Code Transformation by Direct Transformation of ASTs

M. Rizun
Lviv National University of Ivan Franko,
RMoD team, Inria Lille – Nord Europe
mrizun@gmail.com

J.-C. Bach S. Ducasse
RMoD team, Inria Lille – Nord Europe,
University of Lille, CRIStAL, UMR 9189
{jeanchristophe.bach,stephane.ducasse}@inria.fr

Abstract

Software evolves to be adapted to the environment, due to bugs, new features and design changes. Code transformations can be done manually, but that is a tedious and error-prone task. Therefore automated tools are used to assist developers in this maintenance operation.

The Pharo environment includes its own refactoring tool — the *Rewrite Engine* — that allows one to transform methods by directly specifying parts of the AST to be rewritten. In addition, it proposes a parse tree transformation engine. However this tool and the used DSL to express the patterns for matching and transforming trees are complex to understand and master. In this context, writing a transformation rule is not a trivial task.

We present a graphical tool built on the top of the *Rewrite Engine* — the *Rewrite Tool* — that abstracts the creation of transformation rules and proposes high-level AST operations that are simpler to understand than syntactic descriptions. It helps to automate the process of code transformation with a user-friendly interface.

Keywords AST, refactoring, rewriting, code transformation

1. Introduction

Code transformation is a core activity in software engineering. Most of the time of the development process is spent on software maintenance [DDN02]. While simple localized code modifications can be manually done without the help of any tool, it is dangerous and time-consuming for multiple changes in a row. Not only is manual refactoring error-prone, but the refactorings are not reusable for future changes. Therefore it is necessary to use tools to assist and automate code transformations during the whole application development life-cycle. Most current IDEs support refactorings [RBJO96, Tic01]. In addition to mere refactorings, some IDEs support the definition of specific code transformations [VCM+13]. These transformations are not struct refactorings in the sense that they do not have to ensure behavior preservation [FBB+99].A constraint that a transformation-assistant has to fulfill is to be understandable without too much effort. To be helpful and effective, the helping tool should not be more difficult to understand and to use than the code to be refactored.

In this paper, we introduce such a tool — the *Rewrite Tool*[1] — to ease the definition of code transformations in Pharo. The *Rewrite Tool* helps the developer express transformation rules to apply on methods. The tool does not expose a complex DSL driven syntax but offers simple operations on AST (Abstract Syntax Tree) nodes such as transform node into a variable. These rules can be saved for a later reuse. Thus the *Rewrite Tool* helps to improve long-term maintenance efficiency by building a set of refactoring rules.

The paper is organized as follows: Section 2 describes the context and the problem starting from an example. Then, Section 3 presents our solution — the *Rewrite Tool* — and its use. Section 4 discusses about features, limitations and future work about the tool. Section 5 concludes this paper.

2. Context and problem description: intuitive tooling is mandatory

Code transforming process consists of the following steps:

1. Definition of the code to change (or its *shape*)
2. Definition of the desired target code (or its *shape*)
3. Searching all occurrences of the code to replace in the source code
4. Replacing all found occurrences by target code

Doing this process manually can only be considered for very specific modifications, which are localized in the source code. Due to the risks and cost of code transformation and

IWST'15, July 15–16, 2015, Brescia, Italy.

ACM 978-1-4503-3857-8/15/07…$15.00.
http://dx.doi.org/10.1145/2811237.2811297

[1] Available here: http://smalltalkhub.com/#!/~MarkRizun/RewriteTool/ A video tutorial can also be found here: http://myfuncoding.blogspot.fr/

refactoring, a manual process on real applications with large codebase is not effective. Therefore, tools are mandatory to handle such tasks.

Such tools have at least two features to offer to be usable in real environments: a mechanism to express different parts of a transformation (we could consider it as a relation between a source — left-hand side — and a target — right-hand side) and a mechanism to execute the transformation on different scopes.

