Automated Domain-Specific Modeling Languages for Generating Framework-Based Applications[*]

André L. Santos
Faculty of Sciences
University of Lisbon
Campo Grande, 1749-016
PORTUGAL
andre.santos@di.fc.ul.pt

Kai Koskimies
Department of Software Systems
Tampere University of Technology
P.O.BOX 553, FIN-33101 Tampere
FINLAND
kai.koskimies@tut.fi

Antónia Lopes
Faculty of Sciences
University of Lisbon
Campo Grande, 1749-016
PORTUGAL
mal@di.fc.ul.pt

Abstract

The adoption of Domain-Specific Modeling Languages (DSMLs) for generating framework-based applications has proved to be an effective way of enforcing the correct use of frameworks and improve the productivity of application developers. However, the development of the code generator of a DSML is typically a laborious task with difficulties in what concerns complexity, understandability, and maintainability. In this paper, we address this problem with a new approach for developing DSMLs for frameworks that allows to eliminate the need of implementing code generators. The approach relies on the extension of frameworks with an additional layer based on aspect-oriented programming that encodes a DSML. By means of a generic language workbench, framework-based applications can be generated from application models described in that DSML. The proposed language workbench was implemented in a prototype tool and we performed a case study on the Eclipse Rich Client Platform.

1 Introduction

Object-oriented frameworks are an important means for realizing *software product-lines* [4]. *Framework-based applications* are developed by *instantiating* a certain framework. The activities related to developing a framework are known as *domain engineering*, whereas *application engineering* refers to the development of framework-based applications.

Learning how to correctly use a non-trivial framework is a difficult and time-consuming activity. In order to help application engineers to overcome this obstacle, domain engineers may develop tool support for *Domain-Specific Modeling* (DSM) [6]. By doing so, the abstraction level of application development is raised, given that applications are described in terms of high-level concepts expressed in a *Domain-Specific Modeling Language* (DSML). The essential elements of a DSM solution are the *modeling language* and the *code generator*, which generates framework-based code from descriptions in that language.

Using model-driven engineering terminology, the modeling language definition can be given in a *meta-model*, while descriptions in that language, i.e. *(application) models*, are given as instances of the meta-model. The meta-models and code generators may be developed using *language workbenches*[1] for that purpose (e.g. [13, 8, 7]). Figure 1 illustrates conventional tool support for generating framework-based frameworks. As part of the *problem domain*, we have the meta-model that describes domain concepts, and application models that describe instances of those concepts. As part of the *solution domain*, we have the object-oriented framework, and the framework-based applications.

[*]This work was partially supported by FCT, through the Multiannual Funding Programme.

[1]i.e. tools that are suitable for developing some sort of DSL support.

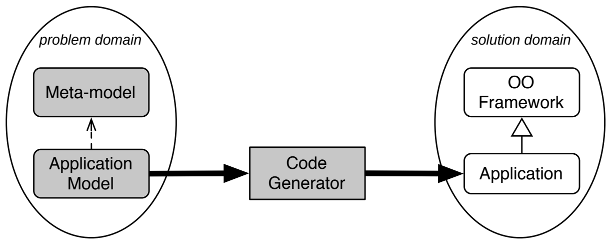


Figure 1. Conventional DSM solution for generating framework-based applications.

DSM approaches claim that it is possible to increase productivity in application engineering activities by up to an order of magnitude [6]. However, these productivity gains imply a significant additional effort in domain engineering activities, since the meta-model and the code generator have to be developed and maintained consistently throughout the evolution of the framework. A DSML is the result of several development iterations, and nevertheless, new increments have to be developed when the domain evolves, implying modifications in the framework, meta-model, and code generator. This makes the evolution of the tool support for DSM challenging.

The difficulty of building and maintaining tool support for DSM depends essentially on the complexity of the mapping between the concept instances expressed in the DSML and the code that has to be generated. In principle, the more straightforward this mapping is, the easier it will be to implement and evolve the code generator. As a matter of fact, *black-box* frameworks are pointed out as being suitable for having tool support for DSM (in [16] referred to as *visual builders*), given that what it has to be generated is simply glue code that composes default components.

Let us assume that the easiest way to develop tool support for DSM is by having a black-box framework as the framework which applications are based on. Although black-box frameworks may have a conceptually clean way of instantiating them, the framework classes do not contain enough information so that the transformation definition between domain concepts and the glue code can be automatically obtained. So it happens because neither the concepts nor their relationships are explicit. The code generator has thus to be developed for implementing this mapping.

One of the main goals of the approach presented in this paper is to relieve domain engineers from developing code generators. We propose a technique for obtaining tool support for DSM solely by enhancing a framework with an additional layer that explicitly encodes the DSML, which we refer to as the *DSM layer*. This layer enables to build framework-based applications at a higher abstraction level, so that the mapping between domain concepts and framework-based code is straightforward, and therefore, it can be automatically obtained. The realization of the DSM layer relies on our previous work that proposes a technique based on *aspect-oriented programming* for modularizing framework extension points in *framework specialization aspects* [18] . These are reusable aspect modules for developing framework-based applications. A DSM layer is composed by several specialization aspects, which are annotated with little additional meta-data for enabling that both the meta-model and the mapping between models and framework-based code can be unambiguously inferred.

