Model-driven Modernisation of Java Programs with JaMoPP

Florian Heidenreich, Jendrik Johannes, Jan Reimann, Mirko Seifert,
Christian Wende, Christian Werner, Claas Wilke, Uwe Assmann
Technische Universität Dresden
D-01062, Dresden, Germany
Email: firstname.lastname@tu-dresden.de

***Abstract*—The history of all programming languages exposes the introduction of new language features. In the case of Java—a widespread general purpose language—multiple language extensions were applied over the last years and new ones are planned for the future. Often, such language extensions provide means to replace complex constructs with more compact ones. To benefit from new language extensions for large bodies of existing source code, a technique is required that performs the modernisation of existing source code automatically.**

In this paper we demonstrate, how Java programs can be automatically migrated to new versions of the Java language. Using JaMoPP, a tool that can create models from Java source code, we enable the application of model transformations to perform model-driven modernisation of Java programs. Our approach is evaluated by applying two concrete transformations to large open source projects. First, we migrate classical for loops to the new for-each style (introduced in Java 5). Second, we convert anonymous classes to closures (planned for Java 8). Furthermore, we discuss how tracing transformations allows to quantify the impact of planned extensions.

I. Introduction

Programming languages evolve over time: new features are added and occasionally old ones are removed. A prominent example of a language that undergoes such an evolution is Java. For example, generics were introduced in Java 5.

All changes—with a few exceptions—that were introduced to Java, preserved backward compatibility. Programs written in older versions do still compile and run with new versions. Still, old programs could be updated using new language features, assuming this improves code readability and therewith maintainability. For small programs, old code fragments can be replaced manually, but for large code bases automatic code modernisation transformations are required.

Source code transformations are known for quite a while and specialised tools exist to perform this task (cf. Sect. V). However, with the advent of Model-Driven Software Development (MDSD) [1], standardised transformation languages (e.g., Query View Transformation (QVT)[1]) became available. If one could use these languages for code transformation, the need for specialised languages would vanish.

To apply a model transformation language to source code, a model of the respective code is required. Also, a metamodel of the language that is subject to transformation is needed. In earlier work, we presented JaMoPP [2], [3]—the Java Model Printer and Parser—a tool that entails a complete metamodel for Java and tooling to convert Java source code to Eclipse Modeling Framework (EMF) [4] models and vice versa. Therefore, JaMoPP enables arbitrary EMF-based tools to work on Java programs.

In this paper, we show how JaMoPP and a standardised model transformation language can be combined to migrate Java code to a new version of the Java language. We present two concrete migration examples. First, existing for loops are transformed to the for-each style that was introduced in Java 5. Second, the conversion of anonymous inner classes to closures, which are planned for the Java 8 release, is performed. We apply the two transformations to a set of large open source Java projects. The results of this transformation can be used to quantify the impact of new language features.

The paper is structured as follows. After giving a brief overview on JaMoPP in Sect. II, we discuss the transformations for the two migration examples in Sect. III. The result of performing the transformations of larger bodies of source code can be found in Sect. IV. We compare our work with related approaches in Sect. V, and conclude in Sect. VI.

II. JaMoPP—Brief Overview

In MDSD, many generic modelling tools exist that can be used in combination with arbitrary languages. This is possible because the tools can be configured with metamodels. A metamodel describes the concepts of a language; also referred to as the abstract syntax of a language. To exchange metamodels between tools, the OMG has standardised the metamodelling languages Meta-Object Facility (MOF) and Essential Meta-Object Facility (EMOF)[2]. A widely used implementation of EMOF is Ecore as part of EMF. To use a generic modelling tool that supports Ecore with a certain language, a metamodel of that language needs to be provided together with tooling to parse (and print) sentences written in the language's concrete syntax (e.g., a Java program) into typed graphs that conform to the metamodel.

With JaMoPP, we provide such an Ecore metamodel and the tooling to parse and print source code for the Java language (currently supporting Java 5). This allows

[1]http://www.omg.org/spec/QVT/

[2]http://www.omg.org/spec/MOF/2.0

developers to apply generic modelling tools on Java source code and hence to use the same tools to work with models (e.g., UML models) and source code. An example of such a tool, which we show in the next section, is the QVT transformation language.

