SPin – A Fujaba Plugin for Architecture Stratification

Felix Klar, Thomas Kühne, Martin Girschick
Fachgebiet Metamodellierung
Fachbereich Informatik
Technische Universität Darmstadt, Germany
felix@klarentwickelt.de, {kuehne, girschick}@informatik.tu-darmstadt.de

ABSTRACT

SPin is a plugin for Fujaba that provides basic support for architecture stratification. It enables Fujaba models to be annotated with refinement directives which may then automatically be executed by the plugin. The corresponding refinement transformations may be defined with a combination of *story driven modeling* and Java coding. These transformations affect both model and associated code, and may be defined interactively, i.e., do not require Fujaba to be shutdown and started up again. In this paper we describe the purpose of the plugin, how to use it, its realization, and some supporting functionality.

Keywords

stratification, model driven development, model transformation, Fujaba plugin

1. INTRODUCTION

Today's software systems have reached such a level of complexity that a single view, e.g., architectural description, is not sufficient anymore. If the system is described from a bird's eye view, using a very high level architecture description, many important details regarding performance, extensibility, etc. remain hidden. If, however, one chooses a view with a much lower level of abstraction, allowing the above properties to be evaluated, the complexity will become unwieldy; it becomes difficult to see the forest for the trees.

Architecture stratification is an approach that connects multiple views on a single system with refinement translations, so that each view describes the whole system on a particular level of abstraction. This way single levels do not only present an optimal mix of overview and detail for various stakeholders, but they also separate and organize a system's extension points, patterns, and concerns [1].

The Fujaba plugin SPin[1] [4] supports the automatic transformation of models into more detailed versions and thus represents basic support for architecture stratification. However, it is not restricted to this particular flavor of model driven development, but supports any development approach that requires annotation-guided model transformations.

In the following, we first describe SPin from a user's perspective (section 2) and then present an example demonstrating the utility of SPin (section 3). Subsequently we take a closer look at the inside of SPin (section 4), before we outline future work (section 6) and finally conclude (section 7).

[1]An acronym for "**S**tratification **P**lug**in**".

2. USING SPin

Figure 1 shows how SPin may be used in the context of Fujaba. SPin supports two kinds of transformations, refinement rules, yielding more concrete models and abstraction rules, yielding more abstract models both of which can be regarded as endogenous transformations [5]. In the following we will concentrate on rules defining refinement transformations only, though.

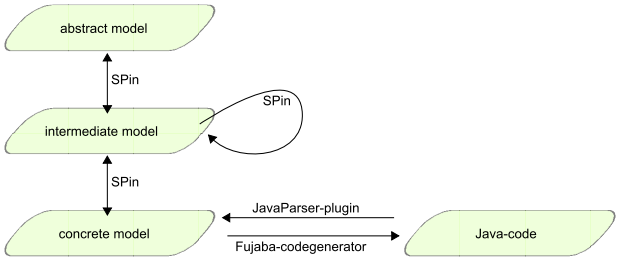

Figure 1: *SPin's role within Fujaba*

Since SPin's transformation rules may not only transform models (e.g., class diagrams) but also any associated code (e.g., method implementations) it can be used to transform a simple system description into a complex one, using multiple steps. The most complex system description can then be used to create an executable system by virtue of the Fujaba code generation engine.

The prerequisite for automatically transforming models in this top-down fashion, however, are directives, i.e., annotations in a model.

2.1 Annotating a model

Annotations specify in which way a model element should be refined in order to obtain a finer grained realization. To provide additional information for the transformation, annotations can be parameterized using basic types (e.g., a string specifying the name of a class that should be generated) or links to other model elements (e.g., specifying one or more of the already existing elements to be used as observers for a subject). We therefore chose a notation similar to UML collaborations in UML class diagrams. Both notations share

the need to specify which elements form a structure—such as which other element(s) should be involved in the transformation process or what other element to use as a parameter to the transformation—and the need to describe the role of the referenced element.

SPin provides a dedicated *annotation editor* to support the introduction and parameterization of annotations. Figure 2 shows a screenshot of the annotation editor displaying the parameters of an "Observer" annotation (see the example in section 3). The annotation is parameterized with two links ("state" and "concreteObserver") and one basic type ("observerClassName").

