ReL: A Generic Refactoring Language for Specification and Execution

Thomas Ruhroth, Heike Wehrheim, and Steffen Ziegert
Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
{thomas.ruhrroth, wehrheim, steffen.ziegert}@uni-paderborn.de

Abstract—Refactoring is a powerful technique for improving the structural quality of software models and programs. Besides informal, example-driven descriptions of refactorings, a number of languages for specifying refactorings have been developed. Such refactoring languages are either specific to particular programming languages or to particular application purposes.

In this paper, we present the generic refactoring language ReL. ReL is a domain-specific language which can be instantiated for any target language with Backus-Naur-style grammar. Thus obtained ReL instances are equally well suited for specifying refactorings and executing them. A toolchain for ReL supports the automatic instantiation for target languages as well as the parsing and execution of refactoring descriptions.

I. INTRODUCTION

Quality assurance is one of the key challenges in software engineering. It has long been acknowledged that the external quality of software (e.g. correctness) is strongly correlated with the internal quality. Refactoring [8], [19] is a frequently applied technique for improving the internal, structural quality of software achieving better readability, extensibility, and maintenance. The purpose of refactoring is to enhance the internal quality of code while keeping its external behaviour. Refactoring has even become a key step in one of the modern software design processes, namely extreme programming [2].

When studying the huge amount of work on refactorings it becomes apparent that it can be broadly divided into approaches specifying refactorings and those providing concepts and tools for executing refactorings. A third dimension is furthermore the programming (or more generally, modelling) language targeted. Starting with the book of Fowler [8], a number of approaches developed ways of defining refactorings, some – like Fowler – rather informal, explained by means of example [1], [12] and some more formal as to enable proving properties of refactorings [18], [17], [9], [26]. Work in the area of refactoring execution focused on technical, machine-readable descriptions of refactorings [29]. Among the latter approaches are languages supplied by Integrated Development Environments like Eclipse [6] or IntelliJ IDEA [10]. While the former techniques are understandable by developers and not by machines, for the latter it is exactly the other way round. Software developers, however, need ways of describing refactorings and ways of communicating these to colleagues. Still, a manual execution of refactorings is practically impossible; machine support is indispensable. Hence, a specification language for refactorings is needed which supports both definition and execution. Moreover, we need to cover different target programming languages with different syntax, but at the same time do not want programmers to have to learn new refactoring languages whenever they move to a different programming language.

In this paper, we present the refactoring language ReL which integrates all these aspects into one single language. ReL is a generic language, not specific to a particular programming language but instantiable for every programming language whose grammar can be given in Backus-Naur-Form [11]. ReL is moreover a general purpose refactoring language: it can be used by programmers to write refactorings as well as execute these on given code.

We have implemented a prototype toolchain for ReL (see Figure 1) supporting these features. It comprises the instantiation of the generic language ReL for a specific target language XX, given both the core grammar of ReL and the target language's grammar. Having thus obtained a grammar for ReL_{XX}, programmers can write refactorings for the target language within ReL_{XX} and deposit them in a refactoring repository. From the grammar of ReL_{XX}, our toolchain furthermore automatically generates a refactoring tool which can read in the refactorings of the repository and automatically execute them. In summary, we have obtained a refactoring language tailored to a specific target language which can be used for modelling and execution of refactorings.

In Section II we present the general idea of ReL. In Section III we demonstrate the approach to define refactorings with ReL by example. This section also deals with some ingenious parts of ReL's semantics that facilitates its suitability for...
Listing 1. An example Java program before refactoring

```java
public class Printer {
    public void printInfo() {
        int i = 5;
        print(i);
        int count = 0;
        while (i != 0) {
            count++;
            i--;
        }
        print(count);
    }

    public void printInfoComplete() {
        for (int i = 0; i < 10; i++) {
            print(i);
            int count = 0;
            while (i != 0) {
                count++;
                i--;
            }
            print(count);
        }
    }
}
```

