University of Waterloo, ECE, Technical Report No. 2007-18

Michał Antkiewicz · Krzysztof Czarnecki

Framework-Specific Modeling Languages

Examples And Algorithms

Draft version 0.3, last update: Apr 25, 2008

Abstract We propose using *framework-specific models* to support the development of framework-based applications. Such models describe how framework-provided abstractions are used or implemented in the application code. Framework-specific models are expressed using *framework-specific modeling languages* (FSMLs), which capture framework abstractions as language concepts. In this paper, we present a concrete approach to defining the abstract syntax and semantics of FSMLs which enables *round-trip engineering*. We apply the approach to three existing frameworks, namely Java Applet, Apache Struts, and Eclipse Workbench, showing that the framework abstractions and usage rules can be adequately captured. We also present a set of algorithms to support round-trip engineering and a generic FSML framework, which we used to implement the sample FSMLs.

Keywords object-oriented frameworks · framework-specific model · framework-specific modeling languages · model-supported engineering · framework completion · round-trip engineering

1 Introduction

Object-oriented application frameworks are one of the most effective and widely used software reuse technologies today. The creation of framework-based applications is often called *framework completion*. The resulting *framework completion code* implements the difference in functionality between the framework and the desired application. A framework provides a set of abstractions, referred to as *framework-provided concepts*. Instances of the concepts may be completely or partially implemented in the framework and it is application's responsibility to correctly use the ready-to-use concept instances and correctly implement the missing parts of the partially implemented instances. Often, a framework only provides an interface for a concept and no instance implementation. In this case, the instance has to be completely implemented in the application and correctly interfaced with the framework. The framework's *application programming interface* (API) specifies the usage and the implementation rules for the concepts. A given concept instance is correctly implemented or used if the completion code *conforms* to the framework's API.

Unfortunately, framework completion can be challenging. The application programmers need to know which concept instances are ready-to-use and which instances need to be or can be implemented. When implementing concept instances, the programmers need to know what are the necessary and optional implementation tasks, and which implementation options are compatibile. They need to maintain the consistency among, potentially, many different kinds of artifacts, such as Java classes, XML configuration files, and Java Server Pages (JSPs), which constitute the completion code. When looking at the completion code, the developers need to distinguish between fragments of code that implement application-specific functionality from the fragments that use the framework. Recognizing concept instances in the code is challenging since some instances, such as collaborations among objects, are usually scattered across the completion code.

We can characterize the process of framework completion as two, interleaving activities: *concept configuration* and *open-ended programming with restrictions*. Concept configuration is deciding which and how many instances of framework-provided concepts are to be created and deciding among framework-stipulated implementation choices for every concept instance. Open-ended programming is implementing application-specific functionality that goes beyond the predefined implementation

This work is partially supported by IBM Centers For Advanced Studies, Ottawa Lab.

Michał Antkiewicz
University of Waterloo
Tel.: +1 519 884 2277
E-mail: mantkiew@uwaterloo.ca

Krzysztof Czarnecki
University of Waterloo
Tel.: + 1 519 888 4567 ext. 37137
E-mail: czarnecki@acm.org

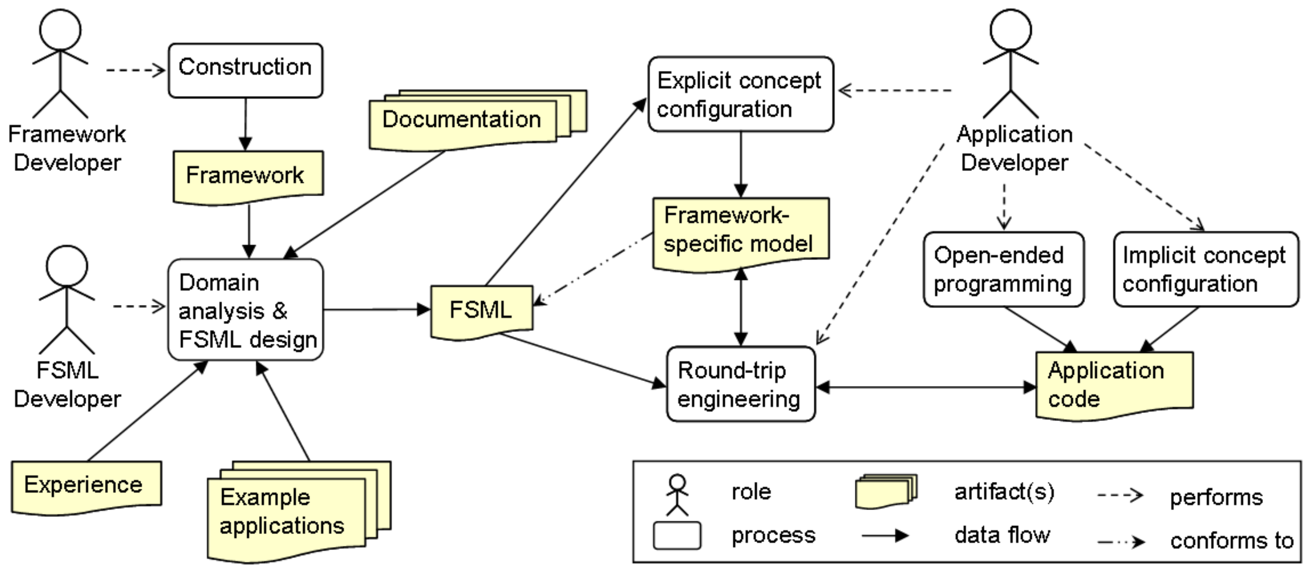


Fig. 1 Overview of model-supported engineering of framework-based applications using FSMLs

steps and choices provided by the framework. Open-ended programming is restricted in the sense that it must not violate the framework's rules of engagement.

Traditionally, both activities are performed by application developers directly in the code. Concept configuration is performed implicitly, code implementing concept instances is created manually, and the rationale for choosing particular implementation options is often lost. In this paper, we propose using *framework-specific models* to support the development of framework-based applications and we present a concrete approach for building *framework-specific modeling languages* (FSMLs) that can be used for expressing such models. We validate the approach by designing three FSMLs, each for one of the open-source frameworks. We show that the concepts can be adequately captured using the presented approach. We propose using *round-trip engineering* to keep the models and the code consistent thoughout the development. We present generic algorithms for reverse-, forward-, and round-trip engineering that interpret the metamodels of the FSMLs. To validate the algorithms we implemented them as a part of a generic paltform for building FSMLs and we have built prototype implementations of the three FSMLs on top of the platform. Finally, we used the prototypes to conduct experiments to validate the effectiveness of reverse- and round-trip engineering using FSMLs.

The rest of the paper is organized as follows. In Section 2 we present an overview of model-supported engineering of framework-based applications. Next, in Section 3 we present a concrete approach to defining modeling languages for expressing framework-specific models. We present the validation of the approach in Section 4, in which we present the metamodels of three FSMLs designed for three open-source frameworks. In Sections 5, 6, and 7 we present algorithms for reverse-, forward-, and round-trip engineering, respectively. We describe a generic platform for building FSMLs in Section 8, which implements the abovementioned algorithms and which was used for building prototype implementations of the three example FSMLs. We conclude with related work in Section 9 and discussion in Section 10.

2 Model-Supported Engineering of Framework-Based Applications

Figure 1 presents an overview of model-supported engineering of framework-based applications, which is an extension of traditional, code-centric engineering. In this development model, application developers still perform open-ended programming and concept configuration manually. However, they now have an option of performing explicit concept configuration, in which they build a framework-specific model by creating instances of the concepts. The model describes the design of the application from the framework viewpoint and contains configurations of concept instances. The model can describe both the current state of the code, the design yet to be implemented, or the original design for the previous state of the code. Since both the model and the code can be developed independently, the developers can perform round-trip engineering to synchronize them. In round-trip engineering, the current model and the current code are first compared to identify the changes that occurred since the last time they were synchronized. Next, the model and the code are reconciled by propagating changes according to developer's decisions. Round-trip engineering propagates changes in both directions and also allows for the creation of a new framework-specific model from the existing code (reverse engineering) and the creation of the new code from the existing model (forward engineering).

Framework-specific models are expressed using FSMLs, that is, the models conform to the metamodels of the FSMLs. The metamodel of an FSML is needed for the concept configuration and the round-trip engineering.

Designing FSMLs is a manual and creative activity. An FSML is a *domain-specific modeling language* [8] designed for an area of concern of an object-oriented application framework. FSML developer performs domain analysis to identify framework-provided concepts. To that end, he analyses the framework itself, all available documentation, and existing applications that use the framework. He may also draw from his own experience in building applications that use the framework. Note, that the FSML developer need not to be a framework developer.

In model-supported engineering and unlike in model-driven engineering, framework-specific models are auxiliary development artifacts, which enables easy adoption of the approach. The models can be automatically retrieved from the existing code and they can provide additional benefits in subsequent development, such as providing the overview of the application from the framework's viewpoint and checking the conformance of the code to the framework's API. Additionally, the FSMLs can be developed incrementally by gradually extending them with new concepts and implementation options.

Next, we present a concrete approach to defining abstract syntax and semantics of FSMLs with support for round-trip engineering.

3 Defining Framework-Specific Modeling Languages

A *framework-specific modeling language* (FSML) is a *domain-specific modeling language* [8] that is designed for a specific framework, called its *base framework*. The abstract syntax of an FSML should capture base framework's concepts and allow concept configuration. The semantics of an FSML should enable round-trip engineering.

3.1 Constituent parts of an FSML

Abstract syntax. An FSML explicitly captures framework-provided concepts as *language concepts* in its abstract syntax. Abstract syntax defines a decomposition of a concept into a hierarchy of *features*. Features are distinguishing characteristics of a concept and allow to tell among the instances of the concept. Models expressed using an FSML are sets of concept instances, where each concept instance is characterized by a configuration (selection) of feature instances. We often refer to feature instances simply as features.

In the feature hierarchy, features can be *essential*, *mandatory*, and *optional* with respect to the parent feature. In a feature configuration, an essential feature must be present if its parent feature is present; a mandatory feature should be present if its parent feature is present; and an optional feature may be present if its parent feature is present. Essential features are also mandatory; however, the difference between the two is that a feature whose essential subfeature is absent cannot exist, whereas a feature whose mandatory subfeature is absent can exists, but is considered to have a *configuration error*. Presence of all essential features can be thought of as the *necessary and sufficient* condition on the presence of their parent feature. The distinction between the mandatory and essential features is important for the ability of the model to capture broken concept instances. The essential features encode the absolute minimum that the implementation of the concept instance must contain in order for the instance to be considered present.

The decomposition of a concept into features may also include additional *well-formedness* constraints, such as a feature *requiring* or *excluding* another feature. Furthermore, features can be grouped and assigned a *group cardinality*, which is an interval specifying how many features from the group have to be present in a configuration. A feature may have a *type* meaning that a value of that type can be associated with the feature in the configuration.

For example, the upper left part of Fig. 2 presents a decomposition of a concept of *Applet* into features. The feature *extendsApplet* is essential. The feature *registers* is mandatory. The feature *listensToMouse* is optional, and the feature *showsStatus* is optional and multiple. The lower left part of Fig. 2 presents a framework-specific model, which is a configuration of the concept *Applet*. The feature *extendsApplet* is present in the configuration and the features *extendsJApplet* and *registers* are absent from the configuration. The absence of the mandatory feature *registers* indicates an error in the configuration of its parent feature *listensToMouse* and, consequently, the instance of the concept *Applet*. Two instances of the feature *showsStatus* are also present. The features *name* and *message* have concrete values.

Mapping of the abstract syntax to the framework API. Features in a framework-specific model represent patterns in the artifacts of framework completion code (referred to as artifact or code patterns) that constitute the implementation of the model. Because framework completion code may consist of multiple artifacts of different kinds, features of a single concept may correspond to patterns scattered across multiple artifacts. Features may also represent certain abstractions or semantic facts about the code, in which cases the features may not correspond to any artifact patterns directly.

The lower part of Fig. 2 presents correspondence links between the features from the model and artifact patterns, which are Java abstract syntax tree (AST) nodes.

