JaMoPP: The Java Model Parser and Printer

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
florian.heidenreich|jendrik.johannes|mirko.seifert|c.wende @tu-dresden.de

Abstract. Model-Driven Software Development is based on standardised models that are refined, transformed and eventually translated into executable code using code generators. However, creating plain text from well-structured models creates a gap that implies several drawbacks: Developers cannot continue to use their model-based tool machinery, relations between model elements and code fragments are hard to track and there is no easy way to rebuild models from their respective code.
This paper presents an approach to bridge this gap for the Java programming language. It defines a full metamodel and text syntax specification for Java. From the latter a parser and a printer are generated to perform the conversion between text and models in both directions. Through this, Java code can be handled like any other model. The implementation is validated with large test sets, example applications are shown, and future directions of research are discussed.

1 Introduction

The goal of Model-Driven Software Development (MDSD) is the (semi-)automatic generation of software systems from models across multiple stages [1, 2]. That is, not only code—such as Java source code—is generated from models, but also models are transformed and refined towards other models. Using a standardised metalanguage, language-independent tools can be utilised to manipulate and analyse models defined in different modelling languages. Since a metamodel defines types and constraints for sentences (i.e., models) of a language, all model manipulations done by different tools can be checked for correctness.

As described above, almost all transformations in an MDSD process produce structured and typed data, even on the level of abstract system modelling (e.g., Use Case modelling). However, the last transformation, from models to code artefacts, is often done in a weak structured and untyped manner using string processing template engines. This is a paradox since type checking and correctness is most important when producing compilable artefacts.

In addition, many modellers—in particular the ones involved in the last steps of an MDSD process—are also programmers. Today's common practices, such as annotating models with (again untyped) Java code, show that a tighter integration between modelling and programming languages is often desired.

We argue that there is a *gap* between modelling and programming languages that is worth closing to tackle many of todays problems in the last steps of MDSD processes. The *gap* is caused by the fact that modelling and programming languages are too often regarded as different things. If a programming language like Java would be handled equally as other modelling languages, the issues discussed above could be addressed: existing modelling tools could handle Java programs as they handle other models—structured and typed—instead of treating them as plain text. By using metamodelling tools for extension and reuse of language specifications, Java (or parts of Java) can be integrated with other modelling languages. As a consequence Java can be utilised as any other modelling language.

To *close the gap* for the Java programming language, we propose the Java Model Parser and Printer (JaMoPP). JaMoPP leverages Java to a modelling language by providing the following:

1. JaMoPP defines a complete metamodel for Java that covers the whole language. The metamodel is defined in the commonly used metamodelling language Ecore [3] which allows it to be processed by metamodelling tools for custom modification, extension or reuse.
2. JaMoPP defines a text syntax that conforms to the Java language specification and from which a parser—to create instances of the metamodel from Java source code—and a printer—to transform instances of the metamodel into Java source code—are generated. Similar to the metamodel, the text syntax—being a model on its own—can be customised, extended and reused and the tooling (i.e., parser and printer) can be regenerated.
3. JaMoPP's Java metamodel reflects the static semantics of Java through cross-references between model elements. These references are established after parsing by an analysis mechanism that implements the specifics of Java's static semantics. This mechanism is implemented in a modular fashion to support customisation, extension and reuse.

With JaMoPP, Ecore-based modelling tools can process Java files in the same manner they process other models. Additionally, the same tools can be applied to the Java metamodel itself. We explore different scenarios later in this paper.

The paper is structured as follows: Sect. 2 gives details about the design and implementation of JaMoPP. This includes our metamodel for the Java language, the text syntax specification, the static semantics analysis, the integration of compiled Java classes and details about the extensive test suite that was used to validate our implementation. Section 3 contains five example applications that demonstrate the benefits of representing Java source code as models. We compare our work to existing approaches in Sect. 4 and draw conclusions in Sect. 5.

2 JaMoPP in Detail

This section introduces the different parts of JaMoPP in detail. In Sect. 2.1 we discuss our Ecore metamodel for Java. Section 2.2 presents details of an EMFText [4] syntax specification for Java and Sect. 2.3 introduces a class file reader. Afterwards, Sect. 2.4 shows how parsing and printing is integrated in the resource handling of the Eclipse Modeling Framework (EMF) [3] to provide transparent access to the Java models for arbitrary tools (e.g., editors or transformation engines) and explores the static semantics analysis implementation. Finally, Sect. 2.5 gives details about the process used to test the different parts of JaMoPP.

2.1 Java Metamodel

There is a huge amount of tools that operate on Java programs and therefore implicitly or explicitly operate on instances of the Java metamodel. Despite the great variety of software that depends on the Java metamodel, it turned out few have an explicit representation of it. On the other hand, there are many language-independent modelling tools that could work on Java programs if an explicit metamodel defined in a standard metamodelling language as Ecore would exist. These tools could then not only be used to work on Java programs, but also to customize the metamodel itself (e.g., to realise Java extensions). Such an explicit Java metamodel needs to reflect the complete syntax and static semantics of Java. Static semantics can be represented in a metamodel using cross-references in which the results of static semantics analysis (e.g., method calls, variable references, constructor calls and type information) are stored. The foundation for JaMoPP is such a metamodel which is derived in the following.

The Java Language Specification (JLS) [5] itself does not provide a complete explicit metamodel. Rather, the syntax and semantics of Java are specified either informally or using syntax diagrams. The closest thing to a metamodel Java itself provides are its reflection facilities. However, these do not assemble a complete metamodel since they do not capture fine-grained elements like statements.

Existing Java parsers (e.g., `javac` or the Eclipse Java Development Tools (JDT)) have internal metamodels written in Java (i.e., a set of classes to create the Abstract Syntax Tree (AST)). These classes represent—in contrast to Java's reflection—all the features and constructs of the Java language, but are still not defined in a standard metamodelling language. One implementation that is closest to a standardized solution is the Java 5 implementation of the Stratego/XT system [6]. However, also this approach does not provide an integration with standard metamodelling tools.