Important properties of JaMoPP are: (1) JaMoPP supports both parsing and printing Java code which allows modelling tools (e.g., model transformation engines) to both read and modify Java source code. This conversion preserves the layout of Java source code. (2) In addition to parsing, JaMoPP performs name and type analysis of the source code and establishes links (e.g., between the usage and definition of a Java class). These links can be exploited by modelling tools to ensure correctness of static semantics properties of the Java source files they generate or modify. (3) JaMoPP itself was developed using our modelling tool EMFText [5]. There, the concrete syntax of Java is defined in an EBNF-like syntax definition. Based on the metamodel and this definition, the parser and printer tooling is generated. This allows us to extend JaMoPP by extending the metamodel and the syntax definition without the need to modify code. With this, JaMoPP can co-evolve with future Java versions and can, in particular, be used to prototype and experiment with new features. An example of this is closure support, which is used in Sect. IV.

JaMoPP has been tested with a large body of source code from open-source Java projects through which stability and support for all Java 5 language features is assured (see [2] for details). Initially [2], we focused on using JaMoPP for forward engineering to generate and compose Java code. In [3] we presented how JaMoPP is used for reverse engineering. In the next section, we demonstrate how JaMoPP is used in combination with QVT for modernisation, which is a combination of reverse and forward engineering.

III. Model-driven Source Code Transformation

In this section we first exemplify Model-Driven Modernisation using our first migration example—the transformation of for loops to the for-each loops. Afterwards we discuss the benefits and challenges we experienced in applying model transformations for source code modernisation in both migration examples (for-each loops and closures). The complete transformation scripts can be found online[3].

To implement our modernisation transformations we used QVT-Operational provided by the Eclipse M2M project[4]. We consider the declarative language QVT-Operational a pragmatic choice for the unidirectional transformations typically required for source-code modernisation. In contrast, its declarative counterparts (QVT-Relations, QVT-Core) are more suitable for bi-directional transformation.

[3]http://jamopp.org/index.php/JaMoPP_Applications_Modernisation/
[4]http://www.eclipse.org/m2m/

```
 1 mapping transformForLoopToForeachLoop
 2   (forLoop : JAVA::statements::ForLoop)
 3   : JAVA::statements::ForEachLoop
 4   when {
 5     forLoop.checkLoopInit() and
 6     forLoop.checkLoopCondition() and
 7     forLoop.checkLoopCountingExpression() and
 8     forLoop.checkLoopStatements()
 9   }{
10   var listType : JAVA::types::TypeReference
11     := getIteratedCollectionType(counterIdentifier);
12   var loopParameter
13     := object JAVA::parameters::OrdinaryParameter {
14       name := "element";
15       typeReference := listType };
16   result.next := loopParameter;
17   result.statement :=
18     map replaceCollectionAccessorStatements(
19       forLoop, loopParameter);
20 }
```

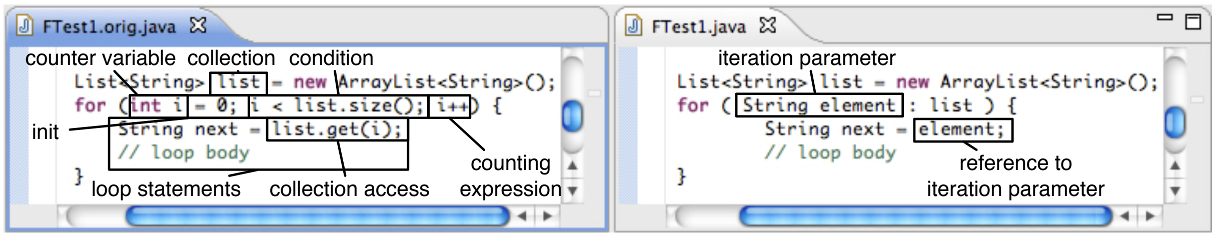
Listing 1. QVT Transformation to replace for loops with for-each loops.

A. Example of Model-Driven Modernisation of For Loops

Listing 1 shows a mapping taken from the transformation of for loops to for-each loops. An example of the expected replacement and a description of the loop elements used in the transformation is given in Fig. 1. The mapping consists of a when-clause (lines 4-9) and a mapping body (lines 10-20). The when-clause defines a number of preconditions that need to be satisfied by a given for loop to be replaced. For instance, that the `init`-statement initialises the loop `counter variable` with `0`, that the loop `condition` ensures iteration among all collection elements, or that the `counting expression` always increases the loop `counter variable` by 1. Furthermore, the `loop statements` are not allowed to refer to the `counter variable` aside from accessing the collection for the next element. If all preconditions are satisfied we calculate the generic type of the collection that is iterated (lines 10-11) and create an `iteration parameter` for the for-each loop that is initialised with this type (line 12-15). Finally, the new for-each loop is initialised with this parameter and its body is filled with the statements of the original for loop, where all statements for `collection access` are replaced with `references to the iteration parameter` of the for-each loop.