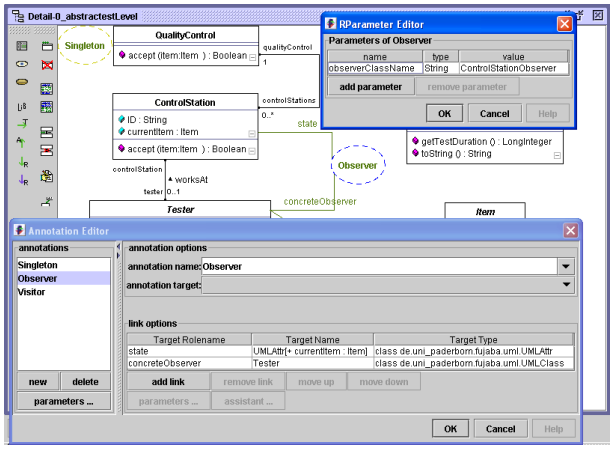

Figure 2: *SPin's annotation editor*

Once a model is completely annotated, the user may use the context menu of an annotation to initiate the corresponding transformation process. Currently, SPin supports manual transformation initiation only, i.e., it is not possible yet to start a recursive unfold process which stops when no more annotations exists.

2.2 Creating a rule

Unfolding an annotation triggers the corresponding refinement rule. Such rules are completely user defined. SPin only provides the machinery for creating, using, and executing rules. The rules themselves are part of a rule library, which can be extended dynamically.

Let us step through the creation of a rule implementing the GoF [3] pattern "singleton". First, we create a UML class diagram and then add a new rule class to it by invoking the "create rule..."-action from the class diagram's context-menu. This causes SPin's "new rule" dialog to open and we use it to specify the rule type and a rule name. In our example we choose a refinement rule with the name "Singleton". The rule's name describes its intent, but will also later be used to annotate a model.

Figures 3 and 4 show a part of what SPin automatically generates after the "new rule" dialog has been closed.

Figure 3 shows the addition of a new refinement class (*RRSingleton*). Among other features it defines an 'apply' method

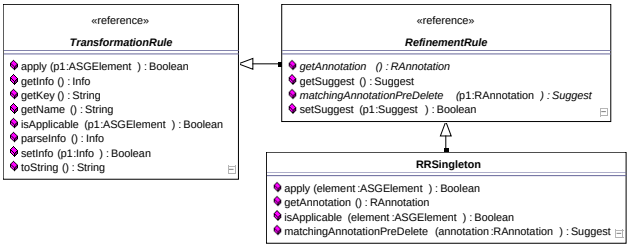

Figure 3: *A new refinement rule*

that contains the actions to be performed when the rule is triggered.

Figure 4 shows the automatically generated body for 'apply'. As one can see Fujaba's *Story Driven Modeling* (SDM) [6] is used to implement the 'apply'-method. This results in a semi-graphical implementation which is more self-explanatory and easier to create and to maintain than handwritten Java-Code. The first check makes sure that the model element to be transformed indeed has the correct annotation ("Singleton" in this case). If yes, a reference to a *UMLFactory* is created so that new UML elements may be created in the core transformation part. Finally the annotation is destroyed, i.e., removed from the diagram, since at this point in time the annotation has served its purpose to create a more concrete realization of its source structure. The rule designer may still change any part of this, but this is how most refinement rules look like, if one leaves out the core transformation part and any further checks as to whether the rule is really applicable.

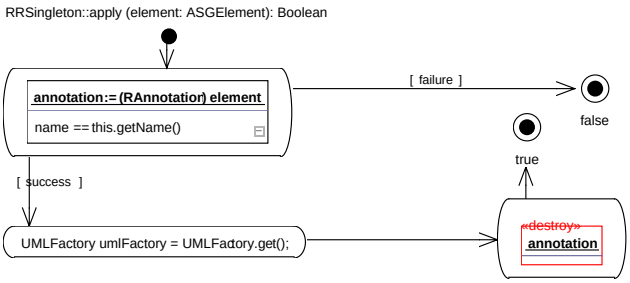

Figure 4: *Generated 'apply'-method*

In our example the rule's precondition has to be enhanced to check whether the annotation is bound to a UML class (see figure 5). If this is the case, the class will be transformed into a singleton class. The transformation code adds an attribute holding the singleton-instance, a private constructor and a get-method that returns the singleton-instance. Once finished, the rule can be exported to the rule library, so that it may be used to transform a UML class into a "singleton".