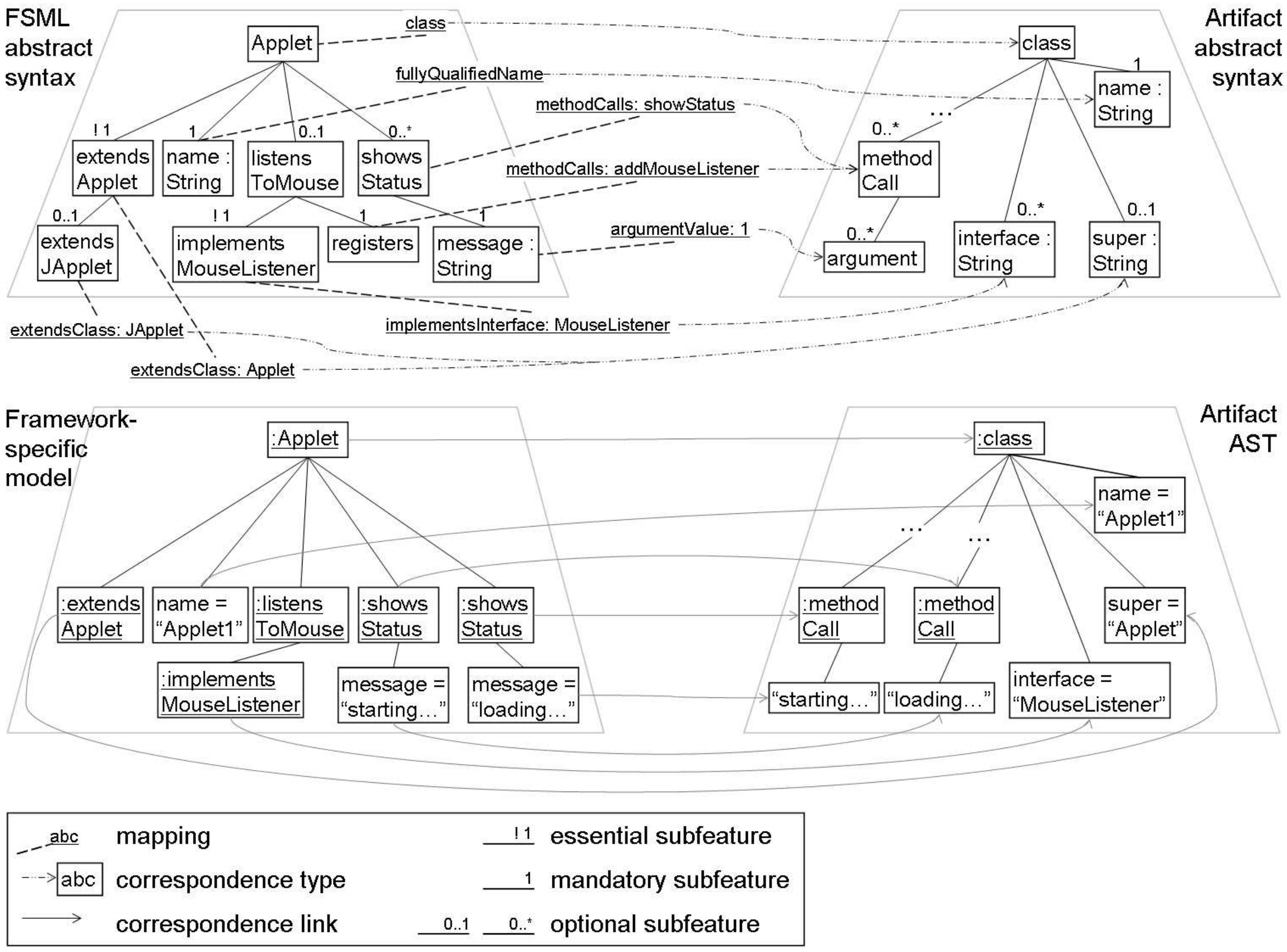


Fig. 2 Abstract syntax, mapping, model, and artifact

The concept instance *Applet* corresponds to the instance of *class*. The feature *extendsApplet* corresponds to the super class declaration *super* of the class. The feature *listensToMouse* does not correspond to any artifact element. The two instances of the feature *showsStatus* correspond to two method call elements. The two instances of the feature *message* correspond to the values of the method call arguments *"starting..."* and *"loading..."*, respectively.

The mapping of the abstract syntax to the framework API defines how concept instances and their features correspond to the artifact patterns in the framework completion code. A feature can be associated with a *mapping definition*, or simply a *mapping*, which is a pattern expression specifying the number and kind of patterns that the feature can correspond to. Each mapping has a predefined *correspondence type*, i.e., the type of corresponding artifact elements. A mapping can also be parametrized. In the upper part of Fig. 2, mappings are attached to features and mapping parameters are specified after a colon (:). For example, the mapping *class* attached to the concept *Applet* specifies that instances of the concept correspond to *classes*. The mapping *methodCalls: showStatus* attached to the feature *showsStatus* specifies that instances of the feature correspond to calls to the method *showStatus*. The mapping *methodCalls: addMouseListener* attached to the feature *registers* specifies that an instance of the feature corresponds to a *set* of method calls. The multiplicity of a feature determines the interpretation of the correspondence. The mandatory feature *registers* is present if the set of corresponding method calls is non-empty. The absence of the feature specifies that the set of corresponding method calls should be empty.

The mapping of abstract syntax to framework API can be realized by *code queries* and *code transformations*. A code query determines the presence or value of a feature in the completion code. A code transformation creates, updates, and removes elements of the completion code that constitute the implementation of a feature. The mapping enables automated round-trip engineering, where the code can be created from the model, the model from the code, and changes made to the code and the model can be identified and reconciled.

Both the code queries and code transformations require a certain scope in which they operate. For example, a mapping *methodCalls: addMouseListener* can be realized by a code query that locates method calls in the body of a class, in the body of a single method, or in the control flow of a single method. Depending on which type of query is used, a different search scope may be required, such as a class or a method. In this paper, we refer to the scope in which a code query or a code transformation operates as *context element*. Examples of context elements include *Java class*, *method call*, *method*, *field*, or *XML element*. The context element required by a mapping attached to a feature can be obtained by locating the closest parent of the feature that corresponds to the artifact element of the required type. Such a parent is referred to as *context feature* for the mapping. Context elements can be a) containers of elements that subfeatures correspond to, b) sources of property values that subfeatures correspond to, or c) sources of semantic facts that subfeatures correspond to.

For example, in the top of Fig. 2, the concept *Applet* is a context feature for those of its subfeatures, whose mappings require a Java class as a context, such as the mapping *fullyQualifiedName* attached to the feature *name*. The mapping *methodCalls* attached to the features *showsStatus* and *registers* requires a Java class or a method as context. In our example, the concept *Applet* provides the context Java class. The mapping *argumentValue* attached to the feature *message* requires a method call as context, which is provided by the feature *showsStatus* in our example.

Using the mapping in forward and reverse engineering. We illustrate forward and reverse engineering using example from Fig. 2. Assume that the model for the instance of the concept *Applet* is given and the completion code elements do not exist. Also, assume that the feature *registers* is present in the model. First, a code transformation creates a class that the applet corresponds to. The name of the class is set based on the feature *name*, and the super class is set based on the feature *extendsClass*. The implementation for the feature *listensToMouse* is created indirectly by the code transformations for the features *implementsMouseListener* and *registers* (assuming its existence). Next, the code transformation for the the feature *showsStatus* creates two method calls for the two instances of the feature. For each method call, the value of the first argument is set to the value of the features *message*. Code transformations may require additional parameters, such as default locations of method calls, which need to be specified in the mapping or obtained interactively from the user.

Now, assume that the model does not exist and the completion code is given. Unlike forward engineering, which is driven by the model, reverse engineering is driven by the abstract syntax because the model does not yet exist. The goal of reverse engineering is to identify instances of the concept *Applet* that are implemented in the code. The mapping *class* specifies that instances of *Applet* correspond to Java classes. In order to find out, which classes could be potential applets, the essential feature *extendsApplet* is used to form a query. The query finds all subclasses of the class *java.applet.Applet*. An instance of *Applet* is created for each subclass. Next, the code query for the feature *extendsJApplet* determines that the context Java class does not extend the class *javax.swing.JApplet* and an instance of the feature is not created. The value of the feature *name* is set to the fully qualified name of the context class. The feature *listensToMouse* does not have any mapping attached to it, and therefore, during reverse engineering, we assume the presence of the instance of the feature. We then proceed with analysis to determine whether all the essential subfeatures of the instance are present. The code query for the feature *implementsMouseListener* determines that the context class implements the interface *MouseListener* and an instance of the feature is created. If the context Java class did not implement the interface *MouseListener*, the assumed instance of the feature *listensToMouse* would be removed because *implementsMouseListener* is an essential subfeature of *listensToMouse*. The code query for the feature *registers* does not find any method calls to *addMouseListener* method in the context class and an instance of the feature is not created. The missing mandatory feature indicates that the applet is implemented incorrectly. Next, the code query for the feature *showsStatus* locates two method calls to *showStatus* and two instances of the feature are created. For each instance of the feature *showsStatus*, the code query for the feature *message* retrieves the value of the first argument of the context method call and sets the value of the feature.

Reverse, forward, and round-trip engineering are described in detail in Sections 5, 6, and 7, respectively.

To summarize, the abstract syntax encodes all valid ways of configuring features, which, through the mapping, define all valid ways of implementing framework-provided concepts.

Concrete syntax. The concrete syntax may offer specialized rendering of the models to enhance their comprehension. In particular, concrete syntax may offer different decomposition of features than the abstract syntax and provide better navigation between features.

3.2 FSMLs address the challenges of framework completion

An FSML with round-trip engineering support addresses the challenges from the previous section.

Knowing how to complete a framework. The creation of a model consists of the creation of concept instances and configuring them by selecting or eliminating features and providing attribute values. Concept configuration is controlled by the abstract syntax and well-formedness constraints, thus guiding the developer in making correct configuration choices.

Code transformations can locate the places where the code implementing a concept instance should be inserted in the completion code. The transformations are executed for a correct concept configuration and, therefore, produce completion code that satisfies the framework rules. A developer can review the changes made by the transformations and learn how to complete the framework.

In the case where the completion code has already been created for a concept instance, changing the configuration of the concept by adding or removing features and modifying attribute values may require updating the completion code by code transformation and subsequent adjustments by the developer.

Obtaining an overview of a framework-based application. Reverse engineering can identify instances of concepts implemented in the code. The identified instances can be presented to the developer in a form of a model, which is, in fact, an overview of the application from the viewpoint of the FSML. Furthermore, the models can be constructed for different versions of the code, allowing the developer to verify whether the current code still conforms to the previous model. Also, reverse engineering can recognize broken or incomplete concept instances that need to be fixed. Finally, correspondence links established during reverse engineering provide traceability between the model and the code.

Following the general rules of engagement for the framework. Forward engineering produces code that conforms to the rules. The reverse mapping helps ensuring that a manual customization of the code does not violate the rules of engagement.

Repetitive code in the domain concept instantiation. Code transformations automate the creation and update of the repetitive code.

Knowing how to migrate completion code after API changes. An FSML provides a framework to help with migration of completion code to a changed API. Code queries can be used to find uses of the deprecated API and specialized code transformations can rewrite existing code to conform to the changed API.

4 Examples of FSMLs

In this section, we present an FSML for each of the following frameworks: Java Applet, Apache Struts, and Eclipse Workbench. For each FSML, we discuss the scope and the challenges addressed by the language, the abstract syntax, and the mapping of the abstract syntax to the framework API. In the remainder of the paper, we refer to the abstract syntax and the mapping as a *metamodel*.

Table 1 Metamodel notation

icon	metamodel element
A ▤	abstract class
▤	concrete class
▭	attribute
▭→	reference
▬→	containment reference
1 , 1..*	mandatory multiplicities
, 0..*	optional multiplicities
!	essential feature

The presented FSMLs have been implemented using a generic FSML framework, which is described in Section 8. The framework interprets the metamodels and provides support for round-trip engineering. The presented metamodels are renderings of fragments of the actual metamodels used in the prototypes.

4.1 Understanding the metamodels and the rendering

We express abstract syntax using class diagrams, which consist of *classes*, *attributes*, and *references*. References can be *containment* or *non-containment*. However, unlike in UML, we render a class diagram as a tree, which represents a containment hierarchy, where attributes and references are children of classes, and classes are children of containment references. Also, to reduce visual clutter, we do not show the type of Boolean attributes and containment references. Instead of showing a class, which is the type of a containment reference, after the colon (:), we render the class and all its subclasses as children of the reference. This way we can clearly see the containment hierarchy and all classes whose instances can be values of containment references. Table 1 summarizes notation used for presenting the metamodels.

