Framework-Specific Modeling Languages with Round-Trip Engineering

Michał Antkiewicz and Krzysztof Czarnecki

University of Waterloo
{mantkiew, kczarnec}@swen.uwaterloo.ca
http://gp.uwaterloo.ca

Abstract. We propose *Framework-Specific Modeling Languages (FSMLs)* as a special category of *Domain-Specific Modeling Languages* that are defined on top of an object-oriented application framework. They are used to express models showing how framework-provided abstractions are used in framework-based application code. Such models may be connected with the application code through a forward and a reverse mapping enabling round-trip engineering. We also propose a lightweight and iterative approach to round-trip engineering. Furthermore, we present a proof-of-concept FSML for modeling the interaction of workbench parts within Eclipse. Finally, we identify a number of challenges, opportunities, and directions for future research on FSMLs.

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*, and means of instantiating them in the framework completion code. The concepts are instantiated by writing the completion code.

Unfortunately, framework completion can be challenging. The application programmers need to know which framework-provided concepts are available and how to instantiate them in order to get the desired effect. The instantiation, which usually involves steps such as implementing interfaces or invoking framework services, is challenging since the implementation choices provided by the framework are not always compatible. Furthermore, the developers need to be able to see how the framework-provided concepts are instantiated in the application code. The latter is challenging since some concepts instances, such as collaborations among objects, are usually scattered in the completion code.

In this paper, we identify the challenges of framework completion and characterize framework-based application development as a mixture of concept configuration and open-ended programming with restrictions. As a main contribution, we show how the challenges of framework completion can be addressed by explicitly capturing the framework-provided concepts as a *Framework-Specific Modeling Language* (FSML) with round-trip engineering. Furthermore, we propose

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 692–706, 2006.

an agile round-trip engineering approach, which is inspired by the *Concurrent Versioning System* (CVS) and its Eclipse user interface [1] and can operate over non-trivial abstraction gaps thanks to mappings enabled by FSMLs. Finally, we describe a proof-of-concept prototype implementation of a FSML with round-trip engineering for an aspect of Eclipse plug-in development and discuss the merits and limitations of our approach.

2 Running Example: Eclipse Workbench Part Interaction

Eclipse [1] 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 Application Programming Interface (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 only 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) [1]. 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 two kinds of part interactions, namely *listens to parts* and *requires adapter*. For example, the Content Outline view listens to *part activation* events by registering itself as a listener with the Workbench *Part Service* and, therefore, it participates in the *listens to parts* interaction. When an editor, such as the Java editor, is activated, the view will receive an activation event. In response to this event, the view will ask the editor for its `IContentOutlinePage` adapter, which is used to display the outline of the editor's input resource. Therefore, the view and the editor participate in the *requires adapter* interaction, with the view as a source and the editor as a target. For a detailed description of the example see [2].

3 Challenges of Framework Completion

Framework completion is often difficult due to the extensive knowledge about the framework design that is needed in order to write and understand the completion code. In particular, application developers face the following challenges.

Knowing how to complete a framework. The developers need to know what are the framework-provided concepts and how the concepts are instantiated in

the code. Creating an instance of a concept involves making implementation choices, some of which are stipulated by the framework's *application programming interface* (API). For example, creating an instance of a framework-provided concept *editor* amounts to implementing `IEditorPart` interface and contributing the editor to the Workbench in a plug-in manifest file. Framework documentation usually provides information on what framework classes should be extended, which interfaces should be implemented, and which API operations need to be called in order to create an instance of the framework-provided concept. Often however, concepts can be instantiated in many different ways and the developers need to know which implementation choices are compatible. For example, an editor can optionally be *multi-page*, in which case it has to extend the framework-provided class `MultiPageEditorPart` and override the abstract `addPages()` method. Furthermore, an editor can optionally have a *contributor*, which is used to contribute editor actions to menus and toolbars. However, if a multi-page editor has a contributor, the contributor has to extend the framework-provided class `MultiPageActionBarContributor`.

