Towards a Generic Infrastructure for Framework-Specific Integrated Development Environment Extensions

Herman Lee, Michal Antkiewicz, Krzysztof Czarnecki

► To cite this version:

Herman Lee, Michal Antkiewicz, Krzysztof Czarnecki. Towards a Generic Infrastructure for Framework-Specific Integrated Development Environment Extensions. Domain-Specific Program Development, 2008, Nashville, United States. pp.6. ⟨hal-00350266⟩

HAL Id: hal-00350266
https://hal.science/hal-00350266v1

Submitted on 6 Jan 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Towards a Generic Infrastructure for Framework-Specific Integrated Development Environment Extensions

Herman Hon Man Lee, Michał Antkiewicz, Krzysztof Czarnecki

Generative Software Development Lab
University of Waterloo
Waterloo, Ontario, Canada
{hlee, mantkiew, kczarnec}@gsd.uwaterloo.ca

Abstract

Object-oriented frameworks are often difficult to use. Framework-specific extensions to integrated development environments (IDEs) aim to mitigate the difficulty by offering tools that leverage the knowledge about framework's application programming interfaces (APIs). These tools commonly offer support for code visualization, automatic and interactive code generation, and code validation. Current practices, however, require such extensions to be custom-built manually for each supported framework. In this paper, we propose an approach to building framework-specific IDE extensions based on framework-specific modeling languages (FSMLs). We show how the definitions of different FSMLs can be interpreted in these extensions to provide advanced tool support for different framework APIs that the FSMLs are designed for.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments/Construction Tools—Graphical environments, Integrated environments, Interactive environments, Programmer workbench; D.2.3 [*Software Engineering*]: Coding Tools and Techniques—Object-oriented programming, Program editors; D.2.13 [*Software Engineering*]: Reusable Software—Reuse models

General Terms Languages

Keywords framework instantiation, framework-specific models, IDE, domain-specific IDEs

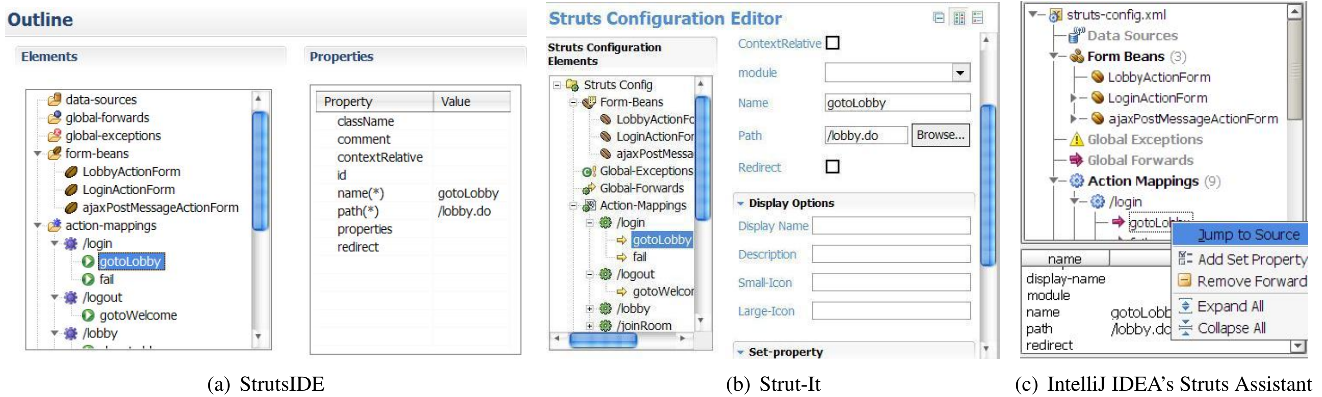
1. Introduction

Object-oriented frameworks are widely used as bases for building applications in many domains. Frameworks implement reusable designs and provide means for implementing instances of domain abstractions, which we call *framework-provided concepts* or *concepts* for short. Applications built on top of these frameworks instantiate concepts through a variety of mechanisms such as extending framework classes, implementing framework interfaces, and making framework API method calls (Antkiewicz et al. 2008). However, writing application code that implements instances of the framework-provided concepts is often challenging, since the concepts are often not well documented and the code may be cross-cutting (Hou et al. 2005).