The features of the concepts can be represented in the abstract syntax as attributes, references, and ⟨containment reference, class⟩ pairs. Multiplicity of a feature is represented by the multiplicity of attribute, reference, and containment reference, respectively. Figure 3 presents abstract syntax for concept *Applet* from Fig. 2 as a class diagram rendered as a containment hierarchy. The feature *Applet* is represented as the containment-reference-and-class ⟨`applets, Applet`⟩ pair; the feature *name* is represented as the attribute `name : String`, the feature *extendsApplet* is represented as the pair ⟨`extendsApplet, ExtendsApplet`⟩; and the feature *extendsJApplet* is represented as the Boolean attribute `extendsJApplet`. We consider a feature to be present in the model if the representing attribute or reference a) has a non-null value for single valued features, b) has a non-empty collection of values for multi-valued features, and c) attribute value is true for Boolean attributes.

The mapping for any abstract syntax element (class, attribute, or reference) is rendered in angle brackets next to the element. Since the whole model describes the completion code, the root element of the model must always correspond to the project, which contains all artifacts that constitute the completion code.

4.2 Applet FSML

The Applet FSML is a simple language whose main purpose is to capture the concept of *applet* and capture framework rules and best practices of applet development. A model expressed in the Applet FSML describes details of applet implementation related to the Applet framework, provides traceability to the fragments of code related to the framework, and allows for checking framework rules encoded as well-formedness constraints in abstract syntax.

Figure 3 presents the metamodel of the Applet FSML. The instance of the root class `AppletModel` corresponds to the modeled project and contains a number of applets through `applets` containment reference. Each instance of the class `Applet` corresponds to a Java class we refer to as *context Java class*.

The attribute `applet : String` corresponds to the fully qualified name of the context Java class. The reference `extendsApplet` corresponds to the fact that the context Java class extends the class `java.applet.Applet`. It is a mandatory and essential feature of an applet, meaning that, it is necessary for a Java class to extend the class `Applet` in order to be an applet. The attribute `extendsJApplet` corresponds to the fact that the context Java class extends the class `javax.swing.JApplet`. It is an optional feature, and applets should extend the class `JApplet` if they also use Java Swing framework for graphical components.

The feature ⟨`overridesRequiredMethods, OverridesRequiredMethods`⟩ is mandatory and represents a framework-rule that an applet should override at least one of the three lifecycle methods: `init`, `start`, and `paint`. This constraint is specified as a feature group `<1-3>` constraint on the class `OverridesRequiredMethods`, and means that at least one and at most three of subfeatures must be present in the implementation. The three subfeatures are represented as `init`, `start`, and `paint` attributes and each corresponds to a lifecycle method: `void init()`, `void start()`, and `void paint(java.awt.Graphics)`, respectively. The mapping specifies that the attributes correspond to methods implemented directly by the class using `inherited: false`. The group cardinality `<1-3>` is marked as essential (using '!'), meaning that if not satisfied, the feature `overridesRequiredMethods` will not be present. Note, that the reference `overridesRequiredMethods` is not marked as essential because it is only mandatory for an applet to override the lifecycle methods, and failing to do so does not mean that a given class is not an applet—it only means that the implementation does not satisfy a framework constraint.

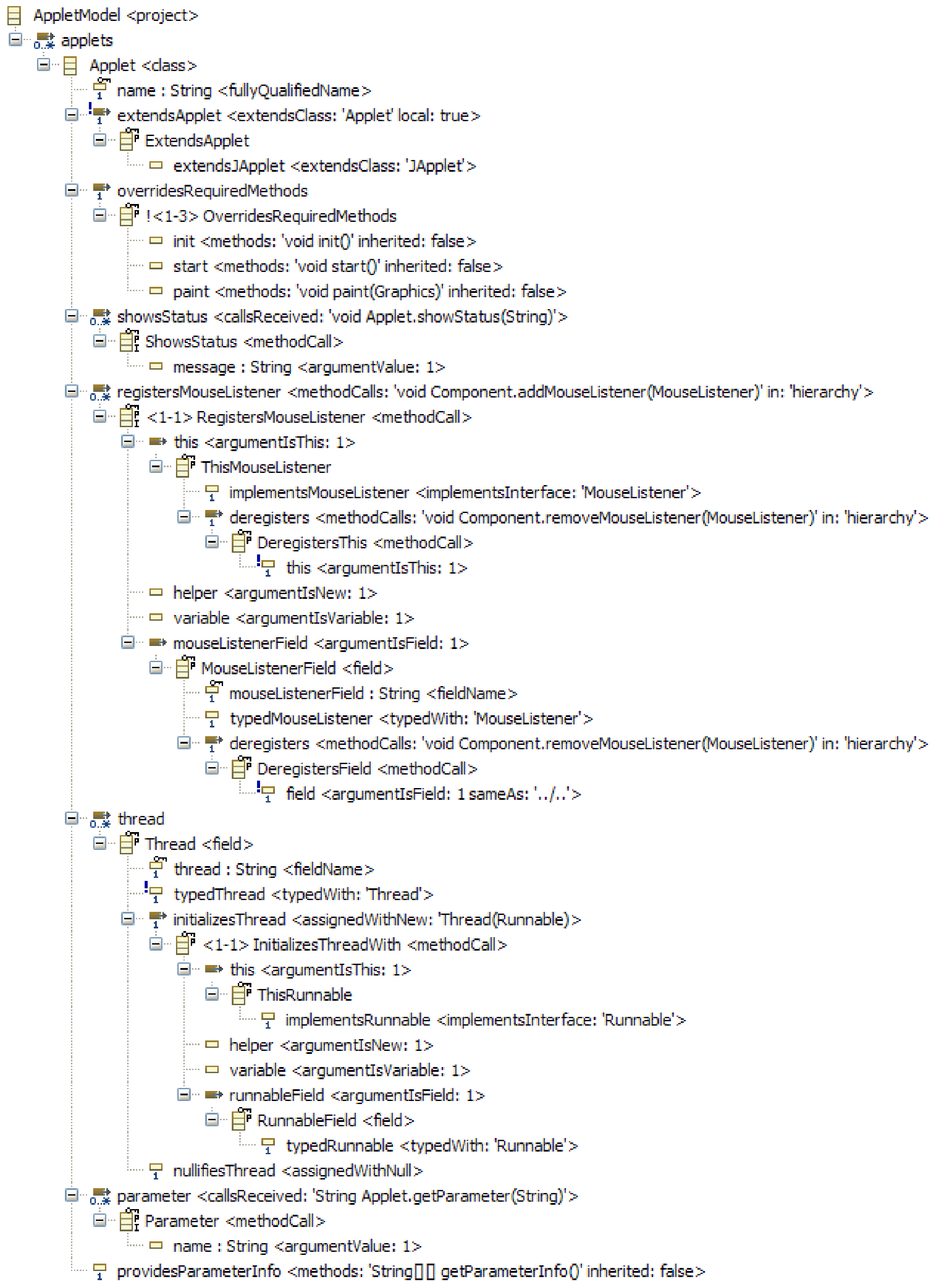
Applets may display status messages by calling `void Applet.showStatus(String)` method. The reference `showsStatus` corresponds to all method calls to the `showStatus` method that can be found in the hierarchy of the context Java class. Each instance of the class `ShowsStatus` corresponds to a single method call[1]. The attribute `message` corresponds to the value of the first argument of the context method call, provided that the value is statically available.

An applet may optionally listen to mouse events. The optional feature ⟨`listensToMouse, ListensToMouse`⟩ represents the fact that an applet is a mouse listener. The attribute `implementsMouseListener` corresponds to the fact that the context Java class implements `java.awt.event.MouseListener` interface. The attribute `registers` corresponds to one or more method calls to `void java.awt.Component.addMouseListener(MouseListener)` method and the attribute `deregisters` corresponds to one or more method calls to `void Component.removeMouseListener(MouseListener)` method. The features `implementsMouseListener` and `registers` are essential for the feature ⟨`listensToMouse, ListensToMouse`⟩ to be present.

Java applets should run long running operations in background threads. As a best practice, a reference to the thread should be stored in a field, the field should be assigned to the new thread and the field should be assigned null, to give the thread a signal to terminate. The feature ⟨`thread, Thread`⟩ represents a number of threads, and each instance of the class `Thread` corresponds to field of the context Java class. Note that a Java field that an instance of the class `Thread` corresponds to becomes a *context Java field*. The attribute `thread : String` corresponds to the name of the context Java field. The attribute `typedThread` corresponds to the fact that the type of the context Java field is `java.lang.Thread` or a subclass. The attribute `typedThread` is marked as essential meaning that only fields of type `Thread` are considered. The attribute `initializesThread` corresponds to the fact that the context Java field is assigned with the call to `new Thread(Runnable)` in the hierarchy of the context Java class. Analogously, the attribute `nullifiesThread` corresponds to the fact that the context Java field is assigned with `null` in the hierarchy of the context Java class. Note, that the mappings *assignedWithNew* and *assignedWithNullIn* require two contexts.

Finally, an applet may retrieve values of named parameters specified on the HTML page by calling `String`

[1] The method call becomes a *context method call* for subfeatures.



```
AppletModel <project>
  applets
    Applet <class>
      name : String <fullyQualifiedName>
      extendsApplet <extendsClass: 'Applet' local: true>
        ExtendsApplet
          extendsJApplet <extendsClass: 'JApplet'>
      overridesRequiredMethods
        !<1-3> OverridesRequiredMethods
          init <methods: 'void init()' inherited: false>
          start <methods: 'void start()' inherited: false>
          paint <methods: 'void paint(Graphics)' inherited: false>
      showsStatus <callsReceived: 'void Applet.showStatus(String)'>
        ShowsStatus <methodCall>
          message : String <argumentValue: 1>
      registersMouseListener <methodCalls: 'void Component.addMouseListener(MouseListener)' in: 'hierarchy'>
        <1-1> RegistersMouseListener <methodCall>
          this <argumentIsThis: 1>
            ThisMouseListener
              implementsMouseListener <implementsInterface: 'MouseListener'>
              deregisters <methodCalls: 'void Component.removeMouseListener(MouseListener)' in: 'hierarchy'>
                DeregistersThis <methodCall>
                  this <argumentIsThis: 1>
          helper <argumentIsNew: 1>
          variable <argumentIsVariable: 1>
          mouseListenerField <argumentIsField: 1>
            MouseListenerField <field>
              mouseListenerField : String <fieldName>
              typedMouseListener <typedWith: 'MouseListener'>
              deregisters <methodCalls: 'void Component.removeMouseListener(MouseListener)' in: 'hierarchy'>
                DeregistersField <methodCall>
                  field <argumentIsField: 1 sameAs: '../..'>
      thread
        Thread <field>
          thread : String <fieldName>
          typedThread <typedWith: 'Thread'>
          initializesThread <assignedWithNew: 'Thread(Runnable)>
            <1-1> InitializesThreadWith <methodCall>
              this <argumentIsThis: 1>
                ThisRunnable
                  implementsRunnable <implementsInterface: 'Runnable'>
              helper <argumentIsNew: 1>
              variable <argumentIsVariable: 1>
              runnableField <argumentIsField: 1>
                RunnableField <field>
                  typedRunnable <typedWith: 'Runnable'>
          nullifiesThread <assignedWithNull>
      parameter <callsReceived: 'String Applet.getParameter(String)'>
        Parameter <methodCall>
          name : String <argumentValue: 1>
      providesParameterInfo <methods: 'String[][] getParameterInfo()' inherited: false>
```

Fig. 3 Metamodel of the Applet FSML

`Applet.getParameter(String)` method. The reference `parameter` corresponds to all method calls to the `getParameter()` method that can be found in the hierarchy of the context Java class. Each instance of the class `Parameter` corresponds to a single method call. The attribute `name` corresponds to the value of the first argument of the context method call, which is the name of the parameter. Applets should also override the `String[][] getParameterInfo()` method, which should return triples of strings ⟨name, type, description⟩ for each named parameter. The attribute `providesParameterInfo` corresponds to the method `getParameterInfo()` declared in the context Java class.

