First Results in Supporting Automated Elaboration of Refactoring Plans

Javier Pérez, Yania Crespo

Universidad de Valladolid
(jperez,yania)@infor.uva.es

1 Introduction to Refactoring Plans

Current approaches to refactoring automation [1] are focused on finding and applying each individual transformation step. When decided which and where a refactoring opportunity exists, development tools are used to automatically apply the refactoring and test its effects. This kind of methodologies are directed to improve certain design characteristics. In some cases deeper redesigning is needed because wider architectural changes are planned. Other refactoring methodologies must be used. Refactorings must be directed to a full system new design proposal. Starting a framework based product-line from legacy software systems is a good example of this.

A sketch of a desirable system design can be obtained from a wide variety of techniques such as Formal Concept Analysis, metrics [2], or just the proposal of skilled software designers. This work assumes that this sketch of the desirable system design exists and explores how to check if the new design could be derived from the original system using an available set of defined refactorings. A small example is presented to show how the full transformation step sequence could be obtained in case it exists.

The transformation sequence is understood as a chain of refactorings that builds up a "Refactoring Plan". This work aims to support the automation of this refactoring process which can be overviewed in Figure 1.

2 Refactoring Plans Issues

The problem of finding the transformation steps that turn a state into another, is seen as a formal language problem. Representing software, specially its inner structural relationships, is straightforward with graphs, so graphs, Graph Grammars (GG) and Graph Rewriting Systems (GRS) seems a natural way to formalize the problem.

The Refactoring Plan can be worked out with a graph grammar where the software system is the starting node, refactorings are the production rules and the desirable design is a “word” formed by the available symbols (classes, fields, methods, . . .). While the research being at a first stage, this “word” is expected to be a new design, where the Refactoring Plan has already been executed, that includes the whole amount of information (all elements) from the current one.

To generate Refactoring Plans automatically, two problems needs to be solved:

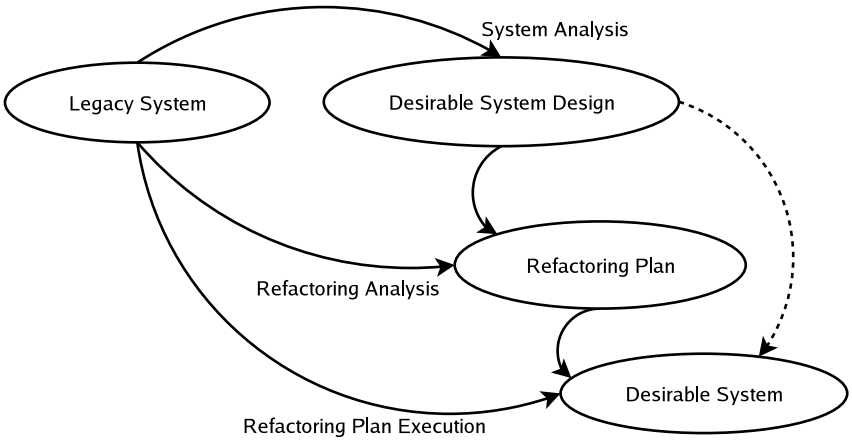

Fig. 1. Overview of the elaboration of Refactoring Plans

- **Validating the desirable design:** to check whether it could be derived or not from the original system, using the available set of defined refactorings. This problem can be seen as the membership problem of formal language theory.
- **Obtaining the Refactoring Plan:** Find the sequence of refactorings that must be performed to redesign the system. This is like obtaining the derivation sequence.

3 Formalizing the problem with Graph Transformations

To formalize refactorings, a formal graph representation of Object-Oriented (OO) software is needed. It must support elements of OO paradigm (classes, fields, methods, ...) and their structural relationships. In order to represent refactorings below method signature level, also method bodies must be represented. The chosen formalism [3], presents a GRS based on directed typed graphs which is already aimed at formalization of refactorings. This formalism defines a representation of refactorings as graph transformation rules. As an important feature related to refactorings, it is designed to allow representation of behavior preserving invariants too. Syntactic correctness of the generated graphs are assured with the use of constraints and well formedness rules. To allow representation of refactoring transformations which must appear in a specific context to be performed, it supports the specification of context-sensitive rules through the definition of negative application conditions, node embeddings and the use of path expressions. The type graph of this proposal, a graph defining the syntactically correct graphs, is shown in figure 2.

