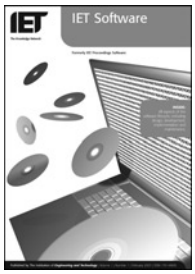
Published in IET Software
Received on 2nd April 2007
Revised on 24th April 2008
doi: 10.1049/iet-sen:20070033



ISSN 1751-8806

Conflict-aware schedule of software refactorings

H. Liu[1,2] *G. Li*[2] *Z.Y. Ma*[2] *W.Z. Shao*[2]

[1]*School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100871, People's Republic of China*
[2]*Software Institute, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, People's Republic of China*
E-mail: liuhui2005@gmail.com

Abstract: Software refactoring is to restructure the internal structure of object-oriented software to improve software quality, especially maintainability, extensibility and reusability while preserving its external behaviours. According to predefined refactoring rules, we may find many places in the software where refactorings can be applied. Applying each refactoring, we may achieve some effect (quality improvement). If we can apply all of the available refactorings, we can achieve the greatest effect. However, the conflicts among refactorings usually make it impossible. The application of a refactoring may change or delete elements necessary for other refactorings, and thus disables these refactorings. As a result, the application order (schedule) of the available refactorings determines which refactorings will be applied, and thus determines the total effect achieved by the refactoring activity. Consequently, conflicting refactorings had better be scheduled rationally so as to promote the total effect of refactoring activities. However, how to schedule conflicting refactorings is rarely discussed.In this paper, a conflict-aware scheduling approach is proposed. It schedules refactorings according to the conflict matrix of refactorings and effects of each individual refactoring. The scheduling model is a multi-objective optimisation model. We propose a heuristic algorithm to solve the scheduling model. We also evaluate the proposed scheduling approach in non-trivial projects. Evaluation results suggest that refactoring activities with the scheduling approach lead to greater effect (quality improvement) than refactoring activities without explicit scheduling.

1 Introduction

Software refactoring is to restructure the internal structure of object-oriented software to improve software quality, especially maintainability, extensibility and reusability [1]. According to new requirements, software is modified again and again (software evolution). As a result, the source code (and the models) is jumbled, and the quality of the design is reduced [1]. Furthermore, the poor quality makes further adaption difficult and expensive. As an attempt to solve this problem, software refactoring was proposed [2]. Object-oriented software (source code) is refactored to improve its quality and facilitate further adaption and extensions. Source code refactoring is so successful that people cannot resist trying to apply refactoring to other software artifacts. Up to now, it has been applied to almost all types of software artefacts, including source code, designs and software requirements [1]. Software refactoring not only improves software quality by restructuring its internal structure, but also guarantees that the external behaviours of software are preserved. So, software refactoring is an effective and relatively safe means to improve software quality [3]. It is accepted as an important activity in software development and maintenance. In eXtreme Programming (XP) [4] and other agile software development processes, refactoring is one of the core activities [5, 6]. In order to facilitate further iterations, we had better refactor software products to improve their extensibility before the next iteration begins. With the popularity of refactoring, more and more CASEs and IDEs, such as Eclipse [7] and Visual Studio [8], begin to provide support for software refactoring. Professional refactoring tools, such as Refactoring Browser [9], are also developed. Tool support is crucial for the successful application of refactoring activities [10].

Experts have defined many refactoring rules according to their experience [3, 11]. For software developers, they just try to find places in software where refactorings should be applied according to the refactoring rules defined by experts. Since refactorings are helpful in improving software quality, developers of course hope to apply refactorings in all the found places so as to achieve maximal quality improvement. However, conflicts among refactorings often make it impossible.

Software refactorings may conflict with each other because the application of one refactoring may change or delete elements that are necessary for other refactorings, and thus disables these refactorings [12]. Conflicts among refactorings may be asymmetrical or symmetrical [12]. If a refactoring can be applied before but not after another refactoring, there is an asymmetrical conflict. If two refactorings can not be applied together in any order, the conflict is symmetrical. Conflicts among refactorings can be detected automatically.

Since the application of some refactorings may disable others, the application order (schedule) of the available refactorings determines which refactorings can be applied (and others will be disabled by these refactorings), and thus determines the total effect of the refactoring activity. So, in order to maximise the total effect (quality improvement), the conflicting refactorings better be scheduled rationally. We should manage an appropriate application order to apply more refactorings, especially the most effective (important) refactorings.

However, how to schedule refactorings is rarely discussed [13]. In this paper, we propose a conflict-aware scheduling approach to schedule conflicting refactorings. The goal of the scheduling approach is to improve software quality as much as possible by arranging an appropriate application order for the available conflicting refactorings.

The rest of this paper is structured as follows. Section 2 presents a motivating example to illustrate the necessity to schedule conflicting refactorings. Section 3 introduces some concepts and technologies related to software refactoring and scheduling of refactorings. Section 4 proposes a conflict-aware scheduling approach. Section 5 presents evaluation of the proposed scheduling approach. Section 6 discusses related work. Section 7 makes a conclusion.

2 Motivating example

We take two refactoring rules from [11] into consideration.

1. Delete Dead Field: If a field of a class is not used by the class or any of its subclass, the field is a dead field which should be deleted [11, Chapter 5].

2. Pull Up Field: If all subclasses of class *A* contain the same field *f*, the field should be pulled up into the superclass *A* [11, Chapter 11].

Suppose that the two refactoring rules have been formalised and automated (implemented) by a refactoring tool and a program which contains three classes *A*, *B* and *C* (as shown in the top left corner of Fig. 1) is being processed by the refactoring tool. Furthermore, the public field *fa* is accessed by methods of class *C*, but not accessed by class *B*

Although the public field *fa* is useful for class *C*, that is, accessed by methods of class *C*, it is definitely useless for class *B*. As a result, the refactoring tool will suggest that the refactoring rule Delete Dead Field can be applied to the field *fa* of class *B*. The result of the refactoring is presented in the bottom left corner of Fig. 1.

The refactoring tool will also suggest to apply refactoring rule Pull Up Field to the same program. Since both subclasses (*B* and *C*) of *A* contain public field *fa*, the common field should be pulled up to the superclass *A* according to refactoring rule Pull Up Field. The result of the refactoring is presented in the top right corner of Fig. 1.

As discussed above, either of the refactoring rules can be applied to the program. However, they cannot be applied to the program together (as shown in the bottom right corner of Fig. 1). When the dead field *fa* of class *B* is removed by applying refactoring rule Delete Dead Field, it is impossible to pull field *fa* from class *B* any more (i.e. applying refactoring rule Pull Up Field). When the common field *fa* is pulled up from subclasses *B* and *C* by applying refactoring rule Pull Up Field, it is impossible to remove field *fa* from class *B* or *A* neither.

Consequently, the result of the refactoring activity on the program depends on the application order (schedule) of the two available refactorings. If Pull Up Field is applied first, the result is the top right corner of Fig. 1. Otherwise, the result is the bottom left corner of Fig. 1.