The Applet FSML does not cover Applet framework completely. Outside the scope of the language are:

1. other kinds of listeners, such as mouse motion listener. We did not include other kinds of listeners because the mechanism is exactly the same as for mouse listener.
2. checking that `String[][] getParameterInfo()` method returns a triple for every parameter used by an applet, and that the applet uses every parameter whose declaration is returned by the method.
3. referential integrity between Java code and HTML page on which an applet is declared. Checking that a HTML page uses correct parameters, and that an applet uses values of all parameters used on the HTML page.
4. communication among applets included on the same HTML page. Applets are assigned symbolic names, which are used for message based communication.

4.3 Apache Struts FSML

Apache Struts is a web application framework built on top of J2EE. Struts is a relatively simple and prescriptive framework, where main abstractions are clear. A web application based on Struts has at least three different kinds of artifacts: Java code for business logic, XML configuration files, which assign roles to Java classes and specify other parameters for the framework, and Java Server Pages (JSPs) for presentation. There are two main challenges with the development of Struts-based applications: maintaining referential integrity among numerous artifacts, and ensuring correctness of XML configuration file. The latter problem arises due to the fact that XML schema of the configuration file only defines few XML attributes as *required*, and the remaining attributes are optional. However, different subsets of XML attributes are required for particular usages of the framework and the schema is not capable of encoding such framework rules. This problem could also be addressed using FSMLs, however, in this section we only show how the problem of referential integrity can be addressed.

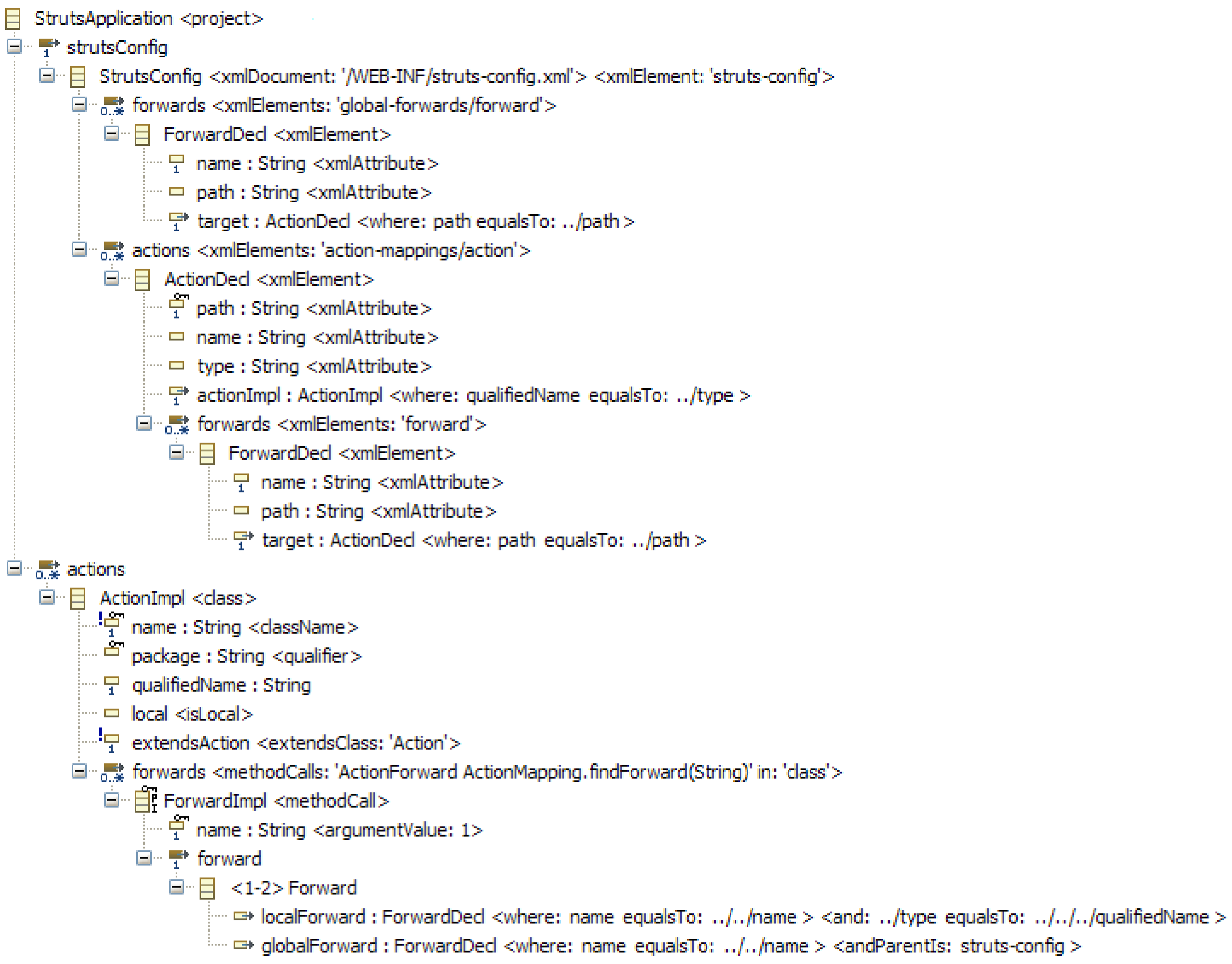
We present a fragment of the Struts FSML's metamodel to illustrate how models expressed using FSMLs can help the developers with maintaining referential integrity among the artifacts. The fragment is concerned only with concepts of *action* and *forward*. An action implements a response to a HTTP request and returns a number of forwards. A forward is a ⟨name, path⟩ pair declared in the XML configuration file, where path is a relative link to an action or a page. The implementation of an action consists of two parts: action declaration in XML configuration file and Java class. Forwards can be *global* and *local* with respect to action declaration. Global forwards can be used by any action, and local forwards can only be used by the declaring action.

The referential integrity problem we are addressing is a) using paths of existing actions in forward declarations, b) using qualified names of existing action Java classes in action declarations, c) using correct forward names in Java code of actions, and d) using paths of existing actions in `<html:link>` tags in Java Server Pages. Case b) includes verifying that a Java class exists for every action declaration, as well as, verifying that action declaration exists for every Java action class (not shown). Case d) is not presented here. By aggregating action and forward information in a single model we can both check referential integrity, and visualize page flow of a Struts application.

This example also illustrates the importance of concrete syntax. Although abstract syntax tree of the model contains all necessary information for the page flow, it is not directly usable and needs to be rendered using a specialized notation.

In our example we only focus on XML and Java. Page flow related information from Java Server Pages in the model includes representation of `<html:link action="`*`path`*`">` tags, where *path* is a value of unique path attribute of action declaration.

Figure 4 presents a fragment of the metamodel of the Struts FSML. Instance of the class `StrutsApplication` corresponds to the modeled project, which contains Java code, XML configuration file, and Java Server Pages. The feature ⟨`strutsConfig, StrutsConfig`⟩ corresponds to the `struts-config.xml` file and, at the same time, to the root XML element of the document called `struts-config`. The feature ⟨`forwards, ForwardDecl`⟩ corresponds to a number of global forward declarations, where each instance of the class `ForwardDecl` corresponds to a single XML element. The attributes `name` and `path` correspond to XML attributes, with the same names, of the context XML element. The reference `target` corresponds to the referential integrity constraint, and specifies that an instance of the class `ActionDecl` retrieved by a model query must exist, such that the attribute `path` of the action declaration equals to the attribute `path` of the forward declaration. If such an action declaration did not exist, the value of the reference would be null and therefore a mandatory feature would be missing. The feature ⟨`actions, ActionDecl`⟩ corresponds to a number of action declarations. The reference `actionImpl` corre-



```
StrutsApplication <project>
  strutsConfig
    StrutsConfig <xmlDocument: '/WEB-INF/struts-config.xml'> <xmlElement: 'struts-config'>
      forwards <xmlElements: 'global-forwards/forward'>
        ForwardDecl <xmlElement>
          name : String <xmlAttribute>
          path : String <xmlAttribute>
          target : ActionDecl <where: path equalsTo: ../path >
      actions <xmlElements: 'action-mappings/action'>
        ActionDecl <xmlElement>
          path : String <xmlAttribute>
          name : String <xmlAttribute>
          type : String <xmlAttribute>
          actionImpl : ActionImpl <where: qualifiedName equalsTo: ../type >
          forwards <xmlElements: 'forward'>
            ForwardDecl <xmlElement>
              name : String <xmlAttribute>
              path : String <xmlAttribute>
              target : ActionDecl <where: path equalsTo: ../path >
  actions
    ActionImpl <class>
      name : String <className>
      package : String <qualifier>
      qualifiedName : String
      local <isLocal>
      extendsAction <extendsClass: 'Action'>
      forwards <methodCalls: 'ActionForward ActionMapping.findForward(String)' in: 'class'>
        ForwardImpl <methodCall>
          name : String <argumentValue: 1>
          forward
            <1-2> Forward
              localForward : ForwardDecl <where: name equalsTo: ../../name > <and: ../type equalsTo: ../../../qualifiedName >
              globalForward : ForwardDecl <where: name equalsTo: ../../name > <andParentIs: struts-config >
```

Fig. 4 Fragment of the metamodel of the Struts FSML

sponds to the referential integrity constraint, and specifies that an instance of the class `ActionImpl` retrieved by a model query must exist, such that the attribute `qualifiedName` of the action implementation equals to the attribute `type` of the action declaration. The feature ⟨`forwards, ForwardDecl`⟩ corresponds to a number of local forward declarations contained by the context XML element.

As we can see, the class `ForwardDecl` is the type of the two references: `StrutsConfig::forwards` and `ActionDecl::forwards`. The reuse is possible because the paths specifying the location of XML elements are specified in the mapping for the references.

The second part of the metamodel represents Java code of actions. The feature ⟨`actions, ActionImpl`⟩ corresponds to a number of Java classes. The attribute `name` corresponds to the class name and the attribute `package` corresponds to the class qualifier. The attribute `qualifiedName` is a derived attribute, whose value is computed by concatenating values of the `qualifier` and `name` attributes. The attribute `extendsAction` is an essential feature and corresponds to the fact that the context Java class extends framework-provided class `org.apache.struts.action.Action`. The feature ⟨`forwards, ForwardImpl`⟩ corresponds to method calls to `org.apache.struts.ActionForward org.apache.struts.action.ActionMapping.findForward(String)` method, which is used to lookup a forward declaration with given name in the XML configuration file. The attribute `name` corresponds to the value of the first argument of the context method call, which is the name of the forward. The feature ⟨`forward, Forward`⟩ corresponds to the referential integrity constraint that a local or global forward declaration must exist for the name provided as argument to the context method call. The reference `localForward` corresponds to the model query, which retrieves an instance of the class `ForwardDecl`, such that value of the attribute `name` of the forward declaration equals to the value of the attribute `name` of the class `ForwardImpl`, and the value of the attribute `type` of parent action declaration equals to the value of the attribute `qualifiedName` of the context Java class. Similarly, the reference `globalForward` corresponds to the model query, which retrieves a global forward declaration.

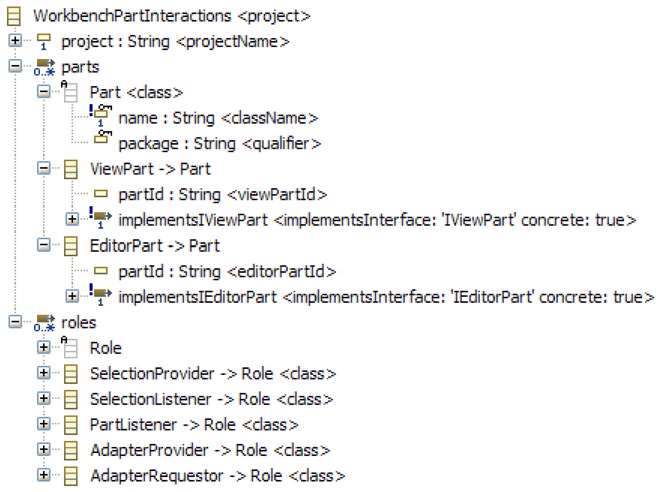
The presented metamodel of the Struts FSML allows checking referential integrity constraints. The model queries of the features that represent referential integrity constraints can be re-evaluated on demand while working with the model, which allows the developers maintaining the artifacts consistent.