Listing 2. An example Java program after refactoring

```java
public class Printer {
    public void printInfo() {
        int i = 5;
        print(i);
        int count = getCountOf(i);
        print(count);
    }

    public void printInfoComplete() {
        for (int i = 0; i < 10; i++) {
            print(i);
            int count = getCountOf(i);
            print(count);
        }
    }

    private int getCountOf(int i) {
        int count = 0;
        while (i != 0) {
            count++;
            i--;
        }
        return count;
    }
}
```

modern programming languages, like Java. The next two sections deal with ReL’s toolchain: Section IV shows how a ReL instance can be generated for a given target language and Section V explains the tool generation step. Related work is discussed in Section VI. Section VII concludes.

II. A COMMON STRUCTURE FOR REFACTORING

It defines a common structure for refactoring descriptions which is first of all independent of the target language. This generic core can be instantiated and made language specific, thereby obtaining a ReL instance. In this section, we start with explaining the basic ingredients of refactoring descriptions. Here and throughout the paper, we illustrate ReL by means of the refactoring “Extract method” (for Java). The purpose of this refactoring is to extract a block of statements, possibly appearing more than once in the code, into a new method and replace the statements by a call to this method. The class `Printer` in Listing 1 is a good candidate for the application of this refactoring: method `printInfo()` and method `printInfoComplete()` share a common block of statements (from the declaration of the integer variable `count` to the print statement). This block of statements could be extracted into a separate method and be replaced by an appropriate method call thus yielding the class in Listing 2. Though conceptually easy to understand, “Extract method” is a particularly challenging refactoring for tools and – being a key refactoring providing the basis for lots of others – has thus been termed the refactoring rubicon [7]. The difficulty essentially lies in the fact that “Extract method” requires extracting specific non-trivial information about method bodies. Due to this property of “Extract method” most of the other approaches to generic refactorings cannot handle this refactoring.

Next, we first of all explain the general structure for describing such refactorings. Most of the natural language and formal descriptions of refactorings (e.g., Fowler [8] and Roberts [24]) use a common scheme for this: they give the code structure before and after applying the refactoring. ReL follows this general scheme. In contrast to the example driven descriptions of Fowler, refactoring languages need to describe patterns of transformations, applicable to arbitrary code adhering to a particular structure. The before and after structure of a refactoring is thus given in the form of (before and after) templates. For “Extract method” the after template for instance needs to describe the introduction of a new method and the replacement of a block of statements by a method call. A template usually mixes language specific parts with metavariables. Meta-variables serve as placeholders for actual code snippets. For example, in “Extract method” a meta-variable is used to stand for the block of statements to be extracted. Upon execution of a refactoring, the values of meta-variables are either already passed to the refactoring by means of a parameter (by the user of a refactoring tool) or are determined by matching templates to actual code. In both cases, metavariables are bound to parts of the code. Often, we would like to restrict this binding (e.g., a meta-variable should only be allowed to take the place of a block of statements but not that of a class definition). To this end, meta-variables get types.

The types of meta-variables are language specific aspects: the type “class” is possible for object-oriented but not for functional languages. In ReL, the possible types for meta-variables are determined by the grammar of the target language. As an example, consider the (excerpt of the) BNF-style grammar for Java in Listing 3. It gives some of the production rules of Java’s grammar, i.e., describes how the left-hand side nonterminals of the grammar can be replaced by the right-hand side. Here, `|` stands for an alternative, `*` is repetition, and symbols in quotation marks (e.g., `"{"`) are terminals. Optional parts are enclosed in square brackets. The types for meta-variables in the ReLJava instance are all nonterminal names of this grammar. Thus, every code part produced by such a nonterminal can be bound to a meta-variable with this type.