Specialized extensions to integrated development environments (IDEs) are often created to ease application development with object-oriented frameworks by offering tools that leverage the knowledge about the framework application programming interface (API). To name a few, Spring IDE was created to support the Spring Framework, StrutsIDE and Struts-It for the Apache Struts framework, FacesIDE for the Java Server Faces (JSF) framework, and Hibernate Tools for the Hibernate framework. All of these example extensions were built as plug-ins to the Eclipse IDE. Similar extensions for these frameworks were also added to the IntelliJ IDEA IDE by the IDE's vendor, JetBrains.

However, current practices require these framework-specific IDE extensions to be custom-built manually for each framework. In this paper, we describe an approach to building a generic infrastructure for framework-specific IDE extensions for any framework formalized as a framework-specific modeling language (FSML) (Antkiewicz 2008) designed for that framework. With the generic infrastructure, the only work required to provide tool support for a new framework is to create an FSML for that framework. In previous works, we have shown that automatic reverse, forward, and round-trip engineering are possible by interpreting the declarative specification of an FSML (Antkiewicz et al. 2008; Antkiewicz 2008). In this paper, we propose using the same definitions of FSMLs to provide interactive tool support typically found in framework-specific IDE extensions, such as code visualization, automatic and interactive code generation, and code validation. We also discuss the potential to enhance these tools beyond the features found in existing framework-specific IDE extensions.

The remainder of this paper is organized as follows: Section 2 surveys some of the existing framework-specific IDE extensions; Section 3 gives an overview of FSMLs; Section 4 presents our approach; Section 5 discusses open questions



(a) StrutsIDE (b) Strut-It (c) IntelliJ IDEA's Struts Assistant

Figure 1. Comparison of Graphical Editors for Struts Config

about the approach and describes future work; Section 6 presents the related work; and finally, Section 7 concludes the paper.

2. Overview of Existing Framework-Specific IDE Extensions

We begin by examining the common features found in existing framework-specific IDE extensions. For the purpose of this paper, we only did a formal comparison between three IDE extensions for Apache Struts: Struts-It, StrutsIDE, and IntelliJ Struts Assistant. Struts is a popular framework for developing web applications. A Struts application consists of a Struts Config XML file and Java implementations of *actions*, *forwards*, and *forms*. Forms accept inputs from the user; actions process the inputs and return forwards that redirect to forms, actions, or web pages. Actions, forwards, and forms also needs to be declared in the configuration file, which creates a consistency problem, since XML declarations often must match class names or other attributes of the Java source code. Information from the Struts Config file can be used, for example, for visualization of the web page flow of an application.

Table 1 compares the features found in three framework-specific IDE extensions for Struts. A quick review of Hibernate tools and Spring IDE revealed that both extensions have many common features as the extensions for Struts. In general, we can categorize the purpose of the features into four broad categories: code visualization (1), automatic code generation (2), interactive code generation (3), and code validation (4).

Code visualization. Current IDE extensions often provide visualization of instances of framework-provided concepts found in the application code through graphical editors, views, and diagrams. Figure 1 shows three implementations of the graphical editor for the Struts Config XML file provided by the three framework-specific IDE extensions from Table 1. Each editor offers document outline and forms for editing the concepts' attribute values. Note the *Jump to Source* functionality provided by the IntelliJ IDEA Struts Assistant.

Extension	Category	Struts-It	StrutsIDE	IntelliJ Struts Assistant
Graphical Editor of XML Config file	1	Yes	Yes	Yes
Code Completion for XML Config file	3	No	Yes	Yes
Visualization of page flows	1	No	Yes	Yes
Primitive Wizards for creating framework concepts	2	Yes	Yes	No
Quick Fixes in XML Config file	4	No	Yes	Yes

Table 1. Feature Comparison of three Framework-Specific IDE Extensions for Struts

Automatic code generation. Current IDE extensions offer code generation for concept instances specified using wizards or graphical editors. Wizards are normally used to create a new project or generate new classes. The generated code is usually intended for manual customization.