Using our technique, domain engineers are able to effectively enhance a framework's implementation with the encoding of a DSML, which can be directly used to generate framework-based applications without the need of developing any additional artifact. This can be achieved by means of a generic language workbench, which extracts meta-models from DSM layers and it is capable of processing instances of those meta-models for generating application code. In addition, we propose a mechanism for realizing *open variation points* [9] based on integrating manual code with code generated from models, given the importance of this issue to DSM-based development.

Comparing to the state-of-the-practice, the approach presented in this paper represents a major strategic difference, given that we propose frameworks to have a "built-in" DSML. Our approach helps to alleviate the problems related to the development and evolution of tool support for DSM. The approach was successfully applied with the Eclipse RCP framework [12] — an industry-strength framework. The proposed language workbench was implemented in an Eclipse-based [7] tool named ALFAMA [17]. The tool supports DSM layers written in AspectJ and defines meta-models in EMF (Eclipse Modeling Framework) [7].

The paper proceeds as follows. Section 2 presents an overview of our approach. Section 3 addresses the development of the DSM layer. Section 4 presents the ALFAMA tool. Section 5 compares the proposed approach with conventional tool support for DSM. Section 6 describes the case study on Eclipse RCP. Section 7 discusses related work, and Section 8 concludes the paper.

2 Approach Overview

This section presents an overview of our approach (Figure 2). In contrast with conventional approaches for generating framework-based applications (as illustrated in Figure 1), domain engineers develop the DSM layer in addition to the framework, while they are relieved of implementing a code generator and of defining the DSML concepts separately (i.e. externally to the framework implementation). Our approach relies on a language workbench that, on the one hand, extracts DSML definitions from DSM lay-

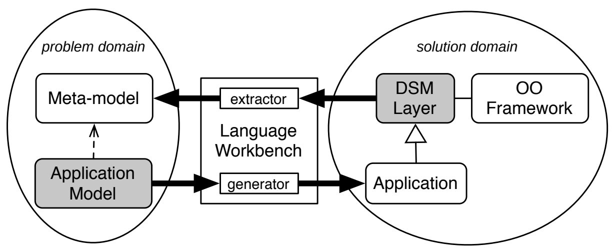


Figure 2. Proposed DSM solution for generating framework-based applications.

ers while, on the other hand, it transforms instances of those models (i.e. application models) into code that is based on the DSM layer. Application engineers develop application models described in the DSML, as if they were using conventional DSM tool support. However, the application models are given as input to the language workbench, instead of a code generator developed specifically for the DSML. The language workbench is generic, in a sense that it can be used for multiple frameworks, as long as the DSM layer is developed in the supported programming language (e.g. AspectJ for Java frameworks) and according to certain rules.

The goal is that the DSM layer precisely represents a conceptual model embedded in its modules, using modeling constructs that are equivalent in terms of expressiveness to those that can be found in meta-modeling technologies. The conceptual model is the meta-model that defines the DSML. Our option was to consider modeling constructs that resemble the ones of EMF [7], which is a Java implementation of the Meta-Object Facility (MOF) [15] — a standard for defining modeling languages. Although with small differences, the meta-models defined in EMF can be considered to have equivalent expressiveness to those that can be defined using the commercial tools such as MetaEdit+ [13] and Microsoft DSL Tools [8].

Representing the DSML in an external format using a meta-modeling technology has advantages such as the possibility of easy integration with other tools that need to access the models, or the standard serialization of meta-models and models.

The definition of a concrete syntax for the DSML is out of the scope of this paper. Concrete syntax is an orthogonal issue, which can be handled independently in addition to the abstract syntax and it does not interfere with the problem of realizing the code generation.

3 DSM Layer

This section addresses the DSM layer. This layer is framework specific and developed by domain engineers. It embodies a DSML definition and, at the same time, the information required for building framework-based applications, given their models expressed in the DSML.

A DSM layer is composed by a set of DSM modules. Each DSM module is associated with a DSML concept and can be either a class or an aspect, with annotations.

In the rest of this section, we proceed as follows. Subsection 3.1 presents the conceptual model in which the notion of DSM layer is based, namely the modeling constructs that can be represented in a DSM layer. Subsection 3.2 introduces the concepts of an example framework fragment. These concepts are used in the running example throughout Subsection 3.3, which addresses the implementation of DSM modules.

3.1 Modeling constructs

The modeling constructs that can be represented in the DSM layer are given in the conceptual model of Figure 3. Most of them are fairly equivalent to those that can be found in meta-modeling and associated technologies, such as EMF.

A *concept* is identified by a *name* and may have several *attributes*, which have a primitive type and are also identified by *name*. A concept may define *relationships* with other concepts, which can be either *composite associations* or *directed associations*. Relationships have an associated *multiplicity* for restricting the number of associations between concepts. A concept may be a specialization of a *super concept*, inheriting its attributes and relationships. A concept may be *abstract*, implying that it cannot be instantiated. In addition to these conventional conceptual modeling constructs, there are two special kinds of concepts related with the integration of manual code and generated code. An *open concept* is a concept that represents an open variation point, where application-specific code can be added. An *accessible concept* is a concept whose instances may be accessed by instances of open concepts, enabling manually given code to access generated code.

