JaTS: A Java Transformation System

Fernando Castor Kellen Oliveira Adeline Souza
Gustavo Santos Paulo Borba*
Centro de Informática
Universidade Federal de Pernambuco

Abstract

In this paper we present JaTS, a transformation system for the Java programming language. The system has been designed with the goal of making the process of creating and applying transformations as agile as possible. Also, it features a transformation language which has a syntax very similar to that of Java, decreasing the semantic gap between the transformation language and the language being transformed. JaTS may be used to specify and apply a wide range of transformations, including refactorings, refinement laws and code generation.

1 Introduction

Program transformation is an important techinque for supporting software engineering activities: refactoring [5], formal software development [1, 3], language translation [4] and code generation [4]. Its use in practical, large scale, projects, however, is not possible without automation. Tool support is vital to the application of program transformations, as it increases productivity and eliminates the danger of introducing new errors.

Several program transformation tools have been implemented. Many of these are not language-specific [4], being able to transform programs from an arbitrary source language to an arbitrary destination language. Although this may be an advantage, it complicates the use of the tools, since they require two kinds of user: the transformation engineer, who configures the tool (encodes the transformations) and the programmer, who uses the tool for software development (applies the transformations). This is usually necessary because, in most cases, the language in which the transformations are encoded is substantially different from the one to which they are applied.

There are also language-specific tools for program transformation [3, 7]. Most of these have the drawback of supporting only a fixed set of built-in transformations. For instance, refactoring systems [7] usually implement a few simple refactorings that can be applied, but a programmer cannot add a new refactoring, unless he has access to the source code of the system. Formal software development systems [3] are usually similar: they support a set of built-in refinement laws that cannot be extended.

*Supported in part by CNPq, grant 301021/95-3. Email: {fjclf,adss,phmb}@cin.ufpe.br. WWW: http://www.cin.ufpe.br/~{fjclf,adss,phmb}. Address: Caixa Postal 7851, Recife, PE, Brazil.

In this context, we present JaTS, a transformation system for the Java [6] programming language. JaTS is able to store and apply transformations written in a language that is a superset of Java, eliminating thus the semantic gap between the transformation language and the language being transformed. Also, the transformation language of the system takes the semantics of Java into account, making it possible to specify transformations that could not be specified in conventional transformation languages. JaTS has been designed as a generic transformation system, that is, it has not been devised with the aim of applying a specific kind of transformation. Instead, it is intended to apply many transformations of different types.

2 Functionalities

The transformation language of JaTS consists of Java extended with JaTS constructions. The goal of these constructions is to allow type (a class or interface) matching and the specification of the new types that are to be generated. From now on, we will use the name JaTS interchangeably for the transformation system and its transformation language.

JaTS transformations consist of three parts: a precondition, a left-hand and a right-hand side. Both sides consist of one or more type (class or interface) declarations written in JaTS. Hereafter, however, we consider transformations having only one type declaration on each side. The type declarations in the left-hand side are matched with the source Java type declarations to be transformed, what implies that both must have similar syntactic structures. The right-hand side defines the general structure of the types that will be produced by the transformation.

2.1 Features of the Transformation Language

The simplest construction of the JaTS transformation language is the JaTS variable, which consists of a Java identifier preceded by the '#' character. JaTS variables are used as placeholders in the transformations. When the left-hand side of a transformation is matched with the source Java type, the result yielded is a set of mappings from variables to Java constructions. This set is called the *result map* of the matching. Roughly, a construction in the source Java type matches another one in the left-hand side of a transformation if they are identical or if the second one corresponds to a JaTS variable. JaTS variables are also present in the right-hand side of transformations, where they are replaced by the values mapped to them in the result map of the matching.

Variables can be declared as being of a specific type. This is necessary because there are situations where it is not possible to determine the intended matching for a certain variable. The type of a JaTS variable refers to the kind of Java construction it should be matched with (an identifier, a name, a list of names, etc.).

Variables do not need to be matched exclusively with simple structures, like identifiers and lists of names. A variable can also be matched with complex constructions like a method, field or constructor declaration. In this kind of matching, the variables must be typed, so that the kind of construction the variables are to be matched with can be determined. So, if we want a certain variable, **#attr**, to be matched with a field declaration, **#attr** must be declared as being of type **FieldDeclaration**. Likewise, it is possible to match a JaTS variable with a set of declarations of the same type (a set of methods, a set of fields, etc.).