Interactive code generation. Specialized code editors implemented by IDE extensions often offer *code completion*, which proposes and inserts available constructs at the given cursor position. For example, the Struts Config editor offers code completion based on the document schema when editing the XML document directly. None of the extensions that we studied have framework-specific support for Java files in this area.

Code validation. Specialized views and editors provided by the IDE extensions often also provide API constraint checking and error highlighting. Some extensions additionally offer automated solutions to common errors called *quick fixes*. However, similar to the code completion, the three IDE extensions in Table 1 only provide quick fixes support in Struts Config based on the document schema.

3. Framework-Specific Modeling Languages

This section describes *framework-specific modeling languages* (FSMLs) (Antkiewicz 2008), which are interpreted

by our proposed infrastructure for framework-specific IDE extensions. Each FSML is designed for a particular framework and formalizes that framework's API concepts and constraints. For example, Eclipse Workbench Part Interaction (WPI) FSML describes concepts such as *views* and *editors* in the Eclipse Workbench API, whereas Struts FSML describes concepts such as *actions*, *forwards*, and *forms* in the Struts API. FSMLs are used to express *framework-specific models*. A framework-specific model models how an application uses a framework's API by describing instances of concepts from the framework's FSML that are implemented by the application.

In FSMLs, concepts are formalized as a cardinality-based feature model (Czarnecki et al. 2004), in which a concept is decomposed into a hierarchy of features. Each feature is a property of a concept and has a cardinality constraint attached to it. The cardinality constraint specifies the number of instances the feature should exist in a concept instance. A feature can also have a *mapping definition* attached, which specifies how the feature can be implemented or located in the code. Semantically, a feature model describes a set of legal configuration of the features.

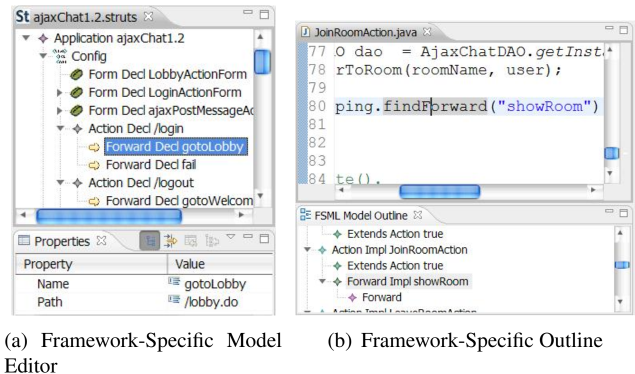
To illustrate, Figure 2 shows a fragment of the feature model of the Struts FSML. The hierarchical nature of the feature model is shown using indentation (subfeatures are further right). Feature cardinality constraints are specified in square brackets and mapping definitions are specified in angle brackets after the name of a feature. For example, the feature `ActionImpl` (line 17) corresponds to a Java class and the feature `forwards` (line 20) corresponds to the method calls to the method `findForward` in the control flow of that class. Any number of instances of both features is possible in a legal configuration as indicated by the cardinality `[0..*]`. An example of a feature that corresponds to an XML element is `ForwardDecl` (line 3). Its mapping definition specifies that each instance of the feature in a feature configuration corresponds to an XML element on the path `global-forwards/forward`, in the XML document that an instance of its parent `StrutsConfig` corresponds to.

