Automating Domain-Specific Modeling Support for Object-Oriented Frameworks

André L. Santos[a], Kai Koskimies[b], Antónia Lopes[a]

[a] *Department of Informatics, Faculty of Sciences, University of Lisbon, Campo Grande, 1749-016 Lisboa, Portugal*
[b] *Department of Software Systems, Tampere University of Technology, P.O.BOX 553, FIN-33101 Tampere, Finland*

Abstract

Domain-Specific Modeling (DSM) support for instantiating object-oriented frameworks has proved to be an effective way of improving the productivity in framework-based development. However, developing and evolving a Domain-Specific Modeling Language (DSML) and its code generator is typically a difficult task. In this paper, we propose a new approach to build DSM support, based on extending frameworks with an additional aspect-oriented layer that encodes a DSML and eliminates the need of implementing code generators. DSM support is automated by a generic language workbench, which is used to build the DSML and to generate framework-based applications from models described in that DSML.

Key words:
Domain-specific modeling, object-oriented frameworks, aspect-oriented programming, code generation, language workbenches.

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.

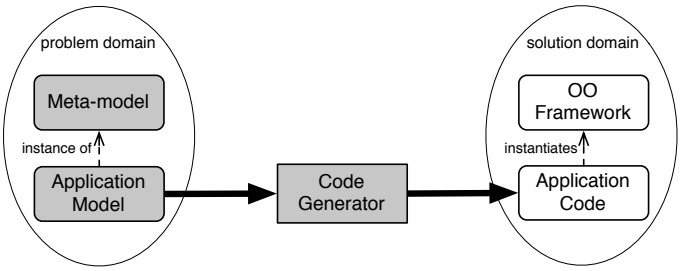
Learning how to correctly use a non-trivial framework is a difficult and time-consuming activity [20]. In order to help application engineers to overcome this obstacle, domain engineers may develop a *Domain-Specific Modeling* (DSM) [6] solution. By doing so, the abstraction level of application development 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 DSM support 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 can be defined by a *meta-model*, whereas *(application) models* are descriptions in that language representing instances of the meta-model. Meta-models and code generators can be developed using *language workbenches* [10]. Language workbenches are tools targeted for developing domain-specific development environments, such as MetaEdit+ [18], Microsoft DSL Tools [12], or Eclipse-based technologies [8] .

Figure 1 illustrates the conventional approach for having DSM support for generating framework-based applications. In the problem domain side, the meta-model describes domain concepts, whereas an application model describes instances of those concepts. In the solution domain side, the object-oriented framework provides a reusable and adaptable system which is instantiated by application-specific code.

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 as the framework evolves. 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 DSM solution challenging.

The difficulty of building and maintaining a DSM solution 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 simpler the mapping is, the easier it will be to implement and evolve the code generator. As a matter of fact, DSM approaches are pointed out to be particularly suited to *black-box* frameworks given that code generation is confined to glue code that composes default components (these solutions are also referred to as *visual builders*,



Email addresses: andre.santos@di.fc.ul.pt (André L. Santos), kai.koskimies@tut.fi (Kai Koskimies), mal@di.fc.ul.pt (Antónia Lopes)

e.g. [24]). However, even though the rules for generating such glue code are typically fairly straightforward, they cannot be inferred automatically on the basis of the framework code, because the mechanisms for instantiating the framework-provided concepts are not explicitly represented in the framework implementation. A code generator has thus to be developed for implementing this mapping.

In this paper we propose a technique for developing DSM solutions solely by enhancing a framework with an additional layer that explicitly encodes the DSML, which we refer to as the *DSM layer*. The realisation of the DSM layer relies on our previous work [27], which proposes a technique based on aspect-oriented programming for modularizing framework *hot-spots* through *(framework) specialization aspects*. These are reusable aspect modules that serve the purpose of developing framework-based applications. By extending a framework with specialization aspects, framework-based applications can be defined at a higher abstraction level. Such abstraction raise shortens the gap between application models and the code of framework-based applications, up to a point where the mapping between them becomes straightforward.

The proposed technique exploits the close relation between application models and application code based on specialization aspects. Comparing to the state-of-the-practice, the approach presented in this paper embodies a major strategic difference, given that we propose frameworks to have a "built-in" DSML encoded by the DSM layer. A DSM layer is composed of several specialization aspects, which are annotated with additional meta-data for enabling that both the meta-model and the mapping between application models and framework-based code can be inferred. Domain engineers are able to effectively extend a framework's implementation with the encoding of a DSML that can be directly used to generate framework-based applications, without the need of developing any additional artifact. We shall show that this can be achieved by means of a generic language workbench, which on the one hand, extracts meta-models from DSM layers, while on the other hand, is capable of processing instances of those meta-models for generating application code. The approach alleviates maintenance problems that are typically associated to the development of code generators.

The proposed language workbench was implemented in an Eclipse-based [8] tool named ALFAMA [25]. The tool supports DSM layers written in AspectJ [7] and defines meta-models in EMF (Eclipse Modeling Framework) [9]. Given the importance and the common need to have *open variation points* [13], our approach also contemplated a mechanism for integrating manual code with code generated from models.

As a proof-of-concept, we have implemented the proposed DSM layer and used ALFAMA to develop a DSML for generating applications based on the Eclipse RCP framework [17]. Eclipse RCP is a framework for developing stand-alone applications based on Eclipse's dynamic plug-in model and UI facilities. A DSML covering the main framework features was successfully developed.

The paper proceeds as follows. Section 2 presents an overview of our approach. Section 3 explains how conventional framework hot-spots can be represented in a modular way in terms of specialization aspects. Section 4 addresses the development of the DSM layer using specialization aspects. Section 5 explains how DSMLs can be automatically derived from DSM layers. Section 6 presents the ALFAMA tool. Section 7 compares the proposed approach with conventional tool support for DSM. Section 8 describes the case study on Eclipse RCP. Section 9 discusses related work, and Section 10 concludes the paper.

2. Approach Overview

This section presents an overview of our approach (see Figure 2). The approach relies on a language workbench that automates the DSM support at the expense of some new development activities. In contrast with the conventional approach depicted in Figure 1, domain engineers have to develop a DSM layer in addition to the framework, while they are relieved of both implementing a code generator and defining the DSML concepts separately (i.e. externally to the framework implementation). The following summarizes how the approach affects the two developer roles:

- *Domain engineers* develop the DSM layer, from which the language workbench extracts the meta-model that defines the DSML. There is no need to develop a code generator for building the DSM support. The language workbench is generic, in the sense that it can be used for multiple frameworks, as long as the DSM layer is developed in the supported programming language (in this work, AspectJ for Java frameworks) and according to certain rules.
- *Application engineers* develop application models using the extracted DSML. These models are given as input to the language workbench, which transforms them into code based on the DSM layer. From a user perspective, application models are described as if conventional DSM support was being used, given that it is transparent to the DSML user how the code of the framework-based application is generated.

