A Language for Specifying Java Transformations

Fernando Castor and Paulo Borba*
Centro de Informática
Universidade Federal de Pernambuco

Abstract

In this paper we present JaTS, a language for specifying transformations for Java programs. The main feature of the language is the similarity of its syntax to the syntax of Java, decreasing the semantic gap between the transformation language and the language being transformed. This feature helps developers already used to the syntax of Java to quickly adapt to the syntax of JaTS. We begin by presenting the syntax and informal semantics of JaTS, specifying some key aspects of the semantics formally. Next, we evaluate the expressive power of the language, comparing it with other languages for specifying program transformations.

Resumo

Neste trabalho nós apresentamos JaTS, uma linguagem para especificar transformações para programas escritos em Java. A principal característica dessa linguagem é a proximidade da sua sintaxe com a de Java, diminuindo o *gap* semântico entre a linguagem de transformação e a linguagem transformada. Essa característica proporciona a desenvolvedores já habituados com a sintaxe de Java uma rápida adaptação à sintaxe de JaTS. Começamos apresentando a sintaxe e a semântica informal de JaTS, especificando alguns aspectos chave dessa semântica formalmente. Em seguida, avaliamos o poder expressivo da linguagem, comparando-a com outras linguagens para especificar transformações.

1. Introduction

Program transformation is a powerful technique for supporting software engineering activities: refactoring [9, 16], formal software development [4, 5], code generation and language translation [8].

In fact, refactoring [9, 16], the process of restructuring code with the purpose of making it easier to understand and maintain without changing its observable behavior, is strongly coupled with program transformation. Refactorings can be specified as parameterized program transformations that obey behavior-preserving preconditions [17]. Nowadays, the use of refactoring to increase quality is considered a very important development practice. For instance, Extreme Programming [2], a recent approach for software development, recommends the use of refactoring as a continuous activity, intimately related to coding.

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

Formal and rigorous software development methods are strong candidates to the use of program transformation as well, since refinement laws can be easily described as program transformations satisfying preconditions that guarantee the soundness of the refinement. This is the case for imperative languages, as in the refinement calculus [14], but also for object-oriented languages [4, 5]. In both cases, algebraic laws for programming languages can be viewed and implemented as program transformations.

Those applications of program transformation show its importance, but its use in practical, large scale, projects is not possible without automation. Tool support is vital to the application of program transformations, in order to provide productivity and eliminate the danger of introducing errors, when performing such a tedious and demanding task. Several program transformation tools have been implemented. Many of these are not language-specific [3, 8], 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. Most of these have the drawback of supporting only a fixed set of built-in transformations. For instance, refactoring systems [12, 16] usually implement a few simple refactorings that can be applied, but a programmer cannot add a new refactoring to this tool, unless he has access to the source code of the system. Formal software development systems are usually similar: they support a set of built-in refinement laws that cannot be extended.

In order to avoid the drawbacks of the general purpose and language-specific transformation tools, we present a language to specify program transformations for the Java programming language [11]. In this language, transformations are specified using a superset of Java, making it easier for programmers to specify the transformations they wish to apply and thus eliminating the need for the transformation engineer. Also, it takes the semantics of Java into account, making it possible to implement transformations that could not be implemented if only the syntax was taken into account. The language has been named JaTS, an acronym for Java Transformation System, the system that actually implements this language, stores and applies transformations to Java programs. We concentrate here on the definition of the language: its syntax, semantics and pragmatics.

This paper is organized as follows. First we present an informal description of the syntax and semantics of JaTS through simple but clarifying examples; we progressively introduce the constructs of JaTS, showing an increasingly more complex transformation. Next we formally specify key aspects of the language using Action Semantics [15, 18]; several aspects are omitted, but are described elsewhere [6]. In Section 4, we evaluate JaTS by comparing its expressive power with that of another transformation system. Finally, we state some concluding remarks.

2. Introducing JaTS

JaTS transformations are written in a language that extends Java with the 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. The simplest among these constructions is the JaTS variable, which consists of a Java identifier preceded by the '#' character.

A JaTS transformation consists of three parts: a precondition, a left-hand and a right-hand side. Both sides consist of one or more type declarations written in JaTS. Hereafter, however, we consider type declarations having only one type declaration on both sides. The left-hand side of a transformation is matched with the source Java type being transformed, what implies that both must have similar syntactic structures. The right-hand side defines the type that will be produced by the transformation.