Besides the parameters and before and after templates, refactoring descriptions have two more important ingredients: preconditions and calculations. Preconditions allow to specify certain applicability conditions for refactorings. For “Extract method” we like to specify that the name of the new method is not equal to a name of an already existing method of the class. In ReL, preconditions are stated in first-order logic using additional so-called analysis functions which extract information from code. Calculations need to be employed when the after template cannot be completely build out of
III. DESCRIBING A REFACTORING WITH RE\mathcal{L}\

When describing a refactoring with RE\mathcal{L}, a major approach is to divide the refactoring into subrefactorings (as long as that is possible). This is the case with “Extract method”. The refactoring can be decomposed into the subrefactorings “Create method” and “Replace code with method call”. Although decomposed, we would like to be able to execute both subrefactorings in one step only. For this purpose RE\mathcal{L} allows to specify refactoring descriptions that are composite refactorings, i.e., they contain other refactorings as subrefactorings. Such a composite refactoring has no after template. Thus, it does not define the code transformations to perform itself. Instead it has some execution statements that state which subrefactorings to apply. These execution statements support conditions as well as loops. A composite refactoring has to guarantee by means of its precondition that applying its subrefactorings is possible.

The scheme in Figure 2 shows the idea of specifying the composite refactoring “Extract method” and its two subrefactorings. Each refactoring description is abstracted by its precondition and effect. The arrows indicate how the preconditions of the two subrefactorings are guaranteed to be valid:

- The precondition of “Create method” has to be lifted to the composite refactoring, i.e., the composite refactoring guarantees that applying “Create method” is possible.
- The precondition of “Replace code with method call” (which is applied after “Create method”) is guaranteed to be fulfilled by a combination of the composite refactoring’s precondition and the effect of “Create method”, i.e., its calculation and its after template. More precisely, the availability of the new method and its correct signature are guaranteed by the effect of “Create method” and the legitimacy of replacing the code fragments with calls to the new method is guaranteed by the composite refactoring.

Next, we are going to deal with the descriptions of the two subrefactorings in detail to exemplify using the concepts of RE\mathcal{L}. In a refactoring description, the general ingredients for defining the refactoring are shown on white background, while the templates code segments which mix the target language’s grammar with meta-variables are shown on grey background.

We begin with the refactoring description to create a new method which is shown in Listing 4.

The refactoring has five parameters. Since every parameter also is a meta-variable, their values can be used within the templates and predicates. The parameter pClassName is used in the before template to specify the class of which the new method becomes a member. The other four parameters specify the new method’s name (pMethodName), its formal parameters (pFormalParams), and its implementation (pStatements, pResult). All of them are used in the after template to build the new method.

Besides the parameter pClassName, the meta-variable tBodyDecl is used within the before template. Considering the source code of Listing 1, tBodyDecl is used to bind the declarations of printInfo() and printInfoComplete(). In contrast to the parameters’ values which are set as input data, the binding of this meta-variable is deduced from the source code when applying the
refactoring. Using this meta-variable in the after template has the effect that the declarations bound to it remain unchanged when the before template is replaced by the after template. Since we want this meta-variable to be able to bind multiple consecutive declarations, namely the declarations of `printInfo()` and `printInfoComplete()`, it is placed within a repetition segment:

```
#rl(tBodyDecl)*
```

A repetition segment is a mark-up of a code segment within the before or after template. Its meaning is that the enclosed code segment may be existing multiple times in the source code matching the template. Repetition segments may be used when there is no information about the number of certain code segments at the design-time of a refactoring description. A repetition segment is inserted into a template by a `#` followed by a distinct name for the repetition segment. The repeated code follows in parenthesis. We can use meta-variables within repetition segments. In this case, the repeated code consists of the meta-variable `tBodyDecl` only. After the parenthesis a restriction for the multiplicity of the repetition is specified. There are three possible multiplicity restrictions available: zero-to-many (marked with `+`), one-to-many (marked with `*`), and zero-to-one (marked with `?`). By choosing the zero-to-many multiplicity `tBodyDecl` may bind an arbitrary number of declarations. Technically speaking, the binding of `tBodyDecl` associates an ordered set of values to the meta-variable `tBodyDecl`. When using repetition segments, for every meta-variable within a repetition segment there are as many values as consecutive source code segments match to the code within the repetition segment. Using repetition segments increases the number of cases the refactoring can be applied as bindings to meta-variables become possible that involve more than one value (or no values at all).