The metamodels for Java that were defined in a standard metamodelling language so far again suffer the drawback of incompleteness. The OMG has published a metamodel [7] that provides coarse-grained concepts of Java (classes, methods, fields, ...), but does not provide blocks, statements or expressions. Another metamodel was developed in the MoDisco[1] project. While, this metamodel

[1] http://www.eclipse.org/gmt/modisco

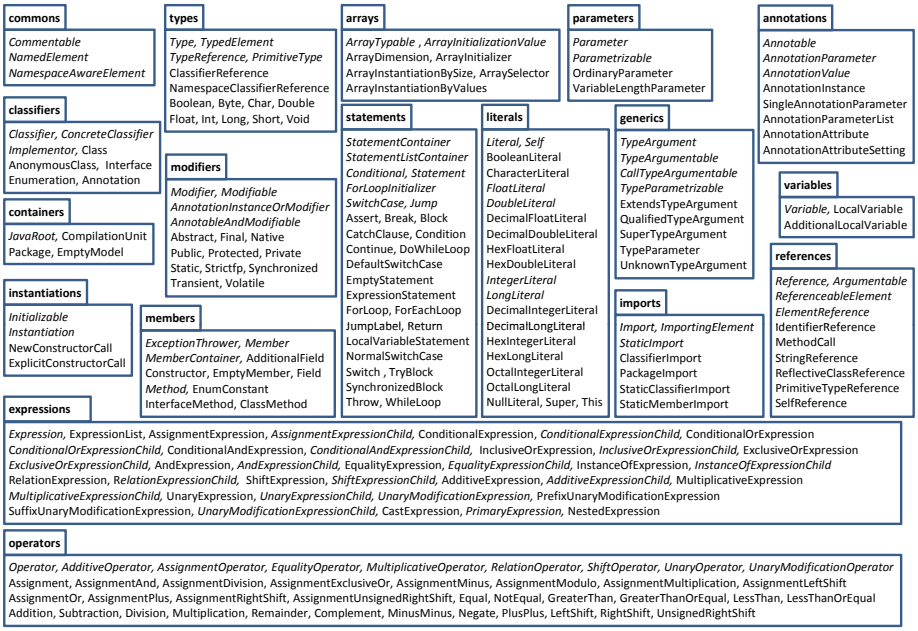

Fig. 1. Metamodel for the Java language

is almost complete, it still has a couple of shortcomings. First, it is purely tree structured—it does not model any static semantics (identifiers are not resolved to their respective elements, but stored as plain strings). Thus, consistency is not necessarily preserved when manipulating models. Second, the MoDisco project creates model elements by traversing the Eclipse JDT AST. This is fine for the purpose of discovering models from Java programs (creating models from source code), but does not support the reverse direction (creating source code from models). From our perspective, the most complete metamodel was defined in the SPOON [8] project. In contrast to the MoDisco model, it contains cross references to express, for example, method calls. It does, however, not provide abstractions for concepts used in different metaclasses (i.e., it contains no abstract metaclasses) and the static semantics analysis is performed by the JDT, leading to a close coupling with the latter. Both the missing abstractions and the close coupling with the hand-coded JDT tooling hinder customization and extension.

As we could not find a metamodel that does both conform to a well established metamodelling language (in particular Ecore) and fulfils our need for completeness, we decided to compare the existing metamodels, extract commonalities and extend them to fully support the JLS. Figure 1 shows the resulting packages and classes of our metamodel. The attributes, references and inheritance

relations are not shown here due to space limitations. However, the complete metamodel is available online at the JaMoPP Website[2].

Our metamodel defines 80 abstract and 153 concrete classes, which are divided into 18 packages. The SPOON metamodel, which we consider the most complete model of the ones mentioned before, contains 140 concrete classes. The higher number of classes found in our metamodel has two reasons. First, we pushed all attributes that were present in multiple metaclasses to an abstract super class to reduce redundancy. Second, elements that can be used exchangeable (e.g., `ForLoop` and `WhileLoop`) share a common super class (`Statement`). This allows to use these elements uniformly.

Besides these additional abstract classes, our metamodel contains all elements of the Java language (e.g., classifiers, imports, types, modifiers, members, statements, variables, expressions and literals) and in particular those that were introduced with the release of Java 5 (e.g., annotations and generics).

2.2 Text Syntax Specification for Java Source Files

To make use of the metamodel defined above, a text syntax specification is needed from which tooling (i.e., a parser and a printer) can be generated. This task is performed by our tool EMFText. For each concrete metaclass we defined a text syntax rule. In the following, we will use the two rules given in Listing 1 to exemplify the generation procedure. Showing all rules is not possible due to space limitations, but the complete set of rules is available on the JaMoPP Website.

```
1 CompilationUnit  ::= ( "package" namespaces[] ( "." namespaces[] )* ";" )?
2                      ( imports )*
3                      ( classifiers )+ ;
4 ClassifierImport ::= "import" ( namespaces[] "." )+ classifier[] ";" ;
```

Listing 1. CS specification for metaclasses CompilationUnit and ClassifierImport.

The language used by EMFText to specify text syntax is called CS. Syntax rules in CS consist of a left and a right-hand side, where the former references a metaclass and the latter defines the syntax for this class. The right-hand side is built from two atomic elements: keywords (enclosed in double quotes) and feature names. Feature names refer to attributes or references defined in the respective metaclass. Furthermore, CS uses concepts from the Extended Backus-Naur Form (EBNF). Elements can be grouped using parenthesis, optional parts are tagged with a question mark (Line 1), optional parts that may occur multiple times are indicated by a star (Line 2) and mandatory, but possibly repeating elements end with a plus sign (Line 3). CS rules are used to generate a parser, a printer and a set of reference resolvers for the defined language. Thus, we will explain next how these three components are derived.

Parsing is based on a Context-free Grammar (CFG), which is derived from the CS specification and implemented using a recursive descending parsing strat-

[2] http://jamopp.inf.tu-dresden.de

CS element	CFG element	Example
Keyword	Terminal symbol	`"package"` (Line 1)
Feature (attribute)	Terminal symbol	`namespaces[]` (Line 1)
Feature (containment reference)	Non-terminal symbol	`imports` (Line 2)
Feature (non-containment reference)	Terminal symbol	`classifier[]` (Line 4)

Table 1. Mapping of CS elements to a CFG

egy. The atomic elements of the CS specification are mapped to a CFG as shown in Table 1.

The keywords form terminal symbols, while features can be mapped either to non-terminals or terminals. This decision depends on the properties of a feature defined in the metamodel. Attributes and non-containment references (i.e., edges that do not belong to the AST) must be followed by square brackets (e.g., `namespaces[]` in Line 1). and will be interpreted as terminal symbols. The value of the terminal (i.e., the concrete text that appears in the source file) is taken as value for the attribute (in this case `namespaces`). If the feature is a non-containment reference (e.g., `classifier[]` in Line 4), the text is interpreted as identifier for the referenced element. This means that the classifier is defined somewhere else (i.e., in the file of the imported classifier) and represented by a symbolic name here. To map this name to the correct imported classifier a reference resolver is generated by EMFText.

For containment references (e.g., `imports` in Line 2) a non-terminal is derived. By looking at the metamodel EMFText knows that `imports` is a reference to objects of type `Import`. Thus, for the non-terminal of `imports` a parser rule with alternatives for all concrete subclasses is created. One of this alternatives is, for example, a `ClassifierImport` (Line 4 in Listing 1).