The application of a JaTS transformation to a Java type is performed in three phases: parsing, transformation and unparsing. The second phase, transformation, can be divided in three steps: matching, replacing and execution. The first matches the parse tree of the left-hand side of the transformation with the parse tree of the source Java type being transformed. Roughly, a node in the source type matches the one in the left-hand side if they are identical or if the second one corresponds to a JaTS variable. A mapping from variables to the values that they were matched with is produced by the matching. This is called the *result map* of the matching. The second step consists of replacing occurrences of JaTS variables in the parse tree of the right-hand side by the corresponding values in the result map. The last step consists of executing some JaTS structures in the parse tree of the right-hand side of the transformation. These executable structures are described later.

2.1 *JaTS Variables*

As mentioned earlier, the most common construction in a JaTS type is the JaTS variable. Variables are used as placeholders in the transformations. For example, in the following transformation:

Left-hand side:	**Right-hand side:**
`class #C extends Object { }`	`class #C { }`

The variable `#C` is used as a placeholder for a class name, in such a way that the left-hand side of the transformation matches with any empty class that explicitly extends the `Object` class, but implements no interfaces. When applied to such an empty class, this transformation yields a similar class with the same name as the original one, except that it does not extend `Object` explicitly. For example, the application of this transformation to the following Java class:

```
class Person extends Object { }
```

produces, as result, the Java class

```
class Person { }
```

Variables can be declared as having a specific type, corresponding to one of the syntactic constructions of Java. In many cases, as in the previous example, this is not necessary since the variable appears in a place that leaves no doubts about the kind of structure it is going to be matched with. There are some cases, however, where the user must declare the type of some variables in order to correctly specify the intended semantics of the transformation. For example, in the transformation

Left-hand side: `class #C extends Object implements #if:Name { }`
Right-hand side: `class #C implements #if:Name { }`

the variable `#if` has been declared as having type `Name`. This means that `#if` should be matched with a single interface name, not with a list of interface names, after the `implements` clause of the source Java type. So, the left-hand side would not match with the following Java class:

```
class Person extends Object implements Runnable, Clonable { }
```

Since this class implements two interfaces. In order to match with this class, we should instead have a transformation with the following left-hand side:

```
class #C extends Object implements #ifs:NameList { }
```

2.2 *Optional Matching and Replacement*

Another useful JaTS construction allows us to specify that the matching of a certain structure is optional. If a structure in the left-hand side of the transformation is optional, it will be matched with the corresponding declaration in the source Java type, if there is one. Otherwise, the optional part will simply be ignored by the matching process. For example, in the transformation

Left-hand side: `class #C extends Object <implements #ifs:NameList> { }`
Right-hand side: `class #C <implements #ifs:NameList> { }`

the `implements` clause of the left-hand side of the transformation is declared optional. That is indicated by the '`<`' and '`>`' enclosing it. So, the left-hand side of this transformation can be matched with both of the following classes:

```
class Person extends Object { }
```

and

```
class Person extends Object implements Runnable, Clonable { }
```

Optional structures may appear in the right-hand side of the transformation, as well. For example, the following right-hand side

```
class #C <implements #ifs:NameList> { }
```

specifies that the `implements` clause will only be part of the class produced by the transformation if some value is mapped to `#ifs` in the result map of the matching. Otherwise, the `implements` clause is simply ignored. It is worth mentioning that if all the occurrences of a variable in the left-hand side of the transformation are enclosed in optional structures, the same must be true for the occurrences of that variable in the right-hand side of the transformation.

2.3 *Matching Declarations*

Variables do not need to be matched exclusively with simple structures, like class names and lists of interface names. A variable can also be matched with a whole method, field or constructor declaration. For example, consider the following transformation:

Left-hand side:

```
class #C extends Object <implements #ifs:NameList> {
    #attr:FieldDeclaration;
}
```

Right-hand side:

```
class #C <implements #ifs:NameList> {
    #attr:FieldDeclaration;
}
```

It removes the `extends` clause of a class declaration, as in the previous example, except for the fact that it expects the body of the source class declaration to have exactly one attribute declaration.

In the same way that it is possible to match a variable with a field or method declaration, it is possible to match a variable with a list of declarations of the same type (a set of methods, a set of fields, etc). The following transformation illustrates the declaration of variables of the type `FieldDeclarationSet`.