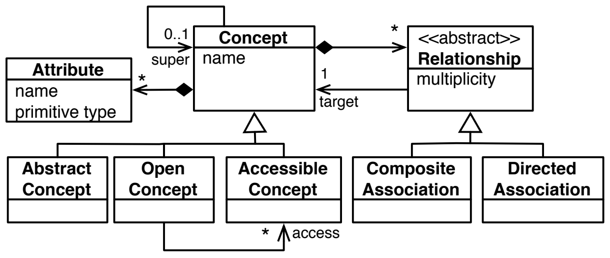


Figure 3. Modeling constructs in DSM layers.

3.2 Example Framework

This subsection presents an example framework for the purpose of explaining how to develop the DSM layer. As a case study, we applied our approach to an existing framework: the Eclipse RCP framework [12], which can be used for building stand-alone applications based on Eclipse's dynamic plug-in model and UI facilities. This case study is discussed in detail in Section 6. Through the rest of the paper, we shall use just a small simplified fragment of it as a running example.

The DSML concepts for the example framework fragment can be defined through the meta-model given as a class diagram in the left-hand part of Figure 4. Each class represents a concept. An *RCP application* has initial window size given by *width* and *height*. It may contain several *ac-*

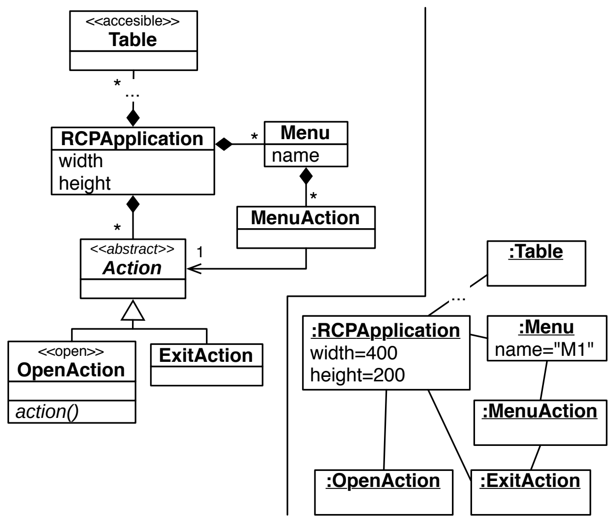


Figure 4. Example framework: Meta-model (left) and application model (right).

tions (abstract concept) and several *menus*. These two containment relations are examples of composite associations. The specific behaviour of an action can be defined in terms of the operation action(). An *open action* is an open concept where the action behavior con be given manually. *Exit action* is a framework-provided action for quiting the application, which can be used by the application engineers as-is. The two latter cases are examples of concept inheritance. A menu has a *name* and may contain *menu actions* which contain references to the application's actions. This is an example of a directed association. An application may contain *tables* indirectly from other child concepts (not shown). A table can be accessed by open concepts, and thus, it is an accessible concept. On the right-hand side of Figure 4 we can see an object diagram representing an instance of the meta-model (i.e. an application model).

Both the meta-model and the application model are used throughout the next subsection. The former is what is represented in the DSM layer, while the latter is used for exemplifying the code generation.

3.3 DSM modules

This subsection explains how the DSM modules can be realized in terms of *specialization aspects* [18]. In what follows, each modeling construct is illustrated with an example presented in a figure with three parts:

- The upper part shows the code of a DSM module, written in Java/AspectJ, plus annotations. The code examples omit irrelevant details, and detailed issues concerning AspectJ's primitives are explained only briefly.
- The middle part presents, on the left-hand side, the meta-model fragment that the modules are encoding and which is extracted by the language workbench. The right-hand side shows a fragment of an application model (instance of the meta-model fragment on the left). The elements drawn with a dashed line are elements that were introduced previously and which are involved in the example.
- The bottom part shows the application code that, for the given application model fragment, is generated by the language workbench (hence, not meant to be given manually).

3.3.1 Concepts and attributes

Each module of the DSM layer is associated with a single application concept, and we assume the module name to be the concept name. A concept is explicitly declared using the annotation @Concept. The concept's attributes can be declared by annotating a constructor of the module with the annotation @Attributes.

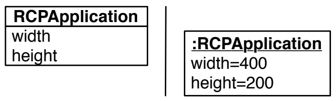
The main class of an application has to implement the framework interface IApplication, which has a method for plugging the *menus* and another one for plugging the *actions*. Figure 5 presents the DSM module that handles the concept *RCP application*. The methods are intended to be empty and non-overridable, since they are going to be advised by other modules (aspects). As the names suggest, their role is to allow the customization of *menus* and *actions*, respectively.

```
@Concept
abstract class RCPApplication implements IApplication {
 @Attributes
 RCPApplication(int width, int height) {
  /* ... */
 }

 final void fillMenuBar(IMenuManager menuBar) { }

 final void makeActions(IBarAdvisor barAdvisor) { }

 /* ... */
}
```



```
class App extends RCPApplication {
  App() {
    super(400, 200);
  }
}
```

Figure 5. Concepts and attributes.