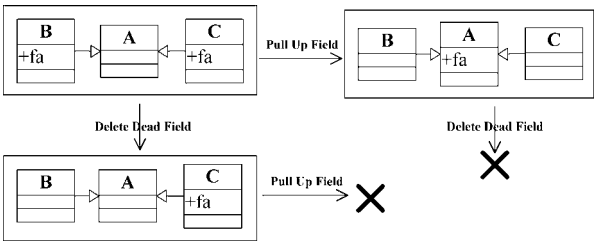


Figure 1 *Conflicts between Delete Dead Field and Pull Up Field*

3 Background

3.1 Software refactoring and its formalisation

Definition 1 (software refactoring): Software refactoring is to restructure object-oriented software to improve its quality, especially extensibility, reusability, and maintainability, while preserving its external behaviours [3].

The word refactoring may have different meaning in different contexts. It may be an abbreviation for software refactoring (a general term, uncountable) representing a technology. It may also represent a specific application of a refactoring rule (a specific case, countable).

As indicated by Definition 1, the goal of software refactoring is to improve software quality. But first of all, we should be aware of what kind of design or structures reduce the quality. Fowler *et al.* [3] call such kind of poor design or structures Bad smells.

Definition 2 (bad smells): Bad smells are the design or structures that may need to be restructured to improve software quality.

Once bad smells are defined, solutions should be proposed to restructure the design so as to improve software quality. A solution to a category of bad smells is usually presented as a refactoring rule.

Definition 3 (refactoring rules): A refactoring rule is a solution on how to restructure a category of bad smells so as to improve software quality.

Refactoring rules can be specified in formal languages, such as graph transformation languages [10, 14, 15]. In graph transformation systems, a transformation rule is represented as a production

$$p : L \xrightarrow{r} R \tag{1}$$

The left pattern L of the production is in fact an abstraction of bad smells. In order to apply a refactorings rule, we have to find occurrences of the left pattern L of the rule, that is, bad smells.

Definition 4 (matches): A match for a refactoring rule $p : L \xrightarrow{r} R$ in software S is a total morphism $m : L \to S$.

As indicated by the definition, a match is a morphism from the left pattern L to the occurrence of the pattern. In this paper, the image of the morphism is also called a matches of the pattern for convenience when the meaning of the word can be distinguished from its context.

Once bad smells (matches) are found out, refactoring rules are applied to restructure them.

Definition 5 (applications of refactoring rules): An application of refactoring rule $p : L \xrightarrow{r} R$ on software S via match $m : L \to S$ is formalised by the following pushout [14].

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ \downarrow m & & \downarrow m* \\ S & \longrightarrow & S' \end{array}$$

where m^* is a co-match of m [14]. By replacing the occurrence of L with a corresponding occurrence of R, the software S is transformed into S'. The application is usually notated as $t : S \xrightarrow{m,p} S'$.

An application of a refactoring rule is also called a refactoring for convenience when the meaning of the word refactoring can be distinguished from its context. The word refactoring means an application of a rule if and only if it is used as a countable noun.

3.2 Conflicts among refactorings

Refactorings (applications of refactoring rules) may conflict with each other [12]. The extract meaning of conflicts among refactorings is defined in Definition 7.

Definition 6 (asymmetric refactoring conflicts): Given two refactorings $t_1 : S \xrightarrow{m_1,p_1} S'$ and $t_2 : S \xrightarrow{m_2,p_2} S''$, t_1 has an asymmetric conflict to t_2 if and only if (iff) t_2 can be applied before, but not after t_1.

We use $t_1 \nrightarrow t_2$ to indicate that t_1 has an asymmetric conflict with t_2.

Conflicts are directed. The fact that carrying out t_1 will disable t_2 does not necessarily means that the application of t_2 will disable t_1. In other words, $t_1 \nrightarrow t_2$ and $t_2 \nrightarrow t_1$ are not equal.

Definition 7 (symmetric refactoring conflicts): Given two refactorings $t_1 : S \xrightarrow{m_1,p_1} S'$ and $t_2 : S \xrightarrow{m_2,p_2} S''$, t_1 has an symmetric conflict with t_2 if and only if (iff) t_2 and t_1 cannot be applied together in any order.

Symmetric conflict between t_1 and t_2 is noted as $t_1 \nleftrightarrow t_2$.

The main cause of refactoring conflicts is that the application of a refactoring may delete or change elements that are necessary for other refactorings. When refactoring $t_1 : S \xrightarrow{m_1,p_1} S'$ is applied, $m_1(L_1 - R_1)$ will be deleted and $m_1^*(R_1 - L_1)$ will be added [14]. If $m_1(L_1 - R_1) \cap m_2(L_2) \neq \emptyset$, refactoring t_1 will disable $t_2 : S \xrightarrow{m_2,p_2} S''$ because elements of $m_1(L_1 - R_1) \cap m_2(L_2)$ which are necessary for match m_2 will be deleted by t_1. As a conclusion, t_1 has an asymmetric conflict with t_2 if and only if (iff):

$$m_1(L_1 - R_1) \cap m_2(L_2) \neq \emptyset \tag{2}$$

doi: 10.1049/iet-sen:20070033

3.3 Automation and schedule of refactorings

Once refactoring rules are formalised, it is possible to automate refactorings [10, 16, 17]. For example, if refactoring rules are formalised as graph transformation rules (productions), they can be automated by state-of-the-art graph rewriting tools, such as PROGRES [18], Fujaba [19] and AGG [20].

When formalised refactoring rules are carried out by graph rewriting tools, such as AGG [20], all applicable refactoring rules are applied in a non-deterministic fashion until no more refactoring rules are applicable [21]. In other words, available refactorings are not scheduled. Instead, they are applied in random.

However, as indicated by the motivating examples in Section 2, different application orders of conflicting refactorings lead to different quality improvement. In other words, in order to improve the software quality as much as possible. The application order of the available refactorings is critical, and obviously random scheduling is not a good solution. Although some rewriting tools also make it possible to apply refactoring rules in a predefined order, there is no guidance available on how to arrange the application order (schedule) of the rules.

For each refactoring rule, there are many places where the rule can be applied. As a result, it also necessary to determine where the rule should be applied first. Furthermore, it is not necessarily the best solution (schedule) to apply a refactoring rule in all the places where the rule can be applied before the next rule gets chances to be applied. It may be more effective to apply the rules alternately in some cases.

Some refactorings can only be semi-automated. Some refactoring rules (such as *Rename Variable*) are hard to formalise. For example, it is not easy to formalise which kind of variable should be renamed, and what the new name should be. For such kind of refactoring rules, experts should find out matches (bad smells) manually or with some help of detection tools and assign parameters or free variables of the refactorings (the new name in this case). And then, refactoring tools (such as Eclipse) can automatically apply the refactorings by restructuring the matches (bad smells) according to corresponding refactoring rules and the assigned parameters, that is, applies the rules via the corresponding matches (Definition 5).

Usually, the application order of the semi-automated refactorings is determined implicitly by software engineers. But it is not only expensive (time consuming), but also inducts uncertainty. Experienced engineers may do it well, but inexperienced engineers may be confused and lead to poor schedule.

All in all, we need a scheduling approach to schedule the large amount of refactorings so as to improve software quality as much as possible.

3.4 Software quality assessment

Even since the very beginning of computer software, software quality has been a big concern, and the assessment of software quality has been a hot topic in both academia and industry.

In order to assess software quality, people have proposed many quality models for software products, such as McCall's quality model [22], Dromey's quality framework [23] and QMOOD [24]. In these quality models, a set of high-level quality attributes are selected to indicate software quality. These quality attributes are then linked to a set of characteristics (or design properties) which are in turn quantitated with automatically gathered metrics.