Left-hand side:

```
class #C extends Object <implements #ifs:NameList> {
    #attrs:FieldDeclarationSet;
    private #type #name;
    #attr:FieldDeclaration;
}
```

Right-hand side:

```
class #C <implements #ifs:NameList> {
    #attr:FieldDeclaration;
    #attrs:FieldDeclarationSet;
}
```

Besides removing the `extends` clause of a class declaration, this transformation removes a private field declared from the body of the source class, as long as this class has at least another field declaration. In fact, the source class may have many more field declarations, since they could be matched to `#attrs`. For example, the type generated by the application of this transformation to the class

```
class Person extends Object implements Runnable, Clonable {
    private String name;
    private Address address;
    private int age;
    private char gender;
}
```

is the following:

```
class Person implements Runnable, Clonable {
    private Address address;
    private int age;
    private char gender;
}
```

since the declarations of the fields `age` and `gender` were matched with `#attrs`, assuming that the user chose `#name` to match with `name` and `#attr` to match with the declaration of the `address` field. In fact, sometimes the user needs to supply arguments indicating how he wishes the matching to proceed, otherwise, the result of applying a transformation cannot be predicted.

Semantically, the result would be non-deterministic in other cases, but, in practice, a particular implementation of JaTS would typically have a default behavior that yields one of the possible results defined by the semantics of JaTS. For example, our implementation of JaTS has the following default behavior: first, field declarations in the left-hand side are matched with field declarations in the source class. In the previous example, "`private #type #name`" is matched with the first compatible field declaration in the source class, "`private String name`"; next, variables of type `FieldDeclaration` are matched with unmatched field declarations. For the previous example, the variable `#attr` is matched with the next unmatched field declaration, "`private Address address`"; finally, if a variable of type `FieldDeclarationSet` is present in the left-hand side of the transformation, the remaining unmatched field declarations in the source class are matched to it. The same procedure applies to the matching of method and constructor declarations.

2.4 *Executable Declarations*

The right-hand side of transformations presented in previous examples contains declarations that appear in the left-hand sides, and maybe some additional fixed declarations. However, it is also useful to have, on the right-hand side, declarations that use information from the original declarations, but are not necessarily identical to them. In order to support this extraction or modification of original declarations, JaTS provides the so called *executable declarations*. They can appear anywhere a variable can, but only on the right-hand side of transformations, usually, executing methods on objects that represent the nodes in the syntax tree being transformed; so, they actually encapsulate Java code that should be executed in order to generate a valid Java declaration or construction.

Executable declarations appear in JaTS transformations enclosed by the "[[" and "]]" symbols. There are two kinds of executable declaration. The first one is called *Information-Extracting Declaration*. For example, the declaration

```
[[ #a.getFieldName(0) ]]
```

yields the name of the field declaration associated to #a, an object returned by the invocation of a method call. The side-effects on the object in which the method is invoked are not important, only the result of the outermost method invocation matters. For example, assume that the JaTS class

```
class Person implements Runnable, Clonable {
    public [[ #a.getFieldType() ]] [[ #a.getFieldName(0) ]]() {
        return [[ #a.getFieldName(0) ]];
    }
}
```

corresponds to the right-hand side of a transformation. Assuming that variable `#a` is mapped to the field declaration "`public String name`", the execution of the executable declarations of this transformation would produce the following class:

```
class Person implements Runnable, Clonable {
    public String name(){ return name; }
}
```

The values associated to JaTS variables are objects that have a set of methods that can be invoked inside the executable declarations. Among these methods, are `getFieldType()` and `getFieldName()`. The argument passed to the `getFieldName()` method indicates that the name desired is the one of the first field declared in that field declaration, since many fields can be declared in only one field declaration. The methods in an information-extracting declaration are executed from the innermost to the outermost, as Java methods would be. The object returned by the outermost method is the result of the information-extracting declaration. After execution, it replaces the executable declaration.