Sometimes, when specifying a transformation, only part of the information obtained from the source Java types is necessary to specify the resulting Java types. For example, it might be useful to know the type of a certain field declaration, but the field declaration itself will not be in the resulting type. JATS addresses this problem by means of two features: *Executable Declarations* and *Iterative Declarations.* Executable declarations extract and modify the information mapped to variables through the execution of Java code. For example, in the transformation

Left – Hand Side	**Right – Hand Side**

`class` #**C** `extends` #**SC** {
 #**ATTR**`:FieldDeclaration;`
 }
	`class` #**C** `extends` #**SC** {
	 `private [[` #**ATTR**`.getType() ]] fd;`
	 #**ATTR**`:FieldDeclaration;`
	 }

The construction `[[` #**ATTR**`.getType() ]]` is an executable declaration and the expression enclosed within the '[[' and ']]' is a simple Java method invocation. The method `getType()` is invoked from the value mapped to the variable #**ATTR**, which consists of a Java object representing a field declaration, and returns the type of that field declaration. This transformation applied to the class

```
class ConcreteTest extends AbstractTest {
    private int code;
}
```

results in the following Java type:

```
class ConcreteTest extends AbstractTest {
    private int fd1;
    private int code;
}
```

Iterative declarations are used for specifying transformations that generate sets of declarations with the same pattern but differing on specific information obtained from a set of declarations in the source Java type. Both, executable declarations and iterative declaration, can only appear in the right-hand side of a transformation.

Some transformations can only be applied if certain preconditions hold. These preconditions are essential components of refactorings and refinement laws, but are also useful for a wide range of program transformations. In JATS, most of the arithmetic, logical, relational and conditional operators of Java can be used to specify preconditions, except for those that need an environment, like '=', '+=' and '++', postifx and prefix. Also, some precondition-specific JATS constructions are allowed.

The transformation language of JATS takes the semantics of Java into account, for applying transformations. By "takes the semantics of Java into account", we mean that JATS does not treat Java structures in a type declaration as simple textual patterns. Their meanings are also taken into account. This feature allows the specification of transformations that could not be specified if only the syntax was taken into account. For example, the matching and replacement of declarations and sets of declarations are not supported by traditional (syntactic-only) transformation languages [4]. A through evaluation of the advantages and disadvantages of JaTS when compared to other transformation systems has been presented elsewhere [2].

2.2 The JaTS Work Environment

JaTS' Work Environment (JWE) has been designed with the goal of making the specification, visualization and application of transformations as easy as possible.

In JWE, transformations are grouped in projects, which consist of sets of related JaTS and Java types. The JaTS types in a project are grouped as left-hand and right-hand sides. To JWE, a transformation is simply an association between left-hand and right-hand sides and is created by selecting the corresponding JaTS types. This way, transformations that have JaTS types in common in the left or right-hand sides may be easily created.

Transformations are presented, in JWE, by means of two panels: one for the types corresponding to left-hand sides of transformations and another one corresponding to the right-hand sides. This division aims to make the visualization of transformations as intuitive as possible, as left-hand side JaTS types are exhibited on the left-hand side panel and right-hand side types on the right-hand side panel. The source and generated Java types are presented similarly. Above each panel is a list of the types belonging to the current project. The type presented in a panel is the one that is selected in the corresponding list.

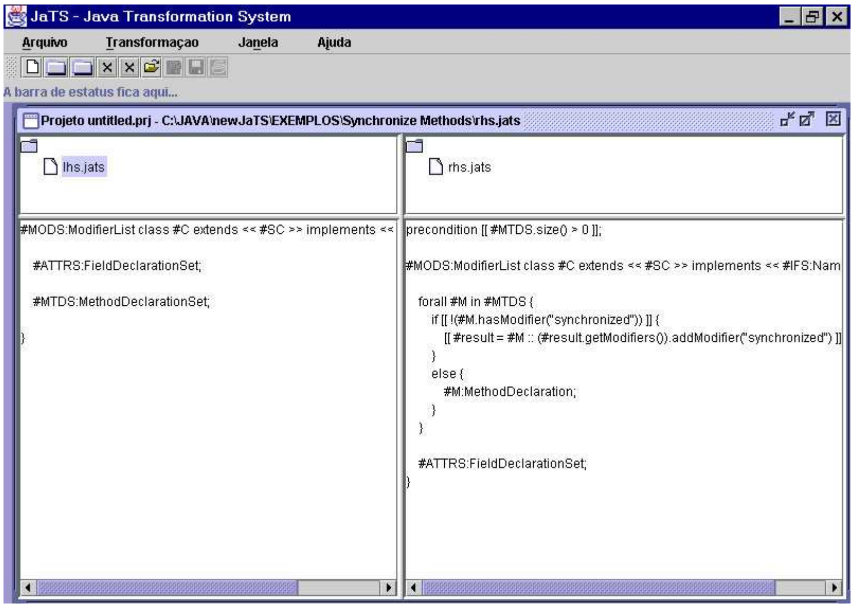

Figure 1: The User Interface of JaTS' Work Environment.

JaTS transformations are applied in the following steps: first, the user choses the types corresponding to the left and right-hand sides of the transformation and selects the 'Store Transformation' option in the 'Transformation' menu. Then, the user selects the Java types to be transformed. The number of Java types selected must be equal to the number of JaTS types in the left-hand side. If this condition does not hold, the application of the transformation fails automatically. After the source types have been chosen, the user selects the 'Apply Transformation' option in the 'Transformation' menu. If the transformation is successfully applied, a number of Java types equal to the number of types in the right-hand side of the transformation is generated. Otherwise, the user is notified that an error has ocurred.