Obtaining an overview of a framework-based application. As the size of the framework completion code grows, it becomes increasingly difficult to obtain overviews of the application from different viewpoints. For example, looking at the code, it is difficult to see how many workbench parts are implemented and how they interact. Creating such an overview involves recognizing instances of concepts in the completion code, which may be challenging since it may require verifying multiple facts across the code or even in multiple artifacts. For example, recognizing that an editor is multi-page requires verifying that the editor class extends `MultiPageEditorPart` and that its contributor, which may be specified in the plug-in manifest file, extends `MultiPageActionBarContributor`.

Following the general rules of engagement for the framework. Some APIs, such as the Eclipse API, expect the developers to follow a set of general rules, which are referred to as *rules of engagement* [1]. Some of the rules are more specific, such as the requirement that certain API classes, e.g., `ContentOutline`, should not be subclassed. Examples of more general rules are that the arguments of a API method call should not be null unless explicitly allowed and long-running user operations should run in separate threads.

Repetitive code in the domain concept instantiation. Creating many instances of the same concept often involves providing repetitive, boilerplate code. For example, such code is needed when contributing a set of editor actions to the Workbench. Creating and maintaining such code manually is tedious and potentially error-prone.

Knowing how to migrate completion code after API changes. As a framework evolves, its API and rules of engagement may also change. Migrating completion code to the changed API is challenging and error-prone since it requires changes in multiple locations. For example, in earlier versions of Eclipse, the creation of a multi-page editor involved extending the `MultiPageEditor` class. Currently, the `MultiPageEditor` class is deprecated and the implementation of an editor should extend the `MultiPageEditorPart` class instead.

Migration of the code to the latest versions of Eclipse requires knowledge about what needs to be changed and how it needs to be changed to conform to the latest API.

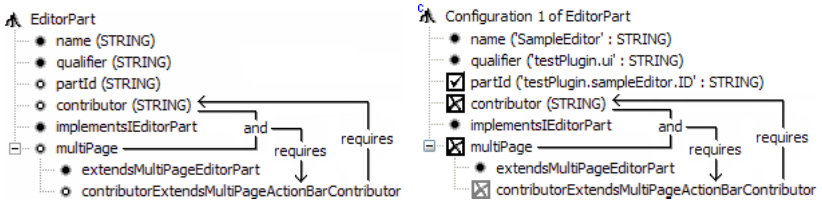
4 Framework Completion as Concept Configuration and Open-Ended Programming

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 choices provided by the framework, such as creating the code that implements a required interface, overriding the default behaviour by subclassing, or implementing code that is entirely outside the scope of the framework. A concrete example from the Eclipse domain is defining a button which will allow the user to enable or disable a part interaction at run-time. Open-ended programming is restricted in the sense that it must not violate the framework's rules of engagement.

Concept Configuration. The set of framework-stipulated implementation choices for a concept and the dependencies among these choices define all correct ways in which the concept can be instantiated as foreseen by the framework design. We can think of the implementation choices as *features* of a concept and formalize the concept's definition as a *feature model*. A feature model is a tree with the concept as its root and children representing its features [3]. Filled circles denote mandatory features and open circles denote optional features. A feature may have an attribute, which is denoted by its type shown in parenthesis. Additional dependencies between features can be expressed as constraints, such as *requires* or *excludes*. Conceptually, a feature model describes a set of all valid configurations (selections) of features.

For example, Fig. 1(a) shows a feature model describing the *editor* concept. Mandatory features have to be implemented by every instance of a concept, for example every editor has to have the `implementsIEditorPart` feature, meaning it has to implement the `IEditorPart` interface. Optional features, such as `multiPage`, are not required in every instance of a concept.

Fig. 1(b) presents a sample feature configuration for the instance of the *editor* concept, where some features have been selected (`partId`) and some eliminated (e.g., `multiPage`) and values of attributes have been specified (e.g., `'SampleEditor'` for `name`). The feature configuration satisfies all constraints implied by the feature model and, therefore, the implementation choices corresponding to the selected features (including the mandatory ones) are consistent. Note that recognizing the implementation of features of a given concept in the code also produces a configuration, which can then be checked for possible constraint violations.