Well-known tools from the Unix community such as *sed*, *awk* or *perl* offer this feature. However, they consider code as character strings without any structure, therefore their *search&replace* language are expressed as flat patterns. As a consequence, developers have to be very careful with their pattern definitions, when performing massive transformations on source code to avoid disasters. Even a simple renaming can be dangerous. A good counter-measure is to consider source code as a tree-structure entity instead of a flat plain text. That is exactly the way tools from the XML world [W3C99, RCD+11], compilers, interpreters and more generally program transformation tools (such as ASF+SDF [BDH+01], Tom [BBK+07], RASCAL [KvdSV09, KHVDB+11, KvdSV11] and Spoofax [KV10]) are managing code. This implies to define the *shape* of the structure the developers want to transform. This *shape* is usually named a *pattern* and is expressed with a specific language. This way to express a transformation has also its own drawback: to define a pattern, one has to know and understand the structure of the tree.

As a modern and fully-tooled environment, Pharo supports creating rules that allows one to perform automated code transformations. This feature is provided by the *Rewrite Engine*[2]. This engine makes it possible to create rewrite rules that perform code transformations at different levels of complexity.

Consider a non-trivial example of transformation of a method in which some collection is initialized. Then, if the size of this collection is greater than 3, some calculations are done and a new element is added to this collection. After that, the result is computed. Assume the method contains the source code shown in listing 1.

All in all, this code has few problems that should be rewritten. First, it could be made more readable by adding a guard clause, *i.e.,* `collection size >= 3 ifFalse: [^self]`. As a result all computations from the `ifTrue: []` block can be extracted, and put after guard clause. Moreover, temporary variables can be moved from the `ifTrue: []` block to temporaries list at the beginning of the method. Second, instead of writing the assignment of the `result` variable two times, it can be done once, by assigning it to result of the `ifTrue: [] ifFalse: []` message. After such transformation, the resulting code would be as illustrated by listing 2.

[2] http://www.refactory.com/the-rewritetool

```
| collection result |
collection := LinkedList
                newFrom: {1.2.3.4.5}.
collection size >= 3
  ifTrue: [
  | temp1 temp2 otherCollection |
  otherCollection := {'a'.'b'.'c'}.
  temp1 := collection size - 1.
  temp2 := otherCollection size+1.
  collection addLast: temp1 / temp2.
  collection last = 1
  ifTrue:
    [ result := self calculateResult ]
  ifFalse:
    [ result := 'Wrong element.' ] ]
```

Listing 1. Source code to transform

```
| collection result
  temp1 temp2 otherCollection |
collection := LinkedList
                newFrom: {1.2.3.4.5}.
collection size >= 3
  ifFalse: [ ^self ].
otherCollection := {'a'.'b'.'c'}.
temp1 := collection size-1.
temp2 := otherCollection size + 1.
collection addLast: temp1 / temp2.
result := collection last = 1
  ifTrue:  [ self calculateResult ]
  ifFalse: [ 'Wrong element.' ]
```

Listing 2. Resulting code after transformation

Obviously, doing such a transformation by hand in each and every method of the system is a very time-consuming and annoying task. Also, there is always a risk to make a mistake during the transformation. Assume, we want to carry out described transformation automatically. In order to do that, the LHS and RHS parts of the rewrite rule have to be implemented using *Rewrite Engine*. Matching and transforming parts of rule should be implemented as shown by listing 3. In this listing, all *metavariables* (variables used by *Rewrite Engine* to match objects) are denoted by a backquote character (‘) which can be associated to other symbols such as **#**, **@**, **.** (dot), **{}** or another **‘**. Semantics of this operators are explained in table 1.