4. An Infrastructure for Framework-Specific IDE Extensions

In previous works, we have shown that automatic reverse, forward, and round-trip engineering of framework-based applications (Antkiewicz et al. 2008; Antkiewicz 2008) is possible by interpreting declarative definitions of FSMLs through framework-specific extensions implemented on top of a *generic FSML infrastructure*. In this section, we describe the existing framework-specific extensions and we propose new ones. The new extensions also require certain new services from the infrastructure. To date, we have prototyped some of the new extensions and implemented the required services.

```
1  StrutsApplication <project>
2    [1..1] StrutsConfig <xmlDocument: '/WEB−INF/struts−config.xml'> <
         xmlElement name: 'struts−config'>
3      [0..*] ForwardDecl <xmlElements: 'global−forwards/forward'>
4        [1..1] name (String) <xmlAttribute>
5        [0..1] path (String) <xmlAttribute>
6        [1..1] target (ActionDecl) <where attribute: path equalsTo: ../path>
7      [0..*] ActionDecl <xmlElements: 'action−mappings/action'>
8        [1..1] path (String) <xmlAttribute>
9        [0..1] name (String) <xmlAttribute>
10       [0..1] type (String) <xmlAttribute>
11       [1..1] actionImpl (ActionImpl) <where attribute: qualifiedName equalsTo: ../
             type>
12       [0..*] forwards <xmlElements: 'forward'>
13         [1..1] name (String) <xmlAttribute>
14         [0..1] path (String) <xmlAttribute>
15         [1..1] target (ActionDecl) <where attribute: path equalsTo: ../path>
16       [0..1] input (String) <xmlAttribute>
17   [0..*] ActionImpl <class>
18    ![1..1] name (String) <className>
19    ![1..1] extendsAction <assignableTo: 'Action'>
20     [0..*] forwards <callsTo: 'ActionForward ActionMapping.findForward(String)'>
21       [1..1] name (String) <valueOfArg: 1>
22       [1..1] forward
23        !<1−2>
24           [0..1] localForward (ForwardDecl) <where attribute: name equalsTo: ../../
                 name> <and attribute: ../type equalsTo: ../../../qualifiedName>
25           [0..1] globalForward (ForwardDecl) <where attribute: name equalsTo:
                 ../../name> <andParentIs instanceOf: 'StrutsConfig'>
26     [0..*] inputForwards <callsTo: 'ActionForward ActionMapping.getInputForward
           ()'>
27       [1..1] name (String) <valueOf attribute: input class: 'ActionDecl'> <where
             attribute: type equalsTo: ../../qualifiedName>
```

Figure 2. Excerpt of the definition of the Struts FSML



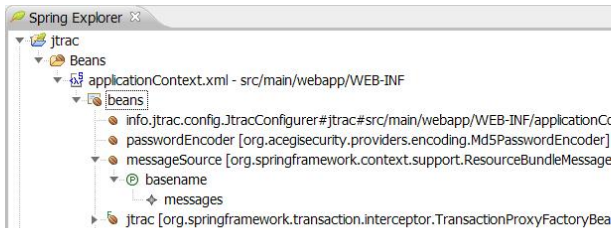
(a) Framework-Specific Model Editor (b) Framework-Specific Outline

Figure 3. Framework-Specific Code Visualization

Code visualization. Currently, reverse engineering is suppoted by all of Applet, Struts, WPI, and EJB FSMLs. Reverse engineering extracts framework-specific models from application code with high precision and recall (Antkiewicz et al. 2008). Additionally, reverse engineering establishes traceability links, which enable *forward navigation* from a feature instance to its implementation, similar to the *Jump to Source* capability as seen in Figure 1(c). Figure 3(a) shows our framework-specific model editor. Similar to the other framework-specific IDE extensions' graphical editors, our editor also displays an outline of the Struts Config file. Unlike the other editors, our editor additionally displays information about the Java implementations of the concept instances, which allows the developers to maintain the con-

sistency between the XML declarations and Java implementations of the concept instances. As another example, Figure 4 shows an editor for Spring FSML in comparison with the Spring Explorer (previously known as beans view) in Spring IDE.