Resolving is needed to map names to the respective referenced elements. An example for an reference is the reference between an `IdentifierReference` (package `references`) used in an expression and the `LocalVariable` (package `variables`) it refers to. Rules for resolving such references are specific to Java and must therefore be added by extending the resolvers generated by EMFText. In addition, there are also external references (e.g., imported classifiers). These references span multiple files and are discussed detailed in Sect. 2.4.

Printing is the inverse process to parsing. EMFText generates a printer from the CS specification that contains separate print methods—one for each concrete metaclass. According to the CS rule that belongs to a class, the printer emits keywords for model elements, the values of element attributes and recursively calls subsequent methods to print contained elements.

For the first rule shown in Listing 1, the print method first emits the keyword `package`, but only if the `namespaces` attribute is set. Afterwards the package identifiers are printed from the sequence of package names stored in the `namespaces` attribute. When printing references the printer distinguishes containment and non-containment references. For a non-containment reference, the printer inverts the reference resolving done by the parser. Instead of the referenced element it-

self, only the respective symbolic name is printed. After processing the package declaration of the `CompilationUnit`, the contained import elements are printed using the appropriate print method (e.g., the one for `ClassifierImport`). Finally the parser prints the `CompilationUnit`'s body using the same mechanism.

As a final remark, we must say that our syntax is less restrictive than the ones used by Java compilers. For example, modifiers (e.g., `abstract`) are allowed for certain elements (e.g., `Field`s) where they cannot actually be used. We could add constraints to our metamodel that can detect such violations after parsing, but this is subject to future work.

2.3 Class File Handling

Our work concentrates on parsing and printing Java source files. Still, a class referenced from a source file might sometimes only be available in byte code format. An example is a reference to the class `java.lang.Object` of Java's standard library which might not be available as source code. For this purpose, we implemented the `ClassFileModelLoader` which uses the BCEL[3] byte code parser and translates the output of the BCEL parser into an instance of the Java metamodel. We require only the public header information of class files and therefore only instantiate elements of the following metamodel packages: `containers`, `classifiers`, `members`, `parameters`, `types`, `arrays` and `generics`. Printing (saving) is not supported for class files.

2.4 Resource Management and Java Classpath

As mentioned in Sect. 2.1, our metamodel is defined in Ecore which is the metamodelling language of EMF. In addition to the fact that Ecore is widely accepted and used, EMF provides many facilities and tools to put metamodels into practical use. In particular, a new language defined in Ecore can be plugged into EMF transparently. EMFText generates printer and parser in a fashion that they can be transparently used by any EMF-based tool. Thus, despite of their specific text syntax, Java models can be handled as any other models by the tools. This section gives a brief overview of the features of EMF used to achieve this (illustrated in Fig. 2).

In EMF, each model is associated with a `Resource` (1) and has a unique identifier—a URI. A `Resource` is responsible for loading and saving its model. EMFText generates a specialised `Resource` from a CS specification and connects it with a generated parser (for loading) and printer (for saving). Figure 2 shows the `JavaResource` (2) together with the `JavaParser` (3) and `Printer` (4) all generated from the Java CS specification presented in Sect. 2.2. Additionally, we extended the `JavaResource` (1) to use the `ClassFileModelLoader` (5) of Sect 2.3.

`Resource`s (1) are manged by `ResourceSet`s (6). Each EMF-based modelling tool (7) (e.g., all tools used in Sect. 3) connects to EMF by instantiating a `ResourceSet`. A `ResourceSet` can acquire `ResourceFactory`s (8) to create new

[3] http://jakarta.apache.org/bcel

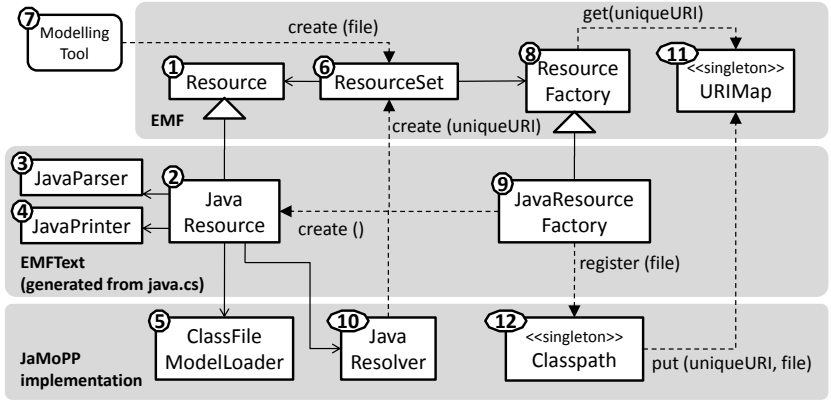

Fig. 2. Java model resource management

resources. A factory is automatically selected based on the extensions of the `Resource`'s URI. The actual encoding of a resource is therefore hidden to the modelling tool (7) that uses a `ResourceSet` (6). For `*.java` and `*.class` files, the `JavaResourceFactory` (9) (also generated by EMFText), is therefore selected.

EMFText also generates stubs for resolver implementations. Resolvers plug into EMF's proxy mechanism and are responsible for resolving non-containment references on demand. In Java this is basically name and type resolution. Our `JavaResolver` (10) realises this for references within one resource as well as for *cross-resource* references. Cross-resource references required our special attention because of the high fragmentation of Java models into several files (usually one class per file). An example of a cross-resource reference is the reference between a `Field` (package `members`) and the `Classifier` (package `classifiers`) that defines its type and resides in another `Resource`. EMF provides the global `URIMap` (11) as a global registry for resources. The `ResourceSet` can use it to find additional model resources on demand. To construct unique URIs for Java classes we provide the `Classpath` (12). Through this, the actual physical location and the encoding of a `JavaResource` (whether it is a source or class file) is hidden.

2.5 Test and Evaluation

The previous sections presented our metamodel, as well as a parser and a printer for the Java language. To show that our approach can handle industrial-sized applications, a large test suite and a test process were created that allow to check whether JaMoPP complies to a reference implementation (the JDT).

The objectives of our test suite are to verify that 1) our parser accepts valid Java programs, 2) our resolver resolves all names and types, 3) the so created model instance has the expected structure, 4) our resolver de-resolves all cross-references to their correct string representation and 5) our printer emits correct

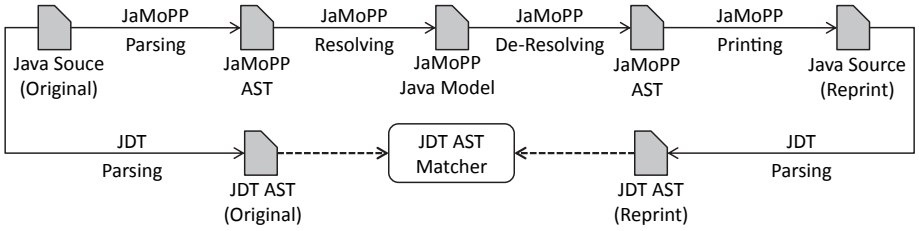

Fig. 3. Test process for validating printer and parser

and complete source code for model instances. To meet these five objectives the test process shown in Fig. 3 was employed.