This rule is clearly non-trivial to implement, not to mention that influential mistake may be made accidentally. For instance, the programmer may miss a dot character when declaring statement metavariable, and instead of statement metavariable `‘.InitializationStatement` he declares `‘InitializationStatement` metavariable.

```
"Left-Hand Side part"
| `@temporaries |
`.InitializationStatement.
`@condition1
  ifTrue: [
    | `@otherTemporaries |
    `@.Statements.
    `@condition2
    ifTrue:
      [ ``@value := ``@calculate ]
    ifFalse:
      [ ``@value := `#wrongLiteral ] ]

"Right-Hand Side part"
| `@temporaries `@otherTemporaries |
`.InitializationStatement.
`@condition1
  ifFalse: [ ^self ].
`@.Statements.
``@value := `@condition2
  ifTrue:  [ ``@calculate ]
  ifFalse: [ `#wrongLiteral ]
```

Listing 3. Pattern expressed with *Rewrite Engine* to capture the method presented in listing 1

This represents a single variable, not a statement as we need. Such an inattention leads to serious consequences. In the previous example, the desired result will not be achieved, as LHS part of rule will not match described source code. When a rule with such a mistake is applied, no error or exception will be shown. Thus, the desired transformation will not be made in the system.

Like similar tools, *Rewrite Engine* has its drawbacks: to create rules, it uses a specific language, whose syntax can be difficult to understand. Table 1 gives an overview of the options that can be associated to the backquote character in order to understand the pattern expression of listing 3. The developer has to be careful when writing complex patterns, to avoid any confusion between symbols. The **dot** symbol expresses the statement for a metavariable after a ‘ or after ‘@, however it is still the instruction separator in Pharo without any ‘. Another important point when using the *Rewrite Engine* syntax is to carefully choose metavariable names, as different metavariable names match different AST nodes.

Plus, when using a DSL, operation semantics may be blurry. For instance, in the case of the *Rewrite Engine*, @ is a construct whose semantics changes whether it is associated to a backquote symbol or a dot symbol. In the general case (and in documentation), @ is associated to *list*. However, it rather means **subtree** when only used with ‘. Therefore, a new Pharoer might be confused by the tool and only use it by mimicing existing examples in a *try&error* pattern.

Symbol	Explanation
#	matches any literal objects
.	matches single statement
@	matches list of objects like statements, message keywords, etc.
‘	when match is found, recursive search is done inside matched node

Table 1. Summary of option symbols used in the *Rewrite Engine* language and their semantics

This is obviously not an acceptable development process in a professional environment.

3. Proposed Solution

To mitigate the problem exposed in section 2, the language can be formalized, meaning disambiguated, but it still could not be accessible to average programmers. Moreover, after the language problem, comes the rule application one: without any clear API or user interface, it is difficult to apply rules to certain environment scope, *i.e.,* packages, classes or methods. Therefore, to take advantage of the power of the *Rewrite Engine*, the rule application process demands an experienced developer. The solution most IDEs chose is to hide complexity and difficulty behind an intuitive, understandable and user-friendly interface. That is exactly the objective of the *Rewrite Tool*: having an intuitive abstraction for the complex and powerful *Rewrite Engine*.

Another attempt to write such a tool was done a few years ago with Flamel[3], but it was not maintained. Therefore we decided to begin to write another one from scratch to ensure a full understanding of the process. Our tool also focuses on the usability and aims at developers, who are not necessarily experimented senior Pharoers.

3.1 Rewrite Tool

We propose to introduce and detail *Rewrite Tool* — a tool which offers a solution to code transformation problems. It abstracts *Rewrite Engine* to take advantage of its power for the creation of rules of different level of complexity, without dealing with the syntax of the specific language. *Rewrite Tool* also allows users to implement and apply code transformations easily. It is consists of two parts:

- First one is *rewrite rule builder* (referenced as Builder, shown in figure 1) that allows users to implement rewrite rules using *Rewrite Engine*. The fact that rule creation process does not require from users to know syntax of this engine is an important and interesting aspect of the tool.
- Second part is *rewrite rule browser* (referenced as Browser, shown in figure 2): it enables users to instantly apply rules to desired environment scope.

