A Constraint Programming Approach to Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring

Minhaz F. Zibran Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada S7N 5C9
Email: {minhaz.zibran, chanchal.roy}@usask.ca

***Abstract*—Duplicated code, also known as code clones, are one of the malicious 'code smells' that often need to be removed through refactoring for enhancing maintainability. Among all the potential refactoring opportunities, the choice and order of a set of refactoring activities may have distinguishable effect on the design/code quality. Moreover, there may be dependencies and conflicts among those refactorings. The organization may also impose priorities on certain refactoring activities. Addressing all these conflicts, priorities, and dependencies, manual formulation of an optimal refactoring schedule is very expensive, if not impossible. Therefore, an automated refactoring scheduler is necessary, which will maximize benefit and minimize refactoring effort. In this paper, we present a refactoring effort model, and propose a constraint programming approach for conflict-aware optimal scheduling of code clone refactoring.**

I. Introduction

Duplicated code, or code clone is a well-known code smell [9], [15]. Programmers' copy-paste-modification practice is regarded as one of the main reasons for such intentional clones that are beneficial in many ways [16]. However, unintentional clones also appear due to a number of reasons. For example, the use of design patterns, frameworks, and similar APIs results in unintentional code clones. Previous studies reported that software systems might have 9%-17% [32] duplicated code, up to 50% [24]. Copying a fragment containing any unknown bugs may result in fault propagation. From the maintenance perspective, the existence of code clones may increase maintenance effort. For example, a change in a clone fragment may require careful and consistent changes to all copies of the fragment. Any inconsistency may introduce new bugs. Nevertheless, in many cases, code clones are unavoidable or desirable. Therefore, to prevent code inflation and reduce maintenance cost the amount of code clones should be minimized by applying active refactoring. However, refactoring is not free, rather is often risky as it might even introduce new bugs and hidden dependencies. Therefore, it is important to have a prioritized refactoring schedule of the potential refactoring candidates so that the maintenance engineers' can only focus on a select list of refactoring candidates considering the existing constraints, potential benefits, risks and available resources.

There are many refactoring patterns [9], [10], all of which are not directly applicable to code clone refactoring. The applicability of certain refactoring activities largely depends on the context. So, for code clones, refactoring activities and the relevant contexts need to be identified in the first place. The consequences of clone refactoring should also be taken into account. The effort required for applying certain refactoring on the underlying code clones should also be minimized to keep the maintenance cost within reach. The application of a subset of refactorings from a set of applicable refactoring activities may result in distinguishable impact on the overall code quality. Moreover, there may be sequential dependencies and conflicts among the refactoring activities. These lead to the necessity that, from all refactoring candidates a subset of non-conflicting refactoring activities be selected and ordered (for application) such that the quality of the codebase is maximized while the required effort is minimized [33].

Software refactoring is often performed with the aid of graph transformation tools [22], where the available refactorings are applied in random, without having been scheduled [19]. Usually, the application order of the semi-automated refactorings is determined implicitly by human practitioners. However, this is inefficient and error-prone. While experienced engineers may do it well, inexperienced practitioners may lead to poor/infeasible schedule. The challenge is likely to be more severe for refactoring legacy systems, or when a developer new to the codebase has to devise the refactoring schedule. Therefore, automated (or semi-automated) scheduling for performing selection and ordering of refactorings from a set of all refactoring candidates is a justified need.

In this regard, this paper makes two contributions. First, we introduce an *effort model* for estimating the effort required to refactor code clones in object-oriented (OO) codebase. Second, taking into account the *effort model* and a wide variety of possible hard and soft constraints, we formulate the scheduling of code clone refactoring activities as a constraint satisfaction optimization problem (CSOP), and solve it by applying constraint programming (CP) technique that aims to maximize benefits while minimizing refactoring efforts. To the best of our knowledge, ours is the first refactoring effort model for OO systems, and we are the first to apply the CP technique in software refactoring scheduling. We choose to adopt CP for two main reasons. First, CP is a natural fit for solving CSOPs such as scheduling problems. Second, this recent technique integrates the strengths from both artificial intelligence (AI) and operations research (OR), and that it has been proved to

be very efficient in solving CSOPs [4], [35].