The DSM layer includes a formal representation of the meta-model embedded in its modules, using modeling constructs that are equivalent in terms of expressiveness to those that can be found in existing meta-modeling technologies. When implementing the ALFAMA prototype as a proof-of-concept of the

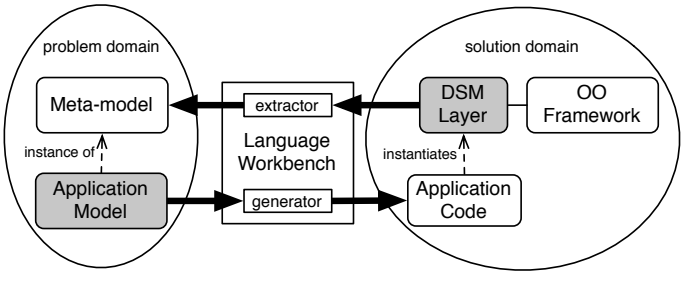


Figure 2: Proposed approach for realizing DSM support.

proposed approach, our option was to consider the modeling constructs that are available in EMF [8]. EMF is a Java implementation of the Meta-Object Facility (MOF) [21], a standard for defining modeling languages. Despite some small differences, the meta-models defined in EMF have equivalent expressiveness to those that can be defined using commercial tools such as MetaEdit+ [18] and Microsoft DSL Tools [12]. 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.

3. Framework Specialization Aspects

In this section, we present the essentials of specialization aspects and explain how the different mechanisms for adapting object-oriented frameworks can be captured with specialization aspects, and how to implement them in AspectJ [7] (additional details can be found in [27]).

3.1. Overview

The traditional development of applications based on frameworks takes place at certain extension points called *hot-spots* [22], which can be adapted for implementing application-specific features. Hot-spots usually involve several classes and it is common that a single class participates in several hot-spots. Specialization aspects were proposed in order to support the representation of hot-spots in a modular way, avoiding the scattering and tangling phenomena.

As defined in [14], an aspect is a concern that cross-cuts the primary modularization of a software system. Like classes, aspects can be abstract and related through inheritance relationships. A *(framework) specialization aspect* is an abstract aspect *A* that modularizes a conventional framework hot-spot, in such a way that variation in this point can be achieved through the definition of concrete subaspects of *A*. These concrete aspects are designated by *application aspects*.

Specialization aspects are intended to be developed for a *base framework*, which is not modified. As illustrated in Figure 3, a set of specialization aspects defines an additional layer on top of an existing framework. This layer can be seen as a higher level interface for developing framework-based applications.

There may exist several application aspects inheriting from the same specialization aspect, representing multiple adaptations of the same hot-spot within the same application. Like regular aspects, specialization aspects may have mutual dependencies and can be structured with aspect inheritance. However, application aspects can be composed exclusively through static references, i.e. an application aspect can be composed with another application aspect only by referencing its module.

Conventionally, the adaptation of a hot-spot in a framework-based application involves different adaptation mechanisms. For instance, the adaptation of a certain framework hot-spot may require to extend a class and override a hook method, while another hot-spot may require to compose a parameterized object with another object exposed by the framework. *Hook method overriding* and *object composition* are the two fundamental mechanisms of hot-spot adaptation [23]. While the former is associated with white-box adaptation, the latter supports black-box adaptation. The following subsections illustrate through examples, how specialization aspects are able to capture hook method overriding (Subsection 3.2), how to structure specialization aspects using aspect inheritance (Subsection 3.3), and how specialization aspects are able to capture object composition (Subsection 3.4). In each case, a specialization aspect that wraps the conventional hot-spot is presented, as well as an example of a concrete adaptation through an application aspect.

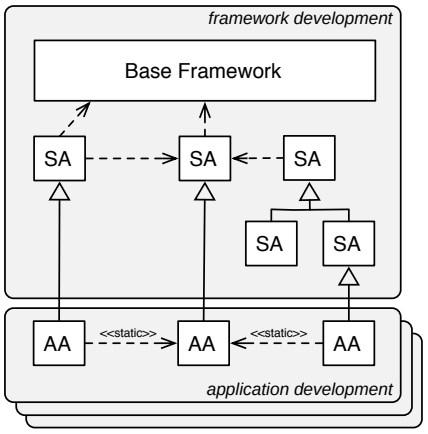


Figure 3: Specialization aspects (SAs) and application aspects (AAs).

3.2. Hook method overriding

Inheritance is a common adaptation mechanism in frameworks, and solutions that apply the Template Method pattern [11] occur very often. These cases require application classes to inherit from a framework class, which is typically abstract, and to override hook methods. As an example, consider the following framework class that can be specialized and adapted by overriding the hook method hook(..), which serves the purpose of plugging an arbitrary number of objects of type Item.

```
1 abstract class BaseClass {
2   ...
3   /* optional override by subclasses */
4   void hook(List<Item> list) {  };
5 }
```

Suppose that the class Item, presented below, is also part of the framework. This class defines that a name is required for the construction of a new item and it offers a method addRelation(..) that supports the addition of a relationship between two items (the owner object and anotherItem).

```
1 class Item {
2   ...
3   Item(String name) { ... }
4
5   void addRelation(Item anotherItem) { ... }
6 }
```

In a framework-based application developed in the conventional way, in order to use BaseClass, we would have to define

a subclass. This subclass may provide a new implementation of hook(..), using the exposed object of type List<Item> (container) for plugging an arbitrary number of objects. There are two hot-spots involved here — *H1* and *H2*. If one wishes to have an application class based on BaseClass without plugging any objects, an empty class as shown below would be defined (*H1*). If one wishes to have an application class based on BaseClass with one or more plugged objects, then the application class would override the hook method hook(..) accordingly (*H2*).

```
1 class ApplicationClass extends BaseClass {
2
3 }
```

After having defined an adaptation of *H1*, the adaptation of *H2* requires to modify the module associated with the adaptation of *H1* (ApplicationClass). The fact that adaptation of hot-spots cannot be realized in a compositional way is a source of complexity in the conventional development of framework-based applications. Although the given example is very simple, recall that a real framework may have tens of hook methods pertaining to different hot-spots.

As mentioned previously, specialization aspects are a means for hot-spot modularization. When having hot-spots based on hook method overriding, the solution using specialization aspects is based on having a separate abstract aspect that "fills in" the behavior of hook methods, so that the hot-spot can be adapted in a compositional way.