4.4 Eclipse Workbench Part Interaction FSML

Eclipse [9] is a universal, open-source platform for building and integrating tools, which is implemented as a set of Java-based object-oriented frameworks. In this paper, we consider a particular part of the Eclipse API, which is concerned with *workbench parts* and their *interactions*. Workbench parts are the basic building blocks of the Eclipse Workbench, which is the working area of an Eclipse user. The parts can interact in various ways, for example, by exchanging events.

In this paper, we consider two kinds of workbench parts, namely *editors* and *views*. An editor is used for displaying and editing the contents of *input resources*. An example of an editor is the Java editor included in the Eclipse Java Development Tools (JDT) [9]. A view is also used for displaying and editing information, but unlike an editor, a view is not associated with any particular input resource. An example of the standard workbench view is *Content Outline*, which is used to display the outline of an input resource opened in an active editor. Editors and views have to be contributed to the Workbench by declaring them in a plug-in manifest files. The Workbench scans manifest files upon startup and makes contributed workbench parts available to the user.

Workbench parts interact in various ways. In this paper, we consider five kinds of part interactions, namely *selection provider*, two kinds of *selection listeners*, *part listener*, and *adapter requestor/adapter provider*. For example, the Content Outline view listens to *part activation* events by registering itself as a listener with the Workbench *Part Service* and, therefore, a class that implements the view plays a role of *part listener*. When an editor, such as the Java editor, is activated, the view receives an activation event. In response to this event, the view asks the editor for its `IContentOutlinePage` adapter, which is used to display the outline of the editor's input resource. Therefore, the view is an *adapter requestor* and the editor is an *adapter provider*. Content Outline view is also a *selection provider* by registering itself as a provider with the Workbench *Selection Service*. The Selection Service broadcasts selection events to all registered listeners. An example of a *selection listener* is



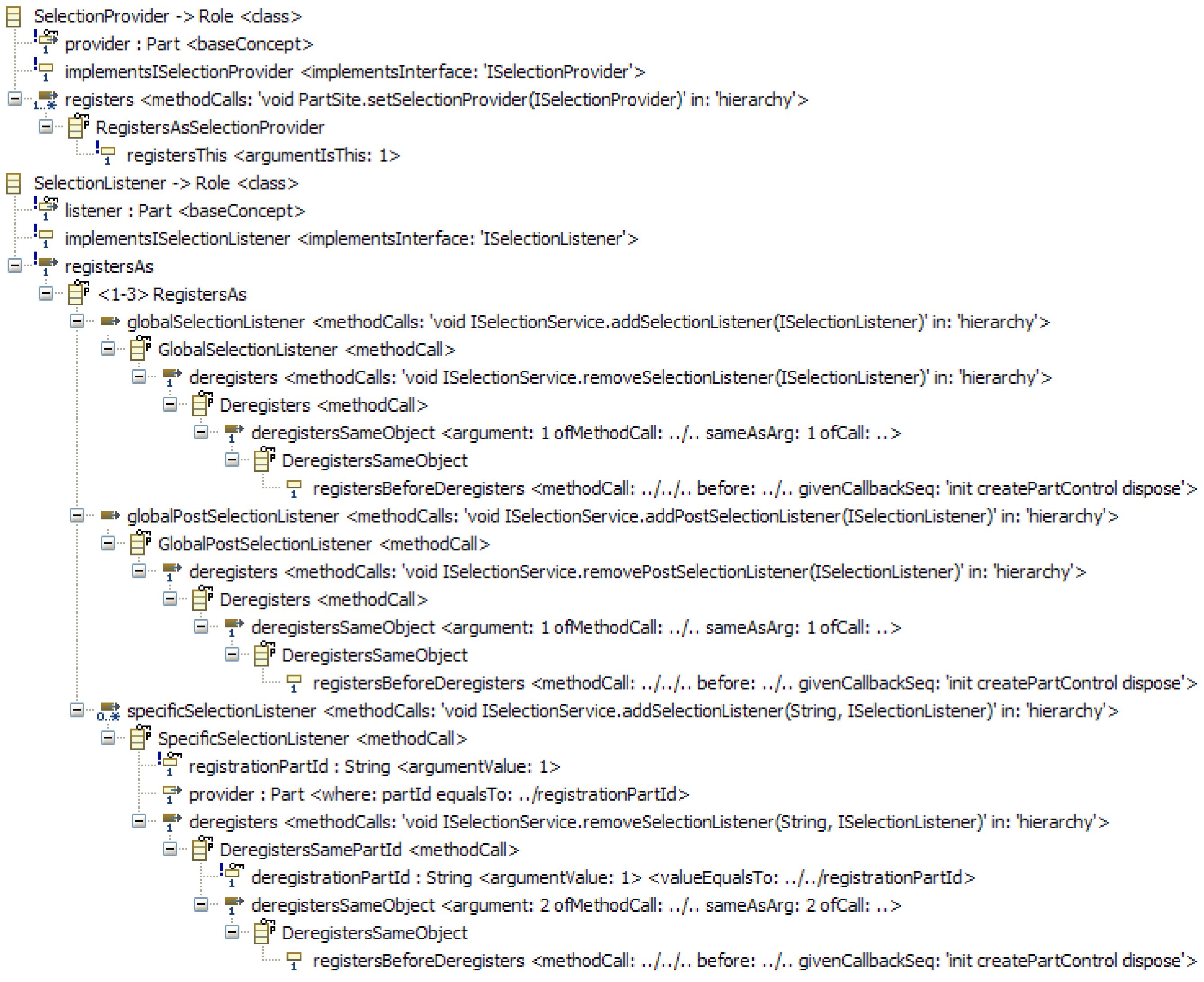
```
WorkbenchPartInteractions <project>
  project : String <projectName>
  parts
    Part <class>
      name : String <className>
      package : String <qualifier>
    ViewPart -> Part
      partId : String <viewPartId>
      implementsIViewPart <implementsInterface: 'IViewPart' concrete: true>
    EditorPart -> Part
      partId : String <editorPartId>
      implementsIEditorPart <implementsInterface: 'IEditorPart' concrete: true>
  roles
    Role
    SelectionProvider -> Role <class>
    SelectionListener -> Role <class>
    PartListener -> Role <class>
    AdapterProvider -> Role <class>
    AdapterRequestor -> Role <class>
```

Fig. 5 Overview of the metamodel of the Workbench Part Interaction FSML

another standard workbench view *Property Sheet*, which displays properties of a selection made anywhere in the workbench.

Figure 5 presents an overview of the metamodel of the WPI FSML. An instance of the class `WorkbenchPartInteractions` corresponds to the modeled project and contains a number parts through the reference `parts` and a number of part interactions through the reference `interactions`. The class `Part` is the type of the reference `parts`. Its instances correspond to Java classes. `ViewPart` is a subclass of `Part` and its instances correspond to Java classes, which implement `org.eclipse.ui.IViewPart` interface. Similarly, instances of `EditorPart` correspond to Java classes, which implement `IEditorPart` interface. The model only represents concrete Java classes because we want to illustrate interactions between parts that can actually be instantiated in the Workbench. That also means that the model must represent behaviour inherited from abstract superclasses. As mentioned before, workbench parts must be declared in plug-in manifest files, in order to become available to the user. The attribute `partId` corresponds to the unique part id specified in the part declaration. In this paper, we abstract from the further details of part implementation, and we focus on part interactions, which are the main area of concern of the WPI FSML.

Instances of the classes `SelectionProvider`, `GlobalSelectionListener`, `SelectionListenerFrom`, `PartListener`, and `AdapterRequestor` correspond to Java classes that play certain roles in the interactions. However, the instances should only correspond to those Java classes that are workbench parts, that is, the Java classes that instances of `Part` correspond to. In the metamodel we refer to the Java classes that other features correspond to using *base-concept references*. Base-concept references can be thought of as a way of *importing context elements*. This powerful mechanism allows us to specify ad-



```
SelectionProvider -> Role <class>
  provider : Part <baseConcept>
  implementsISelectionProvider <implementsInterface: 'ISelectionProvider'>
  registers <methodCalls: 'void PartSite.setSelectionProvider(ISelectionProvider)' in: 'hierarchy'>
    RegistersAsSelectionProvider
      registersThis <argumentIsThis: 1>
SelectionListener -> Role <class>
  listener : Part <baseConcept>
  implementsISelectionListener <implementsInterface: 'ISelectionListener'>
  registersAs
    <1-3> RegistersAs
      globalSelectionListener <methodCalls: 'void ISelectionService.addSelectionListener(ISelectionListener)' in: 'hierarchy'>
        GlobalSelectionListener <methodCall>
          deregisters <methodCalls: 'void ISelectionService.removeSelectionListener(ISelectionListener)' in: 'hierarchy'>
            Deregisters <methodCall>
              deregistersSameObject <argument: 1 ofMethodCall: ../.. sameAsArg: 1 ofCall: ..>
                DeregistersSameObject
                  registersBeforeDeregisters <methodCall: ../../.. before: ../.. givenCallbackSeq: 'init createPartControl dispose'>
      globalPostSelectionListener <methodCalls: 'void ISelectionService.addPostSelectionListener(ISelectionListener)' in: 'hierarchy'>
        GlobalPostSelectionListener <methodCall>
          deregisters <methodCalls: 'void ISelectionService.removePostSelectionListener(ISelectionListener)' in: 'hierarchy'>
            Deregisters <methodCall>
              deregistersSameObject <argument: 1 ofMethodCall: ../.. sameAsArg: 1 ofCall: ..>
                DeregistersSameObject
                  registersBeforeDeregisters <methodCall: ../../.. before: ../.. givenCallbackSeq: 'init createPartControl dispose'>
      specificSelectionListener <methodCalls: 'void ISelectionService.addSelectionListener(String, ISelectionListener)' in: 'hierarchy'>
        SpecificSelectionListener <methodCall>
          registrationPartId : String <argumentValue: 1>
          provider : Part <where: partId equalsTo: ../registrationPartId>
          deregisters <methodCalls: 'void ISelectionService.removeSelectionListener(String, ISelectionListener)' in: 'hierarchy'>
            DeregistersSamePartId <methodCall>
              deregistrationPartId : String <argumentValue: 1> <valueEqualsTo: ../../registrationPartId>
              deregistersSameObject <argument: 2 ofMethodCall: ../.. sameAsArg: 2 ofCall: ..>
                DeregistersSameObject
                  registersBeforeDeregisters <methodCall: ../../.. before: ../.. givenCallbackSeq: 'init createPartControl dispose'>
```

Fig. 6 Metamodel for the WPI FSML concepts related to selection handling

ditional features with respect to artifact elements that other elements in the model, or even in other models, correspond to. Figure 6 presents details of the interactions concerned with selection handling. An instance of the class `SelectionProvider` corresponds to the same Java class that the target (value) of the `provider` base-concept reference corresponds to. The `SelectionProvider` concept specifies that a part is a selection provider if it implements `org.eclipse.jface.viewers.ISelectionProvider` interface and registers with Selection Service by calling `void org.eclipse.ui.internal.PartSite.setSelectionProvider(ISelectionProvider)` method. Similarly, the `SelectionListener` concept specifies that a part is a selection listener if it implements `org.eclipse.ui.ISelectionListener` interface. Concept `GlobalSelectionListener` specifies that a part, which is a selection listener, registers with Selection Service by calling `void org.eclipse.ui.ISelectionService.addSelectionListener(ISelectionListener)`, and deregisters by calling `removeSelectionListener(ISelectionListener)` method. The attribute `registersBeforeDeregisters` corresponds to the fact that the method call the attribute `registersWithService` corresponds to occurs before the method call instance of the class `Deregistration` corresponds to, with respect to the given callback sequence. The callback sequence specifies that the framework calls methods `init`, `createPartControl`, and `dispose` in the specified order. The attribute `deregistersSameObject` corresponds to the fact that the same object is the argument of the registration and deregistration method calls. Concept `SelectionListenerFrom` corresponds to a selection listener which listens to events from parts with given part ids. The feature ⟨`registersWithService`, `RegistersWithService`⟩ corresponds to a number of registration method calls, and the attribute `registration-`

`PartId` corresponds to the part id used for a single registration. The reference `provider` represents a referential integrity constraint, that part ids used for the registration must be part ids of existing parts. The value of the reference is an instance of `Part` retrieved by a model query, where the value of the attribute `partId` of the part equals to the part id used for the registration. The feature ⟨`deregistersWithService, DeregistersWithService`⟩, corresponds to the deregistration method call, whose first argument's value is the same as the part id used for the registration.