3. CASE STUDY

We now demonstrate the utility of SPin by considering an example system that simulates a quality control assembly line. In this example we use the three design-patterns "Sin-

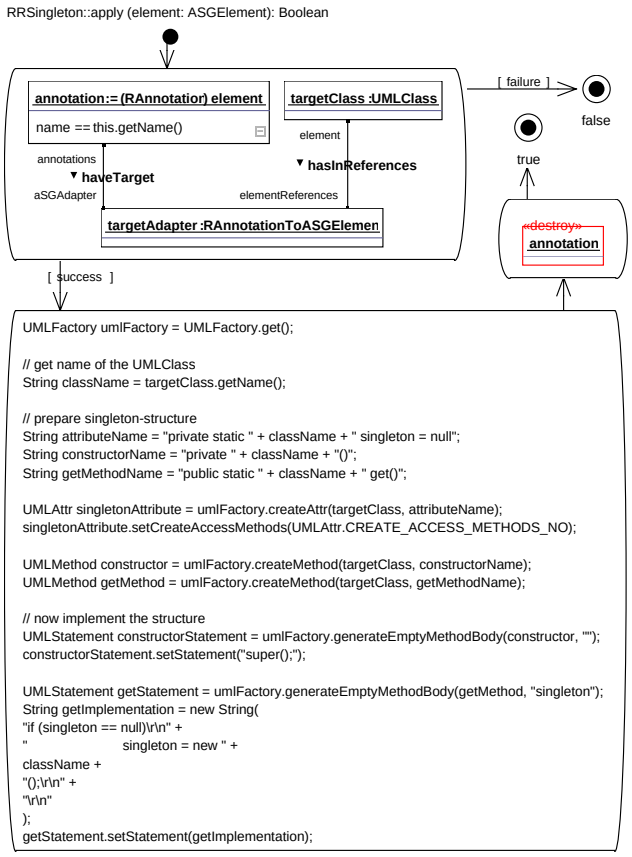

Figure 5: *Fully implemented 'apply'-method*

gleton", "Observer", and "Visitor" [3] in order to obtain a high-level view on the system's structure (see Figure 6).

3.1 System description

The system has a main quality control unit (*QualityControl*) that should be realized as a singleton instance (hence the corresponding annotation). Quality control is realized as an assembly line that consists of a variable number of control stations (class *ControlStation*). These stations check items (abstract class *Item*) passed to them by the assembly line. Our example features only one concrete item type (class *Screw*).

Control stations feature a tester which checks the current item. For each observed item a test report (class *ItemTest*) is created. Testers come in two kinds: manual testers, like humans, that are able to perform very complex tests and automatic testers, like industry-robots that are specialized for testing a single property of an item. Here, a robot (class *Scale*) is used, that checks an item's weight.

Let's have a closer look at the annotations "Observer" and "Visitor". Annotation "Observer" is parameterized with two links. Link "state" binds attribute *currentItem:Item* of class *ControlStation* to the annotation to define which state should be observed. Link "concreteObserver" binds class *Tester* which should observe the specified state. An additional parameter "observerClassName" of base-type *String* with value "ControlStationObserver" has been added to the annotation.[2] This parameter specifies the name of the generated observer interface. Annotation "Visitor" has two links as well: "element" specifies which class should be the element of the visitor pattern and "concreteVisitor" specifies which class should visit the element. For a more detailed description of the rules corresponding to "Observer" and "Visitor" please see [4].

3.2 Refining the system

We now refine this system, by unfolding annotations step by step, until we reach the most detailed system description.