Figure 4(a) presents a specialization aspect for wrapping hot-spot *H2*. In order to make the example richer and more interesting, it was assumed that, in the framework at hand, correct adaptation of the hot-spot is confined to the plugging of objects of type Item, created for that purpose, with application-specific names. The specialization aspect ItemAspect is "parameterized" with a name (lines 4-6) and includes an abstract pointcut appclass() (line 9) that is meant to match an extension of BaseClass (due to the primitive within(BaseClass+), in line 12, the advice may only take effect in the subclasses of BaseClass). Moreover, ItemAspect defines an advice that modifies the behavior of the hook method hook(..) of the class matched by appclass(), adding an object of type Item to the List<Item> object (line 15), using the method create() to create that object. The method create() uses the attribute name for creating the Item object (lines 18-20). The primitive args(list) together with the declaration after(List<Item> list) enables to gain access to the parameter of type List<Item>.

Figure 4(b) presents an application aspect defining an example adaptation of hot-spot *H2* in a given framework-based application. ItemA is an application aspect that inherits from ItemAspect and defines the pointcut appclass() in ApplicationClass (introduced before). Moreover, it specifies that the name of the plugged object is "Item A". If the application needs other items to be plugged in that point, other application aspects extending ItemAspect would have to be defined accordingly.

Notice that, in this solution, Item objects can be cohesively plugged in the application using an application aspect, and no modification or inspection of ApplicationClass is required.

(a) Specialization aspect

```
1  abstract aspect ItemAspect {
2    String name;
3
4    ItemAspect(String name) {
5      this.name = name;
6    }
7
8    /* to match an extension of BaseClass */
9    abstract pointcut appclass();
10
11   after(List<Item> list) :
12     within(BaseClass+) && appclass() &&
13     execution(void hook(List<Item>)) &&
14     args(list) {
15       list.add(create());
16     }
17
18   Item create() {
19     return new Item(name);
20   }
21 }
```

(b) Application aspect

```
1 aspect ItemA extends ItemAspect {
2   ItemA() {
3     super("Item A");
4   }
5
6   pointcut appclass() : target(ApplicationClass);
7 }
```

Figure 4: Specialization and application aspects for hook method overriding.

Moreover, the behavior of the hook method hook(..), which would have to be defined by applications in the conventional adaptation of the hot-spot, is no longer visible to application developers. Finally, the type List<Item> is no longer relevant to framework-based applications.

Now suppose that in the framework we have been considering, objects of certain subtypes of Item, such as SubItem presented below, can also be plugged in a framework-based application.

```
1 class SubItem extends Item {
2   SubItem() {
3     super("default name");
4   }
5 }
```

In order to support the plugging of subtypes of Item, we would have to define a specialization aspect similar to ItemAspect for each of those subtypes. Such solution would lead to a significant amount of code duplication. However, in order to maximize reuse, a set of related specialization aspects may be structured using inheritance, as explained in next subsection.

The objects that are plugged in the application through an application aspect are instantiated within the aspect. Most likely, there is need to enable these objects to collaborate with other objects. As we will show in Subsection 3.4, specialization aspects are also capable of modularizing such collaborations.

3.3. Structuring specialization aspects with inheritance

In order to avoid code duplication and to promote extensibility, inheritance may be used to structure related specialization aspects. If there are specialization aspects whose advices are similar, the commonality may be factored out to a specialization aspect from which they inherit. For instance, in the running example, a specialization aspect for supporting the plugging of objects of type SubItem would be similar to ItemAspect. In order to avoid duplication of code, we can define a specialization aspect that extends ItemAspect and overrides the method create(), so that an instance of SubItem is returned. Figure 5(a) presents an implementation of this solution. Given that the creation of SubItem does not require a name as input, the application aspects that extend SubItemAspect only have to define the pointcut appclass(). Figure 5(b) presents an example of an applicatiton aspect— SubItemB, that plugs an object of type SubItem in the class ApplicationClass (defined previously). An application that, in addition to class ApplicationClass, includes the application aspects ItemA and SubItemB, is adapting hot-spot *H2*, plugging two different objects.

(a) Specialization aspect (inheriting from another specialization aspect)

```
1 abstract aspect SubItemAspect extends ItemAspect {
2   SubItemAspect() {
3     super(null);
4   }
5
6   Item create() {
7     return new SubItem();
8   }
9 }
```

(b) Application aspect

```
1 aspect SubItemB extends SubItemAspect {
2   pointcut appclass() : target(ApplicationClass);
3 }
```

Figure 5: Structuring specialization aspects with inheritance.

3.4. Object composition

Framework adaptation may also rely on compositions of objects that are instantiated in the code of a framework-based application. In this subsection, our aim is to show that the composition of these objects can also be achieved through the definition of application aspects inheriting from a specialization that captures the corresponding hot-spot.

In our running example, as shown in Subsections 3.2-3.3, the definition of application aspects that extend ItemAspect (directly or indirectly) gives rise to the creation of objects of type Item, which are plugged in the framework-based application. Additionally, it is possible to establish relations between objects of type Item through the invocation of method addRelation(..). The fact that the framework allows the definition of relations between pairs of objects of type Item that are plugged in an application, can be represented in terms of the specialization aspect RelationAspect, presented in Figure 6(a).

(a) Specialization aspect

```
1  abstract aspect RelationAspect {
2    Item sourceItem = null;
3    Item targetItem = null;
4
5    /* to match an extension of ItemAspect */
6    abstract pointcut sourceObj();
7
8    after() returning(Item item) :
9      within(ItemAspect+) && sourceObj() &&
10     execution(Item create()) {
11       sourceItem = item;
12       if(targetItem != null)
13         sourceItem.addRelation(targetItem);
14     }
15
16   /* to match an extension of ItemAspect */
17   abstract pointcut targetObj();
18
19   after() returning(Item item) :
20     within(ItemAspect+) && targetObj() &&
21     execution(Item create()) {
22       targetItem = item;
23       if(sourceItem != null)
24         sourceItem.addRelation(targetItem);
25     }
26 }
```

(b) Application aspect

```
1 aspect Relation extends RelationAspect {
2   pointcut sourceItem() : target(ItemA);
3   pointcut targetItem() : target(SubItemB);
4 }
```

Figure 6: Specialization and application aspects for object composition.

The specialization aspect has two abstract pointcuts sourceObj() (line 6) and targetObj() (line 17), which are intended to match the application aspects that plug the source and target object, respectively. The advice related to sourceObj() captures the instantiation of the Item object that is returned by create(), and assigns the variable sourceItem pointing to it (line 11). The advice related to targetObj() is analogous, but assigns the variable targetItem with the target object instead. The fact that the specialization aspect does not know which Item object is created first, requires the conditional invocation of addRelation(..) (lines 12-13, 23-24). However, addRelation() is executed only once given that there is a single invocation of create() application aspect. Figure 6(b) presents the application aspect Relation. The application, in addition to application aspects ItemA and SubItemB, also defines this aspect, which establishes the relation between the two objects plugged in the application by ItemA and SubItemB.