[3] https://pharorwrules.wordpress.com/

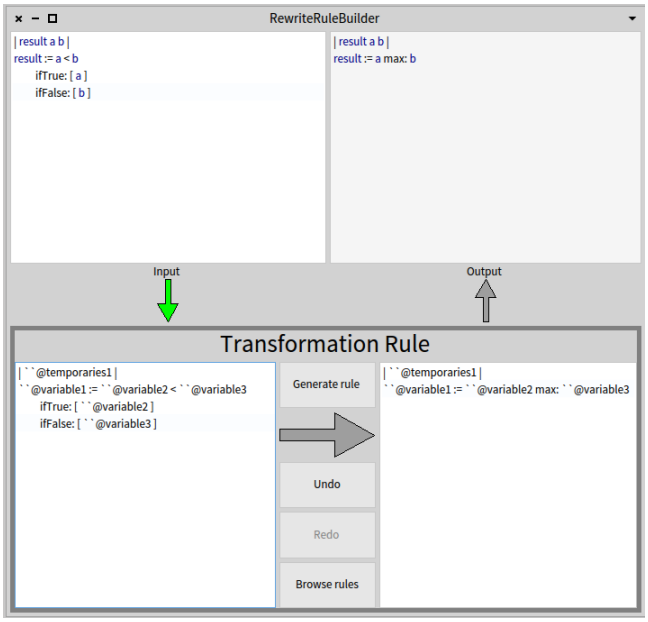

Figure 1. Rewrite Rule Builder

In the bottom section of Builder, a transformation rule is displayed, *i.e.,* matching and transforming parts of rule. While the user creates a rule in this segment of Builder, he can simultaneously see the result of applying this rule to source code from input pane (top left panel). The result is displayed in the output pane (top right panel). It means, that once user modifies a rule, he instantly gets feedback in the output pane. Both input and output panes are located above the transformation rule part of Builder.

Concretely, *Rewrite Tool* offers operations which are accessible via a contextual menu. In addition to general actions such as *Change*, *Rename*, *Undo/Redo*, *Generate Rule* and *Browse rules*, there are many operations that abstract *Rewrite Engine* DSL, avoiding to write a pattern with the dedicated syntax:

- Abstract condition: expresses a pattern with a condition (in Pharo) on matched nodes
- Abstract literal: adds a metavariable representing a literal (*e.g.,* `1`, `'a'`, `true`, *etc.*)
- Abstract object: adds a simple metavariable to match a simple object
- Abstract statement: allows the expression of pattern that watches a statement
- Abstract statements: allows the expression of a sequence of patterns
- Abstract temporaries list: matches a list of temporary variables

These operations have to be used in the left-hand panel and in the right-hand panel to build the source and target patterns composing the transformation rule.

3.2 *Rewrite Tool* at work

In the following subsection, we describe rewrite rule implementation process. Assume that the goal is to transform each piece of code that uses the construction illustrated in listing 4.

```
result := a < b
  ifTrue:  [ a ]
  ifFalse: [ b ]
```

Listing 4. Source code to be refactored

The code transformation should change it to a better implementation as in listing 5 that uses `max:`.

```
result := a max: b
```

Listing 5. Expected result of the transformation

Transformation rule writing. In this example, two objects, possibly numbers are compared. Actually, we are not interested in any detailed information about these objects. In order to create a rule that will do the transformation as presented, we use the Builder. In the input pane we put an example of the source code, that we desire to transform. As well, we have to set the same code in left-hand side part of transformation rule. Next, we put code that we want to have as a result of code transformation, into right-hand side part of the transformation rule. Now, the Builder is ready to work with the rewrite rule itself. For now, in transformation rule section, we have very specific rule, which will change only those places, where specific objects are used, *i.e.,* `result`, `a` and `b`. However, we would like to have a rule that will work in general case, and change each occurrence of such a construction independantly of the objects which are used in it. In order to make it, we should select the `result` variable in matching part of rule, right-click on it and execute *Abstract object* action. The same manipulation has to be done with `a` and `b` variables. Notice, that when we apply *Abstract object* action on `result` or any other variable, it is replaced in both - matching and transforming parts of the rewrite rule. Also, if we had used the `result` variable more than once in any part of rule, all occurrences of it would have been replaced. This little automation, among many others, helps to avoid annoying and time consuming manual work. As a result of these actions, we get the code illustrated by listing 6 in the transformation rule section (bottom left panel).