To evaluate the effectiveness of our scheduler and the code clone refactoring effort model, we also conduct a case study on four software systems written in Java. We find that our scheduler is capable of efficiently computing the optimal refactoring schedule, and our refactoring effort model offers significant help in the estimation of the refactoring efforts.

The remaining of the paper is organized as follows. Section II identifies the refactoring patterns that are suitable for code clone refactoring. In Section III, we describe our clone refactoring effort model. Section IV discusses how the effect of refactoring can be estimated. In Section V, we describe the possible constraints on refactorings, and Section VI presents our CSOP formulation of the refactoring scheduling problem. In Section VII, we illustrate our case study to evaluate our refactoring scheduler and the effort model. Section VIII contains discussion on the related work, and finally Section IX concludes the paper with our directions to future research.

II. Clone Refactoring Operations

Among the software refactoring patterns [9], we find the following patterns suitable for clone refactoring, and we refer to them as refactoring *operators*. Detail about these refactoring patterns can be found elsewhere [9], [10].

- **Extract method (EM)** extracts a block of code as a new method, and replaces that block by a call to the newly introduced method. EM may cause splitting of a method into pieces. For code clone refactoring, similar blocks of code can be replaced by calls to an extracted generalized method (Figure 1).
- **Pull-up method (PM)** removes similar methods found in several classes by introducing generalized method in their common superclass.
- **Extract superclass (ES)** introduces a new common superclass for two or more classes having similar methods, and then applies *Pull-up method*. This may be necessary when those classes do not have a common superclass.
- **Extract utility-class (EU)** is applicable in situations, where similar functions are found in different classes, but those classes do not conceptually fit to undergo a common superclass. A new class is introduced that accommodates a method generalizing the similar methods that need to be removed from those classes.

Besides these prominent refactoring patterns, other low level refactoring operations such as, *identifier renaming*, *method parameter re-ordering*, *changes in type declarations*, *splitting of loops*, *substitution of conditionals*, *loops*, *algorithms*, and *relocation of method or field* may be necessary to produce generalized blocks of code from near-miss (similar, but not exact duplicate) clones.

For code clone refactoring, these refactoring operators will operate on groups of clone fragments having two or more members. We refer to such clone-groups as the refactoring *operands* or *candidates*. Thus, a *refactoring activity* (or simply, refactoring) r can be formalized as,

$$r = \langle op, g \rangle, \text{where } op \in \{EM, PM, ES, EU, ...\}$$

and g is the clone-group, which the refactoring operator op operates on. Note that more than one refactoring operators may be needed to refactor the same clone-group, and thus a complete refactoring of a clone-group may require more than one refactoring activities.

III. Estimation of Refactoring Effort

The effort required for code clone refactoring is likely to depend on the type of refactoring operators and operands. For instance, applying the *extract method* refactoring pattern on exact duplicate code clones will require less effort than that for applying on near-miss code clones. Moreover, refactoring clone code snippets that are scattered across different locations of the codebase and/or inheritance hierarchy may require relatively more effort than that for refactoring clones residing cohesively at certain location of the codebase. To address these issues, we propose a code clone *refactoring effort model* for procedural and OO software systems as follows.

Let us consider a clone-group, $g = \{c_1, c_2, c_3, \ldots, c_n\}$ is extracted as a set of refactoring candidates, where c_i $(1 \leq i \leq n)$ is a clone fragment inside method m_i, which is a member of class C_i hosted in file F_i contained in directory D_i. Mathematically,

$$\begin{aligned} c_i \frown m_i \frown C_i \frown F_i \frown D_i, &\quad \text{for object-oriented code.} \\ c_i \frown m_i \frown F_i \frown D_i, &\quad \text{for procedural code.} \end{aligned}$$

Here, the symbol $\frown$ indicates a containment relationship. $x \frown y$ means, x is contained in y, in other words, y contains x. The relationship preserves transitive property, i.e., $x \frown y \frown z \Rightarrow x \frown z$. Thus, the set $C(g)$ of all classes hosting the clone fragments in g can be defined as,

$$C(g) = \{C_i \quad | \ \forall \ c_i \in g, c_i \frown C_i\}.$$

A. Context Understanding Effort

The applicability of refactoring on certain code clones is largely dependent on the context. Therefore, before refactoring, the developer needs to understand the context pertaining to the refactoring candidate at hand. For understanding the context, the developer needs to examine two things: the caller-callee delegation of methods and the inheritance hierarchy.