After unfolding the "Singleton" annotation we may then unfold "Observer". Note that we have to attach/detach concrete observers to/from their subjects (in this case: class *ControlStation*), so that observers will be notified of state changes. As we want these code-fragments to be placed in method 'setTester(Tester)' in class *ControlStation*, we need to implement this method manually, so it will attach/detach tester instances accordingly. We also want to specify what should be done when an observer is updated by its subject. This is accomplished by adding a call to method 'createItemTest(Item)' within the method 'update(ControlStation)' in class *Tester*.

Finally, we resolve "Visitor". Of course we need to provide the code for the visit-methods in each concrete visitor. This is currently done manually after the transformation step, but alternatively one may also provide the method bodies as parameter values to a correspondingly defined "Visitor" rule.

All other pattern-related method bodies will be automatically generated of by the respective rules. The resulting system structure is visualized in figure 7.

3.3 Completing the system

After the system has been refined to its most detailed version, we now have to complete the implementation by filling in the missing method bodies.

We only need to deal with two methods in our example: (a) 'process(Item)' in class *ControlStation*, which has to notify its observers, if an item receives the focus of a control station and (b) 'createItemTest(Item)' in class *Tester*, which has to initialize the visit-process by invoking method 'accept(ItemVisitor)' on the passed *Item*-instance.

Now that the most detailed model has been completed, Fujaba's codegenerator can be used to generate executable code from it.

4. INSIDE SPin

The following section describes some of the internal aspects of SPin, in particular how SPin provides support for the creation of new transformation rules.

[2]However, this parameter is not visible in figure 6. It may only be seen or changed through the annotation editor (see figure 2).

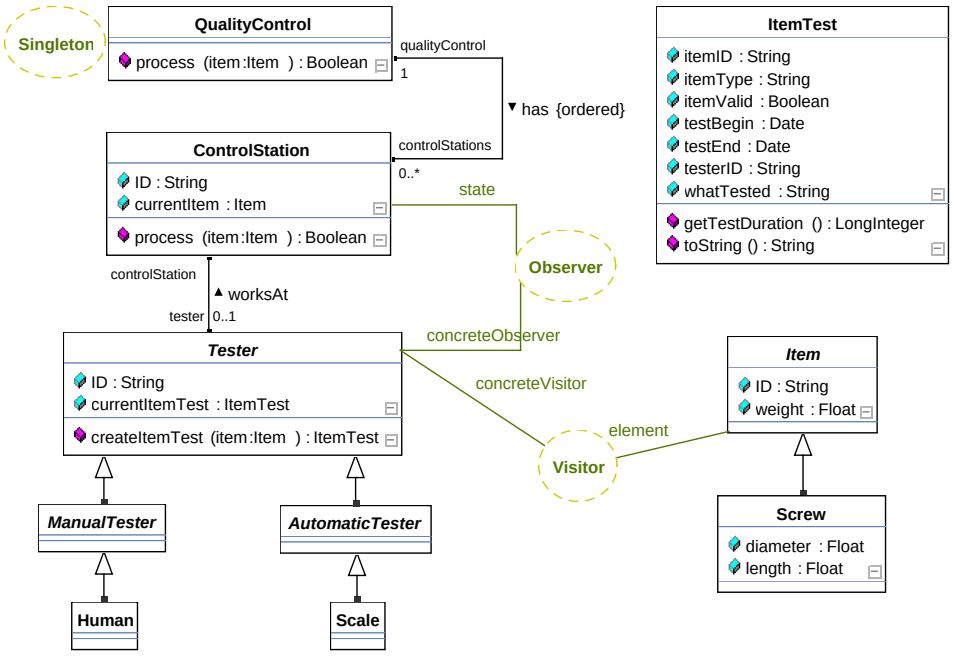

Figure 6: *Most abstract system version of our example*

4.1 Dynamic Rule Additions

Once a rule is created and added to the library, it can immediately be used in models without requiring manual shutdown, recompilation, etc. This is achieved in the following way: After creating or modifying a rule in the form of a UML class, the rule developer asks SPin to export the rule. SPin thereupon first employs Fujaba's code generator to generate the corresponding Java-code and then the Java compiler to create the respective Java bytecode. Subsequently, all new generated files are moved into the rule library directory and finally the library is reloaded.

