A Formal Pattern Language for Refactoring of Lisp Programs

António Menezes Leitão
Instituto Superior Técnico
Dep. de Engenharia Informática
Grupo de Inteligência Artificial
Av. Rovisco Pais
1049-001 Lisboa
Portugal
Tel: 351-218417641
Fax: 351-218417472
E-mail: aml@gia.ist.utl.pt

Abstract

Refactoring tools are extremely important to prevent errors in legacy systems restructuring. Unfortunately, to be effective, these tools require huge computer resources that cause unacceptable inefficiencies or limit the size of the restructured programs or even the programming language expressiveness. Previous approaches also made it difficult for the programmer to extend the tool with new refactoring operations.

Pattern languages exist that describe several techniques to refactor a program. However, they are usually described at a very abstract level that is understandable only by a human programmer and that is hard to implement in refactoring tools.

We present a formal pattern language that solves these problems. The pattern language is used to define refactoring operations that can be as powerful as is computationally possible but that are very easy to write.

The language simplicity also allows for the automatic learning of new refactoring operations from the observation of manual code transformations.

1. Introduction

Program restructuring is the modification of a program and/or documentation in order to make it easier to understand and modify and less prone to errors caused by maintenance activities [3].

The worst problem in program restructuring is the possibility of introducing errors in a program that is being modified [10]. When the modifications are done manually, it is almost certain that errors will occur. The best way to avoid these errors is to use tools that help the programmer in his restructuring tasks.

In this article we will be particularly concerned with one sub-area of program restructuring named refactoring. Program refactoring is the transformation of a program that reorganizes it without changing its behavior [13].

Preserving a program's behavior is extremely important when the modified program is hard to understand, as is the common situation in most legacy systems. In this case, the consequences of a restructuring operation can be difficult to predict and the preservation of the program's semantics is an assurance to the programmer that no errors will be introduced.

Unfortunately, most current programming languages are expressive enough to make refactoring an ideal rather than a reality. The use of higher order procedures, dynamic calls, dynamic typing, explicit evaluation and many other features makes it impossible to build tools capable of assuring semantic preservation across the entire range of refactoring operations [9].

Current refactoring tools also suffer from not being easy to extend in order to deal with new refactoring operations. Either the tools are not extendable or the extension language is hard to use.

Another problem with previous approaches is that new refactoring operations are usually described at an extremely abstract level, usually using natural language or diagrams [7]. Although the patterns are quite helpful to guide a restructuring task, the description level is so abstract that, in fact, it can only be understood by a human programmer and it is a very difficult task to rewrite those patterns in a way digestible by a refactoring tool.

2. A New Pattern Language

Previous work on refactoring tools have been concerned mainly in providing a basis for some fundamental restructuring operations such as variable renaming, expression abstraction, function extraction, etc. Although these operations are very important, it is also important to automate a large number of other restructuring operations including architecture-changing operations. Unfortunately, it is difficult to provide a fixed set of such operations because the programming techniques are continuously evolving and because there are programming languages, such as Common Lisp [2] or Scheme [16], allowing syntactical extensions that traditional approaches have difficulty to deal with [9].

In this section we present a new approach to refactoring operations that is based on a very simple syntax for the description of transformation rules. We are particularly interested in refactoring programs written in Lisp-like languages and it is our opinion that the syntax of the transformation rules should be as close as possible to the syntax of those languages. We will show that the simple syntax of the transformation rules is no impediment to the development of quite complex transformations.

Another important benefit of the simpler syntax is that it is quite easy for the programmer to extend the refactoring tool with more transformations. This activity can be further simplified by a transformation rule generator that creates new transformations based on some manual restructuring operation.

2.1. Transformation Rules

Transformation rules are made of an antecedent pattern and a consequent pattern. When the antecedent pattern matches a code fragment, its pattern variables become bound to the corresponding sub-fragments. In this case, the matched fragment is replaced by an instance of the consequent pattern with its pattern variables replaced with the corresponding sub-fragments they are bound to.