Starting with a valid Java source file (upper left corner of Fig. 3), both our parser and the reference implementation (the JDT) process the given input file and create an AST—which in the case of our parser is a model with unresolved cross-references. Next, our resolver first resolves all names to cross-references and afterwards de-resolves the cross-references to names again. Then the model is printed to its text form, which is again processed by the JDT parser (lower right corner of Fig. 3). The second JDT AST is then compared to the original JDT AST using the JDT's own AST matcher.

The given procedure meets the five objectives given above. 1) If our parser does not accept a valid file it can either throw a parsing exception or run forever—both cases are detected by unit tests. 2) After resolving, the test checks the model for unresolved references and throws an error if it finds any—thus testing if the resolver succeeded. 3) The model instance is checked for completeness through several mechanisms. Elements that are referenced but missing are already detected by the resolver. Furthermore, since the resolving is a complex procedure, also other structural errors in the model lead to failures in the resolver. Any other missing information can not be printed and will thus be detected by the AST matcher. In addition, we manually wrote unit tests for distinct Java language features. They consist of assertions that check whether the correct model elements are created for given pieces of source code. 4) When resolving succeeded, the references are de-resolved again and printed. However, it's still possible that, i.e., a method call gets resolved to the wrong method but printing after de-resolving results in the same method call. These kinds of errors cannot be detected by the AST matcher but are covered by manually written unit tests. 5) Other errors that are caused by wrong printing only are also detected by the AST matcher. When the matcher delivers a failure, we manually compared the original and the reprint to discover the location of the error.

The input for this test process were 79.017 Java files (15.5 million non-empty lines including comments). In the end, all 79.017 Java files passed the test process. Among these files were some self-defined classes for testing individual language features, the sources of two IDEs (Eclipse 3.4.1 and Netbeans 6.5.1), application and web servers (Apache Tomcat 6.0.18 and JBoss 5.0.0 GA), math

libraries (Apache Commons Math 1.2, Mantissa 7.2), web frameworks (Google Web Toolkit 1.5.3, Spring 3.0.0M1 and Apache Struts 2.1.6), an XML Parser (XercesJ 2.9.1), a code generator framework (AndroMDA 3.3), the Sun JDK 1.5.0 Update 16, as well as subsets[4] of the compiler test suite JACKS and the JDT test project. Due to space limitations we kindly refer to the JaMoPP Website for the links to the test sources.

The tests were automated using JUnit and revealed many errors during the specification of the text syntax and the implementation of the resolvers. Due to the specific nature of the test inputs, quite different classes of errors were found. For example, the tests in the JACKS suite are very specific and contain many special corner cases mentioned in the JLS. Escaped unicode sequences used in keywords, optional end of file characters and complex number literals are just a few examples.

In some cases the test process failed at almost every possible step and showed that the complex procedure is worth the effort. To give an example, one of the tests revealed that our parser did not accept `switch` statements where the `default` case was not the last one. Thus, we corrected the CS specification, regenerated the parser, which then in turn accepted the `default` cases. But, the print test was still failing, because the `default` case was represented in the metamodel separately from the other `switch` cases. As a consequence, the `default` cases were parsed correctly, but still printed at the end of the case list. Addressing all cases using one `0..*` reference in the metamodel solved the problem.

In summary we can say that the presented test procedure does not guarantee the correctness or completeness of our implementation, because no proof was established. Nonetheless, the executed tests were performed on industrial-sized applications and give confidence that JaMoPP can be applied in practice.

3 Examples

Bridging the textual and model-based representation of Java programs has several benefits. In this section we present a selected set of applications that can profit from JaMoPP. The applications represent typical development activities (e.g., generating, analysing and visualising code), but also cover more advanced actions (e.g., composing programs or building product lines).

The running example which will be used throughout this section is a contact management application (cf. Fig. 4). A `ContactList` is organised into several `Group`s, each containing a number of `Contact`s. We distinguish `Person` and `Company` contacts. In addition information like `Address`es, or `Relationship`s between `Contact`s is managed.

The first example (Sect. 3.1), shows how a general purpose model transformation language can be used to generate Java code (i.e., models of Java programs) from UML models. As this generation process is performed solely based on models, the resulting code is guaranteed to be syntactically correct. The second

[4] Only compilable classes were used, invalid files were omitted

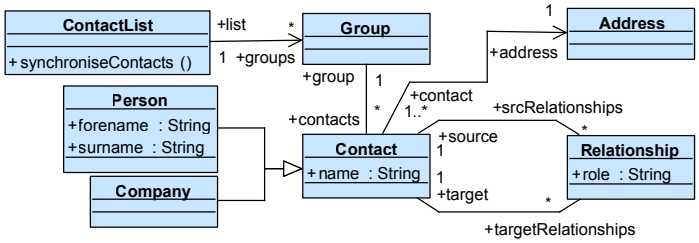

Fig. 4. Class diagram for the contact management application

application (Sect. 3.2) employs JaMoPP for model-based code analysis. Based on the Object Constraint Language (OCL) we show how to query and analyse Java code by means of model-based technology. Furthermore, Sect. 3.3 shows how existing model visualisation tools can be applied to Java source code.

A more advanced application follows in Sect. 3.4 where Java models are used to compose programs in a syntactically safe manner. This is in contrast to common template engines which merely concatenate strings, but can give no guarantees with regard to syntactical correctness. Finally, we will conclude this section by showing how Software Product Line Engineering (SPLE) can benefit from our approach (Sect. 3.5).

The main objective of our applications is twofold. First, they show how existing model-based technology can be easily reused to operate on a textual general purpose language. Second, our applications present advantages that are not solely connected to the reuse of tools, but show that operating based on standardised models can conquer some of the limitations proprietary text-based tools have.

3.1 Java Code Generation with Model-2-Model Transformations

Model transformations are central to any MDSD process. In the literature, such transformation are usually divided into two categories: model-2-model (m2m) and model-2-text (m2t). The former transform models (i.e., instances of metamodels) into other models; the latter models into plain text. An m2t transformation can be used to generate any kind of, not necessarily structured, text (e.g., documentation). Currently, m2t transformations are often used to generate code—not seldom Java code. Since for most m2t engines their output is simply a sequence of strings, they can not guarantee correctly structured results. By regarding Java as a modelling language, we can replace m2t transformations with m2m transformations which are aware of the metamodel of their output. Some benefits of this are shortly examined in this section.