For example, QMOOD proposed by Bansiya and Davis [24] includes six high-level quality attributes: reusability, flexibility, understandability, functionality, extendibility and effectiveness. And these attributes are linked to design properties: hierarchies, abstraction, encapsulation, coupling, cohesion, composition, inheritance, polymorphism, message and complexity. They also weight the linkages from design properties to quality attributes. For example, the quality attribute understandability is linked to design properties by the following formula [24]

$$\begin{aligned}\text{UNDERSTANDABILITY} = {} & -0.33 * \text{Abstraction} + 0.33 * \text{Encapsulation} - 0.33 * \text{Coupling} + 0.33 * \text{Cohesion} \\ & -0.33 * \text{Polymorphism} - 0.33 * \text{Complexity} - 0.33 * \text{DesignSize}\end{aligned}$$

And then these design properties are linked to metrics: DSC (Design Size in Class), NOH (Number of Hierarchies), ANA (Average Number of Ancestors), DAM (Data Access Metric), DCC (Direct Class Coupling), CAM (Cohesion Among Methods of Class), MOA (Measure of Aggregation), MFA (Measure of Functional Abstraction), NOP (Number of Polymorphic Methods), CIS (Class Interface Size) and NOM (Number of Methods). The metrics can be automatically collected by tools such as QMOOD++ [25]. In such a way, we can assess software quality by quantitative quality attributes

$$Q := [Q_{\text{reusability}}, Q_{\text{flexibility}}, \cdots, Q_{\text{effectiveness}}]$$

Quality assessment with other quality models is similar. In general, software quality can be presented as

$$Q := [Q_{\text{attribute}_1}, Q_{\text{attribute}_2}, \cdots, Q_{\text{attribute}_n}]$$

and quality improvement is presented as

$$\Delta Q := [\Delta Q_{\text{attribute}_1}, \Delta Q_{\text{attribute}_2}, \cdots, \Delta Q_{\text{attribute}_n}]$$

Quality improvement achieved by refactorings can also be assessed. The quality improvement can be measured by comparing software quality before and after refactoring. It is also possible to estimate the effect of refactorings on software quality before they are applied [13, 26–28]. The estimation may be qualitative [28] or quantitative [13, 26, 29]. It may be done by experts manually [13, 26] or by computer automatically [29].

Kaotaoka *et al.* [30] illustrate the quantitative evaluation of quality improvement achieved by refactoring. They suggest to apply refactoring in three steps:

1. identify refactoring candidates;

2. validate refactoring effect;

3. apply refactorings.

In the second step, we should assess the quality improvement (refactoring effect). Kaotaoka *et al.* [30] focus on one quality attribute only: maintainability. In that paper, Kaotaoka *et al.* link the maintainability to coupling, and thus the quality improvement (refactoring effect) is evaluated by comparing the coupling before and after refactoring.

4 Scheduling approach

4.1 Overview

In this section, we present and discuss a conflict-aware scheduling approach for conflicting refactorings. The goal of the scheduling approach is to promote quality improvement achieved by refactoring activities.

As the first step of scheduling, conflicts among refactorings are detected. Section 4.2 proposes an approach to detect conflicts among refactorings. And then, Section 4.3 proposes a scheduling model. The target of the scheduling model is the maximal quality improvement achieved by the selected refactorings. And then, a heuristic algorithm is proposed in Section 4.4 to solve the scheduling model proposed in Section 4.3.

4.2 Conflict matrix

Before conflict-aware schedule can be applied, conflicts among refactorings should be detected first.

Suppose there are two applicable refactorings (t_1 and t_2)

$$t_1: S \xrightarrow{m_1, p_1} S_1 \qquad t_2: S \xrightarrow{m_2, p_2} S_2$$
$$p_1: L_1 \xrightarrow{r_1} R_1 \qquad p_2: L_2 \xrightarrow{r_2} R_2$$

As discussed in Section 3.2, refactoring t_1 asymmetrically conflicts refactoring t_2 if and only if

$$m_1(L_1 - R_1) \cap m_2(L_2) \neq \emptyset \tag{3}$$

This is the key for the detection of conflicts between refactorings.

The algorithm to detect the conflict matrix of the refactorings is shown in Fig. 2. Every element $element_j$ involved in matches is scanned twice. In the first scan, the algorithm records the matches $element_j$.AffectedMatches, which will be affected (destroyed) if $element_j$ is deleted, and the matches $element_j$.DestructiveMatches by restructuring which $element_j$ will be removed or changed. In the second scan, the algorithm detects conflicts according to the records made in the first scan: restructuring any match in $element_j$.DestructiveMatches will destroy $element_j$, and thus destroy all the matches in $element_j$.AffectedMatches. At the end, $C(i, i)$ is set to zero because a refactorings will not conflict with itself.

For each pair of conflicting refactorings, for example, an asymmetrical conflict from t_1 to t_2, some elements of the overlap of m_1 and m_2 will be deleted by carrying out t_1. Suppose that the average amount of the deleted elements for each overlap is k, the amount of refactorings is n, and the conflict probability is γ, then the time complexity of the detection algorithm in Fig. 2 is $O(n^2 * \gamma * k)$.

4.3 Scheduling model

All available refactorings and the conflicts among them are formalised as a directed graph G as shown in Fig. 3. Each vertex v_i represents a refactoring t_i. A directed edge from vertex v_i to v_j indicates that the application of refactoring t_i will disable refactoring t_j. If there is also a directed edge from vertex v_j to v_i at the same time, there is a symmetric conflict between refactorings t_i and t_j.

```
Input: ST = {t1, t2, t3 ··· tn}
For each ti ⊆ ST
  For each elementj ⊆ mi(Li)
    //Deleting elementj will affect (destroy) match mi
    elementj.AffectedMatches.add(i)
    If (elementj ∈ mi(Li − Ri))
      //elementj will be destroyed by restructuring
      // match mi
      elementj.DestructiveMatches.add(i)
    End If
  End For
End For
For each ti ⊆ ST
  For each elementj ⊆ mi(Li)
    If (elementj.DestructiveMatches ≠ ∅)
      //The element will be destroyed by restructuring
      //any match in the set
      For each k ⊆ elementj.DestructiveMatches
        For each ℓ ⊆ elementj.AffectedMatches
          //Restructuring match mk will
          //destroy match mℓ
          C(k, ℓ) = 1
        End For
      End For
    End If
  End For
End For
For i ∈ [1, n]
  C(i,i)=0
End For
```

Figure 2 *Detection of conflict matrices*

doi: 10.1049/iet-sen:20070033

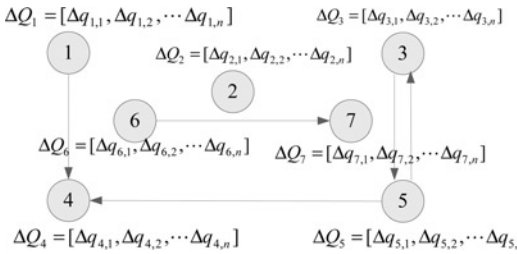


Figure 3 *Refactorings and conflicts among them*

For each refactoring t_i, the effect (quality improvement) of t_i is represented as $\Delta Q_i = [\Delta q_{i,1}, \Delta q_{i,2}, \ldots, \Delta q_{i,n}]$, where $\Delta q_{i,;j}$ is the effect of refactoring t_i on the jth quality attribute f_j. The assessment of software quality and quality improvement has been discussed in Section 3.4. Since the scheduling model is not confined to any specific quality model, the quality improvement (effect of refactorings) is represented in a general form. But before the proposed approach can be applied to a specific project, a specific quality model (such as QMOOD [24]) should be selected for the specific project.