The before template still holds information that we did not address yet: an identification name (specified after `Index`) and a scope (specified after `Scope`). The identification name assigns a distinct name to the code segment specified in the before template. This is needed because `ReL` allows to specify more than one code segment within each template and needs to be able to match a code segment specified within the after template to its corresponding code segment within the before template. In case there is more than one code segment specified in a template, we will call these code segments subtemplates. We will see a useful application of a template with multiple subtemplates when dealing with the refactoring description of “Replace code with method call”. The scope is defined to state what kind of source code segment is allowed to match to the subtemplate. Similar to the meta-variables’ types, possible scopes that a subtemplate is allowed to have are given by the nonterminals of the target language’s grammar. Defining such a scope enables us to use syntactic checking within the subtemplates.

There are some constraints for applying the refactoring that can not be expressed solely by means of the before template. These constraints are formulated using first order logic and considering the meta-variables’ bindings. For example the predicate

```
∀ x ∈ getMethodNames(tBodyDecl) • x ≠ pMethodName;
```

states the requirement that the name of the new method may not yet be in use. This predicate uses the analysis function `getMethodNames(x)` which returns the names of all methods given their declarations `x`. Declarations of other things than methods are disregarded by the analysis function. Analysis functions were already used by Roberts [24] in his approach to a formal analysis of refactorings. They have no side effects and are frequently needed within the predicates to state the precondition properly.

When exchanging the source code segments that matched to the subtemplates of the before template with the subtemplates of the after template, meta-variables’ bindings are used that were given as parameter or deduced by matching the before template to the source code. However, we would like to use other source code segments within the after template that were neither given as parameter nor determined during matching. For the refactoring “Create method” we need to know the type of the return variable which is represented by the meta-variable `cResultType` in the refactoring description of Listing 4. Determining this type is the purpose of the calculation. Depending on two of the meta-variables already known, `pStatements` and `pResult`, a value for `cResultType` is calculated. For this purpose the analysis function `getVarDecl(x, y)` is used which finds the declaration expression for the variable `y` within the statements `x`. Subsequently the analysis function `getType(x)` returns the type of the located declaration expression. The precondition has to guarantee that there is a valid binding for `cResultType` which is done by the second predicate of the precondition. Otherwise the refactoring description would not be sound.
The next subrefactoring of “Extract method” is to replace occurrences of statements with calls to the new method. For this refactoring description we assume that the new method already exists. This is required by its precondition and ensured by the sequential execution of “Create method” and “Replace code with method call”. The refactoring description is shown in Listing 5. Its parameters are similar to the first refactoring description. The most eye-catching difference is that the before template specifies three subtemplates instead of only one. This can be seen by the number of Index keywords used within the before template. Each Index introduces a new subtemplate.

The idea of this refactoring description is to find all occurrences of the statements to be replaced by the method call and replace them at one go. This is done by the third subtemplate which is called “ReplaceByCall”. To state that this subtemplate may be matched multiple times to the source code, we use the optional keyword Repeat with the multiplicity +. Similar to the multiplicity of repetition segments, three different multiplicities are allowed for the subtemplates: zero-to-many (marked with +), one-to-many (marked with *), and zero-to-one (marked with ?). In contrast to the unfolding of repetition segments, the source code segments do not have to occur in consecutive order when matching them to the subtemplate. Instead they can be anywhere within the source code. ReL’s semantics causes a subtemplate with given multiplicity to match as much source code segments as possible, i.e., without violating the precondition.