The described mechanism enables to encapsulate object compositions. In this way, one can implement a composition of objects, which are handled by other application aspects, without knowing where and how the objects are instantiated. Such compositions are an increment that does not modify or requires the inspection of other application aspects. When using conventional hot-spot adaptation, it is not possible to achieve this kind of encapsulation and modularity.

4. DSM Layer

This section addresses the development of the DSM layer. A DSM layer is framework-specific and it is developed by domain engineers. The DSM layer is composed of several annotated specialization aspects, which are referred to as DSM modules. While the specialization aspects are capable of modularizing the adaptation of hot-spots according to framework-provided concepts, the role of the annotations is to explicitly associate concepts to specialization aspects and relationships between them.

In the rest of this section, we proceed as follows. Subsection 4.1 presents the conceptual model on which the notion of DSM layer is based, namely the modeling constructs that can be used in the DSML definition. Subsection 4.2 introduces the concepts of an example framework fragment. These concepts are used in the running example throughout Subsection 4.3, which presents the implementation of the several DSM modules.

4.1. Modeling constructs

The modeling constructs that can be represented in the DSM layer are given in the conceptual model of Figure 7. Most of them are fairly equivalent to those that can be found in meta-modeling technologies, such as EMF or Microsoft DSL Tools.

A *concept* is identified by a *name* and may have several *attributes*, which have a primitive type and are also identified by a *name*. A concept may define *relationships* with other *target* concepts, which can be either *composite associations* or *directed associations*. When representing a composite association in the DSM layer, the target is the parent concept. Relationships have an associated *multiplicity* for restricting the number of links between concept instances. 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 and generated code. An *open concept* is a concept that represents an open variation point, where application-specific code can be added. In other words, open concepts give support to glue arbitrary behavior that is specific to a particular application and that is not covered by the DSML. An *accessible concept* is a concept whose instances may be accessed by instances of open concepts, enabling manually given code to access generated code.

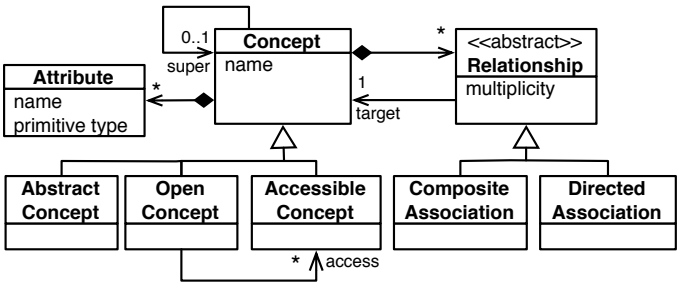


Figure 7: Modeling constructs for defining DSML.

4.2. 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 [17]. This case study is discussed in detail in Section 8. Through the rest of the paper, we shall use just a small simplified fragment of Eclipse RCP classes as a running example.

The framework-provided concepts that will be used in the DSM layer are defined in the meta-model given as a class diagram in the left-hand part of Figure 8. The meta-model exemplifies each of the modeling constructs given in Figure 7. Each class represents a concept. An *RCP application* has initial window size given by *width* and *height*. It may contain several *actions* (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 can be given manually. An *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 RCP application may indirectly contain *tables*, which are contained in instances of 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 8 we can see an object diagram representing an example instance of the meta-model (i.e. an application model). It describes an RCP application with width equals to 400 and height equals to 200. The application has two actions, the exit action, and an open action that accesses the table. It also has a menu "M1" with a menu action that executes the exit action.

Both the meta-model and the application model are used throughout the next subsection. The former is what is to be

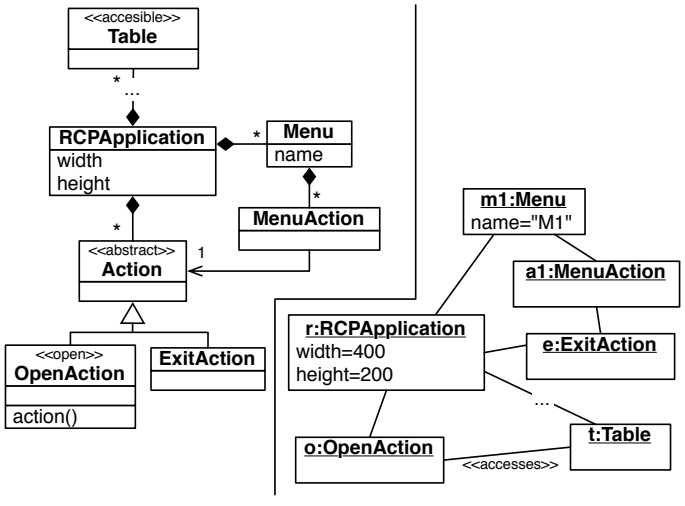


Figure 8: Example framework: meta-model (left) and application model (right).

expressed in the DSM layer, while the latter is used for exemplifying the code generation.

4.3. DSM modules

This subsection explains how to implement the DSM modules making use of the different modeling constructs described in Figure 7. A DSM module is a specialization aspect with additional annotations on its elements, namely on module declaration, constructor signature, and pointcuts. 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. 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. Recall that such a meta-model is to be 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 introduced previously.
- The bottom part shows the application code that realizes the given fragment of the application model. Such code is meant to be generated by the language workbench (hence, not meant to be coded manually).

4.3.1. 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 RCP application has to implement the framework interface IApplication, which has a method for plugging the *menus* and another one for plugging the *actions*. Figure 9 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.

4.3.2. 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.

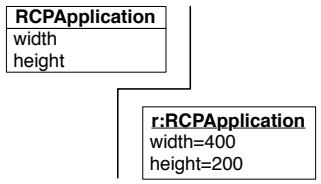
Figure 10 presents the DSM module that handles the concept *menu*. A menu is part of an *RCP application* and can be included by defining the pointcut application() on an extension of RCPApplication (previous module). The multiplicity defines that each application can have several menus. The advice introduces the necessary behavior for plugging the menu in the application.

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

 final void fillMenuBar(IMenuManager menuBar) { }

 final void makeActions(IBarAdvisor barAdvisor) { }

 /* ... */
}
```



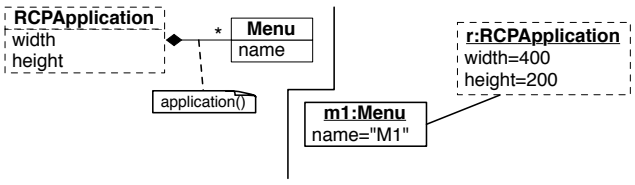

```
class R extends RCPApplication {
  R() {
    super(400, 200);
  }
}
```

Figure 9: DSM module defining a concept with attributes.