The scheduling approach is to select a sequence of the available refactorings to maximise the quality improvement

$$\begin{aligned} &\max\ \mathrm{TQF}_1 = \sum_{j\in[1,k]} \Delta q_{x_j,1} \\ &\max\ \mathrm{TQF}_2 = \sum_{j\in[1,k]} \Delta q_{x_j,2} \\ &\quad\vdots \\ &\max\ \mathrm{TQF}_n = \sum_{j\in[1,k]} \Delta q_{x_j,n} \\ &\mathrm{Seq\ SQ} = \langle t_{x_1}, t_{x_2}, t_{x_3}, \ldots, t_{x_k} \rangle \end{aligned} \tag{4}$$

s.t.

$$\begin{aligned} &1 \le k \le N \\ &1 \le x_i \le N \quad i \subseteq [1, k] \\ &x_i \ne x_j \quad \text{if} \quad i \ne j \\ &\langle v_{x_i}, v_{x_j} \rangle \notin E \quad \text{if} \quad i \le j \quad \text{and} \quad i, j \subseteq [1, k] \end{aligned} \tag{5}$$

where E is the edges of the directed graph G and N the size of G. SQ is a sequence of refactorings selected to be applied. TQF_i is the total effect of the selected refactorings on the ith quality attribute f_i. The last constrain

$$\langle v_{x_i}, v_{x_j} \rangle \notin E \quad \text{if} \quad i \le j \quad \text{and} \quad i, j \subseteq [1, k]$$

guarantees that no selected refactoring will be disabled by the previously applied refactorings.

4.4 Heuristic algorithm

In this section, we propose an algorithm to solve the scheduling model proposed in Section 4.3.

The scheduling problem is a multi-objective optimisation problem (MOP) which is known to be an NP-complete combinatorial optimisation problem. The central ideal of aggregation-based strategy to MOP is to transform MOP into a single-objective optimisation problem (SOP). All of the objectives ($\mathrm{TQF}_1, \mathrm{TQF}_2, \ldots, \mathrm{TQF}_n$) are combined into a single one:

$$\mathrm{TQF} = \sum_{i\in[1,n]} w_i * \mathrm{TQF}_i \tag{6}$$

where w_i is the weight of the ith objective TQF_i.

To transform MOP into SOP, we should decide the weights for the objectives. The weights for the objectives indicate the relative importance of the corresponding quality attributes. Yu *et al.* [31] suggest that we should quantitatively assess software quality so as to decide the priorities of quality attributes (soft goals) in software refactoring activities. The decision is usually made by experts.

Even if the scheduling model is successfully transformed into a single-objective model, it is still hard to compute its optimum solution. The time complexity is $O(N!)$, where N is the total amount of the available refactorings. So, a heuristic algorithm which results in a suboptimum solution with much lower complexity is preferred.

Given a set of refactorings and their conflict matrix C

$$\begin{aligned} &\mathrm{ST} = \{t_1, t_2, t_3, \ldots, t_N\} \\ &C_{i,;j} = \begin{cases} 1 & \text{if } t_i \text{ asymmetrically conflicts } t_j \\ 0 & \text{else} \end{cases} \\ &\mathrm{Size}(C) = (N, N) \end{aligned} \tag{7}$$

The following heuristic algorithm is given to schedule the refactorings.

Step 1 (Initialising G): Draw the initial directed graph G to represent the refactorings and conflicts among them.

$$G = \langle V, E \rangle$$

where V and E are the vertexes and edges of G, respectively. Refactoring t_i is represented as a vertex v_i, and $e_{i,j} \in E$ if and only if $C_{i,j} = 1$

Step 2 (Initialising SQ): Initialise the restructuring sequence SQ to empty.

$$\mathrm{SQ} = \langle\rangle$$

Step 3 (applying uninjurious refactorings): Vertexes without outgoing edges represent uninjurious refactorings which will not hurt (disable) other refactorings. In order to prevent them from being destroyed by other refactorings, they are applied first.

If $|G| = 0$, the algorithm ends.
If there is no uninjurious refactorings, go to Step 4.
Otherwise, if vertex v_i represents an uninjurious refactoring (outdegree = 0), remove v_i and its edges and apply (carry out) the corresponding refactoring t_i (SQ = SQ + t_i).
Go back to Step 3 to apply the next uninjurious refactoring.

Step 4 (computing synthetical effect): For each vertex $v_i \in V$, compute its synthetical effect

$$\mathrm{Syn}Q_i = \sum_{j \in [1,n]} w_j * \Delta q_{i,j} \tag{8}$$

where $\Delta q_{i,j}$ is the effect of refactoring t_i on the jth attribute attribute$_j$, and w_j is the weight of the jth attribute attribute$_j$. So, SynQ_j is the weighted total effect of refactoring t_i.

Step 5 (computing potential effect): The application of a refactoring may disable other refactorings. This should be counted on as potential negative effect. Furthermore, the application of refactoring t_i also reduces the potential negative effect of refactorings which have asymmetrical conflicts to t_i. The potential effect PQ$_i$ of refactoring t_i is computed in the following way

$$\mathrm{PQ}_i = \mathrm{Syn}Q_i - \sum_{e_{i,j} \in E} \mathrm{Syn}Q_j + \sum_{e_{\ell,i} \in E} \mathrm{Syn}Q_\ell \tag{9}$$

Step 6 (selection and application): Select a vertex v_i from G which has the greatest potential effect. Apply the corresponding refactoring t_i.

Step 7 (update potential effect): Once the selected refactoring is applied, the corresponding vertex v_i is removed from G. Before that we should update the potential effect of vertexes adjacent to v_i. For each $e_{i,j} \in E$, that is, there is an edge from v_i to v_j, remove v_j and its edges from the directed graph. The edge from v_i to v_j indicates that the application of refactoring t_i will disable refactoring t_j. As a result, once t_i is applied, refactoring t_j should be removed from candidate (available) refactorings. For each $e_{\ell,i} \in E$, that is, there is an edge from v_ℓ to v_i. Update the potential effect of v_ℓ according to (10)

$$\mathrm{PQ}_\ell = \mathrm{PQ}_\ell + \mathrm{Syn}Q_i \tag{10}$$

Once v_i is removed, the potential quality improvement of v_ℓ should be updated according to (9). The result of the update is equivalent to (10).
If no vertex is left, the algorithm ends. Otherwise, go back to Step 6.

In each iteration, a refactoring (vertex) is selected and applied. And every edge is visited three times: the first and second times in the initialisation of potential effect of its initial and terminal vertexes; the third time in the deletion of the edge. So, if vertexes and edges contain references to each other, the time complexity of the algorithm is $O(|V| + 3|E|) = O(N^2)$, where N is the total amount of the available refactorings.