Note that `"@` characters have been generated and are part of the *Rewrite Engine* language syntax. An informed developer could also directly interact with it. Listing 7 shows the right-hand side of the transformation rule (in the bottom right panel).

```
``@variable1:=``@variable2 < ``@variable3
  ifTrue:  [ ``@variable2 ]
  ifFalse: [ ``@variable3 ]
```

Listing 6. Left-hand side of the transformation rule

```
``@variable1 :=
      ``@variable2 max: ``@variable3
```

Listing 7. Right-hand side of the transformation rule

Now we have a complete rewrite rule, that is usable to perform the desired code transformation. New variables, which replaced `result`, `a` and `b`, are called *metavariables*. They are used by the *Rewrite Engine* to match objects in source code. Each metavariable is recognized by the parser as a node of AST. Classes that represent metavariables are in the AST-Core-Pattern package. For example, metavariable in listing 8 matches any sequence of statements. As a reminder, table 1 in section 2 summarizes *Rewrite Engine* syntax.

```
`.@Statements
```

Listing 8. Metavariable matching any sequence of statements

Rule creation. Clicking the "Generate rule" button finishes the rule creation process. A name for the rule — and therefore the subclass of *RBTransformationRule* representing it — is requested in order to generate it. This newly created class can be used to perform code transformations programmatically, or using Browser which is shown in figure 2.

Rule execution. Browser enables users to apply rule to certain packages, classes and methods. Before performing code transformation, the tool displays changes, which are going to be done. Browser shows all rules, that are present in the current image, in bottom part in the dropbox. All of them can be reused for later code transformations. Programmer can then select the rules and the transformation scope before applying them.

3.3 AST code representation

In order to have reliable automated code transformations, we work with the AST representation of code instead of manipulating source code itself in the left-hand side and right-hand side parts of the rule. The AST provides detailed information about source code and its structure. For instance, figure 3, respectively figure 4, illustrate the AST representing the code of listing 6, respectively listing 7.

Every node of the AST, in the *Rewrite Tool*, provides its own options during rule creation process. For example, when user selects an instance of `RBBlockNode`, he has an option to create a metavariable, that matches any block — instance of `RBPatternBlockNode`. Manipulating AST instead of text allows one to easily compare nodes for equality, which is important to be able to massively transform code in a reliable way. As was mentioned earlier, when we apply *Abstract object* action on a given variable `X`, all `X`s references are replaced in LHS and RHS parts of rewrite rule. This automation is achieved by comparing all AST nodes for equality with `X` variable node. Those changes are also traced: for *Rewrite Tool*, AST keeps track of old nodes for each particular node. Meaning, that when user replaces `X` variable node with some metavariable node, *i.e.,* performs *Abstract object* action on `X` variable, this new node has a property named `#oldNodes` which contains `X` variable node. This information is very valuable for traceability and *undo* actions on specific nodes to return to the previous state.

The AST is obtained by parsing the source code with *RBParser* `RBParser» #parseRewriteExpression:`. *RBParser* also returns corresponding nodes for metavariables as well. As a result, it is possible for us to get an AST representation of any rewrite rule. *Rewrite Tool* functionality is based on this, inasmuch any action, made with LHS or RHS part of rule, has to be applied to AST.

4. Discussion and future work

As discussed in the article, in order to achieve reliable and precise code transformation, *Rewrite Tool*, in particular *Rewrite Engine*, manipulates AST representation of source code. This is the usual way to work in the program transformation community where tools such as Rascal [KvdSV09, KHVDB+11], Spoofax [KV10] and Tom [BBK+07] have been developed for this purpose. In the large Eclipse community, code transformations and refactoring tools are usually plugins relying on AST manipulation with JDT (Java Development Tools) as we do in Pharo.