Within the subtemplate “ReplaceByCall” there are the meta-variables tStatsA and tStatsB. They are used for considering the statements before and after the statements to be replaced. As we do not want to constrain the number of statements before and after the code to be replaced, tStatsA and tStatsB both have the type Statements. Thus, they bind all statements which occur in the same code block as the statements to be replaced. Due to the multiplicity of the subtemplate, there is one such binding to tStatsA, resp. tStatsB, for each source code segment that matches to the subtemplate “ReplaceByCall”.

The first subtemplate which is called “FindClass” is used with the parameter pClassName to bind tClassOrInterfaceBody to the class body that we want to make changes in. This is used by the other two subtemplates with the keyword In to guarantee that subsegments of tClassOrInterfaceBody are considered only. Thus, matching source code segments that are located outside of tClassOrInterfaceBody (e.g., in another class) are not assigned to the subtemplate “ReplaceByCall” and not replaced by the after template’s subtemplate with the same identification name. Without this keyword, source code segments could be assigned to the subtemplate “ReplaceByCall” that are located in a different class.

In order for the reader to understand the purpose of the second subtemplate which is called “BlockMethod”, we need to address the template replacing mechanism. As we allow to define more than one subtemplate in the before template, it is possible that the matching source code segments are overlapping. Overlapping source code segments however impose a problem: if we replaced both source code segments, one replacement would overwrite the other replacement. Actually, the replacement of the first code segment could even destroy the code structure such that we do not know anymore how to replace the other code segment. Instead of replacing both code segments, ReL’s semantics simply disallows assigning overlapping source code segments to subtemplates of the before template. In case of possible overlapping source code segments ReL chooses that source code segment that matches to the subtemplate with the higher priority. They are prioritised in order they appear. So the purpose for the source code segment “BlockMethod” is to bind the source code segment of the new method’s declaration. Thus, it prevents the source code segment from being bound to the subtemplate “ReplaceByCall”. This prevention is necessary because we do not want the statements inside the new method to be replaced by a call to itself.

ReL offers a possibility to make an exception to this no-overlapping rule. This is done with the first subtemplate “FindClass”. The source code segment bound to “FindClass” is used to restrict the other two subtemplates to only match source code segments within the class declaration’s body tClassOrInterfaceBody. This would violate the no-overlapping rule. However, disallowing assigning two over-
lapping source code segments to subtemplates of the before template is necessary only if both source code segments are going to be replaced. Since the source code segment bound to “FindClass” is not going to be replaced, there is no conflict. To indicate that overlapping with a code segment is allowed (which may only be allowed if it is not going to be replaced) we use the special keyword NoReplace.

Having these two refactoring descriptions as well as a description for the composite refactoring outlined in the beginning of this section, we can refactor the program introduced in Section II. While pFormalParams and pResult both are parameters for the two subrefactorings, they do not have to be parameters for the composite refactoring “Extract method”. Instead, they can be calculated by analysing the source code when applying the composite refactoring and passed on to the subrefactorings as parameters. By doing so, the only parameters left are pClassName, pMethodName, and pStatements.

Note that with the preconditions for the refactoring descriptions we imposed some constraints that are stronger than we would like to have for a refactoring definition of “Extract method”, e.g., the fact that the new method has to have a return variable and its declaration has to be done within pStatements. This was done to simplify designing the refactoring descriptions but does not limit the approach of ReL. We can easily define refactoring descriptions for the other cases of “Extract method” and its subrefactorings. It is even possible to define multiple refactoring descriptions with the same signature, i.e. name and parameters, and put them into the same refactoring repository. We call such a refactoring *polymorphic*. The applicability of the multiple refactoring descriptions of a polymorphic refactoring is differentiated by their preconditions, not by their signatures. As a consequence, we do not need to choose which refactoring description to use. Instead, for applying a polymorphic refactoring we use the first refactoring description with valid precondition that is found in the repository.

### IV. Instantiating ReL for a Target Language

In ReL, a refactoring description is written for a specific target language. ReL itself however provides a concept that is independent of the chosen target language.