We used the Atlas Transformation Language (ATL) of the Eclipse M2M Project[5] to define a transformation from a UML class diagram into a set of Java classes. The transformation can handle all concepts of UML used in Fig. 4. ATL is a declarative rule-based language relying on OCL. Consult [9] for more details.

[5] http://www.eclipse.org/m2m

```
1  rule Property {
2    from umlProperty : uml!Property
3    to javaField : java!Field (
4      name <- umlProperty.name,
5      type <- typeReference
6    ),
7    typeReference : java!TypeReference (
8      target <- if (umlProperty.upper = 1) then
9          umlProperty.type
10       else
11         java!Package.allInstances()->any(p | p.name = 'java.lang').compilationUnits->collect(
12                 cu | cu.classifiers)->flatten()->any(c | c.name = 'LinkedList')
13       endif,
14     typeArguments <- if (umlProperty.upper = 1) then
15         Sequence{} --empty type argument list
16       else
17         Sequence{typeArgument}
18       endif
19   ),
20   typeArgument : java!QualifiedTypeArgument(
21     target <- umlProperty.type
22   )
23 }
```

Listing 2. ATL rule that translates a UML `Property` into a Java `Field`

Representatively, Listing 2 shows one rule of our ATL transformation that transforms UML `Properties` into Java `Fields`. In UML, `Attributes` as well as `AssociationEnds` are `Properties` and are thus matched by the rule shown in Line 2. The corresponding concept in Java is `Field` of which one is constructed as seen in Line 3. The name of a UML `Property` is mapped to the name of the constructed Java `Field` (Line 4). For the `Field`'s type, a `TypeReference` is created and assigned (Lines 5 and 7).

Determining the concrete target of the `TypeReference` is a bit more complex: if the UML `Property` has a multiplicity greater than one, it should be mapped to a list type. Therefore, depending on the UML `Property`'s upper bound, we either map to the Java `Type` that corresponds to the UML `Type` of the `Property` (Line 9[6]), or find the `java.lang.LinkedList` class by examining the Java standard library packages (Lines 11–12), which are also represented as models and given as input into the transformation. If a list is used, it should be type-argumented with the correct type. Therefore, a `TypeArgument` is constructed (Line 20), set to the correct type (Line 21) and assigned (Line 17).

The m2m transformation rule presented above, shows how a complex type mapping between UML and Java can be realised without bothering about details of the text syntax. In a m2t transformation one has to keep care of opening and closing angle brackets for type arguments and that all "`;`" are correctly placed.

However, we do not want to imply that code should only be produced by m2m transformations in the future. As one can see, for instance, in Lines 11–12 of Listing 2, m2m transformation rules tend to become complex. With a

[6] here ATL automatically determines the rule that translates the UML `Type` into the Java `Type` (rule not shown here)

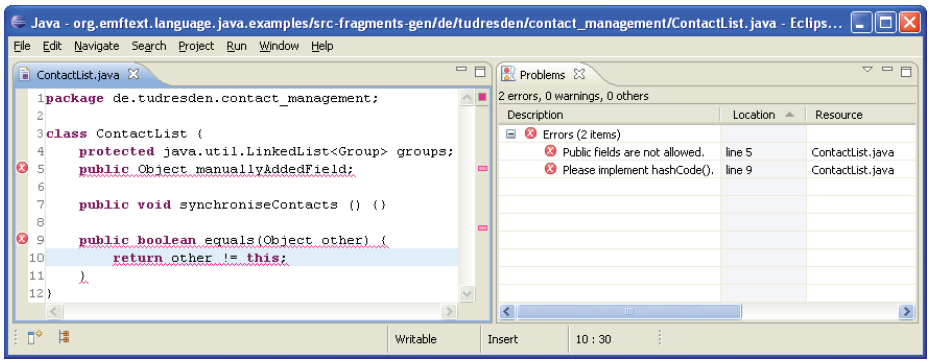

Fig. 5. Screenshot of the RestrictED tool

m2t template, this part can be written down more elegantly. With the Java metamodel at hand, we can start to combine the advantages of m2m and m2t transformations. A typesafe template language for Java could be defined by extending the metamodel with template functionality. Section 3.4 will show that such extensions can now be performed using metamodelling.

3.2 Generic Source Code Analysis based on OCL

Source code analysis is a common approach to find irregularities and bugs in programs or to enforce coding conventions. As a consequence various tools exist to support developers in this task (e.g., Findbugs [10] for the Java language). Based on pattern matching or abstract interpretation these tools can find faults in programs without actually executing them.

Using model-based representations of programs, such analysis can be performed uniformly for arbitrary languages. For example, our analysis tool RestrictED [11] uses declarative expressions written in OCL to specify undesired code patterns. Our earlier experiments [11] showed that static analysis can be performed on small toy languages. With the advent of JaMoPP this approach can be applied to the Java language in its full extent.

Returning to our running example, we might now want to make manual additions to the code generated in the last section. Such additions should not violate the guidelines that were respected by the code generation. We will use RestrictED to present two simple OCL invariants to check common coding conventions.

Listing 3 shows the first OCL expression. This invariant states that fields must not use the modifier **`public`** by collecting the set of modifiers that have the type `Public` and checking that this set is empty.

```
1 context members::Field inv:
2   self->modifiers->select(m|m.oclIsKindOf(modifiers::Public))->size() = 0
```

Listing 3. OCL invariant to find public fields

```
context members::Method inv:
  if (self.name = 'equals') then
    (self.container->members->exists(m | m.name = 'hashCode'))
  else true endif
```

Listing 4. OCL invariant to detect missing `hashCode()` methods

A second invariant (Listing 4) states that whenever a classifier contains a method `equals()` an implementation of the `hashCode()` method must be present too. This is important because the implementation of both methods must match to ensure correct comparison in collections.

Figure 5 shows a screenshot of the RestrictED tool applied to one of the classes generated in Sect. 3.1. The two basic examples indicate how arbitrary OCL expressions can be used to analyse Java source code. In contrast to most other analysis tools for Java, users can add custom restrictions, do so declaratively and use the same language for all software artefacts. Furthermore, JaMoPP enables using model transformations to declaratively repair code according to given rules.

3.3 Tailored Visualisation with GMF

Visual representations raise the level of abstraction and help the comprehension of software systems [12]. Thus, especially for modelling languages graphical syntaxes enjoy a high reputation. To ease the time consuming and cumbersome task of building visualisations or even graphical editors, several model-based visualisation frameworks emerged [13, 14]. With JaMoPP providing a model-based representation of Java these technologies can now be applied to build tailored visualisations for Java programs.