Here is a simplified example of a transformation rule:

```
(deftransform
    (cond (?test ?conseq)
          (t ?altern))
    (if ?test ?conseq ?altern))
```

In the previous rule, all symbols preceded by a question mark are pattern variables that can match anything. All the other elements must match with equal elements in a code fragment. The previous rule states that whenever we have a code fragment that corresponds to a `cond` form with one normal clause and one default clause, and with one result expression on each clause, we can transform it into an `if` form, with `?test`, `?conseq` and `?altern` sub-fragments obtained from the corresponding elements in the `cond` form. Note that, in general, multiple rules match a given code fragment. All the matching rules are presented to the programmer and it is up to him to choose the one he wants to apply.

Preferring an `if` in place of a `cond` is a matter of taste. Sometimes we prefer the opposite. To this end, some transformations are bi-directional, that is, they can be used to transform the antecedent in the consequent or vice-versa. These rules are, in fact, equivalence rules. Here is an example of such rule that relates the use of five Common Lisp functions:

```
(defequivs ((cadr ?x)
            (car (cdr ?x))
            (second ?x)
            (nth 1 ?x)))
```

The previous rule states that any code fragment that matches one of the patterns can be replaced by any of the remaining patterns. The equivalence rule is just a compact way to express several transformation rules, one for each pair of patterns. This is a powerful shortcut to avoid an explosion of transformation rules.

Our pattern language distinguishes two types of patterns: 1) those that can occur on the antecedent position of a transformation rule and 2) those that can occur on the consequent position. Each type of pattern has distinct expressiveness:

- Patterns in antecedent position are intended to match code fragments and they are described using a pattern language similar to the one described on [19] and allowing segment variables, conjunction and disjunction of patterns, predicates over pattern variables, etc.

- Patterns in consequent position are intended to serve as templates to new code fragments, using variable substitution to instantiate the template. These patterns are not intended to match and, in fact they could be replaced by a more traditional approach such as *quasiquotation* [4]. However, since most consequent patterns can also be used in antecedent position, it is preferable to be consistent and use the same syntax. On the other hand, it is useful to be able to operate more complex substitutions on the patterns in consequent position and this asks for more expressiveness in the pattern language.

These different requirements for the pattern language suggest that we divide it into three parts: a common core that can be used for equivalence rules, that is, rules where patterns can swap position, and two extensions, one for patterns in antecedent position and one for patterns in consequent position.

2.2. Patterns in Equivalence Rules

The pattern language used in equivalence rules can be described in BNF notation as follows:

```
<pattern>      ::=<variable>
                 |<atom>
                 |<simple pat>
                 |<segment pat>
                 |(<pattern> . <pattern>)
<patterns>     ::=ϵ
                 |<pattern> <patterns>
<simple pat>   ::=(?is <variable> <predicate>)
<segment pat>  ::=((?* <variable>) <patterns>)
                 |((?+ <variable>) <patterns>)
                 |((?? <variable>) <patterns>)
<variable>     ::=?<symbol>
```

The meaning of the operators is the following:

(?is <***variable***> <***predicate***>**)** Matches <*variable*> with a code fragment as long as it satisfies the <*predicate*>. The predicate is ignored when the pattern is in consequent position.

((?* <***variable***>**)** <***patterns***>**)** Matches <*variable*> with zero or more code fragments (collecting them on a list) and the remaining patterns with the remaining code fragments.

((?+ <***variable***>**)** <***patterns***>**)** Matches <*variable*> with one or more code fragments (collecting them on a list) and the remaining patterns with the remaining code fragments.

((?? <***variable***>**)** <***patterns***>**)** Matches <*variable*> with zero or one code fragment (collecting it on a list) and the remaining patterns with the remaining code fragments.

The following rule is an example of the transformations allowed with this pattern language:

```
(defequivs ((cond (?+ ?clauses)
                  (t (if ?test
                         ?conseq
                         ?altern)))
            (cond (?+ ?clauses)
                  (?test ?conseq)
                  (t ?altern))))
```

The previous rule transform a `cond` form with an `if` form in the default clause into a `cond` form with an extra clause and vice-versa.