If the directed graph G is an directed acyclic graph, the algorithm will stop at Step 3 (applying uninjurious refactorings), that is, there is no need to get into the heuristic part of the algorithm (Steps 4–7). For an directed acyclic graph, the algorithm is equal to topological sorting algorithms.

5 Evaluation

The evaluation purpose is to validate that conflict-aware schedule of conflicting refactorings is helpful in promoting quality improvement achieved by refactoring activities.

5.1 Evaluation subjects

As stated in Section 3.3, quite a few refactorings are not yet automated. As a result, it will be very hard and expensive to refactor large projects for software engineers who are not familiar with these projects. On the contrary, it will be much easier for them to refactor projects they developed. For this reason, we decide to select projects developed in our institute for evaluation to reduce the evaluation cost. Evaluations on projects outside the institute are also in plan.

Table 1 Refactoring rules

Rule number	Refactoring rules
1	Remove Clone
2	Move Method
3	Encapsulate Variable
4	Remove Method
5	Reduce Tight Coupling
6	Remove Variable
7	Move Variable
8	Pull up Method
9	Rename Class
10	Rename Variable
11	Decompose Long Method
12	Replace Literal
13	Pull up Variable
14	Rename Method
15	Decompose Large Class
16	Create Super Class
17	Add Parameter
18	Variable Reuse

IET Softw., 2008, Vol. 2, No. 5, pp. 446–460

doi: 10.1049/iet-sen:20070033

5.1.1 PKU metamodelling tool: As indicated by the name, PKU metamodelling tool (PMMT) is a metamodelling tool developed in Peking University. Users design modelling languages with PMMT, and then PMMT can automatically generate modelling tools which support the designed modelling languages.

In order to reduce risk, the developers decide to develop a prototype based on Eclipse. The prototype has be finished which is composed of 312 files containing about 70 000 lines of source code. It has also successfully generated a UML (Unified Modelling Language) modelling tool according to the UML metamodel [32, 33]. With the success of the prototype, they decide to extend it into a complete metamodelling tool. However, they find that it is very hard to extend the prototype because the source code is jumbled. So, they decide to refactor it before any extension is made.

5.1.2 PKU BPEL modelling tool: PKU BPEL modelling tool (PBMT) is a business process modelling tool based on Business Process Execution Language (BPEL). It is also developed in Peking University by a research team. It looks something like the BPEL Project of Eclipse [34].

5.2 Evaluation process

The purpose of the evaluation is to compare the new refactoring approach, in which refactoring conflicts are explicitly considered and conflicting refactorings are scheduled with the proposed scheduling approach (conflict-aware schedule), with the traditional refactoring approach, in which scheduling is not explicitly proposed (implicit schedule). So, we decide that both the refactoring approaches should be applied to the evaluation projects (two identical copies of the projects in fact) independently.

In traditional refactoring approach (implicit schedule), experts look around, find bad smells restructure the bad smells, and then go ahead for more bad smells. Some of the bad smells are found with the help of professional detecting tools such CCFinder [35]. But as stated in Section 3.3, some complicated bad smells are not easy to formally specified, and thus they usually have to be detected by experts manually.

In the new refactoring approach (conflict-aware schedule), experts find out bad smells (but no refactoring is applied in this stage). And then, the proposed scheduling approach is applied to these candidate (available) refactorings to detect the conflicts among them, and then manage a schedule

Table 2 Conflicts among refactoring rules

Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	×										×				×			×
2					×			×							×			
3		×				×	×			×			×					
4	×	×			×	×		×			×	×		×	×		×	×
5								×							×			
6	×		×		×		×			×			×					×
7			×		×		×			×			×					×
8		×	×	×	×									×	×			
9																		
10	×																	
11	×			×														×
12	×																	
13						×	×			×								×
14		×	×															
15								×	×				×					
16									×									
17																	×	
18	×		×			×	×			×			×					×

(application order). After that selected refactorings are applied according to the schedule.

In the evaluation, we adapt the QMOOD [24] to assess the software quality. In order to determine weights of different quality attributes in the quality model (soft goals of refactoring activities [31] or the priorities of quality attributes), we compare the quality of the evaluation projects with that of similar mature projects to reveal in which aspects (quality attributes) the evaluation projects need improvement. If the comparison suggests that the evaluation projects are poor in quality attribute AttributeA, the quality attribute AttributeA will be given a higher priority (weight). These weights are necessary for the heuristical algorithm proposed in Section 4.4.

PMMT is compared with the Eclipse Modelling Framework Project (EMF [36]) which is a mature open-source metamodelling tool. PMMT and EMF have the same functions: metamodelling. Furthermore, both of them base on the same platform: Eclipse. PBMT is compared with BPEL Project of Eclipse [35]. They also share similar functions and platform.

5.3 Evaluation results and analysis

The refactoring rules used in the evaluation are listed in Table 1 and the conflicts among them are listed in Table 2.

5.3.1 PKU metamodelling tool: Metrics of the PMMT (original version) and EMF are presented in Table 3. Before the quality attributes can be computed, actual metric values should be normalised because actual metric vaules of different ranges are combined in the computation of the quality attributes [24]. Since EMF and PMMT have similar functions and are developed on the same framework, it is reasonable to normalise the metrics of PMMT by dividing them by the corresponding metrics of EMF. The normalised metrics are presented in Table 4, and the quality computed with the normalised metrics are presented in Table 5. Since the software quality presented in Table 5 suggests that PMMT is best in Understandability with respect to EMF (−0.4357 against −0.99), Understandability is assigned the lowest weight (1). Other quality attributes are assigned similarly. Weights of quality attributes are presented in Table 6.

Table 3 Actual metric values of PMMT and EMF

Metrics	Projects	
	PMMT	EMF
design size	344	653
hierarchies	245	347
abstraction	1.63228	1.66922
encapsulation	0.42110	0.93111
coupling	2.91570	4.75498
cohesion	0.44041	0.46982
composition	0.12367	0.33875
inheritance	0.37918	0.29072
polymorphism	0.16279	0.68913
messaging	4.85174	12.50383
complexity	6.00291	16.86371

Table 4 Normalised metric values of PMMT and EMF

Metrics	Projects	
	EMF	PMMT
design size	1	0.526799387
hierarchies	1	0.706051873
abstraction	1	0.977869297
encapsulation	1	0.452254618
coupling	1	0.613188593
cohesion	1	0.937397547
composition	1	0.365062158
inheritance	1	1.304291672
polymorphism	1	0.23622739
messaging	1	0.388020692
complexity	1	0.35596606

After the refactoring activities on PMMT with the two refactoring approaches, respectively, are completed, metrics of the two resulting systems are gathered. The results are presented in Table 7. The normalised metrics and quality attributes are presented in Tables 8 and 9, respectively. The results suggest that the resulting system of conflict-aware schedule has better quality than that of implicit schedule. It is better in almost all aspects (five out of six) as indicated by the evaluation results in Table 9. The improvement in

Table 5 Quality of PMMT and EMF

Quality attributes	Projects	
	EMF	PMMT
reusability	1	0.538462278
flexibility	1	0.260411281
understandability	−0.99	−0.435731526
functionality	1	0.521049561
extensibility	1	0.952599883
effectiveness	1	0.667141027

doi: 10.1049/iet-sen:20070033

Table 6 Weights of quality attributes (PMMT)

Quality attributes	Weight
reusability	4
flexibility	6
understandability	1
functionality	5
extensibility	2
effectiveness	3

Table 7 Actual metric value of resulting systems (PMMT)

Metrics	Conflict-aware schedule	Implicit schedule
design size	348	337
hierarchies	201	269
abstraction	1.33275	1.29000
encapsulation	0.58113	0.56001
coupling	2.90835	2.77745
cohesion	0.77350	0.67928
composition	0.21448	0.18448
inheritance	0.47703	0.45780
polymorphism	0.23056	0.16056
messaging	4.75342	4.55490
complexity	5.91021	5.84273

understandability achieved by the conflict-aware schedule approach is not as great as other quality attributes because the original version of the system is already very good at understandability with respect to EMF as suggested in Table 5, and thus the conflict-aware schedule gives the lowest weight (priority) to understandability.