3.3.2 Composite associations

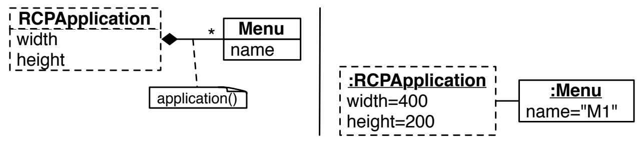
Concepts may have composite associations with other concepts. A composite association is defined by annotating an abstract pointcut with @PartOf, with the parameters concept and mult for defining the parent concept and the association multiplicity, respectively.

Figure 6 presents the DSM module that handles the concept *menu*. A menu is part of an *RCP application* and can be included by defining the pointcut application() on an extension of RCPApplication (previous module). The multiplicity defines that each application can have several menus. The advice introduces the necessary behavior for plugging the menu in the application.

```
@Concept
abstract aspect Menu {
 @Attributes
 Menu(String name) {
  /* ... */
 }

 @PartOf(concept="RCPApplication",mult="*")
 abstract pointcut application();

 after(IMenuManager menuBar) : args(menuBar) &&
  within(RCPApplication+) && application() &&
  execution(void fillMenuBar(IMenuManager)) {
   menuBar.add(createMenu());
  }
 /* ... */
}
```



```
aspect Menu1OnApp extends Menu {
 Menu1OnApp() {
  super("M1");
 }
 pointcut application() : target(App);
}
```

Figure 6. Composite associations.

3.3.3 Abstract concepts

A concept may be declared to be abstract, meaning that it cannot be instantiated. The purpose of having an abstract concept is to have other concepts that inherit from it, reusing its functionality. A DSM module representing an abstract concept is annotated with @AbstractConcept.

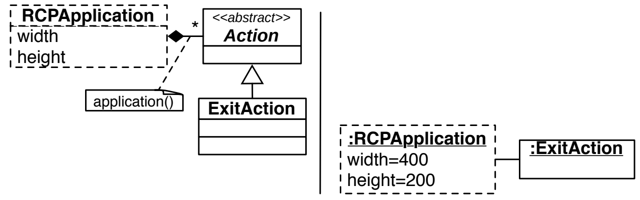
Figure 7 presents the DSM module that handles the concept *action* (Action). It is similar to the previous example. However, it has an abstract method that its extensions should define. The figure also presents the DSM module for handling the concept *exit action* (ExitAction), which inherits from Action, overriding createAction() and leaving the inherited abstract pointcut application() undefined.

```
@AbstractConcept
abstract aspect Action {
 @PartOf(concept="RCPApplication",mult="*")
 abstract pointcut application();

 after(IBarAdvisor barAdvisor) : args(barAdvisor) &&
  within(RCPApplication+) && application() &&
  execution(void makeActions(IBarAdvisor)) {
   IAction action = createAction();
   barAdvisor.register(action);
  }

 abstract IAction createAction();
}
```

```
@Concept
abstract aspect ExitAction extends Action {
 IAction createAction() {
  return ActionFactory.QUIT.create();
 }
}
```



```
aspect ExitActionOnApp extends ExitAction {
    pointcut application() : target(App);
}
```

Figure 7. Abstract concepts.

3.3.4 Directed associations

A concept may declare a directed association with another concept. This can be done by annotating an abstract pointcut with @Association, with parameters concept and mult, as in composite associations.

Figure 8 presents the DSM module that handles the concept *menu action*. An extension of Action is to be defined in the pointcut action(). The first advice captures the *action* creation and keeps its reference. An extension of Menu is to be defined in the pointcut menu(), in order to set the menu which contains the menu action. The second advice adds the action upon the creation of the menu.

3.3.5 Open and accessible concepts

An open concept represents an open variation point, where application-specific code can be added in addition to the generated code. The difference between DSM modules that represent open concepts and the regular ones is that the former declare certain methods to be exposed to application

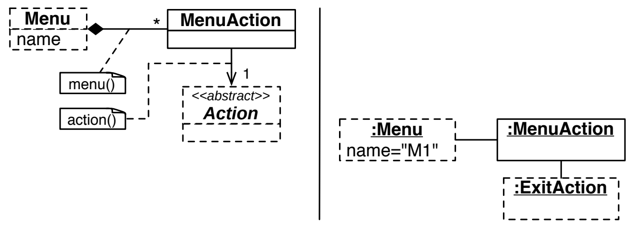
```
@Concept
abstract aspect MenuAction {
 private IAction action;

 @Association(concept="Action",mult="1")
 abstract pointcut action();

 after() returning(IAction a):
  within(Action+) && action() &&
  execution(IAction createAction()) {
   action = a;
  }

 @PartOf(concept="Menu",mult="*")
 abstract pointcut menu();

 after() returning(IMenuManager menu) :
  within(Menu+) && menu() &&
  execution(IMenuManager createMenu()) {
   menu.add(action);
  }
}
```



```
aspect ExitActionOnMenu1 extends MenuAction {
  pointcut action() : target(ExitActionOnApp);
  pointcut menu() : target(Menu1OnApp);
}
```