2.3. Patterns for Antecedents

When a pattern is to be used exclusively in antecedent position in a transformation rule the pattern language is extended in the following way:

```
<simple pat>   ::=(?and <pattern> <patterns>)
                 |(?or <pattern> <patterns>)
                 |(?not <pattern> <patterns>)
<segment pat>  ::=((?* <variable>) <patterns>)
                 |((?+ <variable>) <patterns>)
                 |((?? <variable>) <patterns>)
                 |((?if <expression>) <patterns>)
```

The meaning of these additional operators is the following:

(?and <***pattern***> <***patterns***>**)** Matches when all patterns match.

(?or <***pattern***> <***patterns***>**)** Matches when one of the patterns matches. The patterns are matched from left to right.

(?not <***pattern***> <***patterns***>**)** Matches when none of the patterns match.

((?if <***expression***>**)** <***patterns***>**)** Matches when the <*expression*> is true and the remaining patterns match the remaining expressions.

The default clause of a Common Lisp form `case`, `ecase`, or `ccase` is one example where the extended pattern language is needed. The default clause either has the form (`t` $form_1$ `...` $form_n$) or the form (`otherwise` $form_1$ `...` $form_n$). To match either case we need the pattern:

```
((?or t otherwise) (?* ?actions))
```

2.4. Patterns for Consequents

There are two features that are very useful to have in consequent patterns: arbitrary computations and cascading patterns.

The first one allows us to dynamically compute the transformation, the second one is used to refine transformations. The added grammar is:

```
<simple pat>   ::=(?map <variable> (<sub-rules>))
                 |(?funcall <name> <terms>)
<sub-rules>    ::=ϵ
                 |<sub-rule> <sub-rules>
<sub-rule>     ::=(<antecedent> <consequent>)
<terms>        ::=ϵ
                 |<term> <terms>
<term>         ::=<variable>
                 |<constant>
```

The symbols <*antecedent*> and <*consequent*> correspond, respectively, to the <*pattern*> described in section 2.3 and section 2.4. A <*constant*> is any Lisp literal. We will now discuss these extensions.

2.4.1. Cascading Patterns

Consider the following example of a function *f* applied to a `cond` expression:

```
(f (cond (test1 act11 ... act1n result1)
         (test2 act21 ... act2p result2)
         ...
         (testk actk1 ... actkl resultk)))
```

One possible restructuring operation is to move the *f* function application to each one of the k `cond` branches, applying it to each $result_i$, i.e.,

```
(cond (test1 act11 ... act1n (f result1))
      (test2 act21 ... act2p (f result2))
      ...
      (testk actk1 ... actkl (f resultk)))
```

Without extra syntax it is impossible to write the transformation rule because it is necessary to transform an unknown number of `cond` clauses. We can overcome this problem using a cascaded pattern.

A cascaded pattern is a sequence of pairs of patterns that is mapped over a list. For each element of the list we try to match the first element of each pair of patterns, in sequence. When we have a match, the second element of the pair provides the template to build a new code fragment that is used in place of the matched element. If, for some element of the list, no pattern matches, the transformation is not applicable.

A cascading pattern allows the creation of *local* transformation rules, applicable only when certain conditions are met. Each sub-pattern can use all the expressive power of the transformation language, including the `map` operator, thus allowing an hierarchy of transformations contexts.

Here is the (simplified) rule for the previous restructuring operation:

```
(deftransform
    (?f (cond (?+ ?clauses)))
    (cond
     (?map ?clauses
      (((?test (?* ?acts) ?result)
        (?test (?* ?acts) (?f ?result)))))))
```

In this rule, the antecedent matches any `cond` form. Each `cond` clause is then matched with the cascaded pattern and the sub-transformation is applied. Note that inner patterns can refer to variables bound on outer patterns.

Here is another example that converts a function whose body consists of a `typecase` or `etypecase` form into CLOS methods [15]:

```
(deftransform
  (defun ?x1 ((?* ?pre-args)
              ?x2
              (?* ?pos-args))
    ((?or typecase etypecase) ?x2
     (?+ ?clauses)))
  (progn
    (?map ?clauses
     (((?type (?* ?actions))
       (defmethod ?x1 ((?* ?pre-args)
                       (?x2 ?type)
                       (?* ?pos-args))
         (?* ?actions)))))))
```

The previous transformation is useful to convert from traditional procedural-based programs into modern object-oriented programs.