An approach to the membership problem and to the finding of the derivation sequence, is searched in the world of GGs. A key goal of this work is finding a kind of GG for which the membership problem is solved and define refactoring transformations, with the other chosen formalism [3], within this grammars. A possibly adequate GG family is defined by Layered Graph Grammars [4] (LGGs), and their extension

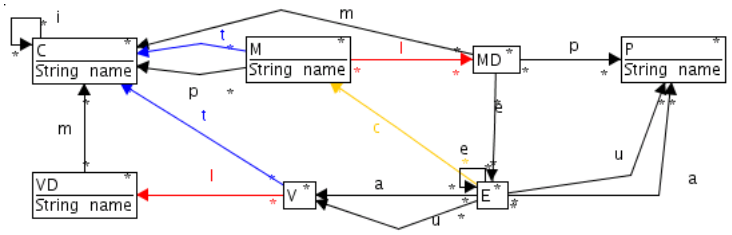

Fig. 2. Type Graph

Contextual Layered Graph Grammars [5] (CLGGs). In CLGGs, the set of available rules are classified in different subsets of ordered layers. To parse a graph, rules are restricted to be applied in the order given by the layers. The membership problem can be managed mainly because application of rules layer by layer reduce the graph and therefore, the space of search is also reduced in each layer step. Rules of a greater layer can only be applied after no more rules of a lower layer can be applied, this assures that the set of applicable rules is also being reduced with each transformation.

The approach presented in this work is inspired by the way that CLGGs behave. A set of refactorings are ordered and assigned different layers. To be sure that these transformations lead to a some kind of reduction of the graph, “normal” refactorings are decomposed into micro-refactorings, based on Opdyke’s low-level refactorings [6]. Micro-refactorings perform small behavior preserving operations such as creation, deletion, renaming and moving of classes, fields and methods. Micro-refactorings are layered to state an application order. To avoid infinite application of canceling micro-refactorings, for example, moving a method back and forth between two classes, some restrictions are applied. For example, a micro-refactoring that moves a method can only be applied once on that method.

4 Example

To explore and explain this approach a simple example is shown. A few rules have been defined, and a small system, with its original and its desirable design, is presented. A graph parser must be fed with rules and both designs, then it will find the rule sequence to transform a design into another.

A small set of rules that perform some micro-refactorings are defined first, represented with the formalism of [3]. A key feature of this rule set is that each rule must create, move or delete a single element of a specific type.

- **Create unreferenced empty Class:** As seen in fig. 3, a new class is introduced in the system if it doesn’t exist.
- **Create unreferenced empty Method:** This rule in fig. 4, a new method is introduced within a class if it doesn’t contain a method with the same signature.
- **Move method to empty method:** As defined in fig. 5, a method body can be moved from a method to another if the target method doesn’t have a body itself. Micro-refactorings have been defined to perform a single step transformation, so, in this

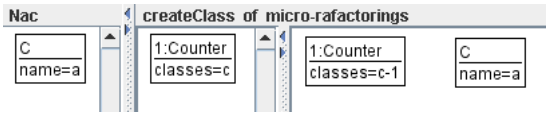

Fig. 3. Create Class

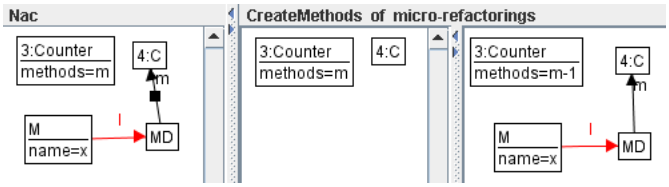

Fig. 4. Create Method

case, the source method signature is preserved and the source body method is replaced by a call to the target method. Another rule would be needed to remove this delegating methods, but this set of rules will be enough for the purpose of the example.

Some mechanisms are established in order to progressively reduce the applicable set of rules. Specific counters are defined for each rule and are used to control the number of times a rule can be applied. Counters are initialized with the estimated times the designer consider that the rule will be applied. Each rule application decrements the corresponding counter, and can only be performed if the counter is above zero. Rules are ordered with layers with the following guidelines:

- **Create new elements:** First of all, every element of the desirable design which doesn't appear on the source design must be created. Classes are created first, then methods and fields are created last.
- **Move elements:** Elements are moved within the system with no specific precedence over class, methods of fields.
- **Delete empty and elements:** Finally, empty and unreferenced elements can be deleted. Methods and fields are removed first and Classes are removed last.

With this principles, the rule set of the example can be ordered: *Create Class* is assigned **layer 0**, *Crete Method* is assigned **layer 1**, and *Move Method* is classified in **layer 2**.