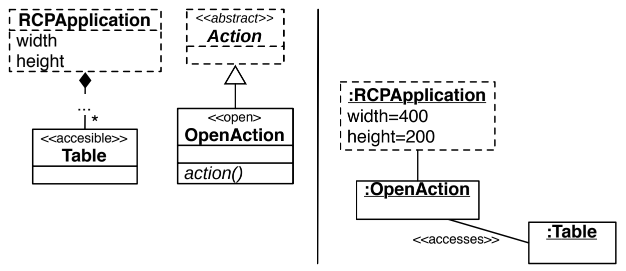
Figure 8. Directed associations.

```
@OpenConcept
abstract aspect OpenAction extends Action {
 @OpenMethod
 abstract void action();

 IAction createAction() {
  return new IAction() {
   public void run() {
    action();
   }
  };
 }
}
```

```
@AccessibleConcept
abstract aspect Table {
 /* ... */

 @AccessibleObject
 TableViewer createTable() {
  return new TableViewer();
 }
}
```



```
aspect Tab1 extends Table {
   /* ... */
}
```

```
abstract aspect AppAction_Adapter extends OpenAction {
 pointcut application() : target(App);

 after() returning(TableViewer o) :
  execution(TableViewer createTable()) && target(Tab1) {
   AppAction.table = o;
  }
}
```

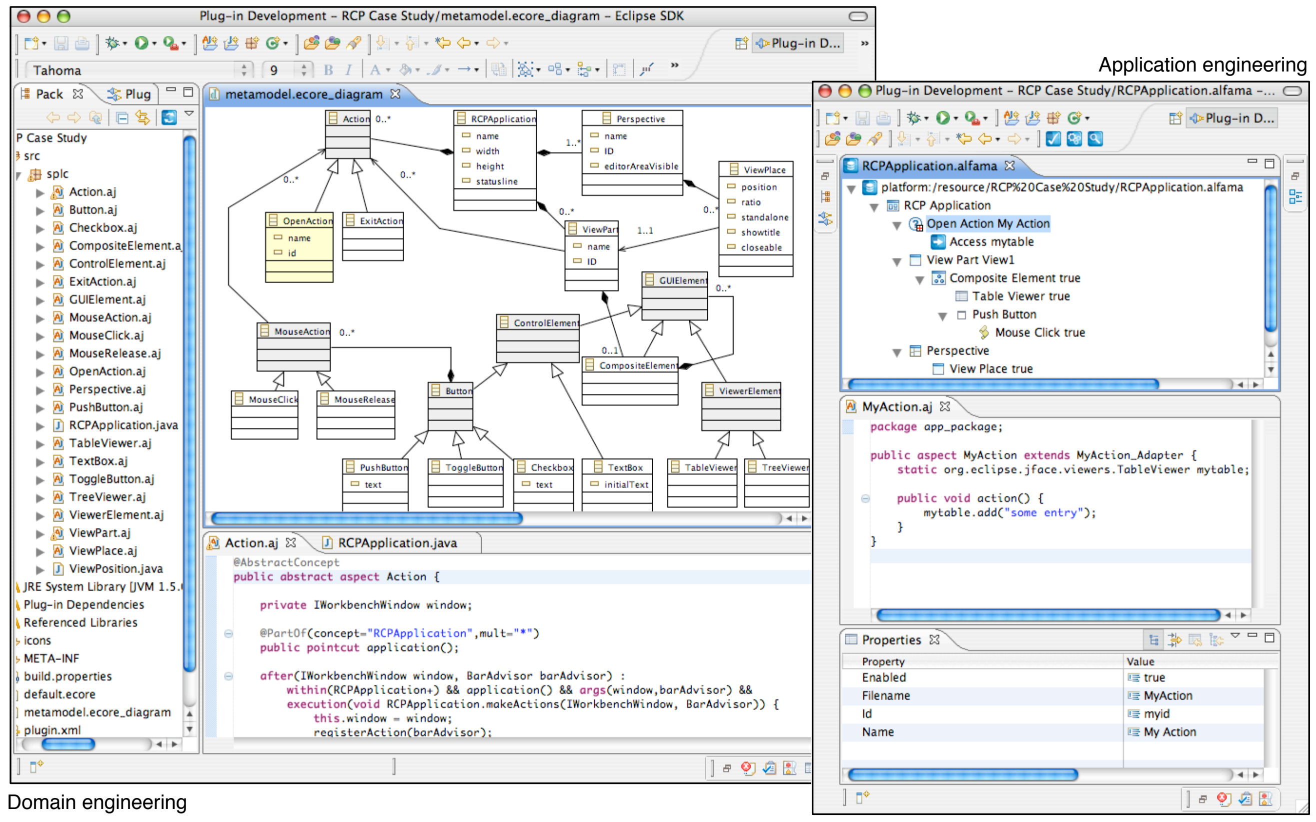
```
aspect AppAction extends AppAction_Adapter {
 /* automatic */
 static TableViewer table;

 void action() {
  table.add("some entry")
 }
}
```

Figure 9. Open and accessible concepts.

engineers. An open concept can be declared by annotating a module with @OpenConcept, while its open methods are annotated with @OpenMethod. An accessible concept is a concept whose instances may expose an object that can be accessed by open concepts. An accessible concept can be declared by annotating a module with @AccessibleModule, and the accessible object can be defined by annotating a method with @AccessibleObject. This implies that the object returned from that method is the accessible object.

