time and embedded component middleware and applications*", OOPSLA 2003, Anaheim, California, United States of America, October 2003.