In addition to framework-specific editors, we prototyped a new extension to support *framework-specific outline view*, as presented in Figure 3(b). Similar to the standard Java outline view that displays members of the currently edited class, the framework-specific outline view displays features of the currently edited concept instance. The framework-specific outline view uses *reverse navigation* to highlight features in the view based on the cursor position in the code editor. Reverse navigation is accomplished by interpreting the traceability links established during reverse engineering in the reverse direction.



(a) Spring Explorer (previously Beans View) in Spring IDE

(b) Spring Framework-Specific Model Editor

Figure 4. Spring Framework-Specific Model Editor vs. Spring Explorer

Automatic Code Generation. Two FSMLs, WPI and Applet, have been shown previously to fully support forward- and round-trip engineering (Antkiewicz 2008). In this paper, we describe the preliminary use of the Struts FSML in forward engineering. Forward engineering creates an implementation for a given framework-specific model by executing code transformations for feature instances. Round-trip engineering, on the other hand, propagates changes between the existing code and the model by executing model updates and code transformations. Round-trip engineering, as currently implemented, supports feature addition, modification, and removal at the model side and code pattern addition at the code side. The *Model-Code Synchronization* view displays results of the comparison between the current model and the current code. For each detected difference, the user decides whether it should be propagated to the code or to the model and invokes the *Model-Code Reconcile* action, which executes model updates or code transformations as requested. Figure 5 shows example results of the comparison between the model and the code.

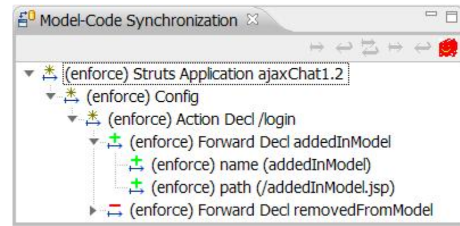


Figure 5. Model-Code Synchornization

A problem with automatic forward and round-trip engineering is that, for some features, the code can be created in many different places and the location depends on the application logic. For example, a method call to the method `findForward` (which implements an instance of the feature `forwards` (line 20)) is by default inserted at the end of the method `execute` of a Struts action class. However, such a method call is typically used inside an `if` statement inside the method's body so that different pages can be displayed under different conditions. Therefore, after the code is inserted, the developer needs to move it to the appropriate place. In such cases, interactive code generation may be a better approach.

Interactive Code Generation. Using the generic FSML infrastructure, we have prototyped two new extensions to support *framework-specific content assist* and *framework-specific keyword programming* in the Java code editor. Although we do not currently have content assist support for Struts Config XML files, we believe that the technique for framework-specific content assist in Java files can be easily extended to XML files.

Framework-specific content assist first identifies all features implemented at the given cursor position using the reverse navigability as described before. Next, by interpreting the definition of the FSML corresponding to the identified features, it proposes the creation of instances of subfeatures of the identified features as *content proposals*. After the developer chooses a proposal, code transformations are executed for the chosen features and all of their mandatory subfeatures (with the lower bound of the cardinality greater than zero). Furthermore, content assist offers subsequent proposals if a choice needs to be made by the developer. The process continues until all required features are instantiated.

For example, if the content assist is invoked when the cursor is placed in a Java class that instantiates the concept `ActionImpl` (Figure 2, line 17), the reverse navigation will first identify that concept's instance in the framework-specific model. Next, by interpreting the Struts FSML, content assist will propose the features that can be instantiated in the current concept instance, `forwards` (line 20) and `inputForwards` (line 26), as shown in Figure 6. If the developer selects the proposal `forwards`, a wizard is displayed requesting the user to enter a value for the feature in