Generally, transformation does not depend on code itself, instead it relies on AST representation of this code. Although *Rewrite Tool* can currently only be used to create rewrite rules at the method level, there is no reason not to be able to transform any other entities represented by AST such as packages or classes. It would allow developers to perform higher level transformations. To work in this direction, a first step before extending the *Rewrite Tool* would be the specification of a full compilation unit in Pharo: an AST representing it, including packages and classes (not only methods), should be defined. Then, *Rewrite Tool* could be adapted to be able to perform transformations on such ASTs. For example, transformations like "Move to class side" refactoring could be managed by the tool. It would also allow one to manipulate a whole class hierarchy or package dependencies.
In the future, we would like to extend the *Rewrite Tool* in order to be able to write major refactoring and program transformations. It would give developer an user-friendly interface for a complex task.

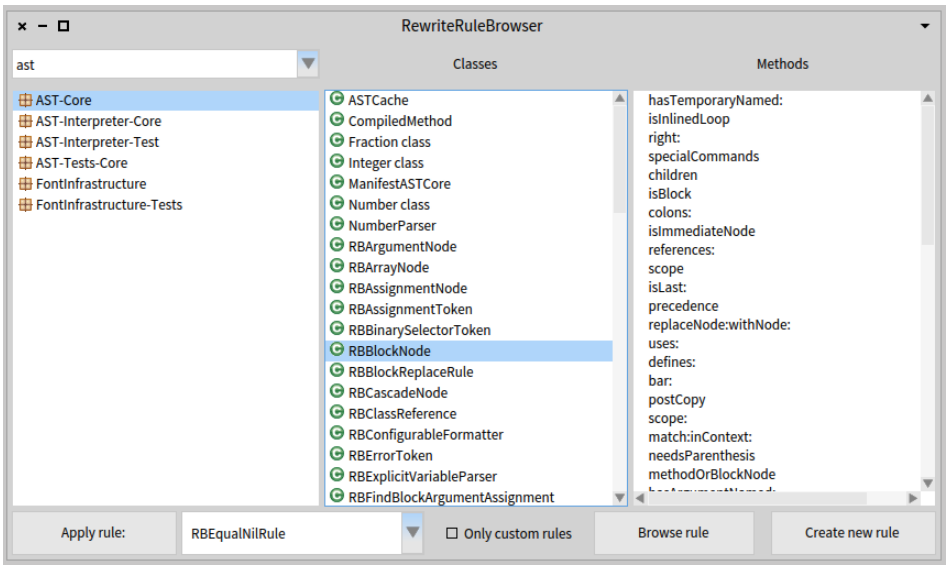

Figure 2. Rewrite Rule Browser

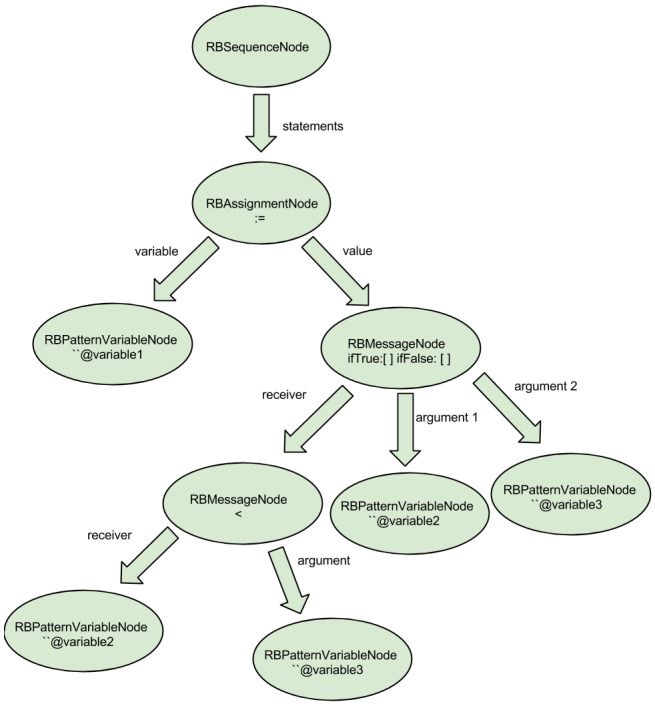

Figure 3. AST representation of listing 6 snippet (LHS)

Another interesting improvement of the tool would be the extension of the graphical user-interface in a way the developer could *draw* the transformation, as in some tools that can be found in the MDE or graph transformation communities. It would imply to integrate a graphical view of the patterns and a way to express an operation and a relation between a source node and a target node. This kind of improvement

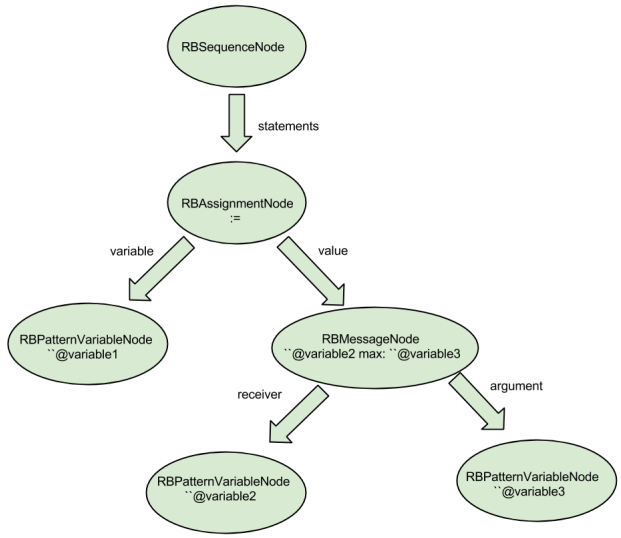

Figure 4. AST representation of listing 7 snippet (RHS)

would not be very useful for a skilled Pharoer but would be interesting for casual programmers, beginners or users from other communities.

5. Conclusion