Figure 7 presents details of the interactions concerned with handling part events and adapters. Concept `PartListener` is very similar in structure to `GlobalSelectionListener` concept, and specifies that a part is a part listener if it implements `org.eclipse.ui.IPartListener` interface and registers and deregisters with Part Service. Note that the same class `Deregistration` is used as the type of the reference `GlobalSelectionListener::deregistersWithService`.

The interaction between adapter requestors and adapter providers is the most complex one and very difficult to see in the completion code. Concept `AdapterRequestor` specifies that a part requests an adapter by calling `Object org.eclipse.core.runtime.IAdaptable.getAdapter(Class)` method. The attribute `adapter` corresponds to the name of type of the requested adapter. The feature ⟨`adapterProvider, AdapterProvider`⟩ corresponds to a number of parts, whose `getAdapter()` method returns object of the requested adapter type. The feature ⟨`providesAdapter, ProvidesAdapter`⟩ corresponds to the method `getAdapter` of the context Java class. The attribute `providesAdapter` corresponds to the fact that the context Java method can return an object assignable to the requested adapter type.

The WPI FSML is an interesing example for a number of reasons. First, it addresses the problem of understanding possible collaborations among objects, which are implemented in the framework and cannot be directly identified in the code without understanding the framework. The *adapter requestor/provider* interaction is particularly difficult to comprehend as it usually occurs together with *part listener* and *selection listener* interactions, where an adapter is requested from the source of an event, in response to the event propagated by a service.

Second, it shows that often specialized static analyses are necessary in order to be able to recognize concept instances. For example code queries of `returnsObjectOfType`, `methodCall: before: givenCallbackSeq:`, and `argument: ofMethodCall: sameAsArg: ofCall:` mappings.

Third, it takes into account behaviour inherited from super classes, by including method calls that can be found in the super classes (as specified by `in:` clause of the `methodCalls` mapping), and inherited methods (as specified by `inherited: true` clause of `methods` mapping).

And finally, it illustrates how multiple FSMLs can be composed by means of *base-concept references.*

5 Reverse engineering

As previously described, the goal of reverse engineering is the creation of a model that describes existing completion code. The process of reverse engineering is driven by the metamodel of an FSML, which consists of the abstract syntax and the mapping of abstract syntax to framework API. Abstract syntax defines the structure of all possible models or, in other words, all possible configurations of features. The mapping provides code queries, which can be used for determining the presence or value of features.

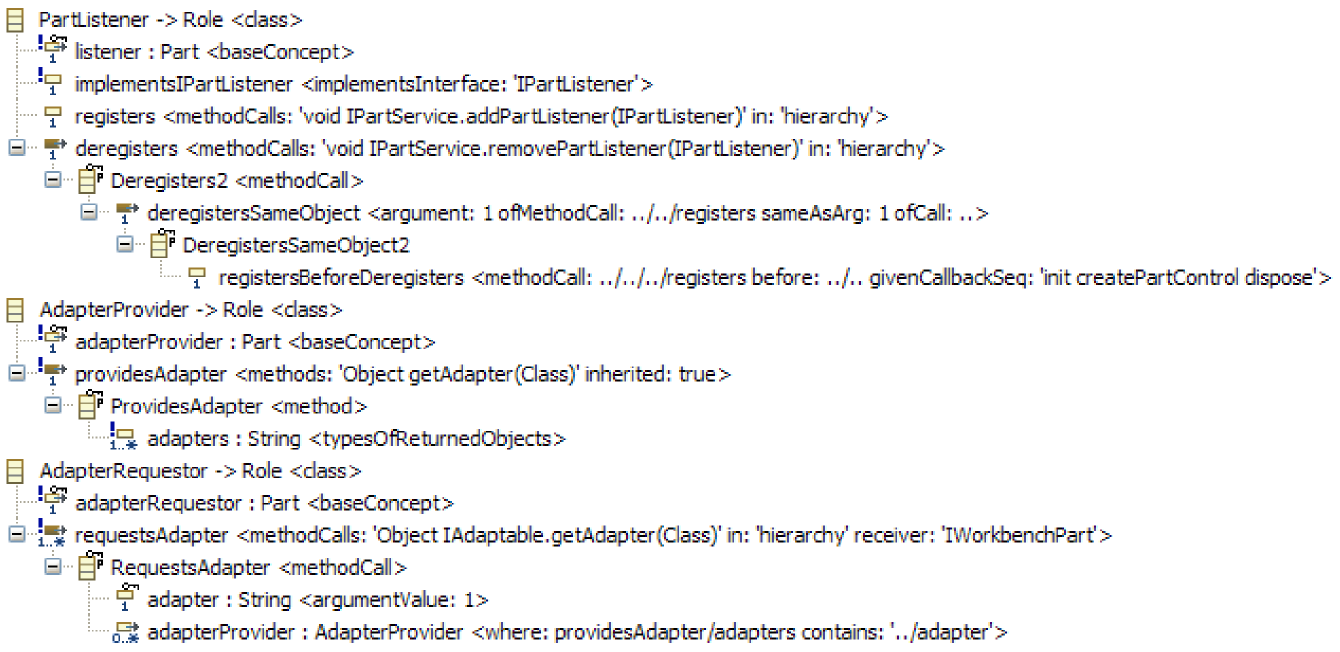
The process of reverse engineering a) creates instances of classes, b) sets instances of classes as values of containment references, c) sets the values of attributes, and d) sets values of base-concept references. Values of non-base-concept and non-containment references, which usually correspond to referential integrity constraints, are never set during reverse engineering. Instead, model queries are executed after the model is created and all data is available.

The reverse engineering process follows the containment hierarchy of the metamodel, in a depth-first manner, and begins with a single instance of the class that corresponds to the modeled project. Each code query may navigate up the already created feature hierarchy to locate a parent context feature that corresponds to the required context element. Classes, attributes and references are processed as follows.

Processing an instance *i* of a class *c*. Attributes and references of the class c, including all inherited attributes and references, are processed in the order of their appearance in the metamodel. The first essential feature determined to be missing stops the process and the instance i is removed from its containing reference. If all essential features are present, a correspondence link is established between the instance i and the corresponding artifact element.

Processing an attribute *a* of instance *i*. A code query is executed for the mapping attached to the attribute to determine the value. The value or a collection of values returned by the query is assigned to the attribute. If a has a non-null value, a non-empty collection of values, or a is true, a correspondence link is established between the attribute and the corresponding artifact element or elements. Note, that a single valued feature can only have one value, but it can still correspond to multiple artifact elements.

Processing a containment reference *r* of instance *i*. Let t be a class which is the type of the reference r. The reference r can contain instances of the



```
PartListener -> Role <class>
  listener : Part <baseConcept>
  implementsIPartListener <implementsInterface: 'IPartListener'>
  registers <methodCalls: 'void IPartService.addPartListener(IPartListener)' in: 'hierarchy'>
  deregisters <methodCalls: 'void IPartService.removePartListener(IPartListener)' in: 'hierarchy'>
    Deregisters2 <methodCall>
      deregistersSameObject <argument: 1 ofMethodCall: ../../registers sameAsArg: 1 ofCall: ..>
        DeregistersSameObject2
          registersBeforeDeregisters <methodCall: ../../../registers before: ../.. givenCallbackSeq: 'init createPartControl dispose'>
AdapterProvider -> Role <class>
  adapterProvider : Part <baseConcept>
  providesAdapter <methods: 'Object getAdapter(Class)' inherited: true>
    ProvidesAdapter <method>
      adapters : String <typesOfReturnedObjects>
AdapterRequestor -> Role <class>
  adapterRequestor : Part <baseConcept>
  requestsAdapter <methodCalls: 'Object IAdaptable.getAdapter(Class)' in: 'hierarchy' receiver: 'IWorkbenchPart'>
    RequestsAdapter <methodCall>
      adapter : String <argumentValue: 1>
      adapterProvider : AdapterProvider <where: providesAdapter/adapters contains: '../adapter'>
```

Fig. 7 Metamodel for the WPI FSML concepts *PartListener* and *AdapterRequestor/Provider*

concrete subclasses of the class t, including instances of the class t if it is concrete. Let c be a concrete class whose instances can be contained by the reference r.

There is a number of possibilities when processing a containment reference, but only one of the following cases can apply. The cases are examined in the given order for each possible class c.

1. The reference r has a mapping attached. A code query is executed for the mapping and instances of the class c are created for each result of the query.
2. The class c has a mapping attached. A code query is executed for the mapping and instances of c are created for each result of the query.
3. The class c contains an base-concept reference ar. An instance of c is created for every possible target of the base-concept reference ar and ar of each instance is set to that target.
4. The class c has an essential feature with a mapping that can be used to form a code query to retrieve elements that instances of c correspond to. A code query is formed and instances of c are created for each result of the query.
5. None of the above cases applies. Instances of the class c are created to satisfy lower bound of the multiplicity of the reference r if r is multivalued, or a single instance of c is created if r is single valued.

Each instance of the class c is processed recursively. Note that correspondence links are not created directly for reference r. Instead, a correspondence link may be created for each instance of the class c. A feature represented as the containment reference r is considered present if it contains at least one child.

Processing of non-containment, non-base-concept references. After the model has been completely created, model queries attached to the non-containment, and non-base-concept references are executed and the values of the references are set.

6 Forward engineering

The goal of forward engineering is the creation of completion code that implements the design specified in a model expressed using an FSML. The forward engineering process is driven by the model. The process traverses the model in the depth first manner and is very similar to the reverse engineering process, with the difference that code transformations are used instead of code queries. The process begins with the root instance of the model which corresponds to the modeled project. Some mappings may not have a code transformation defined.

Processing an instance i of a class c. If a class has an base-concept reference ar, no new artifact element is created, because it has already been created for the target of the base-concept reference, before. If a class has a mapping attached, code transformation is executed to create a new artifact element. In some cases, a mapping attached to the containing reference, i.e., a reference that contains instance i, is used as the source of parameters for the code transformation. Attributes and references of the class c, including all inherited attributes and references, are processed in the order of their appearance in the metamodel.

Processing an attribute *a* of instance *i*. A code transformation attached to the attribute a is executed.

Processing a containment reference *r* of instance *i*. A code transformation attached to the reference r is executed. Each instance contained by the reference r is processed recursively.

Processing of non-containment, non-base-concept references. Non-containment and non-base-concept references are ignored during forward engineering.

Table 2 Key specifications

icon	metamodel element	name
🔑	attribute, reference	key
🔑P	class	parent key
🔑I	class	index key

7 Agile Round-Trip Engineering

The goal of round-trip engineering is keeping a number of artifacts, such as models and code, consistent by propagating changes among the artifacts. Making artifacts consistent by propagating changes is also referred to as *synchronization*. Round-trip engineering is a special case of synchronization that can propagate changes in multiple directions, such as from models to code and vice versa. Round-trip engineering is hard to achieve in a general setting due to the complexity of the non-isomorphic mappings between the artifacts.

FSMLs enable round-trip engineering over non-trivial mappings that close the abstraction gap between the framework-provided concepts and the completion code. The mapping can be precisely defined because the framework prescribes a finite set of framework-stipulated implementation choices.

In this section, we present a particular approach, which we refer to as *agile round-trip engineering*. The approach supports on-demand, rather than instantaneous, synchronization. The artifacts to be synchronized can be independently edited by developers in their local workspaces, and the reconciliation of the differences can be done iteratively. Furthermore, the agile approach assumes that a model can be completely retrieved from the code using static analysis. We believe that our approach fits agile development particularly well because it supports collaborative, CVS-style development and models do not have to be maintained separately if not desired.

Fig. 8 shows the artifacts and processes involved in agile round-trip engineering. The intention of agile round-trip engineering is to synchronize the current *asserted model*, which represents the intended model of the application, and the current *framework completion code*, which may be inconsistent with the asserted model. The asserted model and the completion code that are consistent are also referred to as being *reconciled*. In order to synchronize the asserted model and the completion code, the current *implementation model* is automatically derived from the current code. Furthermore, we assume that the *last reconciled model* contains the latest copy of each feature instance that was archived after the features's most recent synchronization. Special cases occur if any of the three artifacts, namely the asserted model, the last reconciled model, or the completion code, are missing. These cases include situations where the code has to be first created from an existing model, the model has to be first created from existing code, or where independently created model and code need to be synchronized for the first time.