(a) *Editor* concept (b) Sample feature configuration

Fig. 1. Concept definition and concept instance configuration

5 Framework-Specific Modeling Languages

A *Framework-Specific Modeling Language* (FSML) is a Domain-Specific Modeling Language [4] that is designed for a specific framework, called its *base framework*. A FSML consists of an *abstract syntax*, a *mapping of the abstract syntax to the framework API*, and, optionally, a *concrete syntax*.

A FSML explicitly captures framework-provided concepts and their features as *language concepts* in its abstract syntax. The abstract syntax encodes all valid configurations of framework-stipulated implementation choices. Models expressed using a FSML describe concept instances. The concrete syntax may offer specialized rendering of the models to enhance their comprehension.

The mapping of the abstract syntax to the framework API defines how concepts and their features map to the framework completion code. The mapping has two parts: the *forward mapping*, defining how to generate new code or update existing code for a concept instance, and the *reverse mapping*, defining how to recognize an instance of a concept in the code. The mappings are defined for every concept and every feature individually, allowing for a fine-grained control over mapping execution. Together, the forward and reverse mappings enable 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. In situations where only a subset of the FSML benefits considered in this paper is of interest, an FSML implementation may choose to provide only one of the two mappings. Furthermore, the forward mapping may also be limited to code generation only.

A 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 rules, thus guiding the developer in making correct configuration choices.

The forward mapping knows the different places where the code implementing a concept instance should be inserted in the completion code. The mappings are executed for a correct concept configuration and, therefore, produce correct

completion code. A developer can review the changes made by the forward mapping 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.

Obtaining an overview of a framework-based application. The reverse mapping 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, the reverse mapping may be adjusted to recognize broken or incomplete concept instances that need to be fixed. Finally, the reverse mapping also provides traceability between the model and the code by locating fragments of code implementing concept instances.

Following the general rules of engagement for the framework. The forward mapping 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. The forward mapping automates the creation and update of the repetitive code.

Knowing how to migrate completion code after API changes. A FSML provides a framework to help with migration of completion code to a changed API. Reverse mapping can be used to find uses of the deprecated API and specialized forward mappings can rewrite existing code to conform to the changed API.

6 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 reverse and forward mappings 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.

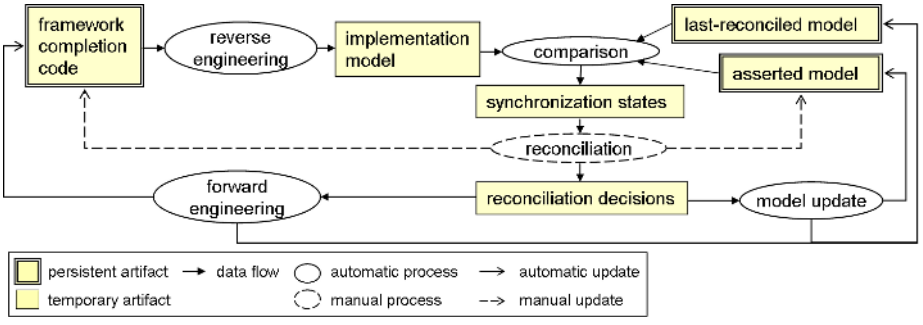


Fig. 2. Artifacts and processes of agile round-trip engineering

Fig. 2 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 concept instance that was archived after the instance'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 following processes:

1. Reverse engineering. The reverse mappings of every concept and every feature are executed on the completion code to create the implementation model. An instance of a concept is created in the implementation model iff all mandatory features are implemented. The requirement that all mandatory features have to be implemented can be relaxed to enable recognizing incomplete or broken concept instances. In the case that there is no code, the implementation model is empty.

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 instances from different models are compared. The correspondence between concept instances is established based on the values of their *key features*, i.e., features which unambiguously identify instances. For example two instances of the *editor* concept will be compared if attributes of features `name` and `qualifier` have the same values.