The system to be redesigned is presented in figure 6. It consist just of three classes and two methods within each class.

The desirable design of the system, in figure 7, shows up that a new class *Connect* has been created and that a couple of methods, *protocol* and *connection*, have been moved. Counters are initialized to allow application of: **1 Create Class rule**, **2 Create Method rules**, and **2 Move Method rules**.

Figure 8 presents the initial state of the system once it has been represented with the formalism of [3]. Successful application of rules by the parser will lead to the creation of the new class, as showed up in figure 9, the creation of the two new methods (see fig.

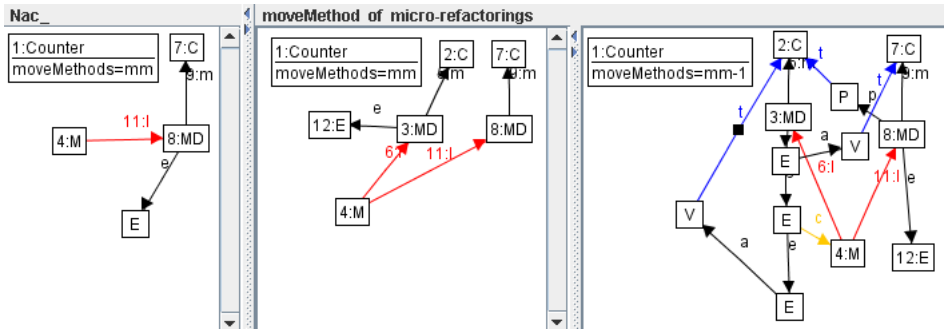

Fig. 5. Move Method

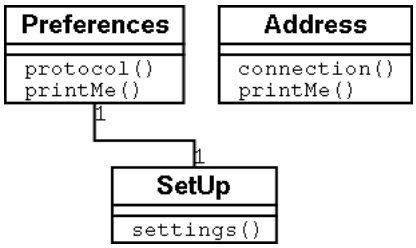

Fig. 6. Start Design

10) and the reallocation of the two moved method bodies within the recently created empty methods (see fig. 11).

5 Results and Future Work

This work present the first results to support the automated elaboration of Refactoring Plans. The main requirements of the problem have been stated. Small examples are used to explore the formalisms that can be applied to support this process. It has been found that graph transformations offer good mechanisms to be used as the relying formalism. To represent software design and specially software refactorings, the formalism presented by [3], can successfully fulfill our requirements. The refactoring plan problem has been identified as the membership problem, which seems to be a promising approach, because graph grammars for which the problem has already been solved, can be used.

Next steps include extending the set of refactorings supported. It must be checked if it is true that every refactoring can be represented as a chain of micro-refactorings. It must be proven that composing micro-refactorings is equivalent to executing “normal” refactorings. The interpretation of the proposed technique as a CLGGs must be formalized too. Validating the process through tools like AGG [7], FUJABA [8] or PROGRES [9], will be a main issue to support automated elaboration of Refactoring Plans. AGG

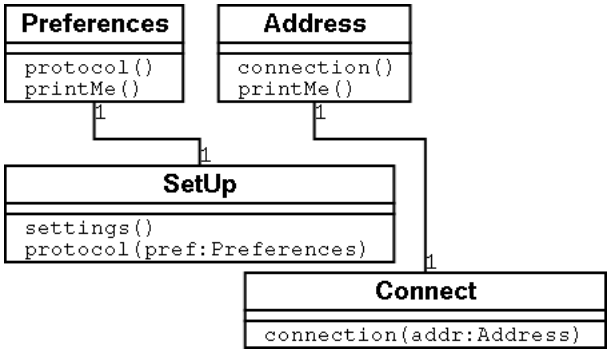

Fig. 7. Desirable Design

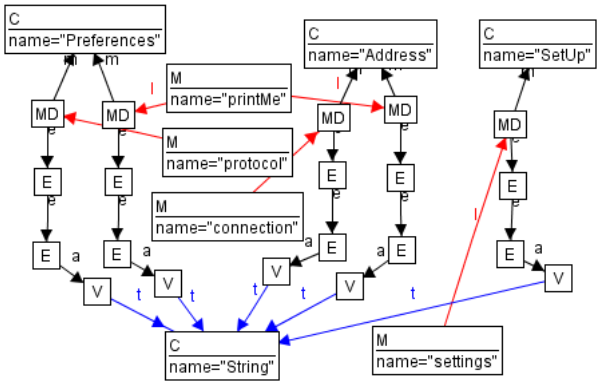

Fig. 8. Initial state of the system. No rules have been applied yet.

seems to be the right tool to use, because it directly supports graph parsing. If needed, extensions to available tools must be developed.