To conclude, we have presented the *Rewrite Tool* which is a pragmatic tool to assist developers in code transformation activities. The base structure it manipulates through patterns is AST. *Rewrite Tool* helps to automate code transformations by providing an clear user-friendly interface for an efficient but complex tool on which it relies (*Rewrite Engine*). It allows one to express Pharo method transformations that can

be saved for a later reuse. In the future, it will be extended for wider and higher level code transformations. This will allow its use as core tool for program transformations.

References

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: piggybacking rewriting on java. In *Proceedings of the 18th international conference on Term rewriting and applications*, RTA'07, pages 36–47, Berlin, Heidelberg, 2007. Springer-Verlag.

[BDH+01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In Reinhard Wilhelm, editor, *CC'01: Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999. ordered but not received.

[KHVDB+11] Paul Klint, Mark Hills, Jeroen Van Den Bos, Tijs Van Der Storm, and Jurgen Vinju. Rascal: From algebraic specification to meta-programming. In *Proceedings Second International Workshop on Algebraic Methods in Model-based Software Engineering (AMMSE)*, pages 15–32, Zurich, Suisse, 2011.

[KV10] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45:444–463, October 2010.

[KvdSV09] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *SCAM*, pages 168–177, 2009.

[KvdSV11] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY Meta-programming with Rascal. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.

[RBJO96] Don Roberts, John Brant, Ralph E. Johnson, and Bill Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.

[RCD+11] Jonathan Robie, Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton, and J Siméon. *XQuery Update Facility 1.0*. W3C, March 2011.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[VCM+13] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In *ECOOP'13*, 2013.

[W3C99] W3C. *XSL Transformations (XSLT) Version 1.0*, November 1999.