Figure 9 presents a DSM module for handling the open concept *open action* (OpenAction), as an extension of Action. The method action(), which defines the action behavior, is declared as being open. The figure also presents a DSM module for handling the accessible concept *table*. When generating the code from an instance of an open concept, two modules are obtained. One module is hidden from application engineers (AppAction_Adapter) and contains code that can be generated from the model, while the other module is exposed to application engineers (AppAction) and contains the open methods. If the open concept accesses an accessible concept, the hidden module also contains code that sets a variable in the exposed module to point at the accessible object. In the example, the open action defines a special association for accessing the table, causing the exposed module to have a variable for accessing the accessible object. In this way, application engineers have a clean mechanism for enabling the manual code to access objects that were instantiated within the generated code, without the need of touching or understanding the latter.


Application engineering
Domain engineering
Plug-in Development – RCP Case Study/metamodel.ecore_diagram – Eclipse SDK
metamodel.ecore_diagram
Plug-in Development – RCP Case Study/RCPApplication.alfama –...
RCPApplication.alfama
Properties

Action.aj — RCPApplication.java

```
@AbstractConcept
public abstract aspect Action {

    private IWorkbenchWindow window;

    @PartOf(concept="RCPApplication",mult="*")
    public pointcut application();

    after(IWorkbenchWindow window, BarAdvisor barAdvisor) :
        within(RCPApplication+) && application() && args(window,barAdvisor) &&
        execution(void RCPApplication.makeActions(IWorkbenchWindow, BarAdvisor)) {
            this.window = window;
            registerAction(barAdvisor);
```

```
platform:/resource/RCP%20Case%20Study/RCPApplication.alfama
  RCP Application
    Open Action My Action
      Access mytable
    View Part View1
      Composite Element true
        Table Viewer true
        Push Button
          Mouse Click true
    Perspective
      View Place true
```

MyAction.aj

```
package app_package;

public aspect MyAction extends MyAction_Adapter {
    static org.eclipse.jface.viewers.TableViewer mytable;

    public void action() {
        mytable.add("some entry");
    }
}
```

Property	Value
Enabled	true
Filename	MyAction
Id	myid
Name	My Action

Figure 10. ALFAMA tool: domain engineering and application engineering perspectives.

4 ALFAMA Tool

We implemented the proposed language workbench in a tool that we refer to as ALFAMA[2] [17]. The tool realizes the tool support for DSM outlined in Section 2, assuming DSM layers developed according to what is presented in Section 3. The tool was implemented as a set of Eclipse [7] plugins. The development of the DSM layer relies on a small subset of AspectJ's primitives, and Java 5 annotations. The DSMLs are extracted from the DSM layer into in EMF [7] models (i.e. meta-models). Optionally, GMF [7] can be used independently for developing a concrete syntax for the DSML.

Figure 10 illustrates the ALFAMA development environment. The domain engineering perspective is shown on the left-hand side. We can see a package of DSM modules and the extracted meta-model that is represented in the DSM layer. The diagram is a visualization of the DSML concepts. The darker elements are abstract concepts. On the bottom part we can see the module Action of the DSM layer being edited. The application engineering perspective is shown on the right-hand side of the figure. On the upper part we can see an application model, which is an instance of the meta-model shown on the domain engineering perspective. The application model is being edited in Eclipse's default EMF tree view editor. The icons are shown in the editor due to a light-weight mechanism for concrete syntax, which consists of associating one icon with each concept. In the middle it is shown the source code associated with an instance of the open concept OpenAction. On the bottom part we can see a property sheet for editing the concept's attributes. In this example, the open module is the only piece of code that the application engineer has to manipulate, while all the rest is generated and it is not meant to be either touched or understood.

[2]ALFAMA: Automatic DSLs for using Frameworks by combining Aspect-oriented and Meta-modeling Approaches.

5 Comparison with conventional DSM

In this section we present the advantages and disadvantages of our approach when comparing with conventional DSM tool support.

5.1 Advantages

Reduces complexity and improves understandability. A code generator is a program that generates another program.

In non-trivial cases, this "indirection" is a source of complexity that may cause a burden for domain engineers. The most structured and intuitive approach to the development of code generators is to use code templates. Still, the implementation of the code generator can easily become complex, for instance, when parts of generated code that result from different model elements are interleaved in common modules and/or have to share instance variables. In our approach, the DSML is encoded in the DSM layer, using a relatively small set of mechanisms based on advices that either complete hook methods or compose objects. As illustrated in the examples of Section 3, the same aspect-oriented mechanisms are used repeatedly. The adoption of *abstract concepts* (e.g. Action in subsection 3.3.3) in the DSM layer allows to add increments in the DSML without difficulty and at a very low cost. Consider for instance the extensions of Action. One could add another *action* just by coding a simple extension, without the need of understanding anything about how the actions are plugged in the framework. The DSML can thus be augmented at these points even by developers that do not master the framework. Just by adding the small module, the new feature becomes ready to be used in the DSML.

Ensures consistency. When using conventional approaches, the consistency between the framework, the modeling language, and the code generator, can be easily broken. A code generator produces text, which is code that instantiates the framework. This code is not checked against compilation until the generator is tested with sample inputs. This brings consistency problems, since a change in the framework may introduce unnoticeable errors in the code that is produced by a not up-to-date generator. Consider the hook method fillMenuBar(..) of Application of the example framework. If, for instance, this method changes its signature, a code generator programmed for overriding the former version of the hook method would not manifest its inconsistency with respect to the framework. The inconsistency would only be noticed when generating code from an application model that involves the hook method. More concretely, the error would be noticed during the compilation of the generated code. In contrast, in our approach, if a module defines an advice that is acting over a non-existent method, one gets a compile-time warning that informs that the module is broken. Nevertheless, compilation errors also occur if the body of an advice is using inexistent framework elements.