5.3.2 PKU BPEL modelling tool: The evaluation is applied on PBMT in a similar way. For space limitation, we only present the final quality of the original system and resulting systems of the two refactoring approaches. The results are presented in Table 10.

The results in Table 10 suggest that the resulting system of the conflict-aware schedule approach is better in all aspects than that of the implicit schedule approach.

5.3.3 Comparing with the expected results: As stated in the beginning of Section 5, the purpose of the evaluation is to validate that conflict-aware schedule of refactorings helps to promote quality improvement in refactoring activities. So, we expect the resulting systems refactored with conflict-aware schedule to be better (higher quality) than those refactored without explicit schedule (implicit schedule). The evaluation results of the two projects are consistent with those expected. In other words, the evaluation results confirm that scheduling conflicting refactorings with the proposed conflict-aware scheduling approach can lead to greater quality improvement.

5.3.4 Threats to validity: The first threat to validity is the quality model. Different quality models usually lead to different evaluation results. In this evaluation, we select a representative quality model QMOOD [24] instead of a customised quality model to reduce this treat.

Table 8 Normalised metric value of the resulting systems (PMMT)

Metrics	Original	Conflict-aware schedule	Implicit schedule
design size	1	1.011627907	0.979651163
hierarchies	1	0.820408163	1.097959184
abstraction	1	0.816494249	0.790306553
encapsulation	1	1.380017245	1.329873823
coupling	1	0.99747996	0.952584383
cohesion	1	1.756312916	1.542389032
composition	1	1.734322349	1.491732421
inheritance	1	1.258074658	1.207351857
polymorphism	1	1.416320475	0.986320475
messaging	1	0.97973426	0.938816221
complexity	1	0.984557153	0.97331676

Table 9 Quality of the resulting systems (PMMT)

Quality attributes	Original	Conflict-aware schedule	Implicit schedule
reusability	1	1.185389323	1.106684854
flexibility	1	1.670955733	1.333348808
understandability	−0.99	−0.689749362	−0.597272438
functionality	1	1.140937527	1.065691033
extensibility	1	1.246704711	1.015697251
effectiveness	1	1.321045795	1.161117026

The second threat to validity is the decision of goals and priorities of quality attributes. In the evaluation, we determine goals by comparing the object systems with corresponding mature systems (references). As a result, the evaluation depends on qualities of these reference systems. In this evaluation, we compare PMMT with EMF and PBMT with Eclipse BPEL. EMF and Eclipse BPEL are open-source mature projects, and their quality is recognised. More important, EMF and PMMT are based on the same framework and implement the same functions. This makes the comparison (comparing PMMT to EMF) more reasonable. The same is true for PBMT and Eclipse BPEL.

The third threat to validity is the experience of the software engineers involved in the evaluation. In the implicit scheduling approach, the results of the refactoring activities depend on the experience of the software engineers. In order to reduce this threat, we select the most experienced engineers to do the implicit schedule. The evaluation results suggest that the scheduling approach is better than implicit schedule done by the most experienced engineers. This comparison makes the evaluation more acceptable.

6 Related work

Software refactoring was proposed by Opdyke [2] in 1992. After that people have made great advances in research and application of software refactoring. In the early days of software refactoring, people mainly focused on source code refactoring. With the success of source code refactoring, more and more software artefacts are considered for refactoring. Philipps and Rumpe [37], and Tokuda and Batory [38] discuss refactorings of architectures. Sunye *et al.* [39] introduce refactoring technology into UML models. Correa and Werner [40] make a step further and apply refactoring technology to OCL (Object Constraint Language) [41].

Definition and detection of bad smells are basics of software refactoring. Fowler *et al.* [3] discuss and bad smells in programs and proposed refactoring rules to restructure them. The work of Wake [11] is similar to that of Fowler *et al.* [3]. Liu *et al.* [42] proposed an approach to detect overlapping use cases. Balazinska *et al.* [43] and Ducasse *et al.* [44] proposed approaches to detect clones in source code. Simon *et al.* [27] detected bad smells with the help of software metrics. Kataoka *et al.* [45] developed a bad smells detection tool Daikon to detect invariables.

Formalisation of refactorings is also an important issue of software refactoring. Mens *et al.* [10, 16] tried to formalise software refactorings as graph transformations, and thus made it possible to automate software refactoring with existing graph transformation tools, such as PROGRES [18], Fujaba [19] and AGG [20]. Cornelio [46] formalised refactorings as refinements described in ROOL (Refinement Object-Oriented Language). Porres [17] formalised refactorings in SWM. Whittle [47] formalised model refactorings with MAUDE. Bottoni and Parisi Presice [48] formalised integrated refactorings as distributed graph transformations.

Table 10 Quality of the resulting systems (PBMT)

Quality attributes	Original	Conflict-aware schedule	Implicit schedule
reusability	1	1.125521315	1.092563464
flexibility	1	1.552158481	1.426418418
understandability	−0.99	−0.777855348	−0.801112577
functionality	1	1.025533815	1.011381680
extensibility	1	1.471043443	1.407481470
effectiveness	1	1.173254643	1.160462100

doi: 10.1049/iet-sen:20070033

People also investigate refactoring constraints to guarantee that software refactoring will not alter the external behaviours of software. Proietti and Pettorossi [49] proposed an approach to detect whether refactorings on Prolog source code is behaviour-preserving or not. Gheyi *et al.* [50, 51] proposed similar approaches for Alloy. But for complex languages without formal semantics, the proof is much harder, if not impossible [1]. For such kind of languages, people propose some looser constraints, such as object-preserving by Bergstein [52] and inheritance constraint by Straeten *et al.* [53] and Liu *et al.* [54].

How to detect conflicts among refactorings is a critical problem in scheduling of refactorings. Mens *et al.* [12] suggested to detect structural refactoring conflicts by critical pair analysis, and Lambers *et al.* [55] proposed a more efficient detection algorithm to detect conflicts in graph-based model transformations. The detection is also supported by tools such as AGG [12, 20, 56].

However, scheduling of refactorings is rarely discussed [13]. Bouktif *et al.* [13] suggest that it is necessary to find out an appropriate schedule to optimise clone refactoring activity under constraints and priorities. The main constraint of the refactoring activity is resource limitation. By selecting the right clones to restructure, we can maximise refactoring effect while satisfying resource limitation. The authors formalise the scheduling problem as a constrained Knapsack problem and solve it with a genetic algorithm.