The result of comparing two concept instances or two features is a *synchronization state*, which characterizes whether a change, such as addition, removal or modification, 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. For example, the synchronization state *forward addition* indicates that a concept instance or a feature has been added to the asserted model (e.g., selecting `multiPage` feature in Fig. 1(b)) and, therefore, needs to be forward engineered to the code. Synchronization state *conflict* indicates that incompatible changes have been made to both the code and the asserted model (e.g., different values have been set for the `partId` feature in the model and in the code). Synchronization states are computed according to decision tables given elsewhere [2]. Here we only explain why using the last reconciled model is important. For example, if a concept instance is present in the asserted model but is missing in the implementation model, then the instance could have been added to the asserted model or removed from the implementation model. If the concept instance is also present in the last reconciled model, then the instance has been removed from the code and, therefore, the synchronization state should be *reverse removal*. On the other hand, if the instance is missing from the last reconciled model, then the instance has been added to the asserted model and, therefore, the synchronization state should be *forward addition*. The last reconciled model also plays an important role in the detection of conflicts.

3. Reconciliation. For all elements with synchronization state other than *consistent*, a *reconciliation decision* needs to be made by the user. A reconciliation decision specifies whether an addition, a removal, or a modification should be propagated from the model to the code or vice versa. For example if the synchronization state for an instance of the *editor* concept is *forward addition*, the possible decisions are *enforce*, meaning that a new editor should be created in the code, and *replace-and-update*, meaning that the asserted model should be updated to be consistent with the code and, therefore, the instance of the *editor* should be removed from the asserted model. In other cases, the possible decisions are *update* and *replace-and-enforce* [2].

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.

4. Forward engineering and asserted model update. Finally, any necessary changes are executed according to the reconciliation decisions. Forward decisions trigger

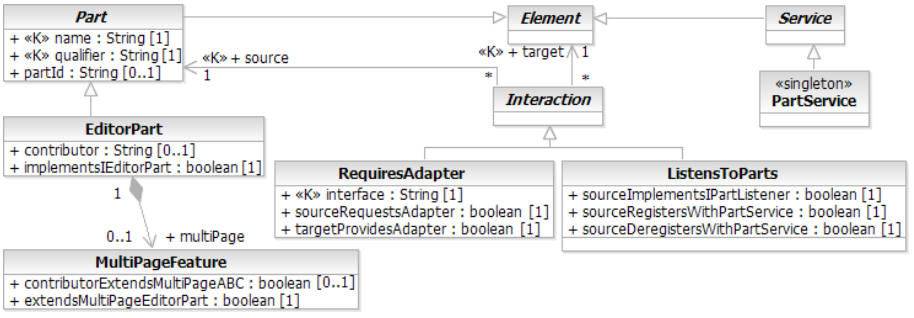


Fig. 3. Fragment of the metamodel of the WPI FSML expressed in MOF

the execution of the forward mappings and reverse 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 reconciled concepts. The execution of the individual forward mappings needs to be properly scheduled in order to be correct.

7 Eclipse Workbench Part Interaction (WPI) FSML

In this section, we present a fragment of the design of a FSML for specifying Eclipse workbench part interactions (WPI). The current prototype implementation of the FSML consists of a metamodel defining the abstract syntax and the forward and reverse mappings, and it supports full round-trip engineering as described in the previous section. Currently, the prototype only provides abstract syntax editor. The complete design of the WPI FSML is described elsewhere [2].

Abstract Syntax. Figure 3 presents an excerpt of the metamodel of the WPI FSML. Classes `EditorPart`, `ListensToParts`, `RequiresAdapter`, and `PartService` are used to represent framework concepts described in Section 2.