2.4.2. Computation

A computation operation is just a bypass to the simpler template-based consequent that allows us to dynamically compute the code fragment after transformation. Let's consider the following illustrative example:

```
(deftransform
  (+ (?is ?x numberp) (?is ?y numberp))
  (?funcall + ?x ?y))
```

This transformation rule is applicable to any expression that adds two numbers. In this case, the result of the rule application is the replacement of the expression with the sum of the numbers.

In practice, the computation is used when part (or all) of the resulting code isn't constant nor derives directly from the transformed code via pattern matching. Here is another example where the rule application must interact with the programmer to require additional information:

```
(deftransform
  ?form
  (progn
    (format *trace-output*
            "Reached point ~A"
            (?funcall read-from-user
                      "Trace point name?"))
    ?form))
```

This transformation is applicable to any expression and is used to insert trace points in the code. Each trace point prints a name that is given by the programmer during the rule application.

The computation is fundamental to certain refactoring operations that must analyze the transformed code and provide any additional semantic-preserving code. For example, a function extraction operation needs to know which are the free variables of a code fragment so that it can use them as function parameters. Here is a (very simplified) example:[1]

[1] A much more complete example is presented in [17].

```
(deftransform
  ?form
  ((?funcall read-from-user
             "Function name?") .
   (?funcall free-variables
             ?form))
```

To see the rule in action, let's suppose the programmer applies it to the expression `(+ 1 x 2 y)` and, when asked regarding the new function's name, he answers `foo`. Then, the expression is replaced by a new expression `(foo x y)`.

The previous examples have shown that we sometimes need transformation rules where some of the elements of the resulting code must be provided by the programmer. This situation is sufficiently frequent to justify a simpler form to specify it. Moreover, it is also frequent that the provided elements must appear on several distinct places of the resulting code (for example, at a function definition *and* at a function call). In this case, it is not only annoying but also error prone to force the programmer to provide that same information over and over again.

To solve this problem, the consequent of a transformation can use a special marker to identify pattern variables that will be bound to some information provided by the programmer. These pattern variables have a distinguishing `$` marker in the beginning in place of the `?` marker.

As an example, we present a transformation rule that converts simple recursive processes into iterative processes. Note that the consequent of the rule adds two new names, namely `$iterate` and `$accum` that will be asked to the programmer and will be systematically replaced in the resulting code.

```
(deftransform
    (defun ?name (?arg)
      (if ?base-case
        ?base-case-value
        ((?is ?combiner associative?)
         ?expr
         (?name ?rec-expr))))
    (defun ?name (?arg)
      (labels (($iterate (?arg $accum)
                 (if ?base-case
                   $accum
                   ($iterate
                     ?rec-expr
                     (?combiner ?expr
                                $accum)))))
        ($iterate ?arg ?base-case-value))))
```

3. Learning Transformations

Most of the rules presented above result from repeated use of some sequence of manual restructuring operations. It is the tediousness of these repetitions that suggests the programmer that it is useful to add a new transformation to the system so that he can apply the entire transformation in a single operation.

As a very simple example, consider one common mistake that occurs in Lisp code: testing that a list *`lst`* contains a single element using the `length` function in an expression such as `(= (length lst) 1)`. This test is a waste of time for any list with more than one element. If we admit the existence of a `singleton?` predicate, we can replace any occurrence of the previous expression with the simpler (and more efficient) `(singleton? lst)`.

This is one example of a rule that the programmer might want to add to the already existing set of transformation rules. Although, in this case, the rule is extremely simple, it is tedious to have to define it (but less tedious than *not* define it and be forced to apply the refactoring manually).

To solve this problem, our refactoring tool can induce a transformation rule from a single example of manual refactoring. To understand this, let's look more carefully at one example of a code fragment before and after a manual refactoring operation:

```
(defun sneps-simple-entailment? (node)
  (and (= 1
          (length (get-nodes node 'ant)))
       (= 1
          (length (get-nodes node 'cq)))))
```

```
(defun sneps-simple-entailment? (node)
  (and (singleton? (get-nodes node 'ant))
       (= 1
          (length (get-nodes node 'cq)))))
```

The inverted square shows the position of the editing cursor before and after the manual refactoring. Although the refactoring was done manually, the expressions before and after the refactoring were recorded by the refactoring tool that will propose the following rule:

```
(deftransform
  (= 1 (length ?x1))
  (singleton? ?x1))
```

As we can see, the rule correctly abstracts the manual operations done by the programmer, independently of the particular sequence of editing operations that he has done. Now, using the proposed rule, the programmer can just position the editing cursor before the expression on the next line and apply the rule to immediately obtain the intended refactoring.