However, Bouktif *et al.* [13] do not deal with conflicts among refactorings. They just assume that clones can be restructured independently. But in fact, refactorings may conflict with each other, that is, carrying out a refactoring may disable other refactorings. As indicated by the second motivating example in Section 2, extracting a clone may destroy other clones and thus prevent them from being extracted.

In this paper, we discuss the schedule of conflicting refactorings in detail and propose a conflict-aware scheduling approach based on a multi-objective optimisation model. The scheduling approach is well integrated with existing software quality models, which makes the assessment of quality improvement of refactorings more accurate, and thus improves the effect of the scheduling approach.

A comprehensive survey of software refactoring is given by Mens and Touwe [1].

7 Conclusion and future research directions

In this paper, we first illustrate the necessity to schedule conflicting refactorings according to their effects and conflicts among them. The motivation is illustrated with an intuitive example. And then, we propose a conflict-aware scheduling approach for conflicting refactorings. We first propose an algorithm to detect conflict metrics among conflicting refactorings. And then, we propose a multi-objective scheduling model. We also propose a heuristic algorithm to solve the scheduling model. The proposed scheduling approach has been validated in projects, and the evaluation results suggest that scheduling refactorings with the proposed approach do help to promote quality improvement.

We also plan to support negative application conditions (NACs) [10, 16, 20, 57]. In order to formalise more refactorings by graph transformation in an intuitive way, NACs are important. More case study and evaluation are also in plan.

8 Acknowledgments

The authors thank the anonymous reviewers for their valuable comments and suggestions. The authors would like to thank Miss Wenmei Liu of Beihang University for her help in the evaluation and proofreading. The authors appreciate the help of the Object-Oriented Modelling research team in software engineering institute of Peking University for their help in the evaluation. The work is supported by the National Grand Fundamental Research 973 Program of China No. 2005CB321805, the National Natural Science Foundation of China No. 60473064, the National High-Tech Research and Development Plan of China No. 2007AA01Z127 and the National Key Technology R&D Program of China No. 2006BAH02A02.

9 References

[1] MENS T., TOUWE T.: 'A survey of software refactoring', *IEEE Trans. Softw. Eng.*, 2004, **30**, (2), pp. 126–139

[2] OPDYKE W.F.: 'Refactoring object-oriented frameworks', PhD thesis, University of Illinois at Urbana-Champaign, 1992

[3] FOWLER M., BECK K., BRANT J., OPDYKE W., ROBERTS D.: 'Refactoring: improving the design of existing code' (Addison Wesley Professional, 1999)

[4] BECK K.: 'Extreme programming explained: embrace change' (Addison Wesley, 2000)

[5] PIPKA J.U.: 'Refactoring in a 'test first'-world'. In MARCHESI M., SUCCI G. (EDS.): Third Int. Conf. eXtreme Programming and Flexible Processes in Software Engineering, May 2002, pp. 178–181

[6] VAN DEURSEN A., MOONEN L.: 'The video store revisited – thoughts on refactorings and testing'. In MARCHESI M.,

SUCCI G. (EDS.): Third Int. Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP2002)), May 2002, pp. 71–76

[7] http://www.eclipse.org/

[8] http://msdn2.microsoft.com/en-us/vstudio/default.aspx

[9] ROBERTS D., BRANT J., JOHNSON R.: 'A refactoring tool for smalltalk', *Theory Pract. Object Syst.*, 1997, **3**, (4), pp. 253–263 (Special issue on object-oriented software evolution and re-engineering)

[10] MENS T., EETVELDE N.V., DEMEYER S.: 'Formalizing refactorings with graph transformations', *J. Softw. Maint. Evol: Res. Pract.*, 2005, **17**, (4), pp. 247–276

[11] WAKE W.C.: 'Refactoring workbook' (Addison Wesley, 2003)

[12] MENS T., TAENTZER G., RUNGE O.: 'Detecting structural refactoring conflicts using critical pair analysis', *Electron. Notes Theor. Comput. Sci.*, 2005, **127**, (3), pp. 113–128

[13] BOUKTIF S., ANTONIOL G., MERLO E., NETELER M.: 'A novel approach to optimize clone refactoring activity'. In CATTOLICO M. (ED.): Proc. 8th Annual Conf. Genetic and Evolutionary Computation (GECCO'06), Seattle, USA, July 2006, pp. 1885–1892

[14] EHRIG H., ENGELS G., KREOWSKI H., ROZENBERG G.: 'Handbook of graph grammars and computing by graph transformation' (World Scientific Publishing Co. Pvt. Ltd., Singapore, 1997), vols. 1–3

[15] GRUNSKE L., GEIGER L., LAWLEY M.: 'A graphical specification of model transformations with triple graph grammars'. In HARTMAN A., KREISCHE D. (EDS.): European Conf. Model Driven Architecture – Foundations and Applications (ECMDA-FA 2005), 2005 (*LNCS*, **3748**) pp. 284–298

[16] MENS T., DEMEYER S., JANSSENS D.: 'Formalising behaviour preserving program transformations'. In CORRADINI A., EHRIG H., KREOWSKI H.-J., ROZENBERG G. (EDS.): Proc. First Int. Conf. Graph Transformation (ICGT'02), Barcelona, Spain, 7–12 October 2002, (*Lecture Notes in Computer Science*, **2505**) pp. 286–301

[17] PORRES I.: 'Model refactorings as rule-based update transformations'. In STEVENS P. *ET AL.* (EDS.): Sixth Int. Conf. Unified Modeling Language (UML 2003), 2003, (*Lecture Notes in Computer Science*, **2863**) pp. 159–174

[18] SCHURR A., WINTER A.J., ZUNDORF A.: 'Graph grammar engineering with progres'. Proceedings of the 5th European Software Engineering Conf., London, UK, 1995 (*Lecture Notes in Computer Science*, **989**) pp. 219–234

[19] NIERE J., ZUNDORF A.: 'Using FUJABA for the development of production control systems'. In NAGL M., SCHURR A., MUNCH M. (EDS.): Int. Workshop on Applications of Graph Transformations with Industrial Relevance, 1999 (*Lecture Notes in Computer Science*, **1779**) pp. 181–191

[20] TAENTZER G.: 'AGG: a tool environment for algebraic graph transformation'. Proc. Int. Workshop on Applications of Graph Transformations with Industrial Relevance, Kerkrade, The Netherlands, September 1999 (*Lecture Notes in Computer Science*, **1779**) pp. 481–488

[21] MENS T.: 'On the use of graph transformations for model refactoring'. Slides of a tutorial at the Summer School on Generative and Transformational Techniques in Software Engineering, Braha, Portugal, July 2005

[22] MCCALL J., RICHARDS P.K., WALTERS G.F.: 'Factors in software quality', Technical Report No., RADC TR-77-369, 1977, US Rome Air Development Center, 1977

[23] DROMEY R.G.: 'A model for software product quality', *IEEE Trans. Softw. Eng.*, 1995, **21**, (2), pp. 146–162

[24] BANSIYA J., DAVIS C.G.: 'A hierarchical model for object-oriented design quality assessment', *IEEE Trans. Softw. Eng.*, 2002, **28**, (1), pp. 4–17

[25] BANSIYA J., DAVIS C.: 'Automated metrics for object-oriented development', *Dr. Dobb's J.*, 1997, **272**, pp. 42–48