The metamodel from Fig. 3 is derived from feature models such as the one presented in Fig. 1(a). Concepts such as `EditorPart` in Fig. 1(a) and composite features such as `multiPage` map to classes. Atomic subfeatures such as `name` or `partId` map to class properties. The multiplicity of a property depends on the corresponding feature type and is `1` for mandatory features and `0..1` for optional features. Properties used to unambiguously identify instances of concepts, i.e., the *key properties* are annotated with the stereotype `<<K>>`. For example, an instance of `EditorPart` is identified by its `name` and `qualifier` properties, and an instance of `RequiresAdapter` interaction is identified by its `source`, `target` and `interface` properties.

Property `partId` is an example of an optional property. An editor is not required to have a part id, in which case, the value of the `partId` property is null and indicates the absence of the feature. Mandatory features which do not have any attributes are represented as Boolean properties. In this case, `false`

indicates absence of the feature. Representing mandatory features as Boolean properties allows us to create instances for concepts partially implemented in the code. The abstract syntax also contains additional well-formedness constraints that correspond to the constraints from the feature model, such as `requires` from Fig. 1(a).

Mapping abstract syntax to the framework API. We define a mapping for every class and class property. A mapping for a property consists of a reverse part and a forward part. The reverse part is a *code query*. The forward part is a *code transformation* that reflects in the code an addition, removal, or modification of a feature in the model. A feature is modified when its attribute value is changed.

In our prototype, we have implemented the mappings in Java. For better presentation, we present the mappings using a concise pseudo-notation. For the queries, we use a number of predefined functions. For transformations we use a mixture of predefined procedures and aspect templates. An aspect generated from a template can be woven into the source code. We specify the templates using Meta-AspectJ [5] as it allows us to use AspectJ pointcuts, method introductions and inter-type declarations to specify where the code should be woven. In Meta-AspectJ, `‘[<code>]` is the quote operator, `#<variable>` and `#[<expression>]` are the unquote operators. The unquote operator splices the value of a variable or an expression.

We present fragments of mappings for *editor* and *listens to parts* concepts to highlight some of the more interesting mechanisms. We start with the mapping declaration for the *editor* concept.

mapping `EditorPart(EditorPart ep <-> Class editor);`

The declaration of the `EditorPart` mapping specifies that `ep` is bound to an `EditorPart` in the model, and `editor` is bound to a `Class` in the code. The mapping can be executed in forward and reverse directions. For example, executing the mapping in the forward direction and providing a concrete `EditorPart` instance and a null reference for `editor` will create a new class in the code. If an actual class is passed as `editor`, that class will be modified to be consistent with the `EditorPart` instance.

The above declaration is followed by mappings for individual features. We start with the key features `name` and `qualifier`.

key *name*
↩ `ep.name = editor.name;`
↦ `RENAME(editor, ep.name);`
key *qualifier*
↩ `ep.qualifier = editor.package;`
↦ `MOVE(editor, ep.qualifier);`

A mapping for a property consists of two parts: a reverse mapping indicated by the ↩ symbol and a forward mapping indicated by the ↦ symbol. For the `name` and `qualifier` properties, the reverse mappings are assignments, and the forward mappings execute the `RENAME` and `MOVE` refactorings, respectively. Mappings for some of the remaining features are as follows.

```
mandatory implementsIEditorPart
↩ ep.implementsIEditorPart = IMPLEMENTS(editor, IEditorPart);
↦ `[ declare parents : #[ep.name] implements IEditorPart ]
optional partId
↩ ep.partId = EDITORID(editor);
↦ EDITORID(ep.qualifier + "." + ep.name, ep.partId);
optional multiPage
↩ ep.multiPage = REVERSE(MultiPageFeature(ep <-> editor));
↦ FORWARD(MultiPageFeature(ep <-> editor));
```

The reverse mapping for the `implementsIEditorPart` property uses the `IMPLEMENTS` function to check if the class implements the `IEditorPart` interface. The forward mapping specifies an inter-type declaration that will add the `implements` declaration to the class, if woven. The mapping for the `partId` property uses the `EDITORID` function to retrieve values from the plug-in manifest file and the `EDITORID` procedure to set the values. Mappings for the `multiPage` property use the `FORWARD` function and the `REVERSE` procedure to execute the `MultiPageFeature` mapping.