```
@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 M1 extends Menu {
 M1() {
  super("M1");
 }
 pointcut application() : target(R);
}
```

Figure 10: DSM module defining a composite association.

4.3.3. 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.

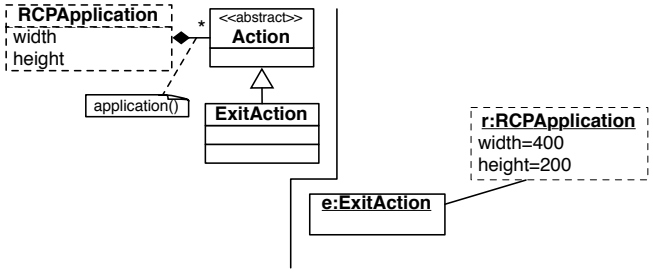
Figure 11 presents the DSM module that handles the concept *action* (Action). It is similar to the previous example. However, it has an abstract method that its extensions should define. The figure also presents the DSM module for handling the concept *exit action* (ExitAction), which inherits from Action, 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 E extends ExitAction {
   pointcut application() : target(R);
}
```

Figure 11: DSM module defining an abstract concept and DSM module inheriting from it.

4.3.4. 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 12 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.

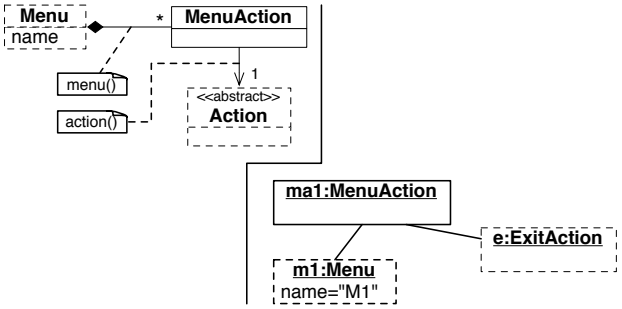
```
@Concept
abstract aspect MenuAction {
 private IAction action;

 @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 MA1 extends MenuAction {
   pointcut action() : target(E);
   pointcut menu() : target(M1);
}
```

Figure 12: DSM module defining a directed association.

4.3.5. 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 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 @AccessibleConcept, and the accessible object can be defined by annotating a method with @AccessibleOb-

ject. This implies that the object returned from that method is the accessible object.

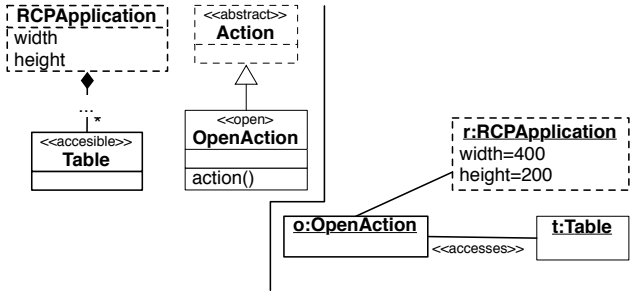
Figure 13 presents a DSM module for handling the open concept *open action* (OpenAction), as an extension of Action. The method action(), which defines the action behavior, is declared as being open. The figure also presents a DSM module for handling the accessible concept *table*. When generating the code from an instance of an open concept, two modules are obtained. One module is hidden from application engineers (in this case, O_Adapter) and contains code that can be generated from the model, while the other module is exposed to application engineers (in this case, O) 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 changing or understanding the latter.

```
@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 T extends Table {
   /* ... */
}
```

```
abstract aspect O_Adapter extends OpenAction {
 pointcut application() : target(R);

 after() returning(TableViewer t) :
  execution(TableViewer createTable()) && target(T) {
   O.table = t;
  }
}
```

```
aspect O extends O_Adapter {
 /* automatic */
 static TableViewer table;

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

Figure 13: Open and accessible concepts.

5. Language Workbench for Building and Using DSMLs

The previous section explained how domain engineers can develop the DSM layer. This section shows how the DSM support for the object-oriented framework can be automated having a DSM layer and a language workbench. As explained in Section 2, recall that domain engineers use the proposed language workbench for extracting a meta-model from the DSM layer. On the other hand, application engineers use the language workbench to generate application code from an application model.

A meta-model is usually an artifact that is described in a specific technology for defining meta-models, such as EMF or Microsoft DSL Tools. Such a technology represents a meta-meta-model, while the meta-models that are developed using that technology represent instances of the meta-meta-model. For instance, Figure 7 represents a meta-meta-model that describes the concepts that can be used to build a meta-model, and the left-hand side of Figure 8 represents a meta-model that instantiates the concepts given in Figure 7.

In order to have a common infrastructure for supporting open and accessible concepts, it is convenient to have a common meta-model that every extracted meta-model extends. The common meta-model is also useful to support the common fea-

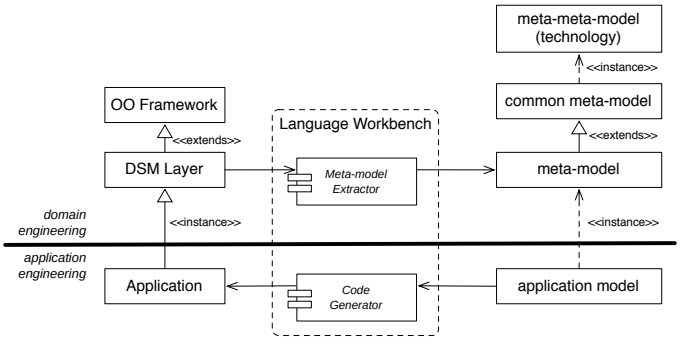


Figure 14: Components of the proposed language workbench.

tures within DSM environment provided by the language workbench. Figure 14 gives an overview of the language workbench with respect to domain and application engineering, depicting the role of the *meta-model extractor* and the *code generator* components. Subsection 5.1 presents the common meta-model, Subsection 5.2 details the extractor component, and Subsection 5.3 details the code generator component.

5.1. Common meta-model

The common meta-model is very thin, and it is composed of four concepts: Concept, AccessibleConcept, Access, OpenConcept. All elements of the extracted meta-models inherit directly or indirectly from Concept. An accessible concept is a concept that can be accessed by open concepts, and thus, the open concept may contain several Access objects to accessible concepts. A variable in the open module with name given by varname is going to point at the accessible object. In Figure 15 we can see the classes of the common meta-model depicted in gray, together with the meta-model used in the previous section depicted in white.

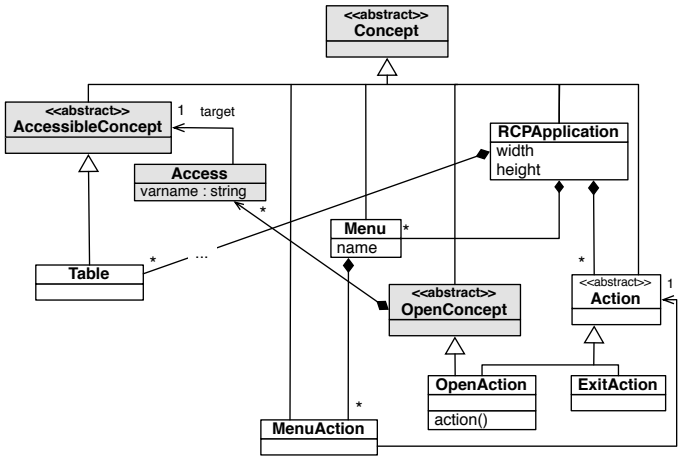


Figure 15: Common meta-model (in gray) which is extended by all the meta-models, and the example meta-model of Section 4 extending it (in white).

5.2. Meta-model extractor

The meta-model extraction is a straightforward and lightweight process that can be executed in one shot. When developing the DSM layer, domain engineers can constantly perform extractions of the meta-model to verify how the DSML is taking shape. Given the one-to-one mapping of meta-model classes and DSM modules, the graphical visualization of the meta-model is also a good way to have an overview of the DSM layer and use it to navigate through its implementation.

Figure 16 presents the algorithm in pseudo-code for obtaining a meta-model from a DSM layer. The algorithm is rather simple, since all the necessary information is explicitly represented in the DSM modules. Each module will have a corresponding class. If the module does not inherit from another module, its class will inherit from Concept, OpenConcept, or AccessibleConcept, according to its type. Otherwise, its class will inherit from the class corresponding to the module from which it inherits. In the case of open concepts, the signatures of their open methods are attached to their class. Composite and directed associations are defined according to the information in the annotations.