3 Implementation

JaTS has been implemented using the Java Development Kit v1.3. The parser for the transformation language has been constructed with the parser generator JavaCC[8]. JavaCC also has a parse-tree generator called jjTree. We have chosen not to use it, though, because syntax-trees generated by it tend to have a large number of unecessary nodes, specially for inherently recursive grammars like the one for Java expressions. Instead, we generate the parse-trees by means of the semantic actions of the parser.

The arquitecture of JaTS is composed by three main parts: JaTS IO, JaTS Engine and the user interface.

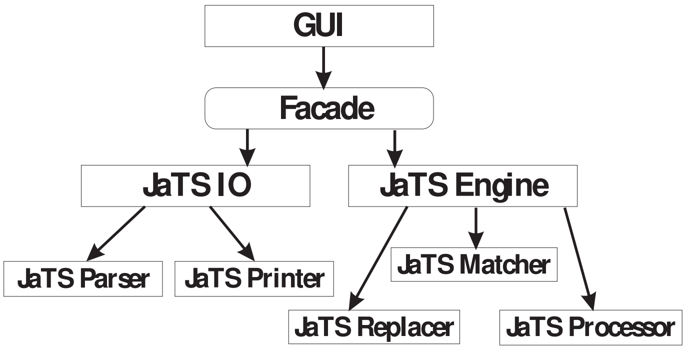

Figure 2: The JaTS Architecture.

JaTS IO is responsible for the parsing of the Java and JaTS types supplied as input and for the generation of the corresponding parse-trees. Also, JaTS IO has the task of doing the pretty printing of the parse trees of the Java types produced by a transformation. Correspondingly, JaTS IO is subdivided in two subsystems: the JaTS Parser, which does the parsing and parse-tree generation, and the JaTS Printer, which does the pretty printing.

The JaTS Engine subsytem applies a transformation to a Java type. In order for a transformation to be applied, the following information must be provided:

1. The syntax-trees of the JaTS types corresponding to the left and right-hand sides of the transformation.

2. The syntax-tree of the Java type being transformed.

3. A (possibly empty) result map containing mappings from variables to nodes.

4. A precondition (optional).

The JaTS Engine is composed by three subsystems: the JaTS Matcher, the JaTS Replacer and the JaTS Processor.

The JaTS Matcher tries to match the parse-tree of the left-hand side of the transformation with the parse-tree of the source Java type. The JaTS Replacer receives as input the parse-tree of the right-hand side of the transformation and the result map of the matching. Its task is to transverse the parse-tree and replace occurrences of variables by the values mapped to them in the result map. Finally, the JATSPROCESSOR is responsible for the processing of the executable and iterative declarations.

4 Conclusion

In this paper we introduced JATS, a transformation system for the Java programming language. We presented the main features of its transformation language, described its work environment and delineated some key aspects of its implementation. The current prototype of the system implements all the functionalities presented. It may be used to write, store and apply transformations. It is restricted, though, to a subset of Java consisting of its declarations and expressions. Statements are not supported yet.

The transformation language of JATS has a simple syntax, not far from Java, the language being transformed, eliminating thus, the need for the transformation engineer. That makes it easier for the user to implement and alter transformations, increasing productivity and reducing the probability of errors being introduced. Also, it is expressive enough to specify several different types of transformations.

Acknowledgements

We would like to thank the anonymous referees, who help improve this paper.

References

[1] Paulo Borba and Augusto Sampaio. The basic laws of rool: An object oriented language. *Revista Brasileira de Informática Teórica e Aplicada*, 7(1), sep 2000.

[2] Fernando Castor. *Definição de uma Linguagem para Especificar Transformações em Java*. Universidade Federal de Pernambuco, 2001. Graduate work. Avaliable for download at http://www.cin.ufpe.br/~fjclf/jats.

[3] Sandrelly Coutinho, Tiago Reis, and Ana Lúcia Cavalcanti. Uma ferramenta educacional de refinamentos. In *XIII Simpósio Brasileiro de Engenharia de Software - Sessão de Ferramentas*, pages 61–64, Florianópolis, Brazil, 1999.

[4] Marcelo F. Felix and Edward H. Hausler. LET:uma linguagem para especificar transformações. In *III Simpósio Brasileiro de Linguagens de Programação*, pages 109–123, Florianópolis, Brazil, may 1999.

[5] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[7] Instantiations, Inc. *The jFactor White Paper*, 2000. Avaliable for download at http://www.instantiations.com/jfactor/docs/default.htm.

[8] Sriram Sankar. *The JavaCC Documentation*. Metamata, Inc., 1998. Avaliable for download at http://www.metamata.com/javacc.