Whereas information-extracting declarations extract information from original declarations, the second type of executable declarations modify original declarations. This is done by invoking methods on a copy of an original declaration. Executable declarations of this type are called *Information-Modifying Declarations*. For example, consider that the class

```
class Person implements Runnable, Clonable {
    [[ #result = #a :: #result.removeModifierName("private");
                       #result.addModifierName("protected");
                       #result.addModifierName("volatile"); ]]
}
```

corresponds to the right-hand side of a transformation. The execution of the executable declarations of this transformation, assuming that `#a` is mapped to "`private String name`", produces the class

```
class Person implements Runnable, Clonable {
    protected volatile String name;

}
```

In fact, the execution of information-modifying declarations does not yield the result of executing one of its method invocations, as in information-extracting declarations. The "`#result=#a`" declaration before the "`::`" operator indicates that the variable `#result` will be mapped to a copy of the object mapped to `#a`. The methods in the body of an information-modifying declaration change the state of the object mapped to `#result`. After their execution, that object is returned as the result of the executable declaration.

It is worth note that if an exception is thrown by a method during the execution of an executable declaration, the application of the transformation is aborted. Special care should be taken so that only valid executable declarations are derived.

2.5 *Iterative Declarations*

JaTS 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. Similarly to executable declarations, they can only appear in the right-hand side of the transformation. Moreover, an iterative declaration has three basic components: a variable denoting a declaration set, also called iteration set, a placeholder variable ranging over this set and a body of declarations possibly referring to the placeholder.

The execution of an iterative declaration yields a copy of the body of declarations for each element of the iteration set, where, for each of these copies, the placeholder variable is replaced by an element of the iteration set.

For example, a transformation that hides the fields of a class and defines "`get`" methods for them can be specified with the iterative declaration of the following transformation:

Left-hand side:

```
class Person extends Object <implements #ifs:NameList> {
    #attrs:FieldDeclarationSet;
    #mtds:MethodDeclarationSet;
}
```

Right-hand side:

```
class Person <implements #ifs:NameList> {

    #mtds:MethodDeclarationSet;
    forall #a in #attrs do
      [[#result=#a :: #result.removeModifierName("public");
                      #result.addModifierName("private");]]

      public [[#a.getFieldType()]]
             [[(#a.getFieldName(0)).addPreffix("get")]]() {
          return [[ #a.getFieldName(0) ]];
      }
    end
}
```

the application of this transformation would generate a new field and a new method for each object in the set associated to `#attrs`. So, applying this transformation to the class

```
class Person extends Object implements Runnable, Clonable {
    public String name;
    public Address address;
    public int age;
    public char gender;
}
```

yields the following class:

```
class Person implements Runnable, Clonable {
    private String name;
    private Address address;
    private int age;
    private char gender;
    public String getName() { return nome; }
    public Address getAddress() { return address; }
    public int getAge() { return age; }
    public char getGender() { return gender; }
}
```

In JaTS, it is possible to specify conditional replacement. For example, the right-hand side shown above could be modified, so that the `get` methods were added only for formerly `public` fields. Due to space constraints, however, we have omitted the construction to specify this kind of replacement.

2.6 *Preconditions*

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, a precondition is specified as an expression preceded by the keyword "`precondition`". Most of the arithmetic, logical, relational and conditional operators of Java can be used to specify a precondition, except for those that need an environment, like "=", "+=" and "++", postfix and prefix. Also, some precondition-specific JaTS constructions are allowed.

Preconditions work much like information-extracting declarations, except for the fact that, after the evaluation of the expression, the result must be a boolean value. The following is an example of a precondition for the transformation of the previous section:

```
precondition #attrs.size() > 0 && !(#attrs in #mtds) &&
             !(#attrs in #attrs.getInitializations());
```

Since the variable `#attrs` is mapped to a declaration set, this precondition says that the variable `#attrs` must be associated to a set with, at least, one element and that the elements of that set are all unreferenced, otherwise, the transformation cannot be applied. The method "`size()`" is invoked in the object that corresponds to the variable `#attrs`. This method returns an integer corresponding to the number of objects in the declaration set.

The "`in`" operator checks if there are references to the value mapped to a certain variable in a certain context and yields `true` if references are found. In case the value is a collection, the check is performed for all of its elements and the result yielded is `true` if any element in the collection is referenced. The kind of check performed by this operator is very useful for describing preconditions for refactorings and refinement laws. In the previous example, it is verified if the elements of the declaration set mapped to `#attrs` are all unreferenced in the set of method declarations mapped to `#mtds` and in the set of the initializations of the field declarations mapped to `#attrs`. The transformation described in the previous section, when using this precondition, implements the *Encapsulate Field* [9] refactoring.