B. Applicability of QVT for Model-Driven Modernisation

For the specification of both transformation scripts we used a set of tools provided by the M2M project for QVT-Operational. The included editor provided advanced editing features like syntax highlighting, code navigation, and code completion. Especially code completion helped a lot in writing expressions that traverse and analyse Java models. A second tool that helped a lot in developing transformations was the QVT debugger. It allows the step-wise evaluation of transformation execution and was indispensable to understand and fix problems in our complex transformation scripts. Third, the QVT interpreter generates



```java
List<String> list = new ArrayList<String>();
for (int i = 0; i < list.size(); i++) {
    String next = list.get(i);
    // loop body
}
```

```java
List<String> list = new ArrayList<String>();
for ( String element : list ) {
    String next = element;
    // loop body
}
```

Figure 1. Example of for loop replacement explaining the elements of for- and for-each loops.

tracing information for each execution of the transformation. The trace records all mappings applied and enabled the quantitative analysis of our examples. We think that the reusability and maturity of these generic tools provide some good arguments for applying a standardised transformation language.

A benefit of model-driven modernisation was the graph-structure of models, which, compared to tree-structures often provided by code parsing tools, allow for more convenient navigation and analysis of references between declarations and uses of code elements (e.g., variables). For example, this eased the specification of the preconditions for the for loop migration that searches the method body for statements that use the counter variable declared in the loop header.

In both examples it is not trivial to come up with an exhaustive set of patterns that identify source code that can be modernised. We consider this a challenge for source code modernisation in general. However, we also learned that some idioms (like the for loop presented in Fig. 1) are quite common and occurred in all Java projects we investigated, as can be seen in the next section.

Some drawback of using model transformations for code modernisation was the focus on abstract syntax, i.e., the language metamodel. It required a good knowledge of the JaMoPP metamodel and some training to represent patterns of source code in abstract syntax. On the other hand, the strict structure of an explicit metamodel was beneficial to ensure the well-formedness of the produced source code.

IV. Evaluation

To evaluate the performance of our source code modernisation approach, we applied the transformations from Sect. III to a set of Java frameworks available to the public. Our goal was to answer the following questions: First, we wanted to know whether a general purpose modelling environment like EMF is scalable enough to handle such a large set of models. Second, we were curious how many resources are required to perform a transformation of this scale with QVT—a generic model transformation language. To answer these questions, we transformed 16.402 Java files from 10 open source projects.

A. *Performance*

To evaluate the transformation performance, we measured the time needed to perform the transformations on individual compilation units. This includes all types referenced by this unit, but excludes other, unrelated parts of the source code. We used a machine with a Dual Core AMD Opteron running at 2.4 GHz with 4 GB RAM. We used only one core of the machine to avoid problems with Eclipse plug-ins that are not thread-safe.

Framework	Files	For loops		Closures	
		min	repl./occur.	min	repl./occur.
AndroMDA 3.3	698	3	4/959	8	8/201
Apache Ant 1.8.1	829	5	24/1028	21	12/99
Comns. Math 1.2	395	1	25/845	5	0/25
Tomcat 6.0.18	1127	4	65/1437	15	52/125
GWT 1.5.3	1850	5	29/1044	23	26/670
JBoss 5.0.0.GA	6414	16	472/2744	70	197/591
Mantissa 7.2	242	1	7/652	3	18/29
Spring 3.0.0.M1	3096	8	82/680	43	31/1403
Struts 2.1.6	1035	3	7/130	13	8/158
XercesJ 2.9.1	716	3	21/1111	12	62/94
	16402	49	736/10630	213	414/3395

Figure 2. Transformation time and ratio of found and replaced elements.

The results of our measurements are shown in Fig. 2. From these numbers one can derive that the pure average transformation time per source file is 0.2 seconds (for-each loops) and 0.8 seconds (closures). These values can be obtained by dividing the total time (given in minutes in columns 3 and 5) by the number of total source files. In addition to the transformation time one must also take into account the time needed to load the input models. This can take up to a few minutes for very complex source files, but is usually done within few seconds. For a migration task, which is performed once for every new release of a programming language, this renders the approach still feasible.

B. *Quantification of Language Extensions*

To quantify the impact of a planned language extension, one can count the number of replaceable language constructs. This is usually only a subset of all cases where a new language construct is applicable. Some potential applications of a language construct may simply not be detected, because developers used structures not covered in the transformation script.