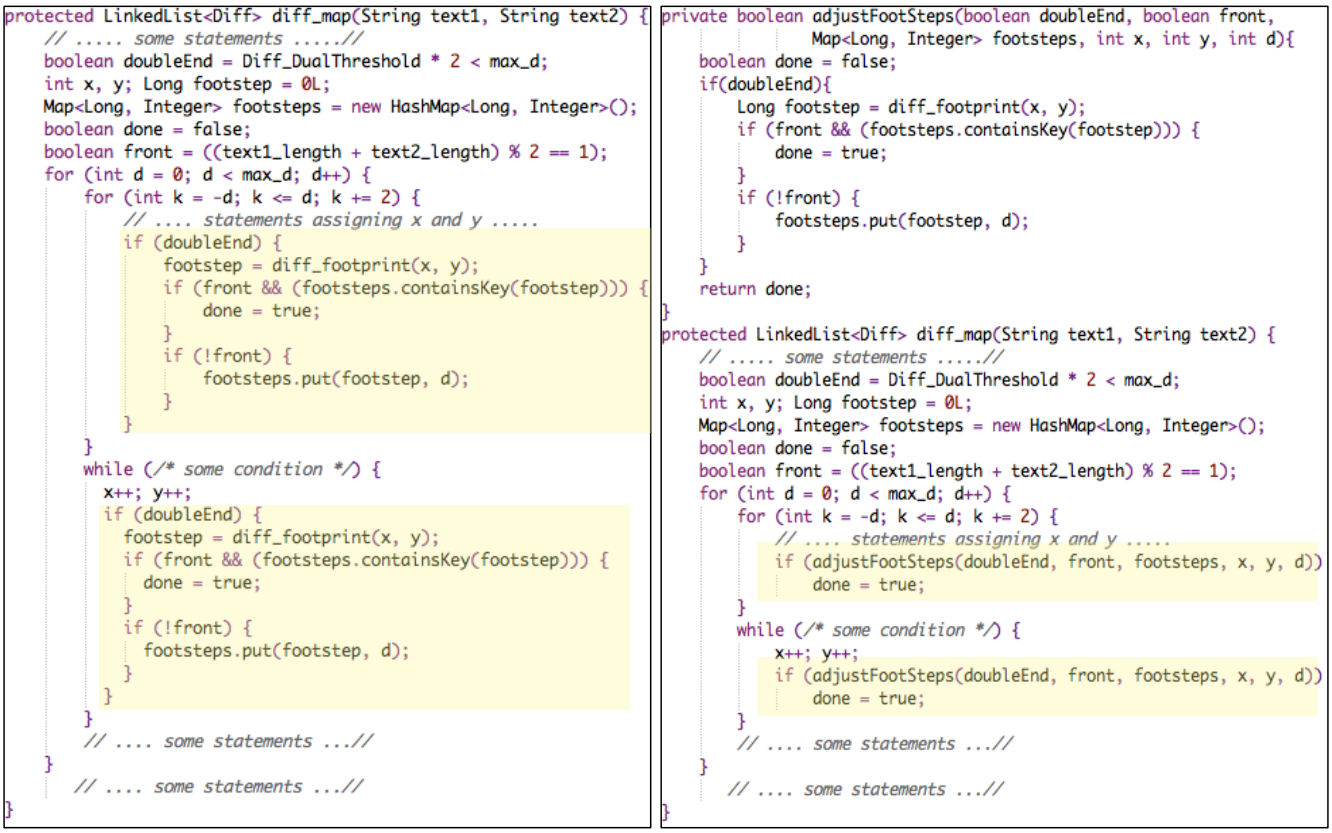
Effort for Understanding Method Delegation. To understand the delegation of methods involving the concerned code fragment $c_i \in g$, the developer needs to understand the chain of methods that can be reached from c_i via caller-callee relationships. Let, $M_r(c_i)$ be the set of all such methods. The developer will also need to comprehend the set $M_f(c_i)$ of all the methods from which c_i can be reached via caller-callee relationships.