Given at least the asserted model or the completion code, the synchronization procedure involves the processes described in the following subsections.

1. Reverse engineering. The outcome of the reverse engineering (see Sec. 5) is the implementation model. In the case that there is no code, the implementation model contains only one instance of the class that corresponds to the modeled project.

2. Comparison. This process compares the asserted model and the implementation model using the last reconciled model as a reference. The comparison is similar to the *three-way compare* in the CVS, where the comparison of two files uses their most recent common revision as a reference. Corresponding concept and feature instances from different models are compared. The correspondence between concept instances or feature instances is established based on the values of their *FSML ids*, i.e., two instances are considered as corresponding if their FSML ids are the same. Table 2 presents icons used in the metamodels from Figures 3-6 to specify how unique FSML ids are calculated for concept and feature instances. An FSML id for an instance i of class c includes a) name of the class c, b) index of i in the collection of children of the parent of i, if c is annotated with *index key*, c) a value of every attribute annotated with *key*, d) an FSML id of target of every reference annotated with *key*, e) the FSML id of the parent of i, if c is annotated with *parent key*. An FSML id for an attribute or reference f of instance i includes a) the FSML id of i, b) name of f, c) value of f, if f is a multivalue attribute or reference.

The result of comparing two concept instances or two features is a *synchronization state*, which characterizes whether a modification, such as addition, removal or change, has occurred exclusively in the model, exclusively in the code, or consistently in the code and the model, or inconsistently in the code and the model. Synchronization states are computed according to decision tables presented in Table 3 for class instances and table 4 for attributes and non-containment references. The column *state* contains synchronization state decided for an element if conditions from the previous columns were satisfied. The column *detected situation* contains descrip-

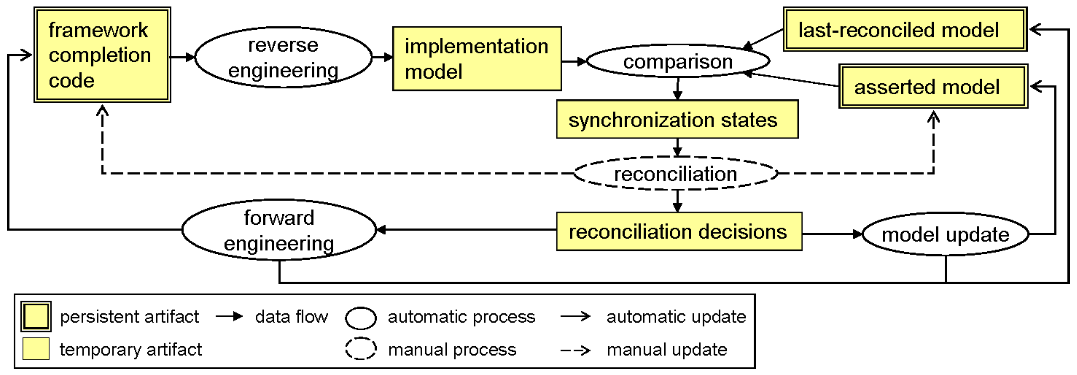


Fig. 8 Artifacts and processes of agile round-trip engineering

#	a	i	l	ac	f	r	aloc	state	detected situation
1	1	1	1	1	0	0	0	✓	unchanged
2				0	1	1	0	⇄	at least one forward and one reverse feature change, no conflict features
3				0	1	0	0	✳→	at least one forward feature change, no conflict features
4				0	0	1	0	✳←	at least one reverse feature change, no conflict features
5				0	0	0	1	✳↔	at least one conflict feature
6	1	1	0	1	0	0	0	+	added consistently to the model & the code
7				0	-	-	-	+↔	added inconsistently to the model & the code, at least one conflict feature
8	1	0	1	-	-	-	-	−←	removed from the code
9	1	0	0	-	-	-	-	+→	added to the model
10	0	1	1	-	-	-	-	−→	removed from the model
11	0	1	0	-	-	-	-	+←	added to the code
12	0	0	1	-	-	-	-	−	removed from the model & the code

Table 3 Class instance synchronization state decision table

#	av	iv	lv	condition	state	detected situation
1	1	1	1	$av = iv = lv$	✓	unchanged
2	1	1	1	$av = iv \wedge av \neq lv$	✳	changed consistently in the model & the code
3	1	1	0	$av = iv$	+	added consistently to the model & the code
4	1	1	1	$av \neq iv \wedge av = lv$	✳←	changed in the code
5	1	1	1	$av \neq iv \wedge iv = lv$	✳→	changed in the model
6	1	1	1	$av \neq iv \neq lv \neq av$	✳↔	conflict: changed inconsistently in the model & the code
7	1	1	0	$av \neq iv$	+↔	conflict: added inconsistently to the model & the code
8	1	0	1	$av = lv$	−←	removed from the code
9	1	0	1	$av \neq lv$	−←✳→	conflict: removed from the code, changed in the model
10	1	0	0	-	+→	added to the model
11	0	1	1	$iv = lv$	−→	removed from the model
12	0	1	1	$iv \neq lv$	−→✳←	conflict: removed from the model, changed in the code
13	0	1	0	-	+←	added to the code
14	0	0	1	-	−	removed from the model & the code

Table 4 Attribute and non-containment reference synchronization state decision table

tions of the modifications made to the artifacts identified by the comparison.

We have identified 15 distinct synchronization states, assuming that the last reconciled model is never changed manually. In this paper we use icons to represent synchronization states for brevity. Symbols +, *, and – indicate *addition*, *change*, and *removal*, respectively. Arrows in the icons suggest the direction of change propagation needed to reestablish consistency. The arrow $\hookleftarrow$ indicates that a change occurred in the code and suggests model update. We refer to the synchronization states that contain only $\hookleftarrow$ as *reverse states*. The arrow $\mapsto$ indicates that a change occurred in the model and suggests code update. We refer to the synchronization states that contain only $\mapsto$ as *forward states*. The absence of an arrow indicates that no change propagation is necessary. The arrow $\leftrightarrow$ and the icons $\overset{\leftrightarrows}{*}$ and $\overset{\leftrightarrows}{*}$ indicate that incompatible changes occurred in both the model and the code. Each synchronization state is described in the column *detected situation*.

The decision tables should be read from left to right. Table 3 specifies how a synchronization state is computed for the corresponding class instances. The columns a, i, and l contain 1 if an instance is present in the asserted model, implementation model, and last reconciled model, respectively. The columns *ac*, *f*, *r*, and *aloc* contain 1 if all subfeatures are consistent (ac), no conflicts and at least one subfeature has a forward state (f), no conflicts and at least one subfeature has a reverse state (r), and at least one subfeature has a conflict state ($aloc$). Value "-" indicates that given entry does not influence the decision. For example, the row #8 specifies the case, where an instance of a class is present in both the asserted model and the last reconciled model, but is missing in the implementation model. The resulting synchronization state is $\overset{-}{\hookleftarrow}$ because the instance must have been removed from the code and therefore the corresponding instance has to be removed from the asserted model to re-establish consistency. The rows #6 and #7 specify the cases where new class instances were added to both the model and the code. If all subfeatures of the instances are consistent, than the instances were added consistently (row #6). Otherwise, if at least one subfeature has a conflict, than the instances were added inconsistently (row #7) and the conflict must be resolved in order to make the added instances consistent. Note, that the subfeatures can have no "*" and no "–" states; the only possible states for the subfeatures in this case are "+" states.

Table 4 specifies how a synchronization state is computed for the corresponding attributes or a non-containment references. Containment references are not compared directly; the values of containment references (i.e., class instances), are compared instead. The columns *av*, *iv*, and *lv*, contain 1 if an attribute or a reference has a value in the asserted model, the implementation model, and the last reconciled model, respectively. The column *condition* specifies additional condition the values must satisfy in order to reach the decision. For example, the row #1 specifies the case, where an attribute or a reference has the same values in the asserted, the implementation, and the last reconciled models. The resulting synchronization state is ✓ (unchanged). The row #2 specifies the case, where the value was changed in the implementation model, and therefore the asserted model needs to be updated. The resulting synchronization state is $\overset{*}{\hookleftarrow}$. The row #12 specifies the case, where the value appears only in the asserted model, and therefore it must have been added to the asserted model. The table also illustrates the role of the last reconciled model in determinig the changes that occurred in the model and in the code. For example, the rows #9 and #11 specify the cases where incompatible changes ocurred in the model and the code. Also, the last reconciled model allows distinguishing between *addition* and *change* as, for example, in the rows #5 and #7.

3. Reconciliation. For all elements with synchronization state other than ✓, +, *, and − a *reconciliation decision* needs to be made by the user. A reconciliation decision specifies whether an addition, a removal, or a change should be propagated from the model to the code or vice versa. For the forward states, the possible decisions are *enforce* and *override and update*. For the reverse states, the possible decisions are *update* and *override and enforce*. For the state $\overset{\mapsto}{\hookleftarrow}$, the possible decision is *enforce and update*. Decision *ignore* is applicable to any state.

Reconciliation may also require manual editing of the completion code or the asserted model (e.g., by providing new values for the attributes), in which case the synchronization states need to be recomputed.

Table 5 describes the modifications executed for a given synchronization state (column) and reconciliation decision (row). Letters m and c stand for the model and the code, respectively. Symbol "↓" indicates processing of subfeatures. Symbol "-" indicates no modification. No symbol indicates that the given state/decision combination is invalid.

4. Code and asserted model update. Finally, any necessary changes are executed according to the reconciliation decisions. The enforce decisions trigger the execution of the code transformations and the update decisions force an update of the asserted model with the values from the implementation model. The last reconciled model is updated with the copies of the reconciled class and feature instances. Additionally, the last reconciled model needs to be updated for the states ✓, +, *, and −.

Code transformations are executed according to the order described in Section 6, but only for elements whose synchronization states are different than ✓, +, *, and −.

	Forward & reverse states							Conflict states				Consistent states			
	+↦	✳↦	−↦	⇆	+↩	✳↩	−↩	✳↔	+↔	⇆	⇆	✓	+	✳	−
enforce	+c	*c	−c												
update					+m	*m	−m								
enforce & update				↓											
override & enforce					−c	*c	+c	*c	*c	+c	−c				
override & update	−m	*m	+m					*m	*m	−m	+m				
ignore	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 5 Reconciliation actions executed for given synchronization state (column) and reconciliation decision (row)

8 Framework for building FSMLs

The prototype implementations of the FSMLs presented in section 4 were build on top of a generic FSML framework. The FSML framework implements generic algorithms for forward and reverse engineering, model comparison, and code and model update, as described in the previous sections.

The framework supports pluggable mapping interpreters. By default, it includes an interpreter for Java mappings. We also implemented custom mapping interpreter for XML used by the Struts FSML and a custom mapping interpreter for handling Eclipse plug-in manifest files used by the WPI FSML. Each interpreter declares which mappings it supports and is called to execute code queries and code transformations at different times.

The framework provides generic support for traceability to code by means of persistent correspondence links, the view *Model-Code Navigation*, which displays correspondence links for a model element, the view *Model-Code Synchronization*, which displays results of comparison, and base classes for implementing new model wizard and synchronize action. The reconcile action is generic and is provided by the framework. The custom part of the synchronize action only involves plugging-in custom mapping interpreters.

The FSML framework is based on the Eclipse Modeling Framework (EMF). EMF provides a metamodeling notation called *Ecore*, which we used for creating the metamodels. We express the mappings using metamodel annotations (tagged values). The metamodels presented in Section 4 were automatically rendered using the *FSML Ecore* view, which is also a part of the FSML framework.

Implementing a new FSML involves a) creating abstract syntax using Ecore, b) annotating metamodel elements using annotations defined by the available mapping interpreters to specify the mappings, c) annotating metamodel elements using key annotations, d) generating metamodel implementation using EMF generator, e) implementing a new model wizard, f) implementing a synchronize action, and g) implementing new custom mapping interpreter, if necessary. Activities a-d) usually require multiple iterations. Activities e-f) are very simple. For example, the new model wizard of the WPI FSML has 28 lines of code and the synchronize action has 27 lines of code. The implementation of a mapping interpreter is non-trivial and includes implementing code queries and code transformations, which support incremental query and update of the artifact, context element handling, support for traceability links, and about 20 callback methods, which are called by the FSML framework at various points of the execution of generic reverse- and forward engineering, and reconciliation algorithms.