Using the Graphical Modeling Framework (GMF) [14] we generated the editable diagrammatic representation for Java packages depicted in Fig. 6. It provides a comprehensive overview of a package's structure and its type interrelations and could help the exploration of big Java libraries and frameworks. As a simple example we visualised the code generated in Section 3.1.

GMF provides a model-driven approach for generating editors for EMF-based languages. Graphical primitives like nodes and arrows are used to visualise model instances. These primitives are related to and customised with regard to classes, class properties, and references of our Java metamodel. This adaptable mapping provides flexibility in to ways: First, visualisations can be tailored to the requirements of a specific engineering task. Second, an existing visualisation can easily be extended to other languages by adjusting the visual mappings.

The capabilities of the editors generated with GMF go beyond just visualising models graphically. They also allow for editing models using their graphical representation. Since the JaMoPP printer serializes Java models to source files, these changes are transparently mapped to the underlying source code. Thus, JaMoPP contributes to the development of visual tools for software development with Java.

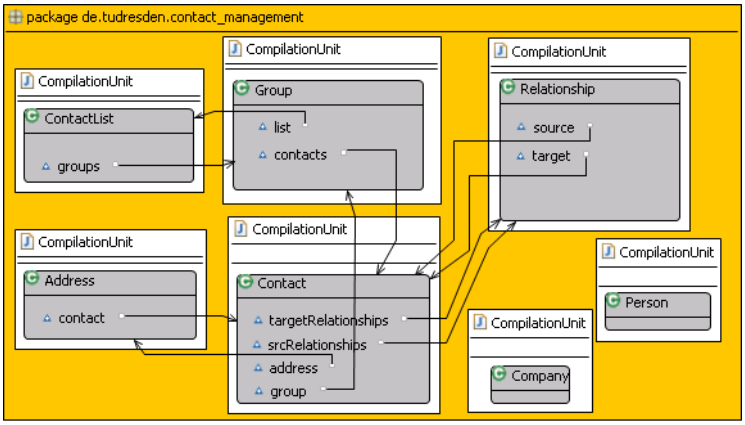

Fig. 6. Visualisation of the package structure generated for the contact management application

3.4 Java Extension to Separate Generated and Hand-Written Code

To demonstrate and experiment with new language concepts, Java is often extended with new features. These features are implemented in prototypes that can often only treat subsets of Java due to a lack of language engineering tool support. Existing tools, like editors, can no longer be used when unaware of the extension.

In MDSD, tools for metamodelling and agile language development exist that are aware of evolution and change of languages, which makes it considerably easier to perform language extension while preserving tool support. The GMF definitions demonstrated in the last section, for instance, can be easily adjusted if changes to the Java metamodel are made—and the GMF editor can be regenerated. Same applies to EMFText's CS-specifications, which have an import mechanism to extend a text syntax along with a metamodel.

The scenario we examine in this section is separation of generated code skeletons (as produced by the transformation of Sect. 3.1) from the list of statements of a method body directly defined in Java. Usually these can not be separated into single artefacts in Java, because a list of statements on its own is not a compilable unit. Furthermore, it would be helpful, if a transformation to Java can indicate where additional code is required when the information for generating it is missing. Java does also not provide a specific construct for that.

In this section, we extend Java with the two features mentioned above and utilise the Reuseware Composition Framework[7] to define a preprocessor that composes models of the extended Java language into plain Java. To do so, the Java metamodel is extended by defining a new package `reuse` with the classes 1) `StatementUnit`, which is a subclass of `JavaRoot` and has a reference

[7] http://reuseware.org

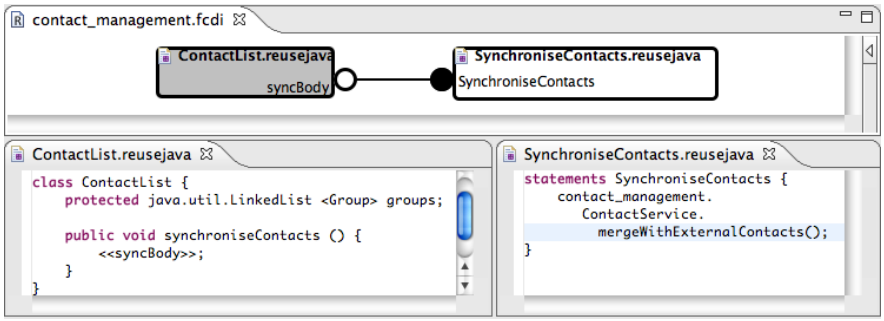

Fig. 7. Screenshot of using the extended Java with Reuseware

`statements` of type `Statement`, and 2) `StatementVariationPoint`, which is a subclass of `Statement`. Both are also `NamedElements`. This extension can be done without altering the original Java metamodel by using the import feature of Ecore.

The text syntax is extended in a similar manner. Listing 5 shows the complete specification. In Line 1, the Java metamodel (`http://...`) and its text syntax (`java`) are identified for import. In Lines 2 and 3 two additional rules that give the text syntax for the new constructs are specified. EMFText takes all the rules of the imported syntax into account when generating printer and parser. Therefore, the new rules are woven into the existing syntax: `StatementUnit` can be used as alternative for other `JavaRoots` (e.g., `CompilationUnit`) and `StatementVariationPoint` as alternative for other `Statements`.

Figure 7 shows a usage of the extended Java. The transformation of Sect. 3.1 can be extended to produce a `StatementVariationPoint` for empty methods (lower left of Fig. 7), or wherever else information is missing. EMFText's editor remains working for the extended Java and can be used to define statement lists stand alone (lower right of Fig. 7). These can be composed with the generated code by defining composition programs in Reuseware (top of Fig. 7)—thus keeping a clean separation between generated and hand-written (or hand-modeled) code.