Then, the set of methods required to investigate for understanding the delegation effort concerning c_i, is determined as,

$$delegation(c_i) = M_f(c_i) \cup M_r(c_i) \cup \{m_i\}.$$

Hence, for understanding delegation concerning all the clone fragments in g, the set of methods required to examine, becomes

$$Delegation(g) = \bigcup_{c_i \in C} delegation(c_i).$$



```
protected LinkedList<Diff> diff_map(String text1, String text2) {
    // ..... some statements .....//
    boolean doubleEnd = Diff_DualThreshold * 2 < max_d;
    int x, y; Long footstep = 0L;
    Map<Long, Integer> footsteps = new HashMap<Long, Integer>();
    boolean done = false;
    boolean front = ((text1_length + text2_length) % 2 == 1);
    for (int d = 0; d < max_d; d++) {
        for (int k = -d; k <= d; k += 2) {
            // .... statements assigning x and y .....
            if (doubleEnd) {
                footstep = diff_footprint(x, y);
                if (front && (footsteps.containsKey(footstep))) {
                    done = true;
                }
                if (!front) {
                    footsteps.put(footstep, d);
                }
            }
        }
        while (/* some condition */) {
          x++; y++;
          if (doubleEnd) {
            footstep = diff_footprint(x, y);
            if (front && (footsteps.containsKey(footstep))) {
              done = true;
            }
            if (!front) {
              footsteps.put(footstep, d);
            }
          }
        }
        // .... some statements ...//
    }
      // .... some statements ...//
}
```

```
private boolean adjustFootSteps(boolean doubleEnd, boolean front,
                Map<Long, Integer> footsteps, int x, int y, int d){
    boolean done = false;
    if(doubleEnd){
        Long footstep = diff_footprint(x, y);
        if (front && (footsteps.containsKey(footstep))) {
            done = true;
        }
        if (!front) {
            footsteps.put(footstep, d);
        }
    }
    return done;
}
protected LinkedList<Diff> diff_map(String text1, String text2) {
    // ..... some statements .....//
    boolean doubleEnd = Diff_DualThreshold * 2 < max_d;
    int x, y; Long footstep = 0L;
    Map<Long, Integer> footsteps = new HashMap<Long, Integer>();
    boolean done = false;
    boolean front = ((text1_length + text2_length) % 2 == 1);
    for (int d = 0; d < max_d; d++) {
        for (int k = -d; k <= d; k += 2) {
            // .... statements assigning x and y .....
            if (adjustFootSteps(doubleEnd, front, footsteps, x, y, d))
                done = true;
        }
        while (/* some condition */) {
            x++; y++;
            if (adjustFootSteps(doubleEnd, front, footsteps, x, y, d))
                done = true;
        }
        // .... some statements ...//
    }
      // .... some statements ...//
}
```

Fig. 1. Example of clone refactoring in VisCad: the method on the top-right corner is extracted by generalizing the clone pairs (shaded blocks on the left)

Thus, for the clone-group g, the total effort for understanding method delegation understanding can be estimated as,

$$E_d(g) = \sum_{m \in Delegation(g)} LOC(m)$$

where, $LOC(m)$ computes the total lines of code in method m including the comments, but excluding all blank lines.

Effort for Understanding Inheritance Hierarchy. Suppose, $C_p(g)$ be the set of all lowest/closest common superclasses of all pairs of classes in $C(g)$. The developer also needs to understand those classes in the inheritance hierarchy that have been overridden or referenced to method m_i containing any code clone $c_i \in g$. Let, $C_s(g)$ be the set of all such classes. Then $C_h(g) = \{C_p(g) \cup C_s(g) \cup C(g)\}$ becomes the set of all classes required to examine for understanding the inheritance hierarchy concerning the code clones in g, and the effort $E_h(g)$ required for this can be estimated as,

$$E_h(g) = \sum_{C \in C_h(g)} LOC(C).$$

B. Effort for Code Modifications

To perform refactoring on the refactoring candidates, the developer usually needs to modify portions of source code.

Token Modification Effort. Developer's source code modification activities typically include modifications in the program tokens (e.g., identifier renaming). Let, $T = \{t_1, t_2, t_3, \ldots, t_k\}$ be the set of tokens such that a token $t_i \in T$ is required to be modified to t'_i, and the edit distance between t_i and t'_i is denoted as $\delta(t_i, t'_i)$. Then the total effort $E_t(g)$ for token modifications can be estimated as,

$$E_t(g) = \sum_{i=1}^{k} \delta(t_i, t'_i).$$

Code Relocation Effort. When the developers need to move a piece of code from one place to another, they typically select a block of adjacent statements and relocate them all at a time. Hence, the code relocation effort $E_r(g)$ can be estimated as, $E_r(g) = |\beta|$, where β is the set of all non-adjacent blocks of code that need to be relocated to perform the refactoring.