4.2 Modifying Activity Diagrams

Often rules need to transform behavior in the form of method bodies. In Fujaba, method bodies are specified with storydiagrams, a combination of activity and collaboration diagrams [6].

In order to support the analysis and modification of such storydiagrams SPin introduces a standard mechanism (available through class *UMLActivityDiagramModifier*) that enables rule developers to visit each element of a storydiagram. It simply iterates over each element and invokes element-specific code on a visitor (class *UMLActivityDiagramVisitor*) that performs operations on those elements. This mechanism can easily be adapted by rule developers through implementing their own visitor classes and passing them to the diagram modifier. For instance, rule developers may use it to modify text within method bodies.

4.3 Element Creation and Initialization

A common activity of rules is the creation of new model elements. SPin provides factories (see *UMLFactory* in Figure 4), whose task is to reduce the effort for creating model elements to a minimum. In the ideal case just one function call is required to create and initialize an element and then add it to a diagram. SPin provides two factories: One supporting the creation of transformation rules and the other to create elements of the annotation metamodel.

One aspect of these factories—creation of elements—will be obsolete with Fujaba 5, when the creation of UML elements through factories will be natively supported. Yet, the other aspect—initialization of elements—will still be useful, since Fujaba 5 will still require manual initialization, requiring potentially many function calls. Hence, SPin's UML factory may even proof to be useful in Fujaba 5.

4.4 Metamodel Synchronization

One way to specify transformation rules is to use the SDM approach available in Fujaba [2]. Figure 4 shows how a story diagram is used to check the applicability of a rule. The transformation process itself may also be specified using SDM (instead of using Java).

However, in order to be able to employ SDM in this manner, Fujaba requires a UML classdiagram containing the appropriate metamodel to be present in the current Fujaba-project. The challenge is hence how to get this metamodel into a Fujaba-project, since one clearly does not want to do it manually. This would not only be a very time-comsuming but also a very error-prone task.

One solution to automate this process is the *JavaParser*-plugin (see also Figure 1). It can be used to generate a UML classdiagram from the Java sourcecode which represents the metamodel. If the source code and the actual metamodel built into Fujaba differ, though, e.g., because of a version mismatch, incompatibilities will arise.