This section showed how a Java extension can be easily realised using MDSD technologies with JaMoPP. Only two additional metaclasses and 5 lines of text syntax specification were needed. Therefore, JaMoPP enables the definition of custom extensions for Java tailored to any specific MDSD process.

```
1 IMPORTS { java : <http://www.emftext.org/java> WITH SYNTAX java }
2 RULES {
3         StatementUnit ::=            "statements" name[] "{" statements* "}" ;
4         StatementVariationPoint ::=  "<" "<" name[] ">" ">"  ";" ;
5 }
```

Listing 5. Extended Java syntax

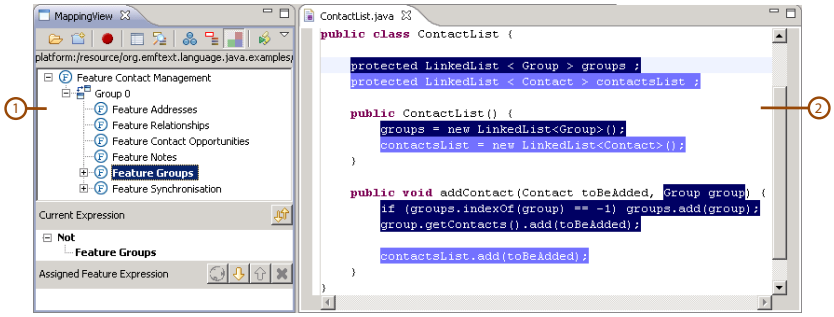

Fig. 8. Mapping features to Java code using the FeatureMapper

3.5 Software Product Line Engineering with JaMoPP

SPLE [15] defines a method to develop a set of related software applications by explicitly managing their common and variable features. Single product features are mapped to specific realisation artefacts of the Software Product Line. Thus, a concrete application can be automatically derived, by removing all realisation artefacts which are not mapped to features selected for the product variant.

To map realisation artefacts in accordance to specific features, traditionally annotation techniques like #ifdef statements, Frames [16], and Traits [17] or composition techniques like mixin layers [18], aspects [19], CaesarJ [20], or AHEAD [21] are used. While annotation techniques do not consider the structure of the annotated implementation language and, thus, can not assure the correctness of the resulting source code, compositional approaches are restricted to a coarse module granularity and can not represent fine granular feature realisations [22]. All techniques share the common drawback that they do not provide good tool support for defining and maintaining feature mappings. In addition, the lacking tool support makes it hard for developers to comprehend feature mappings.

The tool FeatureMapper[8] provides a visual paradigm for mapping features to artefacts of EMF-based modelling languages [23]. JaMoPP now enables the application of the FeatureMapper to the Java language and targets the problems of the aforementioned mapping paradigms.

Figure 8 depicts a screenshot of the FeatureMapper. The MappingView on the left (1) displays the features of a product line. EMFText's editor (2) shows a Java file. Mappings can be created by assigning code elements to features. Here the level of detail in the Java metamodel introduced by JaMoPP is of key importance. The selectable code elements comply to the granularity of the metaclasses. A concrete selection and the result of its removal from the source code can be checked for well-formedness regarding the Java metamodel. This

[8] http://www.featuremapper.org

makes the mapping as fine granular as annotation based mappings and as safe as in compositional approaches.

To address the problem of comprehensibly for developers the FeatureMapper provides several visualisation techniques. Figure 8 depicts one example—the context view. This visualisation colours the elements in the source code in accordance to the colour of the features in the feature model they are mapped to. This allows to investigate how features are realised in different product variants and how feature realisations interact. The example shows two variants of the `ContactList`. Source code elements highlighted in a darker shade belong to a variant of the `ContactList` were `Contact`s are managed in `Group`s. The Java code elements in a brighter shade represent the implementation of a simplified `ContactList` without the Feature *Group*s. Further visualisation techniques used in the FeatureMapper are discussed in [24].

The presented example shows how the model-based representation of Java code provided by JaMoPP enables a SPLE tool developed for modelling languages to be used at implementation level.

4 Related Work

In principal, the gap between the model of a system, specified using a modelling language, and its implementation, written in a programming language, can be closed in two ways. First, through traditional code generation techniques (e.g., template-based approaches) code can be directly generated from models. Second, the programming language can be lifted to the level of modelling languages, as done for Java in this paper, to create programs on the modelling level and then convert these, utilising a concrete syntax mapping, to text that can be processed by existing tools for the programming language. The second procedure introduces an additional intermediate representation that enables a number of additional opportunities as discussed for Java above.

Consequently, this section splits into three parts. First, we compare our work to existing code generation approaches that do not explicitly treat Java as a modelling language. Second, we relate our approach to other works done in the area of textual syntax mapping. Third, we look at existing work that aims at raising the Java language to the level of modelling languages.

Code Generation can also be considered as meta programming, which in turn can be performed in different ways. One can either use the concrete or the abstract syntax of the target language, or emit code using string literals [25]. Template languages are based on the concrete syntax and are translated to meta programs containing string literals in order to execute them. Most Computer-Aided Software Engineering (CASE) tools (e.g., Borland Together[9], MagicDraw[10], or TOPCASED[11]) utilise template languages of this type and source code is therefore typically represented as plain text. That means, the

[9] http://www.borland.com/de/products/together/
[10] http://www.magicdraw.com/
[11] http://www.topcased.org/

structure of the target language is not regarded when specifying the generator. JaMoPP follows the suggestion of [25] and enables the use of an object-language aware template language. This avoids the following issues that can be found in the text based code generation techniques. First, it is hard to ensure that the generated code conforms to the target language. Second, the generated implementation is only loosely connected to the models it was derived from. Establishing trace links between the models and the generated source code is hard if code generation is performed on the level of strings. Third, because of the loose connection between models and code, CASE tools must use reverse engineering to get back to the modelling level. Fujaba [26] provides a particular interesting approach in this area. To reverse engineer models from generated code Fujaba uses a template-based parsing approach. However, this approach works only well for code originally generated by Fujaba. Manually added code can only be parsed if it strictly conforms to the conventions of the used code templates. JaMoPP cleanly separates the two concerns (mapping models to Java and mapping Java to text) and enables traceability for code generation because this is performed as a model transformation. Thus, both the conformance to the target language (i.e., Java) and the loose connection problem are addressed by our approach.

Textual Syntax Mapping Besides EMFText, a number of tools to provide textual syntax for models exist. A comprehensive classification in [27] shows that most of these tools share many features of EMFText. EMFText was chosen for JaMoPP—in addition to the fact that we are familiar with the tooling—because of: a) its resolving mechanism that can be tailored to the needs of a complex language as Java b) its capability to generate printer and parser from a single specification c) its import mechanism, which allows for modularisation, extension and reuse of a syntax specification d) its tight integration with EMF through which developers using EMF-based tools can directly profit from JaMoPP.

To the best of our knowledge, none of the other tools analysed in [27] provides all these functionalities in combination. TCS [28], for instance, provides a nice way to specify scoping rules declaratively. Unfortunately, these are not sufficient to implement the complex resolution rules of Java and do not support resolving among multiple resources. Xtext and Xpand [29] could be used in combination, with Xtext mapping text to models and Xpand doing the inverse. However, this leads to the need of maintaining two separate syntax specifications (one for each direction). This is error-prone for a large language like Java and complicates reuse and extension. In the above settings alternatives to Xpand like MOF2Text [30] or the Epsilon Generation Language [31] could also be used for the mapping from models to text. We also did not find a suitable import mechanism for language extension in any of the other tools.