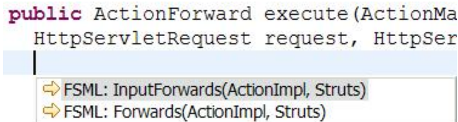


Figure 6. Framework-Specific Content Assist for Struts

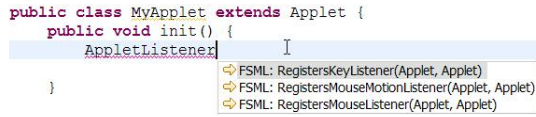


Figure 7. Framework-Specific Keyword Programming

order to satisfy the cardinality constraint of the subfeature `name` (line 21). Next, the mandatory feature `forward` (line 22) and its two subfeatures specify that for each forward name used in the code, a corresponding forward declaration must exist in the Struts Config file. Line 23 specifies that at least one and at most two such declarations must exist. The features `localForward` (line 24) and `globalForward` (line 25) represent *referential integrity* constraints and they are references that point at the corresponding local or global forward declarations in the model. Therefore, in order to satisfy the framework-specific constraint, code assist displays a wizard allowing the developer to choose a forward type: a `localForward` or a `globalForward`. After the developer selects the forward type, the referential integrity constraint is evaluated. For example, if the developer selects `globalForward`, a global forward declaration with the same name as the name specified previously will be matched. If such a declaration does not exist, code assist will automatically create a new instance of the feature `ForwardDecl` (line 3) and set the value of its subfeature `name` (line 4) to the value of the feature `name` (line 21). For example, assuming that the user entered the value `"success"` as the forward name, a method call `mapping.findForward("success")` is generated at the cursor position and an XML declaration

```
<global-forwards>
  <forward name="success"></forward>
</global-forwards>
```

is generated in Struts-Config.xml.

The support for framework-specific keyword programming filters possible content proposals to those that contain the user-entered keywords in the feature's name and it is an improvement to the traditional prefix-based auto completion. For example, Figure 7 shows framework-specific content proposals for the keyword `Listener` for a different FSML, the Java Applet FSML. Based on the cursor position and the entered keyword, content assist identified three features, representing the different kinds of listeners that can be instantiated.

We have also added support for interactive code generation into the model editor. The model editor grays out concept instances that are not yet implemented in the code and suggests code content and code location for these concept instances based on previous instances of the same concept. Beside the unpredictability of a concept's code location as described previously, code content also varies for framework concepts because of reasons such as method overloading and object inheritances.

Code Validation. In the generic FSML infrastructure, we implemented model validation that evaluates cardinality and other constraints specified in an FSML. Since the model represents the usage of the API by an application, constraint violation in the model indicates a constraint violation in the code. For example, since a framework-specific model of a Struts application contains features related to both XML (Figure 2, lines 2-16) and Java (lines 17-27), referential integrity constraints can be checked. We encode such constraints as mandatory features. For example, the feature `actionImpl` on line 11 corresponds to the constraint that for every action declaration, which is an instance of the feature `ActionDecl`, a corresponding action implementation must exist. Figure 8 shows an example of the result of model validation when a forward declaration is missing for an instance of the feature `forwards`. The feature `forward` (line 22) will only be present if at least an instance of the feature `localForward` or `globalForward` is present (as specified by the essential feature group on line 23).

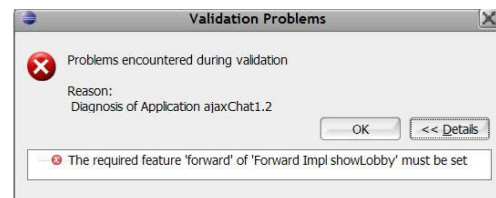


Figure 8. Framework-Specific Model Validation

Using the model validation, we can implement *API error highlighting* by continuously running reverse engineering in the background to obtain an updated model of the code and creating problem markers in the code editor for the concept instances with constraint violation.

5. Discussion and Future Work

At the core of our approach is the idea that framework-specific IDE extensions can be implemented generically and parameterized with an FSML definition. All framework-specific extensions based on FSMLs interpret the same language definition. However, they require different infrastructure services, such as reverse engineering, traceability links, or incremental code addition. Sometimes, the services need to be adapted to support new extensions. For example, existing incremental code addition for Java was extended to generate code in the middle of a method body to support content assist. The core of the generic FSML infrastructure is independent from the types of source artifacts and sup-

ports pluggable mapping interpreters that process mapping definitions. For example, adding support for Java and XML requires plugging in appropriate mapping interpreters. As a result, feature instances in a model can correspond to code patterns spanning multiple source artifacts of different types. In comparison, the existing framework-specific IDE extensions we analyzed typically are implemented manually and support only one artifact type: XML.