ReL is a family of refactoring description languages. A concrete ReL instance can be generated by combining a language-independent core grammar with a language-dependent grammar. The former one is the same in every ReL instance. It provides the syntax for specifying a refactoring’s name, parameters, meta-variable declarations, as well as its precondition and calculation predicates. The latter one is generated from a given grammar of the target language. This grammar is necessary for the code within the templates. It corresponds to the target language’s grammar added with the possibility to use meta-variables and repetition segments. Furthermore, it provides the types that can be used in meta-variable declarations.

For generating a ReL instance there is a prototype implementation. It takes the grammar of a target language as input. To obtain a corresponding ReL instance, it modifies the grammar and connects it with ReL’s language-independent core grammar. While the approach works for any BNF, our implementation is currently restricted to LL(k) grammars. Input and output formats are JavaCC [3] grammar files.

#### A. Core grammar and its connection to the language-dependent grammar

Listing 6 shows some production rules of the core grammar. The first three production rules define parts of the syntax for a refactoring’s description. These production rules are independent of the target language, i.e., every ReL instance has the same core rules. The reuse of these core rules in different target languages is possible due to ReL’s generic approach to describe refactorings with templates, meta-variable declarations, precondition predicates, and calculation predicates.

The production rule for the nonterminal NTSCOPETYPE connects this core grammar with the instance-specific part that is generated from the target language’s grammar. To be more precise, this rule can produce every nonterminal that is available in the target language’s grammar. For the target language’s grammar of the example in Section III the production rule for the nonterminal NTSCOPETYPE in shown in Listing 7. The rule for this nonterminal demands a literal immediately before the code of a subtemplate that corresponds to the nonterminal producing the code. This literal defines the scope of the subtemplate. Demanding such a scope in refactoring descriptions, yields knowledge about the syntactic structure of the subtemplate’s code. Thus, we can decide which production rule to take for building an abstract syntax tree of the subtemplate’s code. Furthermore, when matching the subtemplate to the source code on the level of abstract syntax trees, many source code nodes that can not match to the template’s root node do not need to be taken into account.

#### B. Adaptation of the target language’s grammar

Reconsider the target language’s grammar of Section III. Every nonterminal within this grammar can be produced by the production rule of NTSCOPETYPE. However, besides writing subtemplate code in the syntax of the target language’s grammar, meta-variables and repetition segments are allowed. To allow the use of meta-variables and repetition segments within the code of a subtemplate, the target language’s grammar is modified.

---

**Listing 6. Parts of ReL’s core grammar**

<table>
<thead>
<tr>
<th>Production Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>REFAC_DESC ::= HEADER PARAMETER BEFORETEMPLATE</code></td>
<td>Defines the refactoring description</td>
</tr>
<tr>
<td><code>BEFORETEMPLATE ::= &quot;BeforeTemplate&quot;: BT_DEFINE</code></td>
<td>Specifies the before template</td>
</tr>
<tr>
<td><code>BT_DEFINE ::= ( BT_SUBTEMPLATE )</code></td>
<td>Defines the before template</td>
</tr>
<tr>
<td><code>BT_SUBTEMPLATE ::= &quot;Index&quot;: INDEXNAME ( NoReplace )?</code></td>
<td>Specifies the index of a subtemplate</td>
</tr>
<tr>
<td><code>PRECONDITION CALCULATION AFTERTEMPLATE</code></td>
<td>Specifies the precondition and calculation</td>
</tr>
</tbody>
</table>

**Listing 7. Connection of core grammar and language-dependent grammar**

<table>
<thead>
<tr>
<th>Production Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`NTSCOPETYPE ::= &quot;CompilationUnit&quot; Compilation Unit</td>
<td>Defines the compilation unit</td>
</tr>
<tr>
<td></td>
<td>&quot;ClassOrInterfaceBody&quot; ClassOrInterfaceBody</td>
</tr>
<tr>
<td></td>
<td>&quot;FieldDeclaration&quot; FieldDeclaration</td>
</tr>
<tr>
<td></td>
<td>&quot;MethodDeclaration&quot; MethodDeclaration</td>
</tr>
<tr>
<td></td>
<td>&quot;Block&quot; Block</td>
</tr>
<tr>
<td></td>
<td>&quot;Statements&quot; Statements</td>
</tr>
<tr>
<td></td>
<td>&quot;BlockStatement&quot; BlockStatement</td>
</tr>
</tbody>
</table>