[26] SAHRAOUI H., GODIN R., MICELI T.: 'Can metrics help to bridge the gap between the improvement of OO design quality and its automation?'. Proc. Int. Conf. Software Maintenance (ICSM'00), San Jose, California, USA, 11–14 October 2000, pp. 154–162

[27] SIMON F., STEINBRUCKER F., LEWERENTZ C.: 'Metrics based refactoring'. Proc. Fifth European Conf. Software Maintenance and Reengineering, 2001, pp. 30–38

[28] TAHVILDARI L., KONTOGIANNIS K.: 'A metric-based approach to enhance design quality through meta-pattern transformations'. Proc. Seventh European Conf. Software Maintenance and Reengineering CSMR'03, 2003, pp. 183–192

[29] BOIS B.D.: 'Opportunities and challenges in deriving metric impacts from refactoring postconditions'. Proc. Fifth Int. Workshop on Object-oriented Reengineering (Co-located with the 18th European Conf. Object-Oriented Programming), 15 June 2004 (*Lecture Notes in Computer Science*, **3344**) pp. 177–186

[30] KATAOKA Y., IMAI T., ANDOU H., FUKAYA T.: 'A quantitative evaluation of maintainability enhancement by refactoring'. Proc. 18th Int. Conf. Software Maintenance (ICSM'02),

doi: 10.1049/iet-sen:20070033

Montreal, Quebec, Canada, 3–6 October 2002, pp. 576–585

[31] YU Y., MYLOPOULOS J., YU E., LEITE J.C., LIU L.L., D'HOLLANDER E.: 'Software refactoring guided by multiple soft-goals'. The 1st Int. Workshop on Refactoring: Achievements, Challenges, Effects, November 2003

[32] Object Management Group. 'UML 2.0 infrastructure specification'. Technical Report, formal/05-07-05, 2005

[33] Object Management Goup. 'UML 2.0 superstructure specification'. Technical Report, formal/05-07-04, 2005

[34] http://www.eclipse.org/bpel/

[35] KAMIYA T., KUSUMOTO S., INOUE K.: 'CCFinder: a multilinguistic token based code clone detection system for large scale source code', *IEEE Trans. Softw. Eng.*, 2002, **28**, (6), pp. 654–670

[36] http://www.eclipse.org/modeling/emf/

[37] PHILIPPS J., RUMPE B.: 'Refinement of information flow architectures'. First IEEE Int. Conf. Formal Engineering Methods, Hiroshima, Japan, 1997, pp. 203–212

[38] TOKUDA L., BATORY D.: 'Evolving object-oriented designs with refactorings', *Autom. Softw. Eng.*, **8**, (11), pp. 89–120

[39] SUNYE G., POLLET D., TRAON Y.L., JEZEQUEL J.M.: 'Refactoring UML models'. In GOGOLLA M., KOBRYN C. (EDS.): Proc. 4th Int. Conf. The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Toronto, Canada, 1–5 October 2001, (*Lecture Notes in Computer Science*, **2185**) pp. 134–148

[40] CORREA A., WERNER C.: 'Applying refactoring techniques to UML/OCL models'. Seventh Int. Conf. Unified Modeling Language (UML 2004), 14 October 2004, (*Lecture Notes in Computer Science*, **3273**) pp. 173–187

[41] Object Management Group. 'UML 2.0 OCL specification'. Technical Report, ptc/03-10-14, 2003

[42] LIU H., SHAO W.Z., ZHANG L., MA Z.Y.: 'Detecting overlapping use cases', *IET Software (IEE Proceedings Software)*, 2007, **1**, (1), pp. 29–36

[43] BALAZINSKA M., MERLO E., DAGENAIS M., LAGUE B., KONTOGIANNIS K.: 'Advanced clone-analysis to support object-oriented system refactoring'. Proc. Working Conf. Reverse Eng., 2000, pp. 98–107

[44] DUCASSE S., RIEGER M., DEMEYER S.: 'A language independent approach for detecting duplicated code'. Int. Conf. Software Maintenance (ICSM '99), 1999, pp. 109–118

[45] KATAOKA Y., ERNST M., GRISWOLD W., NOTKIN D.: 'Automated support for program refactoring using invariants'. Proc. Int. Conf. Software Maintenance, 2001, pp. 736–743

[46] CORNELIO M.L.: 'Refactorings as formal refinements', PhD thesis, Federal University of Pernambuco, Brazil, 2004

[47] WHITTLE J.: 'Transformations and software modeling languages: automating transformations'. In JEZEQUEL J.-M., HUSSMANN H., COOK S. (EDS.): Proc. UML 2002 – The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th Int. Conf., Dresden, Germany, 2002, (*LNCS* **2460**) p. 227C242

[48] BOTTONI P., PARISI PRESICCE F., TAENTZER G.: 'Specifying integrated refactoring with distributed graph transformations'. Proc. Applications of Graph Transformations with Industrial Relevance, 2003, pp. 227–242

[49] PROIETTI M., PETTOROSSI A.: 'Semantics preserving transformation rules for prolog', *ACM SIGPLAN Not.*, 1991, **26**, (9), p. 274C284

[50] GHEYI R., MASSONI T.: 'Formal refactorings for object models'. 20th Annual ACM SIGPLAN Conf. Object-oriented Programming, Systems, Languages, and Applications, New York, 2005, p. 208C209

[51] GHEYI R., MASSONI T., BORBA P.: 'A rigorous approach for proving model refactorings'. In REDMILES D.F., ELLMAN T., ZISMAN A. (EDS.): 20th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2005), Long Beach, California, USA, November 2005, p. 372C375

[52] BERGSTEIN P.L.: 'Object-preserving class transformation'. ACM SIGPLAN Conf. Object-oriented Programming, Systems, Languages, and Applications, New York, 1991, p. 299C311

[53] STRAETEN R.V.D., JONCKERS V., MENS T.: 'Supporting model refactorings through behaviour inheritance consistencies'. In BAAR T., STROHMEIER A., MOREIRA A., MELLOR S.J. (EDS.): The Unified Modeling Language (UML) 2004, Berlin Heidelberg, 2004, (*LNCS*, **3273**) p. 305C319

[54] LIU H., MA Z.Y., SHAO W.Z.: 'Description and proof of property preservation of model transformations', *J. Softw.*, 2007, **18**, (10), pp. 2369–2379 (in Chinese)

[55] LAMBERS L., EHRIG H., OREJAS F.: 'Efficient detection of conflicts in graph-based model transformation', *Electron. Notes Theor. Comput. Sci.*, 2006, **152**, (27), pp. 97–109

[56] HAUSMANN J.H., HECKEL R., TAENTZER G.: 'Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph

transformation'. Proc. 24th Int. Conf. Software Engineering, 2002, pp. 105–115

[57] LAMBERS L., EHRIG H., OREJAS F.: 'Conflict detection for graph transformation with negative application conditions' In CORRADINI A., EHRIG H., MONTANARI U., RIBEIRO L., ROZENBERG G. (EDS.): Proc. Third Int. Conf. Graph Transformations (ICGT'06), Natal, Rio Grande do Norte, Brazil, 17–23 September 2006 (*Lecture Notes in Computer Science*, **4178**) pp. 61–76

IET Softw., 2008, Vol. 2, No. 5, pp. 446–460
doi: 10.1049/iet-sen:20070033