```
Input: DSM Layer (set of Module)
Output: Meta-model (set of Class)
foreach Module dmod do
    Create Class c (getName(dmod));
    c.setAbstract(hasAbstractConceptAnnotation(dmod));
    if hasOpenConceptAnnotation(dmod) then
        c.addSuperClass(OpenConcept);
        c.attachAnnotation(getOpenMethods(dmod));
    end
    if hasAccessibleConceptAnnotation(dmod) then
        c.addSuperClass(AccessibleConcept);
    end
    foreach Attribute att in getAttributesAnnotation(dmod) do
        c.addAttribute(att.getType(),att.getName());
    end
end
foreach Class c do
    Module dmod = getModule(c);
    if extendsModule(dmod) then
        c.addSuperClass(getClass(dmod.getSuper()));
    end
    if hasNoSuperClass(c) then
        c.addSuperClass(Concept);
    end
    foreach PartOfAnnotation ann in
    dmod.getPartOfAnnotations() do
        Class parent = getClass(ann.getConcept());
        Create Association ca (c, ann.getMultiplicity(),
        ann.isOrdered());
        ca.setContainment(true);
        ca.addAnnotation(ann.getPointcut());
        parent.addAssociation(ca);
    end
    foreach AssociationAnnotation ann in
    dmod.getAssociationAnnotations() do
        Class target = getClass(ann.getConcept());
        Create Association a (target,ann.getMultiplicity());
        a.setContainment(false);
        a.addAnnotation(ann.getPointcut());
        c.addAssociation(a);
    end
end
```

Figure 16: Algorithm for obtaining a meta-model from the DSM layer.

5.3. Code generator

Throughout Subsection 4.3 several fragments of application models were given, together with the corresponding application aspects that realize those model fragments. As mentioned before, the code of the application aspects can be obtained automatically. From the examples of Subsection 4.3, one can observe that the given set of application aspects has, in fact, a very close relation with the given application model. Moreover, the mapping between the objects of the application model to the code elements of the application aspects is uniform. By

uniform, we mean that modeling elements is always mapped in the same way, independently from the the situation. Every object of the application model is mapped to a module that inherits from the specialization aspect with the same name as the object's class. All the object's parameters are mapped to arguments to a call to the constructor of the specialization aspect. Every composite association between a parent and a child object is mapped to a pointcut definition in the child object. Every directed association between a source and target object is mapped to a pointcut definition in the source object. Having this uniformity of mappings, one can have a generic transformation definition applicable to every meta-model encoded by a DSM layer. This characteristic of the approach is what enables the code generator to be generic, i.e., applicable to every DSM solution developed using our approach.

Figure 17 presents the algorithm of the generic code generator in pseudo-code. For each object in the application model there will be a corresponding application aspect (AA). The name of the DSM module that an application aspect has to inherit from is equal to the name of the object's class. In the case of an open concept, an additional application aspect is created (the open module). Each attribute value is translated to an argument in the call to the constructor of the DSM module. Each containment association link results in a pointcut definition in the application representing the child object. If an open module has accesses, it is augmented with a static variable with a type equal to the type of the accessible object and a name equal to the value of the attribute varname, and accordingly, the application aspect is augmented with an advice definition for setting the static variable to point at the accessible object. Each non-containment association link results in a pointcut definition in the application aspect.