3. An Action Semantics for JaTS

As JaTS introduces several nontrivial constructions and is intended to support critical activities such as formal development and refactoring, it is important to precisely define the semantics of those constructions. This leads to a comprehensive unambiguous understanding of the language and provides a sound basis for guiding the implementation of transformation systems using JaTS as its language.

In order to formally specify the semantics of JaTS, we chose the Action Semantics [15, 18] formalism, which was created with the goal of making the formal specification of programming languages more readable and easier to reuse and modify. Indeed, the main advantage of Action Semantics over other formalisms for specifying the semantics of programming languages is its readability. Besides that, the notation has built-in constructs for specifying non-trivial concepts like exceptions and iteration, which adds up for the readability, abstractness and ease of use of the formalism.

An Action Semantics (AS) specification is composed by the abstract syntax of the language being specified, which defines the overall structure of the language without worrying about matters such as ambiguity, semantic functions, mapping each element of the syntax to the semantic entities representing its behavior and semantic entities, which represent the implementation-independent behavior of programs, as well as the contributions that parts of programs make to its overall behavior. In this section, we will not specify the abstract syntax neither the semantic entities of JaTS. These will be mentioned throughout the parts of the specification, when necessary.

3.1 *Semantic Functions*

The specification of the semantics of the JaTS language contains four main semantic functions: run, match, execute and replace.

The run function specifies the meaning of applying a transformation. It receives as arguments a transformation (left-hand side, right-hand side and precondition), source Java types and a possibly empty map with variable-value mappings provided by the user (see Section 2.3). The match and replace functions have a simple semantics, roughly corresponding to the well known concepts of pattern-matching and substitution; their specifications are omitted for brevity. We focus on the execute semantic function and on its auxiliary functions, due to the complexity of the execution process.

3.1.1 *Execution of Iterative Declarations*

The semantics of iterative declarations is slightly complicated. The main reason for that is the fact that the execution of an iterative declaration involves the concepts of both execution and replacement. Every time the placeholder variable is mapped to an element in the iteration set, a new replacement has to take place before the declarations in the body of the iterative declaration can be executed, and that happens for each element of the iteration set. This is sketched below:

needs: **Java Action Semantics [6].**
introduces: execute _ , execute block declarations _ , respectively execute block _ using _ and _.

- execute _ :: IterativeDeclaration → action [escaping | giving a syntax-tree] [using current binds].

```
(1)    execute [[ “forall” $V_1$:Variable “in” $V_2$:Variable “{”D:IterativeBlockDeclaration$^*$“}”]] =
           ||| check (already-matched $V_2$) and check not (already-matched $V_1$)
           ||then
           ||| give the declaration-set bound to $V_2$ then clone it
           |then
           || respectively execute block D using $V_1$ and (the given declaration-set)
           or
           || check not (already-matched $V_2$) or check (already-matched $V_1$)
           |then
           || escape.
```

In Action Semantics, vertical bars are responsible for identation and indicate which actions are passed as arguments to each action or action combinator. The or action introduces non-determinism in an AS specification. The then action works as a sequential composition operator for transient information. Finally, the escape action roughly corresponds to the throwing of an exception.

The execute function verifies if the placeholder variable and the iteration set are valid and, if so, calls the following function:

- respectively execute block _ using _ and _ :: IterativeBlockDeclaration* → Variable → declaration-set → action [escaping | giving a syntax-tree] [using current binds].

This function binds the placeholder (PH) variable to each of the values in the iteration set and, for each of these values, calls the execute block declarations _ function –responsible for calling the replace and execute functions for each declaration in the body of declarations–. It also removes the value just mapped to the PH variable from the iteration set and checks if the iteration set is empty. The result of this function is a parse-tree consisting of the result of executing the iterative declaration.

3.1.2 *Execution and Replacement of Executable Declarations*

As mentioned earlier, there are two types of executable declaration. Both of them have a semantics which is more related to Java than to JaTS and, as such, has been previously described [6]. The replace semantic function, on the other hand, has interesting semantics when treating information-modifying declarations.

The execute function evaluates all Java method invocations in the body of an executable declaration and yields a syntax-tree corresponding to the result of its execution. It is defined as follows:

needs: **Java Action Semantics [6]**
introduces: execute _ .

- execute _ :: ExecutableDeclaration → action [escaping | storing | diverging | giving a syntax-tree] [using current binds].

We will not further specify the execute function, since its semantics is very coupled with the Java semantics for method execution, and that has already been specified using Action Semantics [6].