The main advantage of our approach is that generic extensions, once implemented, can be easily used for multiple frameworks under the condition that the frameworks' APIs are formalized as FSMLs. The main disadvantages are that the implementation of generic services and extensions is not trivial and that an FSML needs to be built for a given framework. However, since FSMLs are specified declaratively, the effort is mostly limited to modeling API concepts and constraints as feature models and specifying mapping definitions. An FSML design method is available (Antkiewicz 2008, Ch. 3).

There are several possible directions for future work.

Incremental reverse engineering. Currently, reverse engineering always processes the entire application, which is too slow for supporting highly interactive extensions such as error highlighting and content outline. Reverse engineering should be performed incrementally, only for the changed source artifacts.

Smart keyword programming. Support for more advanced keyword programming, such as the use of synonyms, and a formal evaluation of the value of framework-specific models in keyword programming as opposed to the extraction of keywords from source code (Little and Miller 2007) remains future work.

Full framework-specific IDE. Beyond the four categories of features in framework-specific IDE extensions described previously, an IDE, in the traditional sense, should support all phases of software development including compilation, debugging, testing, deployment, and version control. This area of research remains future work. However, for version control, we have implemented support for model comparison, similar to the one used in round-trip engineering.

6. Related Work

The creation of language programming tools and IDEs based on programming language description has long been the subject of research, dating back to the 80's (Reps and Teitelbaum 1984). Of these, recent efforts include Eclipse IDE Meta-tooling Platform (IMP) (Charles et al. 2007), TOPCASED (Farail et al. 2006), and Textual Generic Editor (TGE) (ATLAS 2008). These existing works focus on providing direct editing support for programs written in a given programming or modeling language, whereas our work exposes two abstraction levels for editing at the same time: the code and its abstractions as models.

Another class of programming environments is one that integrate hand-written code with code generated from models targeting frameworks. The integration can happen through different mechanisms, such as protected blocks, subclassing, calls, aspects, open-classes, or partial-classes. For example, Cook described the use of protected blocks in his keynote about domain-specific IDEs (Cook 2008). In our work, we do not explicitly separate between the hand-written and the generated code: the developers can navigate to the fragments of code implementing feature instances from the model through traceability links and they can freely move fragments of code as long as that does not violate API constraints.

7. Conclusion

In this paper, we presented an approach to creating framework-specific IDE extensions based on a generic FSML infrastructure. Instead of rewriting all the extensions for each new framework, we propose a set of generic framework-specific IDE extensions that interpret the definition of any FSML. We have examined the common features in current custom-built framework-specific IDE extensions and demonstrated the feasibility of the approach by describing the approach in terms of these features on a popular framework.

References

Michal Antkiewicz. *Framework-Specific Modeling Languages*. PhD thesis, University of Waterloo, 2008.

Michal Antkiewicz, Thiago Tonelli Bartolomei, and Krzysztof Czarnecki. Fast extraction of high-quality framework-specific models from application code. *Journal of Automated Software Engineering*, 2008. Accepted and recommended for publication.

ATLAS. ATLAS Megamodel Management (AM3) home page, 2008. `http://www.eclipse.org/gmt/am3/`.

Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton Jr. IMP: A meta-tooling platform for creating language-specific IDEs in Eclipse. In *ASE*, pages 485–488, 2007.

Steve Cook. The domain-specific IDE, 2008. Keynote at Code Generation.

Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration using feature models. In *SPLC*, pages 266–283, 2004.

Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project: a toolkit in open source for critical aeronautic systems design. In *ERTS*, 2006.

Daqing Hou, Kenny Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *IWPC*, pages 87–96, 2005.

Greg Little and Robert C. Miller. Keyword programming in Java. In *ASE*, pages 84–93, 2007.

Thomas Reps and Tim Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, 1984.