```
Input: Application model (set of Object)
Output: Application code (set of AA [application aspect])
foreach Object obj do
    Create AA aa;
    aa.setName(generateUniqueName(obj));
    aa.setSuperModule(getClass(obj).getName());
    if isOpenConcept(obj) then
        aa.setAbstract(true);
        Create AA openaa;
        openaa.setName(obj.filename);
        openaa.setSuperModule(aa);
    end
    foreach AttributeValue value in obj.getAttributeValues() do
        aa.addArgumentInSuperConstructorCall(value);
    end
    foreach AssociationLink ca in
    obj.getContainmentAssociationLinks() do
        Object child = ca.getTarget();
        if isAccess(child) then
            AccessibleObject ao =
            child.getTarget().getAccessibleObject();
            openaa.addStaticVariable(ao,child.varname);
            aa.addAdviceForStaticVariable(ao,child.varname);
        end
        else
            String pointcut = getPointcutFromAnnotation(ca);
            child.addPointcutDefinition(pointcut,aa.getName());
        end
    end
    foreach AssociationLink da in
    obj.getNonContainmentAssociationLinks() do
        Object target = da.getTarget();
        String pointcut = getPointcutFromAnnotation(da);
        aa.addPointcutDefinition(pointcut,getUniqueName(target));
    end
end
```

Figure 17: Algorithm for generating application code based on the DSM layer from an application model.

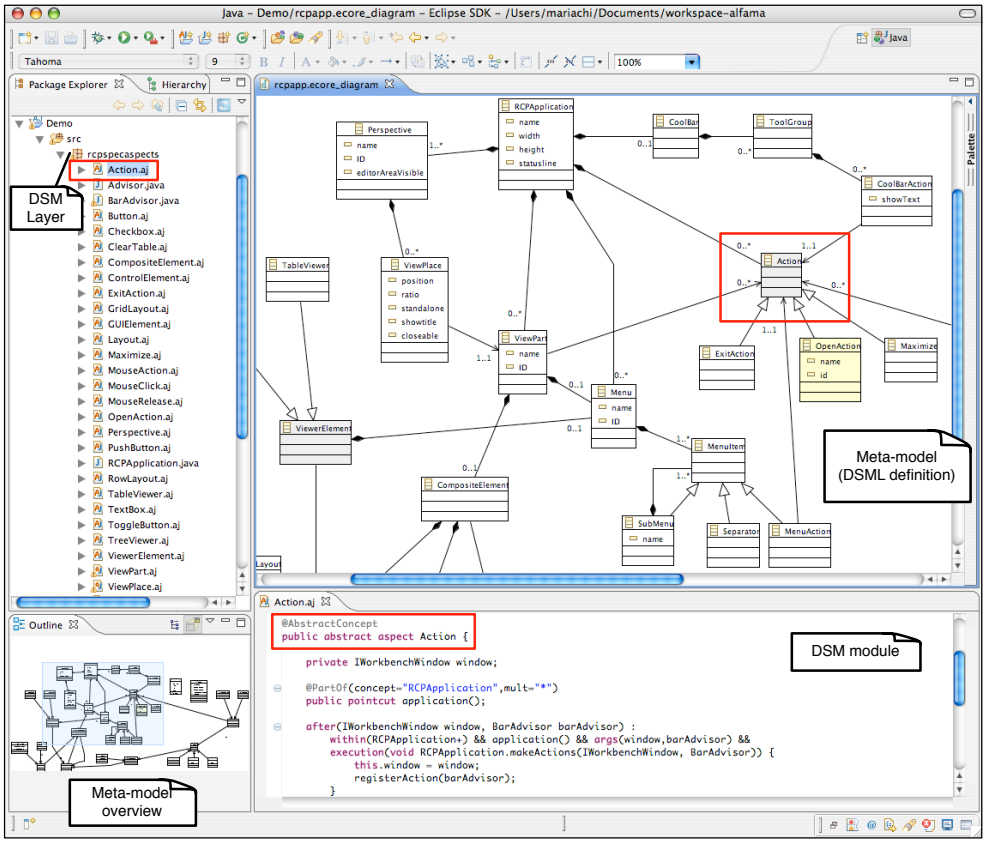
6. ALFAMA Tool

We implemented the proposed language workbench in a tool named ALFAMA [25]. The tool realizes the approach for developing DSML presented in this paper. The tool was implemented as a set of Eclipse [8] 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 EMF [8] models (i.e. meta-models). Application models can be edited in a default tree-view editor. Figure 18 shows two screenshots of the ALFAMA tool, considering the different perspectives of domain and application engineering.

From a domain engineering perspective, the DSM layer is implemented for the base framework using the regular programming language editors. The language workbench enables a one-shot extraction of the meta-model representation into an EMF model. Such a model can be visualized in a graphical way, as shown in the figure. Therefore, while implementing the DSM layer, the developer may constantly check how the DSML is taking shape. The graphical visualization of the DSML concepts is useful both in terms of understandability and complexity management. The DSML concepts in the graphical model can be traced to the DSM modules that represent them (one-to-one mapping). On the left-hand side of Figure 18 (a) we can see the package explorer with the package rcpspecaspects containing several DSM modules for the Eclipse RCP framework. For instance, in the bottom part of the right-hand side we can see the DSM module Action opened in the editor. On the right-hand side of the upper part we can see a diagram representing the meta-model that is extracted from the package containing the DSM layer.

From an application engineering perspective, the language workbench provides a generic tree-view editor for application models. Such editor represents the containment of concept instances by nesting the items in the tree nodes. Optionally, GMF [8] can be used independently for developing a concrete syntax for the DSML. The attribute values of the concept instances are edited in a property list. The tool enables a one-shot generation of the application aspects that implement the application. The generated code is divided in two sets: public and hidden. As the names suggest, the public set contains modules that are intended to be used externally or manipulated (open modules), whereas the hidden set contains modules that are not meant to be seen, touched, or understood by an application engineer. In

(a) Domain engineering environment



(b) Application engineering environment

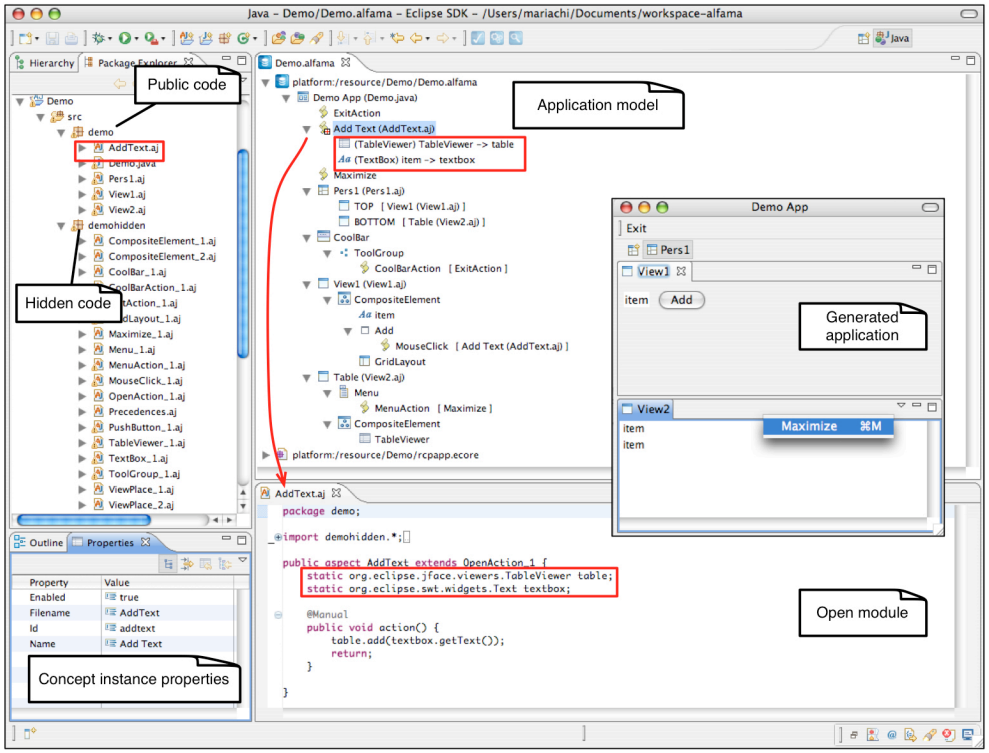


Figure 18: ALFAMA tool: Automatic DSLs for using Frameworks by combining Aspect-oriented and Meta-modeling Approaches.

the case of instances of open concepts, the application developer may navigate to the open module in order to complete it manually. On the right-hand side of the upper part of Figure 18 (b) we can see an application model (instance of the meta-model of Figure 18 (a)) describing an application based on Eclipse RCP. On the left-hand side we can see a package containing the code that was generated from the application model, divided in two packages (public and hidden). On the right-hand side of the bottom part we can see the open module AddText opened in the editor. The marks in the figure highlight how the access declarations on the application model affect the open module. In the application model, an object of type OpenAction (open concept) has two nested accesses to objects of type TableViewer and TextBox. In the open module AddText, the two variables table and textbox are generated by the language workbench and assigned automatically, so that they can be used in the code (as shown in the example).

7. Comparison with conventional DSM

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

7.1. 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 4, the same aspect-oriented mechanisms are used repeatedly. The adoption of *abstract concepts* (e.g. Action in Subsection 4.3.3) 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 4, 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, the code that processes composite instances of the meta-class MenuItem would have to be changed, in order to generate the code that plugs the actions. In our approach, a module encapsulates the concept (as in Subsection 4.3.4), 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.

7.2. 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, in order to overcome such a difficulty, our work in [26] presents a set of design patterns that provide a good aid for developing specialization aspects for a DSM layer. Moreover, the use of aspects to manage variability has been applied successfully in another approach [15], 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 the same modeling construct could be contemplated, we found no practical significance in doing so. Moreover, having such different ways would compromise simplicity. Perhaps when trying the approach on more frameworks this option could be revised, in case DSM modules are found "inelegant". 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.

8. Case Study