Nonetheless, the number of places in existing source code where a new language construct can be applied, does at least give some indication about its impact. The ratio for loops that can be replaced by for-each loops to all for loops found, and the ratio of anonymous inner classes that are replaceable by closures to all inner classes found is shown in Fig. 2.

On average, 6.9% of all for loops were transformed to the for-each style. The percentage of anonymous classes that were replaced by closures was 12.2%. Certainly, these numbers can be increased if more patterns are covered by the transformation scripts. However, given the very restrictive scripts which we used, the numbers are quite high. Thus, one can reason that actual benefit can be gained by using automatic transformations here and that both language extensions are useful additions to the Java language.

V. Related Work

There exists a large amount of work and tools for source code transformation.[5] One prominent approach is Stratego/XT [6], a tool set for strategic rewriting of source code. While Stratego/XT and other approaches are proved to be useful and applicable in academic and industrial projects, they do not provide a standardised transformation language. With JaMoPP one can transform source code to a standardised intermediate representation (i.e., EMF-based models) where we apply a standardised transformation language (i.e., QVT). This can be generalised to other programming languages and other transformation languages.

MoDisco [7] aims at discovering models from legacy applications. It handles Java code as well as other artefacts (e.g., configuration files). Based on the Eclipse JDT parser, MoDisco creates models from source code files. The metamodel for these models is defined in Ecore similar to JaMoPP. Thus, transformations can be specified using existing languages (e.g., QVT). However, MoDisco does not preserve the layout of the source code when printing back transformed models back to their original representation (i.e., Java source code). Approaches for metamodel evolution (e.g., [8]) are inherently limited to the model level and do not allow to print models after performing evolution steps.

Architecture-driven Modernisation (ADM)[6] of the OMG goes beyond what is presented in this paper by applying modernisation efforts on all artefacts involved in the software development process (e.g., source code, database definitions, configuration files, ...). However, strategies that are built on top of existing OMG standards (e.g., similar to the combination of EMOF and QVT in our approach) can fit nicely in the overall goal of ADM.

VI. Conclusion & Future Work

In this paper, we implemented and evaluated two examples of Java modernisation to show that JaMoPP in combination with the standardised model transformation language QVT can be used for Model-Driven Modernisation of Java programs. In applying the transformations on a set of open-source projects we experienced that the transformations can be performed in acceptable time and transformed a reasonable part of the code (on average, 6.9% of all for loops and 12.2% of all anonymous classes were considered as candidates for transformation). So far we only used our own judgement based on our experience with Java to determine the semantic correctness of the transformation rules. In future we plan to automatically validate correctness by running the test suites of the open-source projects on the modernised code. Further, we did not yet compare the effort of writing QVT transformations for Java with alternatives such as using Java-specific source code transformation tools or other model transformation languages. Doing this comparison by implementing the two modernisation transformations with different languages and tools is subject to future work.

[5]http://www.program-transformation.org/ provides an overview.
[6]http://adm.omg.org/

Acknowledgment

This research is co-funded by the European Commission within the projects MODELPLEX #034081 and MOST #216691, by the German Ministry of Education and Research (BMBF) within the projects feasiPLe and SuReal; by the German Research Foundation (DFG) within the project HyperAdapt and by the European Social Fund and Federal State of Saxony within the project ZESSY #080951806.

References

[1] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development*. John Wiley & Sons, 2006.

[2] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, "Closing the Gap between Modelling and Java," in *Proc. of 2nd Int'l Conf. on Software Language Engineering (SLE'09)*, ser. LNCS, vol. 5969. Springer, Mar. 2010, pp. 374–383.

[3] ——, "Construct to Reconstruct—Reverse Engineering Java Code with JaMoPP," in *Proc. of Int'l Workshop on Reverse Engineering Models from Software Artifacts (R.E.M. 2009)*, Oct. 2009.

[4] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *Eclipse Modeling Framework (2nd Edition)*. Pearson Education, 2009.

[5] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende, "Derivation and Refinement of Textual Syntax for Models," in *Proc. of ECMDA-FA'09*, ser. LNCS, vol. 5562. Springer, Jun. 2009, pp. 114–129.

[6] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," in *Domain-Specific Program Generation*, ser. LNCS, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Spinger, Jun. 2004, vol. 3016, pp. 216–238.

[7] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering," in *Proc. of ASE'10*. ACM, 2010, pp. 173–174.

[8] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "COPE - Automating Coupled Evolution of Metamodels and Models," in *Proc. of ECOOP'09*, ser. LNCS, S. Drossopoulou, Ed., vol. 5653. Springer, 2009, pp. 52–76.