...
Listing 8. Adaptation of target language’s grammar (step 1)

| Statements ::= ( BlockStatement ) * |
| Statements ::= ( MVAR | Statements_A ) |
| Statements_A ::= ( BlockStatement ) * |

Modification:

Statements ::= ( BlockStatement ) *
⇓

Statements ::= ( MVAR | Statements_A )
Statements_A ::= ( BlockStatement ) *

Listing 9. Adaptation of the target language’s grammar (step 2)

| Statements ::= ( MVAR | Statements_A ) |
| Statements_A ::= ( BlockStatement ) * |
| Statements ::= ( MVAR | Statements_A ) |
| Statements_A ::= ( BlockStatement ) * |
| Statements ::= ( BlockStatement ) * |
| Statements ::= ( BlockStatement ) * |
| Statements ::= ( BlockStatement ) * |
| Statements ::= ( BlockStatement ) * |

For every nonterminal in the grammar, it shall be possible to write a meta-variable instead of a source code segment. Thus, for every production rule another rule is created. Every appearance of an old rule is substituted with the new rule. The new rule either produces a meta-variable or points to the old rule. Listing 8 shows such a modification for an arbitrary production rule. The production rule for the nonterminal Statements of the target language’s grammar is replaced by a rule with a choice in the adapted grammar. The new rule allows to expand Statements into a MVAR node as alternative to the old right-hand side which is reshipped into the new nonterminal Statements_A. An expansion of Statements into MVAR represents the appearance of a meta-variable that is typed as Statements. Due to such modifications we were able to use meta-variables as replacement for source code segments in the example of Section III.

To deal with repetition segments we need to further extend the grammar. If a production rule contains a repetition symbol like *, +, or ? on its right-hand side, another new rule is needed for each sequence of symbols that is enclosed by such a repetition symbol. These new rules allow us to use repetition segments instead of ordinary source code segments. Listing 9 shows this kind of extension for a rule that contains exactly one repetition symbol. Statements_A replaces each sequence of symbols that is enclosed by a repetition symbol – in this case the nonterminal BlockStatement only – with a new production rule. The new rule Statements_B1 allows a repetition segment as alternative to the BlockStatement. Thanks to Statements_A which retains the repetition symbol enclosing Statements_B1, repetition segments and ordinary source code segments can be used side by side as is done with the meta-variable tBodyDecl in the after template of the refactoring description shown in Listing 4.

Note, that the nonterminal MVAR represents the appearance of a meta-variable but does not state the type of the meta-variable it represents. Instead, its type is given by a declaration within the refactoring description (e.g. the meta-variables tStatsA, tStatsB, and pStatements are declared as Statements in Listing 5). It has to be ensured that each meta-variable’s type agrees to its usage within the subtemplates which is constrained by the target language’s grammar. In other words, a MVAR has to be used as a code segment that can be derived by the production rule of its declared type.

According to the grammar of Listing 3 there is only one appearance of Statements within the right-hand side of the production rule of the nonterminal Block (which is the scope of the subtemplate “ReplaceByCall”). However, we use three meta-variables with Statements as type, namely tStatsA, tStatsB, and pStatements. Thus, it has to be ensured that these three meta-variables may be used in a sequence when only one derivation of Statements is required. Since Statements can be derived into (BlockStatement)* which can be replaced by three segments of (BlockStatement)* (one for each meta-variable), this is allowed.