The semantics for replacement in information-modifying declarations is a little more complicated than in other JaTS constructs. That happens because these declarations use an auxiliary variable as a placeholder and this variable is not replaced by the value associated to it in the result map. Instead, it is replaced by the value associated to another variable. The replace semantic function for information-modifying declarations is specified below:

needs: **Java Action Semantics [6]**
introduces: replace _ .

- replace _ :: ExecutableDeclaration → action [escaping | giving a syntax-tree] [using current binds].

```
(1)     replace [[ "[[" V:Variable "=" V₁:Variable "::" P:MethodCall⁺ "]]" ]] =
              |check (already matched V) then escape
              or
              || check not (already-matched V)
              |then
              ||| give the object bound to token of V₁
              ||then
              |||| clone the given object
              |||then
              ||||| (bind token of V to the given object and rebind) hence (replace P)
```

The function first verifies if the V variable has already been matched. This variable will be used as a placeholder for the value mapped to V_1 in the scope of the executable declaration and cannot be previously matched. Then, a copy of the value mapped to V_1 is mapped to V and the variables in the sequence of method calls of the information-modifying declaration are replaced by the values mapped to them, where the clone action returns an identical "deep" copy of the given object or syntax tree and the hence action works as a sequential composition operator for scoped information.

4. Evaluating JaTS

Since there are many languages and tools for specifying program transformations [3, 8, 12, 17], it is interesting to better compare JaTS with one of these languages, in order to assess its ease of use, expressive power and limitations. We then chose LET [8], a language to specify program transformations in a higher level of abstraction than that supported by the TXL language [7].

In LET, transformations are specified as a 4-tuple consisting of a lexical component, a syntactic component, a transformational component and an auxiliary component. The first and second describe, respectively, the non-terminal and terminal symbols of the grammar of the source language. The transformational component consists of mappings from constructions in the source language to constructions in the destination language. The transformations are applied by pattern matching and replacement. The auxiliary component consists of TXL auxiliary routines.

Implementing the transformation shown in Section 2.5 using LET would be easy. Abstracting from the details of encoding the grammar of Java in LET, the task would consist of programming only the transformational component of the LET specification. No TXL code would be necessary.

A powerful feature of LET is the capability of using TXL functions as part of the transformation, adding all the expressive power of TXL to LET. This feature, however, presents the drawback of requiring the user to know TXL as well. JaTS, on the other hand, is a superset of Java and, as such, is easier to use and get used with, at least for Java programmers. Since it is a language-specific transformation language, JaTS takes into account the semantics of Java when performing a transformation, so that JaTS transformations can only generate well-formed Java programs.

Some problems arise when trying to implement some JaTS constructions using LET. The first one arises when trying to code a JaTS transformation that uses optional matching. Since LET does not support optional structures, we have to treat each optional structure as being two different structures. For example, the JaTS pattern:

```
class #C <implements #ifs:NameList> { (...) }
```

would be implemented in LET as two patterns: one corresponding to a class declaration where the `implements` clause is present and one corresponding to a class declaration where it is not.

In spite of the duplication of code, that solution would be fair if only the `implements` clause of a class declaration could be declared optional. The truth, however, is that many structures in a JaTS specification may be declared optional. If we tried to code the JaTS pattern

```
class #C <extends #SC> <implements #ifs:NameList> { (...) }
```

using LET, we would have to create four different patterns, so that all the possibilities of matching could be covered. The number of patterns required grows exponentially with the number of optional structures present, making it impractical to implement transformations with optional structures in LET.

Another problem arises when we try to implement the semantics for declaration matching in LET. For example, consider the following JaTS transformation:

Left-hand side:

```
class #C {
    #a:FieldDeclaration;
    #m:MethodDeclaration;
    #n:MethodDeclaration;
}
```

Right-hand side:

```
class #C {
    #a:FieldDeclaration;
    #n:MethodDeclaration;
}
```

This very simple transformation removes one of the method declarations from the body of a class that has exactly one field declaration and two method declarations. Implementing this transformation in LET as a series of patterns, one for method declarations, one for field declarations, etc, is not possible, since we cannot define a transformation rule that behaves differently when matching identical patterns.

An alternative approach would be to define a rule matching the whole class declaration. The pattern to be matched in this case would be a class declaration containing a field declaration followed by two method declarations. This solution works fine, as long as the order of the declarations in the source class does not change.