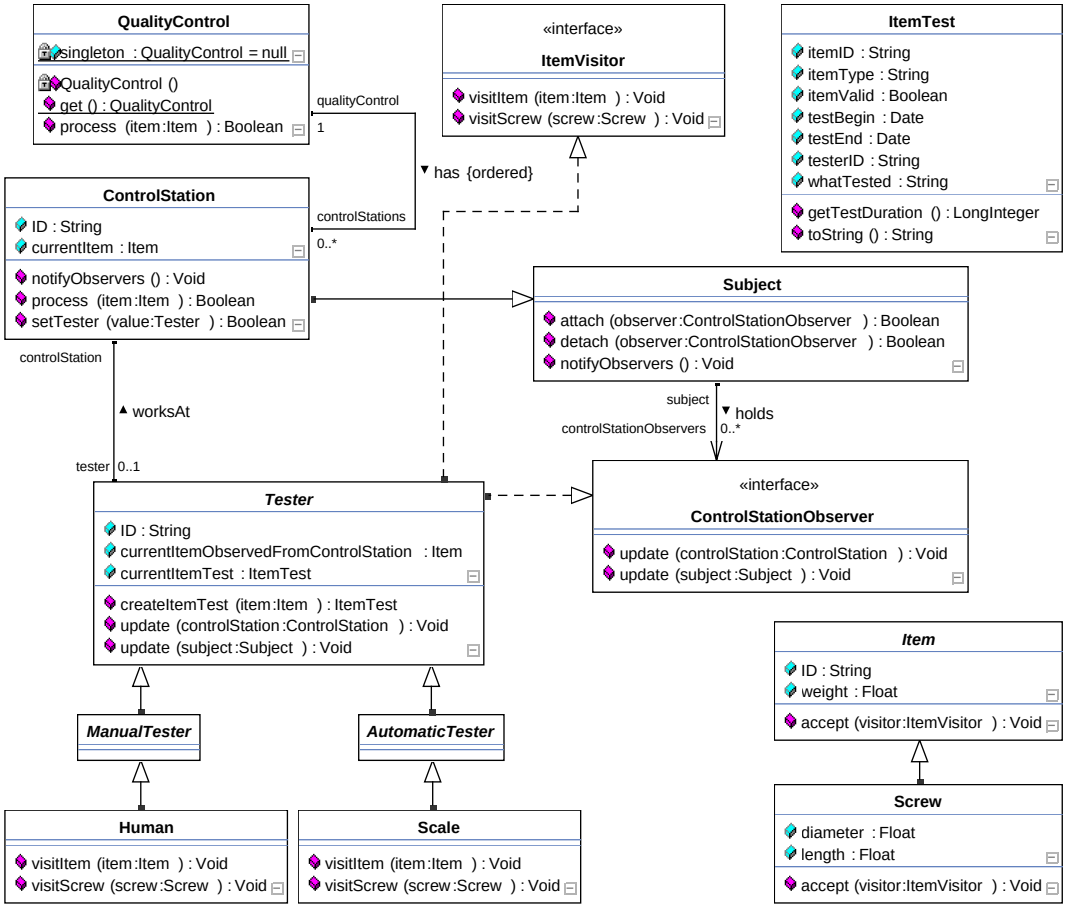

Figure 7: *Most detailed system version of our example*

SPin therefore uses a *Virtual Machine synchronizer* (VM-synchronizer) [4]. It takes the actual—and thus guaranteed to be of the correct version—Java bytecode of the respective part of Fujaba as input, extracts structural information using a Java classloader in conjunction with the Java reflection API and generates a UML model from the extracted information using a UML factory. Finally a post processing step detects associations between UML classes and adds them to the metamodel which then can subsequently be used by SDM.

5. RELATED WORK

Most MDA tools specialize on generating code from a model or migrating models in one modeling language to another, i.e., exogenous transformations [5]. Tools that support model refactorings can—according to [5]—be classified as supporting horizontal endogenous transformations. In contrast, architecture stratification realizes vertical endogenous transformations. In other words, refactorings maintain the same level of abstraction whereas architecture stratification creates different levels of abstraction expressed in the same modeling language.

Only few commerical tools, such as Together Architect[3] and ArcStyler[4] provide basic support for defining vertical endogenous transformations as well.

Together Architect provides an extendable template-based mechanism for creating patterns. This mechanism can be used to create rules that perform vertical endogenous transformations. A pattern manager is used to apply patterns to class diagram elements. In contrast to SPin, however, this way transformations are executed in a step by step fashion, whereas SPin automates the transformation of all annotations of one kind and will eventually support a fully automated application of all applicable transformations from top to bottom.

The MDA tool ArcStyler supports both model-to-model and model-to-code transformations, which are defined in so called *cartridges*[5]. A cartridge defines a source and target metamodel (or inherits the UML metamodel) and the transformations from the source to the target. UML stereotypes may be used to guide the transformation process. In addition so called *marks* are used to allow further parameterization of the model, e.g. for different target platforms. Transforma-

[3] http://www.borland.com/us/products/together/

[4] http://www.interactive-objects.com/

[5] http://www.interactive-objects.com/support/doc/doc/Carat_Guide.pdf

tions are described using the script language JPython. This is supplemented by the concept of blueprints which are similar to *model templates*. ArcStyler follows the MDA approach where a platform independent model (PIM) is completely parameterized and then transformed to a new platform specific model (PSM). If this approach is used in a staged, incremental manner, it very much resembles the abstraction level stratification approach of SPin.

Neither Together Architect nor ArcStyler support Fujaba's *Story Driven Modeling*, which is very useful for the semi-graphical specification of transformation rules as used in SPin.

6. FUTURE WORK

The current version of SPin offers a limited set of transformation rules. Although these are user extensible, the utility of SPin would be increased if it came with a rich set of ready-to-use rules. By applying the stratification process to big and complex software systems it will be possible to extract useful rules which can then be added to SPin.