Being able to induce a rule from a single example is a very desirable feature but the programmer must understand that there are several possible rules that replicate the given example. To minimize this problem the rule generator is specially tailored to generate "good" rules. These rules are "good" in the sense that they have deep knowledge of the programming language in use so that they can understand the transformed code. In particular, they understand the difference between normal and special forms and between required and optional arguments in function calls.

The idea behind rule generation using just two expressions e_1 and e_2 is to build a rule $p_1 \Rightarrow p_2$ whose application to the first expression e_1 results in an expression equals to e_2. Given the fact that in a transformation rule both the antecedent and the consequent must have the same set of variables, the matching between p_1 and e_1 must generate the same set of substitutions as the matching between p_2 and e_2. The key to the rule generation is then to find an appropriate set of substitutions. The *inverse* application of that set to e_1 and to e_2 creates the patterns p_1 and p_2 needed to define the rule.

Usually, there are several sets of substitutions that are appropriate. In fact, its number grows exponentially with the number of nodes of the abstract syntax tree of the first expression. In [17] we present an algorithm that creates a stream of substitutions sets that is ordered according to several heuristics: 1) substitutions for special operators or macro operators aren't good because these operators have special semantics and can't be easily replaced, 2) substitutions for arguments must take into account the number of required and optional parameters of each function or macro operator, 3) maximization of the number of nodes of the substitutions for abstracted syntax trees, and 4) minimization of the number of substitutions in the set. This stream of substitutions sets is then presented to the programmer, one at a time, until he sees one that he thinks is the most appropriate. In practice, the first one is usually the best one.

4. Conclusion and Related Work

Program transformation is a vast research area that studies how to transform a program into another program. This transformation can be done (1) between programs written in two different languages or (2) between programs written in the same language.

In the first case, the transformation process might be carried from high- to low-level programming languages, in a process usually known as compilation [1], or from formal specifications to high-level programming languages, in a process usually known as program synthesis [21], or in the reverse direction, from machine language to higher-level languages, known as decompilation [6] or from high-level languages to specifications, known as reverse engineering [5].

In the second case, the transformation process aims at rewriting the program in a different form but using the same programming language. When the goal is to optimize the program the process is named program optimization [1], when the goal is to improve a program's design to increase its understandability and maintainability, the process is named program refactoring [8]. In the present article we are particularly interested in this last type of program transformation.

Program refactoring is a technique that transforms a program without changing its meaning. This is of utmost importance when we deal with legacy code because it is extremely difficult to predict the impact of changes. Refactoring solves this problem by avoiding changes that might affect program behavior. Griswold [9] presented the theory behind several refactoring transformations, demonstrating its use on programs written in a subset of the Scheme language. Opdyke [20] did the same for object-oriented languages such as C++. Fowler [8] presents a catalog of refactoring transformations but these are described in an informal way and thus it is necessary to formalize them in order to build tools that automate its use. Several tools [23, 12, 14] attempt to do that.

Unfortunately, these refactoring tools suffer from not being easy to extend in order to deal with new refactoring operations, specially because most refactoring operations are described in very abstract ways and are difficult to implement in machine-understandable form. One way to solve this problem is to use pattern languages to formally describe the refactoring operations, so that they can be used by refactoring tools. In [11, 22], graph rewriting rules are used to transform the system's architecture. Although the rules can be automatically applied, they operate on the system's architecture, i.e., a high-level description of the system. Other approaches [9, 18] focus on low-level refactoring operations, such as function extraction or variable renaming.

What we think is missing is a tool that includes transformations in the entire spectrum of refactoring operations, from low-level transformations to architectural changes. Moreover, this tool must be flexible enough to be easily extended by the programmer. This suggests the use of simple pattern-based refactoring rules.