The semantics for the matching of declaration sets could not be implemented in LET either, due to limitations in the expressive power of the language. The problems encountered were similar to the ones mentioned in the previous example.

5. Conclusions

In this paper we introduced JaTS, a language for specifying program transformations for the Java programming language. We presented its syntax and informal semantics, formally specified some key aspects of its semantics and analyzed the language comparing it with another language for specifying program transformations. A prototype system implementing all the constructions presented in this paper and capable of storing the transformations and applying them to Java programs has been devised.

The JaTS language may be used to specify a wide range of program transformations, since it was not designed with a specific kind of transformation in mind. Many refinement laws [4, 5] can be easily implemented using JaTS. Some refactorings can also be defined as JaTS program transformations, making it a useful tool in the development of software using Extreme Programming [2], for example. JaTS may also be used to simply generate code, freeing the programmer from the task of generating tedious code. For example, a program transformation that generates “`set`” and “`get`” methods for a class that has only fields, has already been implemented and tested.

Comparing JaTS with related approaches for specifying program transformations, JaTS offers significant advantages. 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, JaTS constructions allow a higher level of abstraction for specifying transformations, not just because the syntax of JaTS is close to Java, but also because specific concepts of Java are taken in to account by JaTS constructions.

Many works related to program transformation have been devised. Among the tools supporting refactoring [12, 13, 17, 19], there is one [19] that deserves some special attention, since it focuses on the Java programming language. It supports a set of well-known refactorings, some of them not yet supported by JaTS. It has limitations, however, since the user cannot specify new refactorings, even simple ones. As for general-purpose transformation systems [3, 8], we briefly evaluated the expressive power of one of them, LET [8], in Section 4. Some JaTS constructions could not be implemented using LET, since it cannot take the semantics of Java into account, only its syntax.

Acknowledgements

We would like to thank Hermano Perelli for his support on specifying the semantics of JaTS. We are also grateful to the anonymous referees, who helped to improve this paper.

References

[1] Débora Aranha and Paulo Borba. Parameterized packages and Java. *In II Brazilian Symposium on Programming Languages*, pages 204-218, Campinas, Brazil, September 1997.

[2] Kent Beck. *Extreme Programming Explained : Embrace Change*. Addison-Wesley, 1999.

[3] U. Bergmam, A.F. Prado and J.C.S.P. Leite. Desenvolvimento de Sistemas Orientados a Objetos Utilizando o Sistema Transformacional Draco-PUC. *In X Simpósio Brasileiro de Engenharia de Software*, pages 173-188, São Carlos, Brasil. 1996.

[4] Paulo Borba and Augusto Sampaio. The Basic Laws of ROOL: An Object-Oriented Language. Revista Brasileira de Informática Teórica e Aplicada, Volume VII, Número 1, Setembro, 2000.

[5] Paulo Borba. Where are the laws of object-oriented programming? *In I Brazilian Workshop on Formal Methods*, pages 59-70, Porto Alegre, Brazil, October 1998.

[6] Deryck F. Brown and David A. Watt. JAS: a Java Action Semantics. *In Proceedings of AS'99*, BRICS notes series, 1999.

[7] James R. Cordy and Ian H. Carmichael. The TXL Programming Language Syntax and Informal Semantics. Department of Computing and Information Science. Queen´s University at Kingston. Kinston, Canada, June 1993.

[8] Marcelo F. Felix and Edward H. Hausler. LET: Uma Linguagem para Especificar Transformações. *In III Simpósio Brasileiro de Linguagens de Programação,* pages 109-123, Porto Alegre, Brazil, May 1999.

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

[10] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented software*. Addison-Wesley, 1995.

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

[12] William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *In ACM Transactions on Software Engineering and Methodology*, Vol 2, No 3, July 1993, Pages 228-269.

[13] Ivan Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. *In Procedings of OOPSLA' 96,* pages 235-249, USA, 1996.

[14] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1994.

[15] Peter D. Mosses. A Tutorial on Action Semantics. *Notes for Formal Methods Europe' 96,* BRICS Notes Series, 1996.

[16] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[17] Don Roberts, John Brant and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, pages 253-263, 1997.

[18] David A. Watt. *Programming Language Syntax and Semantics*. Prentice-Hall, 1991.

[19] The jFactor Documentation. Avaliable at http://www.instantiations.com/jfactor/docs/default.htm