The proposed approach for developing the DSM layer went through an iterative process where its applicability was checked against two frameworks, JHotDraw [28] and Eclipse RCP [17]. 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). The meta-model fragment visible in Figure 18 (a) corresponds to a subset (roughly half) of the DSM modules that were developed. We were able to handle the main application concepts, and fully executable code could be successfully generated from the application modules. We tested the extracted DSML by developing toy applications, which made use of the framework features covered by the language.

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, including all the concepts that are visible in the meta-model of Figure 18. Concepts that inherit from other concepts are represented nested under the super concept, and abstract concepts are in italic. 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 anyhow more complex than the ones used throughout Subsection 4.3. From the data in the table, we can see that roughly one fifth of the code uses 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 4.3.3.

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 relevant in the context of product-lines.

Table 1: LOC for Eclipse RCP concepts.

Concept	J+AspJ	Concept	J+AspJ
RCPApplication	71+0	CompositeElement	18+6
Action	14+7	*ControlElement*	7+0
ExitAction	8+0	Text Box	19+0
OpenAction*	22+0	*Button*	35+0
Maximize	8+0	PushButton	9+0
ViewPart	24+15	ToggleButton	7+0
Perspective	19+7	*MouseAction*	12+13
ViewPlace	22+19	MouseClick	4+1
Menu	26+20	CoolBar	7+6
MenuItem	6+13	ToolGroup	28+4
SubMenu	16+0	CoolBarAction	11+3
Separator	7+0	*ViewerElement*	7+0
MenuAction	9+6	TableViewer	8+0

* - open **Total Java:** 424 **Total AspectJ:** 120 **Total:** 544

9. Related Work

Approaches based on *feature-oriented programming* (FOP), such as AHEAD [3], CaesarJ [19], 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 [16] 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 gen-

erated 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+ [18] and Microsoft DSL Tools [12] 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 7.2, 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, the DSM support relies only on the framework implementation, while ALFAMA is capable of automating the DSML construction.

10. Conclusion

In this paper we presented a new approach for developing DSM support for an object-oriented framework. The approach relies on extending frameworks with an additional layer that encodes a DSML for generating framework-based applications. We validated the approach by implementing a prototype tool and performing a case study on the Eclipse RCP framework. Our approach is suitable for product-lines implemented as object-oriented frameworks.

Although the concrete syntax of DSMLs is an important issue from a user's perspective, its definition is out of the scope of this paper. The ALFAMA prototype supports a generic tree-based representation of application models with no domain-specific (graphical) notations. Concrete syntax is an orthogonal issue that can be handled independently from abstract syntax and semantics, and hence, it does not affect the necessary components for generating framework-based applications.

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.

References

[1] Antkiewicz, M., Czarnecki, K., 2006. Framework-specific modeling languages with round-trip engineering. In: MoDELS'06: Proceedings of the 9th International Conference Model Driven Engineering Languages and Systems.

[2] Apel, S., Leich, T., Saake, G., 2006. Aspectual mixin layers: aspects and features in concert. In: ICSE '06: Proceedings of the 28th International Conference on Software Engineering.

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

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

[5] Czarnecki, K., 2004. Overview of generative software development. In: Proceedings of the International Workshop on Unconventional Programming Paradigms.

[6] DSM Forum, 2009. Workshops on domain-specific modeling, 2001-2008. http://www.dsmforum.org/DSMworkshops.html.

[7] Eclipse Foundation, 2009. AspectJ programming language. http://www.eclipse.org/aspectj.
URL `http://www.eclipse.org/aspectj`

[8] Eclipse Foundation, 2009. Eclipse platform and projects. http://www.eclipse.org/projects.

[9] Eclipse Foundation, 2009. EMF – Eclipse Modeling Framework. http://www.eclipse.org/emf.

[10] Fowler, M., 2008. Martin Fowler's Bliki. http://www.martinfowler.com/bliki/.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc.

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

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

[14] Kiczales, G., Lamping, J., Lopes, C., Hugunin, J., Hilsdale, E., Boyapati, C., October 2002. Aspect-Oriented Programming (united states patent). http://www.pmg.lcs.mit.edu/ chandra/publications/aop.html.

[15] Kulesza, U., Alves, V., Garcia, A. F., de Lucena, C. J. P., Borba, P., 2006. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In: ICSR'06: Proceedings of the 9th International Conference on Software Reuse.

[16] Kulesza, U., Lucena, C., Alencar, P. S. C., Garcia, A., 2006. Customizing aspect-oriented variabilities using generative techniques. In: SEKE'06: Proceedings of the 18th International Conference on Software Engineering & Knowledge Engineering.

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

[18] MetaCase, 2008. MetaEdit+ tool. http://www.metacase.com.

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

[20] Moser, S., Nierstrasz, O., 1996. The effect of object-oriented frameworks on developer productivity. Computer 29, 45–51.

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

[22] Pree, W., 1995. Design patterns for object-oriented software development. ACM Press/Addison-Wesley Publishing Co.

[23] Pree, W., 1999. Hot-spot-driven development. In: Building application frameworks: object-oriented foundations of framework design. John Wiley and Sons, Ch. 16, pp. 379–394.

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

[25] Santos, A. L., 2008. ALFAMA: Automatic DSLs for using Frameworks by combining Aspect-oriented and Meta-modeling Approaches. AOSD'08 Demonstrations Track.

[26] Santos, A. L., Koskimies, K., 2008. Modular hot spots: A pattern language for developing high-level framework reuse interfaces. In: EuroPLOP'08: 12th European Conference on Pattern Languages of Programs.

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

[28] SourceForge, 2008. JHotDraw framework. http://www.jhotdraw.org.