In this article we presented a new pattern language for refactoring operations. The pattern language is specially adapted to deal with programs written in Lisp-like languages, using a very similar syntax that is immediately understood by the Lisp programmer.

The pattern language is based on template matching using pattern variables to match arbitrary code fragments. The pattern language also includes several operators that allow for sophisticated pattern matching, including segment variables, disjunction, conjunction, arbitrary predicates, arbitrary computations and cascaded patterns, among others. This results in a simple yet very expressive pattern language. On one end of the expressiveness range we can design a transformation rule using a pattern that match anything but that uses an arbitrarily complex predicate to check any necessary pre-conditions and that computes the resulting code using some arbitrarily complex code construction. On the other end of the range we can find pure pattern-matching syntactical transformation. In between, there's a

huge range of transformations that can be described using a mixed approach of pattern-matching and more complex tests and code building operations.

Another good feature of our pattern language is that it is possible to automatically generate a transformation rule from a single example of a refactoring operation. The generated rule can then be extended by the programmer using more expressive pattern matching operators.

Our pattern language is used at the core of the **C**^{3}**PO** (***C**areful **C**ode **C**onversion to **P**revent **O**bsolescence*) refactoring tool. The tool was extensively used in the refactoring of SNePS [24], a large legacy system still in use in the Artificial Intelligence area.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[2] ANSI and ITIC. *American National Standard for Information Technology: programming language — Common LISP*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1996. Approved December 8, 1994.

[3] R. Arnold. *Tutorial on Software Restructuring*. IEEE Society Press, Washington, DC, 1986.

[4] A. Bawden. Quasiquotation in Lisp. In *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy, San Antonio, January 1999.*, pages 4–12, Jan. 1999.

[5] E. J. Chikofsky and J. H. Cross, II. Reverse engineering and design recovery: A taxonomy. In R. S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[6] C. Cifuentes and K. J. Gough. A methodology for decompilation. In *Proceedings of the XIX Conferencia Latinoamericana deInformatica*, pages 257–266, Buenos Aires, Argentina, Aug. 1993.

[7] S. Demeyer, S. Ducasse, and S. Tichelaar. A pattern language for reverse engineering. In P. Dyson, editor, *Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999*, Konstanz, Germany, July 1999. UVK Universitätsverlag Konstanz GmbH.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[10] W. G. Griswold and D. Notkin. Computer-aided vs. manual program restructuring. *ACM SIGSOFT Software Engineering Notes*, 17(1):33–41, January 1992.

[11] J. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. In S. Demeyer and H. Gall, editors, *Proceedings of the ESEC/FSE Workshop on Object-Oriented Re-engineering*. Technical University of Vienna, Information Systems Institute, Distributed Systems Group, Sept. 1997. Technical Report TUV-1841-97-10.

[12] jFactor. http://www.instantiations.com/jfactor/.

[13] R. E. Johnson and W. F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, Nov. 1993.

[14] JRefactory. http://jrefactory.sourceforge.net/.

[15] S. E. Keene. *Object-Oriented Programming in Common Lisp: A programmer's guide to CLOS*. Addison-Wesley Publishing Company, Cambridge, MA, 1989.

[16] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998. With H. Abelson, N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, and M. Wand.

[17] A. M. Leitão. *Reengenharia de Programas*. PhD thesis, Instituto Superior T´ecnico, Universidade T´ecnica de Lisboa, 2000.

[18] J. Morgenthaler. *Static Analysis for a Software Transformation Tool*. PhD thesis, University of California, San Diego, Dept. of Computer Science and Engineering, 1997. Tech. Report CS97-552.

[19] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, California, 1992.

[20] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992.

[21] H. A. Partsch. *Specification and Transformation of Programs - a Formal Approach to Software Development*. Springer, Berlin, 1990.

[22] A. Radermacher. Support for design patterns through graph transformation tools. In *AGTIVE*, pages 111–126, 1999.

[23] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[24] S. C. Shapiro and W. J. Rapaport. The SNePS family. *Computers & Mathematics with Applications*, 23(2–5):243–275, Jan.–Mar. 1992.