Promotes composability and contributes to low change impact. In general, code generators are not implemented in cohesive and composable modules. This implies that adding increments to the generator involves modifications in existing generator modules. For instance, recall the example given in Section 3, and suppose that there is no support in the DSML for including *actions* in a *menu*. In the case of having a conventional code generator, the support for generating code for including the actions would require modifications in the generator part that handles the instances of the meta-class Menu. Namely, code that processes composite instances of the meta-class MenuItem, in order to generate the code that plugs the actions. In our approach, a module encapsulates the concept (as in Subsection 3.3.4), consisting of a non-invasive increment to the DSM layer. Moreover, DSM modules can be composed to form different variants of the DSML. One can make combinations of modules and obtain different DSMLs without needing to understand any internals of these modules.

5.2 Disadvantages

Without a supporting methodology, a domain engineer may take some time to master the development of DSM modules, due to their different design style. However, the use of aspects to manage variability has been applied successfully in another approach [10], reinforcing our belief that aspects are useful when combined with frameworks. Despite the learning issues, the main disadvantage of our approach is related with flexibility, which we detail next.

Uniform representation of modeling constructs. The mechanisms to represent the meta-model elements in the DSM layer are not very flexible. Each modeling construct has a single way of being represented. For instance, a meta-class (i.e. concept) must be represented in a module and the attributes must be represented in a constructor of that module. Although different ways to represent a same modeling construct could be contemplated, we found no practical significance in doing so, and simplicity would be compromised. Perhaps when trying the approach on more frameworks, if DSM modules are found "inelegant", this option could be revised. In conventional DSM approaches, meta-classes, attributes, etc, can be mapped freely, in a sense that it is up to domain engineers to decide how to map modeling elements to implementation elements. Therefore, the automation gains of our approach compromise flexibility to a certain extent.

Generation of other artifacts. Some frameworks require that applications have to provide descriptors (e.g. in XML) in addition to code. The information contained in those descriptors can also be represented in the DSML. Currently, our approach does not address the mapping of DSML concepts to different artifacts other than the framework instantiation code. However, given that the DSML is defined independently in a standard format, there is no obstacle in having a separate generator that processes the same applications models with the purpose of generating other artifacts.

6 Case Study

The proposed approach for developing the DSM layer went through an iterative process where its applicability was checked against two frameworks, JHotDraw [19] and Eclipse RCP [12]. This section focuses on the latter, which is definitely more complex and can be considered an industrial-strength framework. Eclipse RCP is a framework for building stand-alone applications based on Eclipse's dynamic plug-in model and UI facilities, such as menus, action bars, listeners, tree views, table views and controls (e.g. buttons, labels, etc).

We developed an independent DSM layer for Eclipse RCP, without modifying or inspecting its implementation internally (i.e. only the interfaces had to be known). Figure 10 shows a subset of the DSM modules that were developed, within a project that imports the required libraries for an RCP-based application. We were able to handle the main application concepts, and fully executable code could be successfully generated from the application modules. Although we only present a few illustrative framework concepts, the case study was more extensive, where the DSML had a total number of concepts that was more than the double of the concepts presented here.

Programming in AspectJ is effectively programming in Java plus aspects. In order to give an idea of the size of a DSM layer, Table 1 shows the number of lines of code (LOC) of Java and AspectJ of the modules associated with Eclipse RCP concepts, covering those that are visible in the meta-model of Figure 10. Concepts that inherit from other concepts are represented nested under the super concept. Recall that the whole tool support for DSM covering these concepts relies solely on the DSM modules. The AspectJ primitives that were necessary to implement these modules were not in any case more complex than the ones used throughout Subsection 3.3. From the data in the table, we can see that about one fifth of the code uses to AspectJ primitives, while the rest is regular Java. Notice that there are concepts (mostly sub-concepts) that do not use any AspectJ primitives (the aspect keyword was not considered as such). Most of these sub-concepts are as simple as the ExitAction that is given in Subsection 3.3.3.

Table 1. LOC for Eclipse RCP concepts.

Concept	J+AspJ	Concept	J+AspJ
RCPApplication	71+0	*GUI Element*	5+6
Action	14+7	CompositeEl.	18+6
ExitAction	8+0	*ControlEl.*	7+0
OpenAction*	22+0	Text Box	19+0
ViewPart	24+15	*Button*	35+0
Perspective	19+7	PushButton	9+0
ViewPlace	22+19	ToggleButton	7+0
MouseAction	12+13	Checkbox	9+0
MouseClick	4+1	*ViewerEl.*	7+0
MouseRelease	4+1	TableViewer	8+0
		TreeViewer	9+0
* - open	**Total Java:** 333	**Total AspectJ:** 75	**Total:** 408

Covering the same concepts, and actually using the same meta-model, we have experienced to develop a (conventional) code generator in Java. Although this code generator was slightly bigger in terms of LOC (approximately 10%), we consider the difference between the sizes not significant. However, it is worth to emphasize that the DSM layer is easier to build and evolve, due to the reasons pointed out in Section 5. Moreover, the DSM layer also embodies the meta-model definition, which in the conventional approach is defined separately.