Employing stratification in its intended form with SPin is currently hindered by the fact that only manual, stepwise initiations of transformations are supported. In order to fully automate the generation of a complex system from a given simple and abstract view, it is necessary to automate the process of unfolding annotations. This includes the specification of the order in which annotations are to be unfolded. However, this ordering is neither difficult to work out, nor should it be part of an automated process. Annotations exhibit natural dependencies and lend themselves to generate levels of system concerns [1]. It is therefore the task of the system architect to select which of the annotations are addressed at each specific abstraction level. As a result, future versions of SPin should provide a configuration system that allows users to specify and store their annotation processing orders.

SPin will significantly benefit from the new features of Fujaba 5. For instance, the then available support for multiple projects will enable developers to create rules in one project and immediately apply them in another. Moreover, users then might be able to easily navigate back and forth between different levels of abstraction.

Moving up in the hierarchy of abstraction levels is already possible as long as the more detailed versions were generated by SPin. This is accomplished by using Fujaba's built in "undo" mechanism.

In the current release of SPin the transformation rules are expressed using arbitrary Java code. This implies that there is no way to execute them "backwards" or to automatically generate inverse rules. The "reverse application" of forward directed generation rules is, however, an attractive facility for reverse engineering systems. This way, one may start from a complex system and simplify the system by either creating and applying "abstraction rules" or by using refinement rules in the "reverse" direction. This, however, requires rules that can either be transformed into their inverse or rules that work bi-directionally. Hence, we are planning to investigate replacing Java for specifying transformation rules with an approach that supports bi-directional rule application from the start, e.g., Triple Graph Grammars [7].

7. CONCLUSION

SPin is the result of a diploma thesis conducted at the metamodeling department of the Darmstadt University of Technology [4]. Although it currently provides only basic support for architecture stratification, it represents a starting point that can be extended into full stratification support.

Even the current SPin version, however, already demonstrates the feasibility of transforming both model elements and associated code in sync with each other, in order to obtain a fully specified complex system from a simple system, after a number of transformation steps. Current rule definitions sometimes require manual edits to the generated models—for user code, such as implementations of "visit" methods for the visitor pattern—however, this is no principle limitation of the approach. More sophisticated rules and corresponding annotation dialogs will allow the continuous and fully automated generation of models from top to bottom.

Transformation rules are user-definable, typically using a convenient mix of SDM (for pattern matching) and Java (for an unconstrained definition of transformations). Since SPin is able to dynamically integrate new rules, the development of the main system model and corresponding rules, can proceed in an interleaved and very interactive manner.

The creation of new rules in SPin is heavily assisted by a number of convenient utilities, such as support for modifying method bodies, element creation, and the synchronization of the UML metamodel.

In addition, the work on SPin resulted in a pattern-like notation for refinement annotations that enable transformation parameters to be specified both graphically (through labeled links) and non-graphically (through primitive parameter types entered into a corresponding dialog). This way one achieves fine grained control of the transformation process, using a concise notation.

We believe that SPin already represents an interesting starting point for supporting architecture stratification, but as we have outlined in the previous section on future work, we are convinced that it has an even higher potential for further development.

8. REFERENCES

[1] C. Atkinson and T. Kühne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software*, 20(1):81–89, 2003.

[2] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language and Java. Technical report, AG-Softwaretechnik, Fachbereich 17, Universität Paderborn, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster*. Addison-Wesley, 1996.

[4] F. Klar. SPin – Ein Werkzeug zur Realisierung von Architektur-Stratifikation. Diplomarbeit, Technische Universität Darmstadt, April 2005, http://www.klarentwickelt.de/doc/science/diplomarbeit_klar.pdf.

[5] T. Mens, K. Czarnecki, and P. V. Gorp. A taxonomy of model transformations. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005.

[6] U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. Technical report, Computer Science Department, University of Paderborn, 2000.

[7] A. Schürr. Specification of graph translators with triple graph grammars. Germany, June 1994. Herrschin, Springer Verlag. Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science.