Finally, we present a mapping for the *listens to parts* interaction.

```
mapping ListensToParts(ListensToPart ltp <-> Class s)
when Part(sp <-> s);
mandatory sourceRegistersWithPartService
↩ ltp.sourceRegistersWithPartService =
  CALLS(s, `[IPartService.addPartListener(IPartListener)]);
↦ `[private void #[sp.name].registerWithPartService() {
      getSite().getPage().addPartListener(this);
    }]
```

The reverse mapping for the property `sourceRegistersWithPartService` uses the `CALLS` function to determine whether there exists a call to `addPartListener()` method in class `s` or any of its superclasses. The forward mapping for the `sourceRegistersWithPartService` property creates a new method, `registerWithPartService()`, which contains the required registration call. Note that the programmer can move the registration call elsewhere and remove the generated method and yet, the reverse mapping will still be able to recognize the registration call.

WPI FSML prototype. We developed a prototype of the WPI FSML as an Eclipse plug-in. Abstract syntax of the language, including well-formedness constraints, is implemented using Eclipse Modeling Framework (EMF) and its model validation framework. Reverse mappings use the AST, query, and pattern matching API of Eclipse's Java Development Tools (JDT) and type inference engine of the *Infer Generic Type Arguments* refactoring [6]. Forward mappings use Eclipse's JDT Java Model and AST rewriting API. The prototype supports agile round-trip engineering. The reverse mappings are completely implemented. To date, the forward mappings support the creation of classes with methods implementing the framework-stipulated behaviour, addition of interfaces and

superclasses, and handling the plug-in manifest files. Weaving of *before* and *after* advices, and code fragment removal are not yet implemented.

The initial evaluation of the prototype involved round-trip engineering of a few Eclipse UI plug-ins as well as some of our own plug-ins. For all of these plug-ins, we were able to completely reverse engineer the models from the plug-ins' code. Furthermore, we were able to synchronize the models and the code after modifying each of them. A more thorough evaluation of the precision and recall of the reverse engineering and the correctness of the forward engineering remains a future work. An on-line demonstration of the prototype is available at our web page.

8 Related Work

There is a large body of related work; however, for space reasons, we can only highlight a few 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 [7]. They advocate that black-box frameworks are particularly well-suited for use with a DSML on top. However, as we discussed in Section 4, 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 FSMLs with round-trip engineering support.

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 [8]) that allow code querying for typical dependency structures such as call graphs and include dependencies. In contrast to these tools, our approach uses whatever specialized analyses are needed for detecting a domain-concept instance. For example, in order to recognize the *requires adapter* interaction, a set of exact types of objects returned by a method needs to be computed.

The other subcategory groups works on detecting design patterns in code (e.g., [9]). 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.

Framework instantiation. Most approaches in this category only support forward mapping to code. 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.

Many approaches have been proposed to assist the framework-instantiation process through active documentation [10, 11, 12, 13], which specifies and interactively guides the developer through available hotspots, instantiation tasks and possible implementation choices. Attempts for automating the framework instantiation such as [10] 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 [14] 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 [15], 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.

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.

9 Discussion and Future Work

The prototype implementation of the WPI FSML provided us with many insights regarding the usefulness of the presented approach to modeling and round-trip engineering.

Most of the features of framework-provided concepts in WPI correspond to small implementation steps such as implementing an interface or invoking a service. However, features corresponding to higher-level requirements can also be represented and mapped to implementation features using constraints.

The reverse mappings of FSMLS are restricted by the available static code analysis techniques. Our agile round-trip engineering approach requires the design to be retrivable 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.