Java as a modelling language As mentioned in Sect. 2.1, the first step towards lifting Java to the level of modelling languages is to provide a metamodel. Existing Java compilers, primarily javac[12], Jikes[13], and GJC[14], use internal rep-

[12] http://www.java.sun.com
[13] http://jikes.sourceforge.net/
[14] http://gcc.gnu.org/java/

resentations of Javas concepts, i.e., they represent them by a set of Java classes. In contrast, using a standardised, Ecore-based representation of abstract syntax, allows to coordinate different activities and tools for software development with Java. This need has been discovered before, but, as described in detail in Sect. 2.1, the resulting metamodels lack features we found indispensable. The most complete, Ecore-based metamodel we found was specified in the SPOON project. SPOON relies on the Eclipse JDT to parse source code and print modifications. Language extensions involving new syntactical constructs, as we perform them with JaMoPP, can not be achieved with SPOON.

5 Conclusion

Recent publications (e.g., [32]) show that treating a programming language as modelling language is needed to close the gap that was introduced by traditional code generation techniques. The JaMoPP approach contributes 1) a comprehensive metamodel of Java defined in the widely used metalanguage Ecore that captures the static semantics of Java in cross-references, 2) an extensible text syntax specification from which parser and printer are directly generated and 3) a modular implementation of Java's static semantics analysis. All components conform to the JLS and cover the complete Java language from high-level constructs like `Package`, `CompilationUnit`, or `Class` to the lowest granularity of individual `Operator`s including also comments. Therefore, JaMoPP exceeds the level of detail of other approaches for connecting specification models with (Java-based) system implementations. JaMoPP was tested for completeness with an extensive test suite. This shows that the idea of representing source code as models is feasible and can therefore be transferred to other programming languages.

With JaMoPP, we closed the gap between modelling languages and the implementation language Java. We showed how different problems caused by this gap are tackled with JaMoPP. On the meta-level, the JaMoPP specifications can be extended or reused to enhance the quality of code-generation templates or to reuse parts of Java in other modelling languages. The model-based representation of Java code enables the usage of MDSD tools in areas like software visualisation or software product line engineering.

The presented applications can only indicate the full potential of JaMoPP. They encourage further investigations and open new perspectives to leverage MDSD. To name a few, metamodelling, (bi-directional) model transformations, model-based tracing and other MDSD concepts can now be used to build type-safe Java template languages, separate hand-written from generated Java code or to improve round-trip engineering of Java programs. Generally spoken, Java can now be treated as a modelling language and therefore integrated into MDSD as deeply as any other modelling language.

Acknowledgement

This research has been co-funded by the European Commission within the FP6 project MODELPLEX #034081, the FP7 project MOST #216691 and by the German Ministry of Education and Research (BMBF) within the projects feasiPLe and SuReal.

References

1. Ritsko, J.J., Seidman, D.I.: Preface. IBM Systems Journal – Special Issue on Model-Driven Software Development **45**(3) (2006)
2. Völter, M., Stahl, T.: Model-Driven Software Development. John Wiley & Sons (2006)
3. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd Edition. Pearson Education (2008)
4. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In Paige, R.F., Hartman, A., Rensink, A., eds.: Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2009). Volume 5562 of LNCS., Springer (2009) 114–129
5. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The (3rd Edition) (Java Series). Addison-Wesley Professional (2005)
6. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72**(1-2) (June 2008) 52–70 Special issue on experimental software and toolkits.
7. Object Management Group: Metamodel and UML Profile for Java and EJB Specification Version 1.0. formal/2004-02-02 (2004)
8. Pawlak, R.: Spoon: Compile-time Annotation Processing for Middleware. IEEE Distributed Systems Online **7**(11) (2006)
9. ATLAS Group: ATLAS Transformation Language (ATL) User Guide. http://wiki.eclipse.org/ATL/User_Guide (February 2006)
10. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Notices **39**(12) (2004) 92–106
11. Seifert, M., Samlaus, R.: Static Source Code Analysis using OCL. In Cabot, J., Van Gorp, P., eds.: Proceedings of the MoDELS 2008 Workshop on OCL Tools: From Implementation to Evaluation and Comparison (OCL 2008). (2008)
12. Diehl, S.: Software Visualization. Springer (2007)
13. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Towards Graph Transformation based Generation of Visual Editors using Eclipse. Proceedings of the Workshop: Visual Languages and Formal Methods (VLFM) (2004)
14. GMF Team: Graphical Modeling Framework. http://www.eclipse.org/gmf/
15. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer (2005)
16. Jarzabek, S.: XVCL: XML-based Variant Configuration Language. Proceedings of International Conference on Software Engineering (ICSE) (2003)
17. Reppy, J., Turon, A.: Metaprogramming with Traits. Proceedings of the European Conference on Object-Oriented Programming (ECOOP) (July 2007)
18. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-oriented Implementation Technique for Refinements and Collaboration-based Designs. ACM Trans. Softw. Eng. Methodol. **11**(2) (2002) 215–255

19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented Programming. Proceedings of European Conference on Object-Oriented Programming (ECOOP) (1997) 220–242
20. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: Overview of CaesarJ. Transactions on Aspect-Oriented Software Development (February 2006)
21. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering (2004) 355–371
22. Kästner, C., Trujillo, S., Apel, S.: Visualizing Software Product Line Variabilities in Source Code. (ViSPLE 2008), Limerick, Ireland (September 2008)
23. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), New York, NY, USA, ACM (2008) 943–944
24. Heidenreich, F., Şavga, I., Wende, C.: On Controlled Visualisations in Software Product Line Engineering. In: ViSPLE 2008. (September 2008)
25. Bravenboer, M.: Exercises in Free Syntax. Syntax Definition, Parsing, and Assimilation of Language Conglomerates. PhD thesis, Utrecht University, Utrecht, The Netherlands (2008)
26. Bork, M., Geiger, L., Schneider, C., Zündorf, A.: Towards Roundtrip Engineering—A Template-Based Reverse Engineering Approach. In: Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008). Volume 5095 of LNCS., Springer (2008)
27. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008). Volume 5095 of LNCS., Springer (2008)
28. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE 2006), ACM (2006)
29. Efftinge, S., Völter, M.: oAW xText: A framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit. (2006)
30. Object Management Group: MOF Models to Text Transformation Language Final Adopted Specification. ptc/2006-11-01 (2006)
31. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: The Epsilon Generation Language. In: Proceedings of the 4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA 2008). Volume 5095 of LNCS., Springer (2008)
32. Angyal, L., Lengyel, L., Charaf, H.: A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering. In: 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS 2008), IEEE Computer Society (2008) 463–472