Eclipse RCP by itself cannot be considered a product-line. However, it is definitely a platform which product-lines can be built on top of. Typically, a product-line would have a narrower scope, with its own *actions*, *view parts*, and *perspectives*, which could be combined to obtain different products. Therefore, we consider this case study as relevant in the context of product-lines.

7 Related Work

Approaches based on *feature-oriented programming* (FOP), such as AHEAD [3], CaesarJ [14], or aspectual-mixin layers (AML) [2], propose systems to be constructed using high-cohesive feature modules, enabling different systems to be obtained by defining a feature configuration. If we consider a feature model to be a DSML, variants of these systems can also be generated in a straightforward way from a valid configuration of the feature model. Our approach is different in the sense that the conceptual models that can be represented in the DSM layer are more expressive when comparing to the feature model that a system built using FOP can represent. The variants of a system built using FOP are a set of pre-planned applications within a finite configuration space, while frameworks usually support the development of an infinite set of applications by composing both default and application-specific components (e.g. Eclipse RCP). Issues regarding the expressiveness of DSMLs are discussed in more detail in [5].

The work in [11] presents a generative technique where code that specializes abstract aspects is generated from feature model instances. The fact that abstract aspects are being specialized by code generation is common to our approach, but the way to develop these aspects is considerably different, and both the concepts (feature model) and the mappings are defined manually. The issue that was raised regarding the expressiveness of feature models applies to this approach, too.

In [1], the authors present the idea of having a Framework-Specific Modeling Languages (FSMLs) with support for round-trip engineering. As well as in our approach, code generators do not have to be developed. The FSMLs are defined manually in the form of feature models, and code generation relies on embedding mappings to the framework elements in the FSML concepts. The use of a FSML supports a development paradigm where an application is obtained by generating code that is meant to be completed manually (if needed). Our approach does not intend to support round-trip engineering. Instead, our option was to have a clear separation between generated and manually written code, where the former is not intended to be manipulated or understood in any case, following the DSM philosophy of raising the abstraction level by complexity hiding.

MetaEdit+ [13] and Microsoft DSL Tools [8] are two examples of commercial language workbenches for developing conventional tool support for DSM. In contrast to these approaches, ours encodes the DSML as part of the framework. Domain engineers are not required to master meta-modeling nor code generation technologies, but instead, they have to use aspect-oriented programming. Due to the reasons pointed out in 5.2, these approaches, as well as FSMLs, are more flexible in what concerns the mapping and what is generated from the models (e.g. XML files). While our approach is less flexible, there are automation gains and tool support for DSM relies only on the framework implementation.

8 Conclusion

In this paper we presented an approach for enhancing an object-oriented framework with a DSM layer based on aspect-oriented programming, encoding DSMLs for generating framework-based applications. We implemented a prototype tool and we performed a case study on the Eclipse RCP framework. Our approach is suitable for product-lines implemented as object-oriented frameworks. A DSM approach based on what we propose allows the DSM tool support to be automatically up-to-date with a product-line platform, at the cost of having the additional DSM layer. After carrying out the research presented in this paper, we strongly believe that it is possible to realize DSM solutions that rely solely on the framework implementation. Adopting such an approach is well motivated by the difficulty of developing and maintaining code generators, the iterative nature of framework-based development, the unavoidable framework evolution, and obviously, the benefits of building applications using a DSML.

9 Acknowledgements

We would like to thank Juha-Pekka Tolvanen, Tommi Mikkonen, and anonymous reviewers for their valuable comments on earlier versions of this paper.

References

[1] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. In *MoDELS*, 2006.

[2] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceedings of the 28th international conference on software engineering*, 2006.

[3] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, 2003.

[4] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., 2000.

[5] K. Czarnecki. Overview of generative software development. In *UPP*, 2004.

[6] DSM Forum. Workshops on domain-specific modeling, 2001-2006. http://www.dsmforum.org/DSMworkshops.html, 2007.

[7] Eclipse Foundation. Eclipse platform and projects. http://www.eclipse.org/projects, 2007.

[8] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Frameworks, Models and Tools.* John Wiley and Sons, 2005.

[9] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 2001.

[10] U. Kulesza, V. Alves, A. F. Garcia, C. J. P. de Lucena, and P. Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *ICSR*, 2006.

[11] U. Kulesza, C. Lucena, P. S. C. Alencar, and A. Garcia. Customizing aspect-oriented variabilities using generative techniques. In *SEKE*, 2006.

[12] J. McAffer and J.-M. Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.

[13] MetaCase. MetaEdit+ tool. http://www.metacase.com.

[14] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *ACM Conference on Foundations of Software Engineering (FSE-12)*, 2004.

[15] OMG. *Meta Object Facility Specification (MOF) 1.4*. OMG, 2002.

[16] D. Roberts and R. E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.

[17] A. L. Santos. Automatic support for model-driven specialization of object-oriented frameworks using ALFAMA. In *OOPSLA'07 Demonstrations Track*, 2007.

[18] A. L. Santos, A. Lopes, and K. Koskimies. Framework specialization aspects. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[19] SourceForge. JHotDraw framework. http://www.jhotdraw.org, 2006.