References

1. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30** (2004) 126–139
2. Crespo, Y., López, C., Manso, E., Marticorena, R.: From bad smells to refactoring, metrics smoothing the way. In: Object-Oriented Design Knowledge: Principles, Heuristics, Best Practices, ... Idea Group Inc (2005) xx to appear.
3. Mens, T., Eetvelde, N.V., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations: Research articles. J. Softw. Maint. Evol. **17** (2005) 247–276
4. Rekers, J., Schurr, A.: Defining and parsing visual languages with layered graph grammars. Journal of Visual Languages and Computing **8** (1997) 27–55
5. Bottoni, P., Taentzer, G., Schrr, A.: Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In: VL '00: Proceedings of the 2000

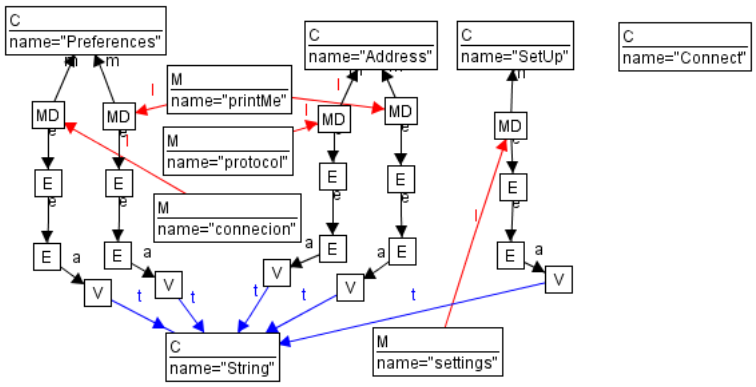

Fig. 9. Class Created

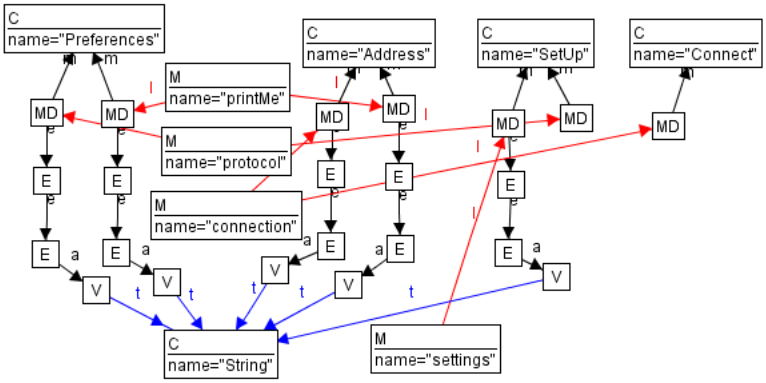

Fig. 10. Method Created

IEEE International Symposium on Visual Languages (VL'00), Washington, DC, USA, IEEE Computer Society (2000) 59

6. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (1992) also Technical Report UIUCDCS-R-92-1759.
7. Taentzer, G.: Agg: A graph transformation environment for modeling and validation of software. In: AGTIVE. (2003) 446–453
8. Nickel, U., Niere, J., Zündorf, A.: The fujaba environment. In: ICSE. (2000) 742–745
9. Münch, M.: Programmed graph rewriting system progres. In: AGTIVE. (1999) 441–448

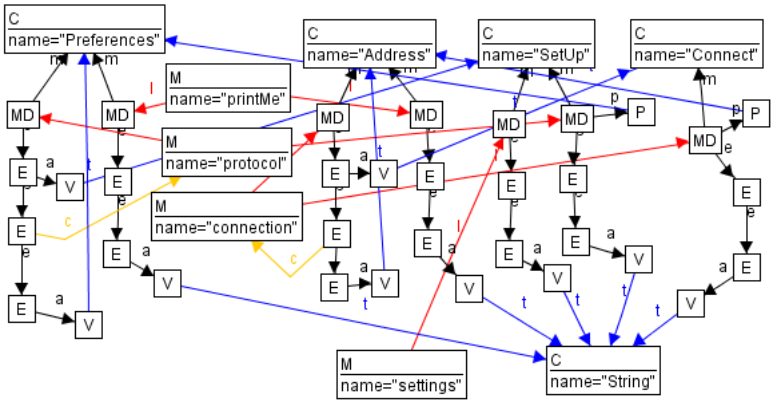

Fig. 11. Methods Moved