V. GENERATING A REFACTORING TOOL

To apply a refactoring given its description, a refactoring tool based on ReL’s semantics has to accomplish the following tasks:

1) Transforming the source code and parameters into abstract syntax trees.
2) Matching the before template’s subtemplates to source code and deducing values of meta-variables along the way.
3) Checking the precondition predicates considering the meta-variables’ values.
4) Computing the values of additional meta-variables according to the calculation.
5) Filling the after template with meta-variable information and replacing source code segments that matched to subtemplates of the before template.
6) Transforming the restructured abstract syntax tree back into source code.

The prototype that accomplishes these tasks is generic. It works for any pair of target language and ReL instance as long as they belong to each other, i.e., the ReL instance is generated from the target language. Both the target language’s grammar and the ReL instance’s grammar have to be provided as JavaCC grammar file. While the tasks 2) – 5) are provided as a generic core library, the code transformation tasks depend on JavaCC parsers which can be generated by these grammar files. A refactoring tool based on ReL is generated by combining both parsers with the core library.

VI. RELATED WORK

The existing work on refactoring languages can broadly be divided into three categories. First of all, there are refactoring descriptions for specific target languages like Java and C++. A lot of already mentioned approaches fall into this category, for instance [21], [24], [28]. The example driven descriptions of Fowler are of this type as well as he always gives his examples in a specific languages. In addition, methods for refactoring UML diagrams (e.g. [4], [16]) belong to this category. The focus of these approaches lays on the actual definition of specific refactorings, the more, the better.

The second category studies languages in general, i.e., the question of how to define refactorings. Most of these approaches are based on meta models and thus can be used for any meta model based target language. The focus is usually
not on defining refactorings, but rather on showing that a certain language is suitable for describing refactorings. In these approaches techniques for defining model transformations like graph grammars [18], [15], [25], [5] or QVT [13] are employed.

Finally, there is work on generic refactoring languages like ours. Generic approaches aim at extracting the basic concepts of refactorings into a generic part which can afterwards be instantiated towards specific target languages. The approach of Lämmel [14] lifts concrete refactorings to a very abstract level (e.g., on which it is possible to define an extraction refactoring being able to extract any kind of entity), however, as a consequence leaves a significant amount of work (coding of refactoring transformations, checking preconditions) to the instantiator. Reimann et al. [22] develop a generic approach to refactoring domain specific languages, which allows to specialise given refactorings for a DSL family via role mappings. The works of [20] and [27] on the other hand provide a number of pre-defined refactorings for a specific meta model. Their meta models capture essential concepts of object-oriented languages. When this generic meta model can be mapped onto the meta model of the particular target language, the refactorings can be transferred. While this framework allows to completely dispose with writing refactorings for target languages, it is also limited in its applicability. First, the framework can only be applied when the target language matches the concepts of the generic meta model, basically only when the target language is an object-oriented language as well. Second, these approaches cannot handle refactorings involving the extraction of detailed information about method bodies, like “Extract method” does. In contrast, our technique can be applied to arbitrary target languages (object-oriented as well as imperative or functional programming language and even modelling languages) as long as they have a BNF-based syntax. Furthermore, by adapting to a target language and using the built-in concept of language specific analysis functions and their application in precondition and calculation, we can also handle complex refactorings like “Extract method” requiring sophisticated code analysis.

VII. CONCLUSION

In this paper we have presented ReL, a generic refactoring language. ReL is a language which can be used by programmers to write as well as execute refactorings. ReL can be instantiated for arbitrary target languages with a BNF-style grammar. We have build a toolchain for ReL which allows for automatic instantiation and implementation. We have explained the use of ReL by examples of Java refactorings. Our examples demonstrate that ReL can indeed handle the refactoring rubicon “Extract method”. The two example refactorings and their composition are part of an existing refactoring repository for Java. A refactoring tool for ReL_{Java} using this repository has been generated. When designing refactorings for this repository, i.e., “Extract class” and “Extract superclass”, we have made some experience that resulted in improvements to ReL’s semantics which allow for more intuitive specification and more efficient execution of refactorings [30]. These improvements include support for the

REFERENCES