WPI FSML currently does not use flow analysis to properly implement the `CALLS` function, which should check whether there exists a call to the given method within the control flow of an instance of a class in question. Also, constant propagation and data flow analysis would improve the precision in some other cases. Currently, we are in the process of designing a FSML for a part of Eclipse's Graphical Modeling Framework (GMF). Reverse engineering of GMF's completion code requires more powerful static analysis techniques than the ones used in WPI, such as techniques typically used in partial evaluation and program slicing. 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 forward mappings are not required to produce fully functional code. A 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. Furthermore, the forward mappings need a better infrastructure in terms of automatic scheduling of the execution of individual mappings and a more declarative way of specifying the mappings, such as offered by scripting languages for refactoring [17]. In general, forward mappings designed to update the code, which are code transformations, are usually harder to devise than reverse mappings.

FSMLs can potentially be used for automatic or semi-automatic code migration as described in Section 5. Although we do not have any practical experience with this aspect yet, we think that the specialized forward mappings can be defined for different versions of the API, or even for different frameworks. Currently, we are investigating the possibility of using FSMLs for the migration of code from the Struts framework to the Java Server Pages 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. 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.

10 Conclusion

In this paper, we propose 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. Compared to round-trip engineering support in the context of a general purpose modeling and programming languages such as UML and Java, FSMLs can enable round-trip over non-trivial mappings. This

more powerful round-trip engineering is possible because the framework API allows capturing the design structures in the application code more explicitly. Furthermore, the ability to freely modify application code manually gives the developer more customization flexibility than the alternative approach of strictly separating generated code from customizations.

Acknowledgements. We would like to thank Bran Selic, Todd Veldhuizen, and the anonymous reviewers for valuable comments on previous drafts. This work is partially supported by IBM Centers For Advanced Studies, Ottawa.

References

1. Eclipse Foundation: Eclipse. `http://www.eclipse.org/` (2006)
2. Antkiewicz, M., Czarnecki, K.: Eclipse workbench part interaction FSML. Technical Report 2006-09, ECE, University of Waterloo (2006) `http://gp.uwaterloo.ca`.
3. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: a progress report. In: International Workshop on Software Factories. (2005)
4. DSM Forum: Workshop on domain-specific modeling (2001-2006) `http://www.dsmforum.org/DSMworkshops.html`.
5. Zook, D., Huang, S.S., Smaragdakis, Y.: Generating AspectJ programs with Meta-AspectJ. In: GPCE'04. Volume 3286 of LNCS., Springer (2004) 1 – 18
6. Tip, F., Fuhrer, R., Dolby, J., Kieżun, A.: Refactoring techniques for migrating applications to generic Java container classes. IBM Research Report RC 23238, IBM T.J. Watson Research Center (2004)
7. Roberts, D., Johnson, R.: Evolving frameworks: A pattern language for developing object-oriented frameworks. In: PLoP'96, University of Illinois, Addison-Wesley (1996)
8. De Volder, K.: JQuery: A generic code browser with a declarative configuration language. In: PADL'06. Volume 3819 of LNCS., Springer (2006) 88–102
9. Shi, N., Olsson, R.A.: Reverse engineering of design patterns from Java source code. In: ASE 2006. (2006)
10. Braga, R.T.V., Masiero, P.C.: Building a wizard for framework instantiation based on a pattern language. In: OOIS'03. Volume 2817 of LNCS., Springer (2003) 95–106
11. Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A., Viljamaa, J.: Generating application development environments for Java frameworks. In: GCSE 2001. Volume 2186 of LNCS. (2001) 163–176
12. Ortigosa, A., Campo, M.: Smartbooks: A step beyond active-cookbooks to aid in framework instantiation. In: TOOLS'99, IEEE Computer Society (1999) 131
13. Tourwé, T., Mens, T.: Automated support for framework-based software evolution. In: ICSM'03), IEEE Computer Society Press (2003) 148–157
14. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Transactions on Software Engineering (2004)
15. Hou, D., Hoover, H.J.: Using SCL to specify and check design intent in source code. IEEE Transactions on Software Engineering **32**(6) (2006) 404–423
16. Sendall, S., Küster, J.: Taming model round-trip engineering. In: Workshop on Best Practices for Model-Driven Software Development. (2004)
17. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a scripting language for refactoring. In: ICSE'06. (2006)