9 Related Work

There is a large body of related work; however, we highlight only a few important works in each category.

Domain-Specific Modeling Languages (DSMLs) and frameworks. The idea of putting a DSML on top of a framework is not new. Roberts and Johnson consider language-based tools on top of frameworks as the highest maturity level in framework evolution [15]. They advocate that black-box frameworks are particularly well-suited for use with a DSML on top. However, configuration alone does not allow fine-grained customization, and it often has to be combined with open-ended programming in practice. We are not aware of any work exploring DSMLs for frameworks with round-trip engineering support except for [2], which this paper is an extension of.

General-purpose code analysis tools for architecture recovery and program comprehension. There is an enormous body of work in this category. Two subcategories are prominent. The first subcategory includes tools (e.g., JQuery [7], JTL [6], XIRC [10]) that allow code querying for typical static code structures, and dependencies. Such tools usually build a database of facts about the structure of the program, and execute queries over the database. XIRC also allows representing artifacts other than program code in its XML database, which allows for checking referential integrity constraints. In contrast to these tools, our approach uses whatever specialized analyses are needed for detecting instances of framework concepts. For example, in order to recognize the *adapter requestor/provider* interaction, an analysis is needed to determine, whether a method can return an object of a certain type, which is different than simply querying for the return type of the method. Also, rather than building a complete database of program facts, FSMLs extracts only the information relevant with respect to FSML concepts, which drastically reduces memory and time footprints. Often, an FSML requires the retrieval of static values (string or type literals) of variables and method call arguments, which are not provided by the generic code query tools.

The other subcategory groups works on detecting design patterns in code (e.g., [17]). The main problem with these approaches is that a design pattern can be implemented in the code in a multitude of different ways. Our approach avoids this problem by limiting itself to the detection of API-stipulated concepts and features, which is more tractable because the framework prescribes all possible ways of implementing a concept. Also, frameworks usually implement a particular variants of design patterns.

Framework instantiation. Most approaches in this category only support forward engineering without incremental update. They usually utilize wizards and scripts, as implemented in many industrial tools, including Eclipse. Unfortunately, such wizards or scripts can usually be run only once since they cannot take manual customizations into account. This problem is sometimes addressed by strictly separating the generated code from the manual one using techniques such as protected regions, subclassing of generated classes, and partial classes in C#. However, we believe that the separation approach affords less flexibility in customizing the generated code, in particular, when the generated code dictates the structure of customizations, such as in the case of template-based code generation.

Many approaches have been proposed to assist the framework-instantiation process through active documentation [4,12,14,18], which specifies and interactively guides the developer through available hotspots, instantiation tasks and possible implementation choices. Attempts for automating the framework instantiation such as [4] offer code generation based on developer's choices, but cannot analyze existing code for correctness. Also, the generator (the wizard) is unable of analyzing existing code in order to determine which choices have been made in the previous run.

AHEAD [3] offers concept configuration controlled by feature models, where features represent modular slices through multiple artifacts, such as code and XML files. The slices may be composed to produce framework completion code. Step-wise refinement is a generative approach, which supports only forward engineering without the ability to update customizations.

Approaches, such as SCL [13], allow framework developers formalizing framework rules using a constraint language. The constraints can be checked on demand against the completion code and detect rule violations. Such approaches could be used to define the reverse mappings of FSMLs. Another way of specifying constraints over programs is a pluggable type system [1], where user-defined typing constraints enrich the type system of the programming language. Pluggable type systems could potentially be used for specifying constraints and rules for framework concepts, however currently there is limited support for handling variability in concepts, as well as for handling multiple types of artifacts.

Design Fragments [11] is an approach to framework instantiation, where a commonly found patterns of framework usage are encoded as design fragments. Design fragments are mined from existing example completion code. Similarly to SCL [13], the conformance of the completion code to a given design fragment can be automatically checked. However, the completion code has to be manually bound to the design fragment it implements, whereas in the FSML approach, the code is bound to the features automatically via the correspondence links. Also, completion code developers have to manually browse the catalog of available design fragments to find the one that fits the requirements. In the FSML approach, developers simply select the features they want to have implemented.

Round-trip engineering. According to Sendall and Küster the main difference between round-trip engineering and forward and reverse engineering is that round-trip engineering takes both artifacts into account with the intention of reconciling them, whereas forward and reverse engineering typically create new artifacts, potentially replacing the old versions [16].

Round-trip engineering between UML and object-oriented languages such as Java is supported by several commercial UML modeling tools. The provided synchronization can be instantaneous or on demand as in our approach. However, the mappings supported by these tools are rather simple one-to-one mappings between UML classes and Java classes.

Round-trip engineering is often provided in the context of Enterprise Java Beans (EJBs) where a single EJB component maps to a class and multiple interfaces. The mappings, however, are simple sructural mappings.

10 Discussion and Future Work

The presented FSML approach makes contributions in several areas. First, we show that framework-provided concepts can be captured in metamodels of FSMLs, and we provide numerous examples taken from the three analysed frameworks. We demonstrate how framework-rules and referential integrity constraints can be captured as abstract syntax constraints. Second, we show that semantics of a DSML can be precisely specified in the context of a framework through a mapping of the abstract syntax to the framework API. Third, we present a generic framework for building FSMLs, which we used for implementing prototype implementations of the presented FSMLs. The framework is highly extensible and supports pluggable mapping interpreters, which can handle mappings for arbitrary kinds of artifacts.

The code queries of FSMLs are restricted by the available static code analysis techniques. Our agile round-trip engineering approach requires the design to be retrievable from the code, which may not always be possible using purely static analysis. This problem could be addressed by injecting design information into the source code, e.g., as code annotations. The FSML could also suggest to the application programmer how to restructure the code to make its design more explicit in the static code structure. In general, the effectiveness of the reverse engineering depends on the programming language and the type of the framework. This aspect requires further research.

In our approach, the code transformations are not required to produce fully functional code. An FSML is intended to be used in an interactive manner. The generated or transformed code is intended to be further customized. We think that generation of code fragments demonstrating the use of the framework can help application developers overcome the initially steep learning curve. In general, code transformations designed to update the code, are usually harder to devise than code queries.

FSMLs can potentially be used for automatic or semi-automatic code migration between two versions of the same framework or between two frameworks. Cheema describes a semi-automatic approach for migration of code completion from Apache Struts to Java Server Faces framework [5]. The process of migration is controlled by the model extracted using an FSML for Struts and specialized export wizard is used to rewrite the code for JSF framework.

We think that, in practice, a single FSML will typically cover a small area of a framework's concern, and multiple FSMLs will be provided for a single framework. For example, in Eclipse, in addition to WPI, another FSML could be used to specify the graphical appearance of workbench parts, and yet another to specify workbench part's menus, toolbars, and actions. Furthermore, round-trip engineering affords manual integration of completion codes created for multiple frameworks. Such integration may be difficult for completion code generated from code templates because such code can be customized in only limited ways. Integration of multiple FSMLs remains future work, however using base-concept references makes it possible to describe other aspects of a concept instance from another model, as demonstrated by the WPI FSML.

Validation. The validation of the proposed approach to modeling requires validating multiple aspects. In this paper, we present the results of applying the approach to three existing frameworks. We conducted the study to see to what extent framework-provided concepts can actually be captured as an FSML. We demonstrated in Section 4 that we were able to capture a variety of concepts and framework rules in the metamodels for a variety of frameworks.

We also validated the feasibility of reverse engineering by applying the prototype implementations of FSMLs to a number of concrete applications. We reverse engineered 71 applets using the Applet FSML. The prototype retrieved 97% of instances of the feature `showStatus`, and 79% of instances of the feature `message`, 96% of instances of the feature `getParameter`, and 73% of names of parameters. We were able to retrieve 100% of all other feature instances. We reverse engineered 3 Struts applications: Apache Roller, Mailreader, and Cookbook using the Struts FSML. The prototype retrieved 99.5% of names of forwards used in 212 method calls to the method `findForward`. We were able to retrieve 100% of all other features. We reverse engineered one Eclipse PDE plug-in, which depends on many other UI plugins using the WPI FSML. The prototype retrieved 90% of adapter types used in method calls to the method `getAdapter`. We were able to retrieve 100% of all other features. The complete data is under preparation.

We have preliminary experience with round-trip engineering; however full validation remains future work. Furthermore, user studies to confirm that the challenges of framework completion are actually addressed are necessary.

11 Conclusion

In this paper, we proposed the concept of FSMLs with round-trip engineering support. The concept addresses a number of challenges in framework-based application development, such as knowing how to write framework completion code, being able to see the design of the completion code, and the migration of the code to new framework API versions. We presented a concrete approach to representing the abstract syntax of an FSML and its mapping to code. We applied the approach to three existing frameworks, namely Java Applet, Apache Struts, and Eclipse Workbench, showing that the framework-provided concepts, features, and usage rules can be adequately captured. We also presented a set of algorithms to support round-trip engineering and a generic FSML framework, which is used to implement our sample FSMLs.

Acknowledgements We would like to thank Herman Lee for help with the implementation of the custom mapping interpreter for XML.

References

1. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. In: OOPSLA'06, pp. 57–74 (2006)
2. Antkiewicz, M., Czarnecki, K.: Framework-Specific Modeling Languages with round-trip engineering. In: MoDELS, *Lecture Notes in Computer Science*, vol. 4199, pp. 692–706 (2006)

3. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling stepwise refinement. IEEE Transactions on Software Engineering **30**(6), 355–371 (2004)
4. Braga, R.T.V., Masiero, P.C.: Building a wizard for framework instantiation based on a pattern language. In: OOIS'03, *LNCS*, vol. 2817, pp. 95–106. Springer (2003)
5. Cheema, A.P.: Struts2JSF - framework migration in J2EE using Framework-Specific Modeling Languages. Master's thesis, University of Waterloo (2007). http://hdl.handle.net/10012/3031
6. Cohen, T., Gil, J.Y., Maman, I.: JTL: the java tools language. In: OOPSLA'06, pp. 89–108 (2006)
7. De Volder, K.: JQuery: A generic code browser with a declarative configuration language. In: PADL'06, *LNCS*, vol. 3819, pp. 88–102. Springer (2006)
8. DSM Forum: Workshop on domain-specific modeling (2001-2006). http://www.dsmforum.org/DSMworkshops.html
9. Eclipse Foundation: Eclipse (2006). http://www.eclipse.org/
10. Eichberg, M., Mezini, M., Östermann, K., Schäfer, T.: XIRC: A kernel for cross-artifact information engineering in software development environments. In: WCRE'04, pp. 182–191 (2004)
11. Fairbanks, G., Garlan, D., Scherlis, W.: Design fragments make using frameworks easier. In: OOPSLA'06, pp. 75–88 (2006)
12. Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A., Viljamaa, J.: Generating application development environments for Java frameworks. In: GCSE 2001, *LNCS*, vol. 2186, pp. 163–176 (2001)
13. Hou, D., Hoover, H.J.: Using SCL to specify and check design intent in source code. IEEE Transactions on Software Engineering **32**(6), 404–423 (2006)
14. Ortigosa, A., Campo, M.: Smartbooks: A step beyond active-cookbooks to aid in framework instantiation. In: TOOLS'99, p. 131. IEEE Computer Society (1999)
15. Roberts, D., Johnson, R.: Evolving frameworks: A pattern language for developing object-oriented frameworks. In: PLoP'96. University of Illinois, Addison-Wesley (1996)
16. Sendall, S., Küster, J.: Taming model round-trip engineering. In: Workshop on Best Practices for Model-Driven Software Development (2004)
17. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from Java source code. In: ASE 2006 (2006)
18. Tourwé, T., Mens, T.: Automated support for framework-based software evolution. In: ICSM'03), pp. 148–157. IEEE Computer Society Press (2003)

Michał Antkiewicz is a Ph.D. candidate at the University of Waterloo.

Krzysztof Czarnecki is an associate professor at the University of Waterloo, where he leads the Generative Software Development lab.