C. Navigation Effort

Effort for source code comprehension, modification and relocation is also dependent on the number of files and directories involved, and their distributions in the file-system hierarchy. To capture this, our effort model includes the notion of navigation effort, $E_n(g)$, calculated as follows.

$$E_n(g) = |F_d(g) \cup F_h(g)| + |D_d(g) \cup D_h(g)| + DCH(g) + DFH(g)$$

where,

$$\begin{aligned}
F_d(g) &= \{F_i \mid m_i \frown F_i, m_i \in Delegation(g)\} \\
F_h(g) &= \{F_i \mid C_i \frown F_i, C_i \in C_h(g)\} \\
D_d(g) &= \{D_i \mid F_i \frown D_i, F_i \in F_d(g)\} \\
D_h(g) &= \{D_i \mid F_i \frown D_i, F_i \in F_h(g)\} \\
DCH(g) &= \max_{C_i, C_j \in C_h(g)} \{\partial(C_i, C_j)\} \\
DFH(g) &= \max_{F_i, F_j \in F_d(g) \cup F_h(g)} \{\eth(F_i, F_j)\}
\end{aligned}$$

Here, $DCH(g)$ refers to the *dispersion of class hierarchy* having $\partial(C_i, C_j)$ denoting the distance between class C_i and class C_j in the inheritance hierarchy. More detail about $DCH(g)$ can be found elsewhere [11]. $DFH(g)$ is a similar metric that captures the *dispersion of files*, and $\eth(F_i, F_j)$ denotes the distance between files F_i and F_j in the file-system hierarchy.

Thus, the total effort $E(g)$ needed to refactor clone-group g is estimated as,

$$E(g) = w_d \times E_d(g) + w_h \times E_h(g) + w_t \times E_t(g) + w_r \times E_r(g) + w_n \times E_n(g)$$

where w_d, w_h, w_t, w_r, and w_n are respectively the wights on the efforts for understanding method delegation, understanding inheritance hierarchy, token modification, code relocation, and navigation. By default, they are set to one, but the software engineer may assign different weights to penalize certain types of efforts.

IV. Prediction of Refactoring Effects

The expected benefit from code clone refactoring is the structural improvement in the code base, which should also enhance the software design quality. Obvious expected benefits include reduced source lines of code (SLOC), less redundant code, and so on. For procedural code, procedural metrics (e.g., SLOC, Cyclomatic Complexity) as well as structural metrics (e.g., fan-in, fan-out, and information flow) can be used to estimate software quality after refactoring. For object-oriented systems, these metrics can be supplemented by object-oriented design metrics, such as *QMOOD* [3] and *Chidamber-Kemerer* [6] metric suites. Quantitative or qualitative estimation of the effect of refactoring on the quality metrics can be possible before the actual application of the refactoring [5], [20], [26], [29], [30].

Having chosen a suitable set of quality attributes, let, $Q = \{q_1, q_2, q_3, \ldots, q_\eta\}$ be the set of quality attribute values before refactoring, and $Q_r = \{q'_1, q'_2, q'_3, \ldots, q'_\eta\}$ be the estimated values of those quality attributes after applying refactoring r. The improvement in quality can be assessed by comparing the quality before and after refactoring. Hence, the total gain in quality $\overline{Q}_r$ from refactoring r can be estimated as,

$$\overline{Q}_r = \sum_{j=1}^{\eta} \vartheta_j \times (q'_j - q_j)$$

where ϑ_j is the weight on the j^{th} quality attribute. By default, $\vartheta_j = 1$, but the software practitioner can assign different values to impose more or less emphasis on certain quality attributes.

In our work, we use the QMOOD metric suite for estimating the effect of refactoring on object-oriented codebase. QMOOD is a prominent quality model for object-oriented systems, which is widely used by many other researchers [5], [19], [20]. We choose QMOOD, mainly because, this quality model has the advantage that it defines six high level design quality attributes (Table I) from the 11 lower level structural property metrics (Table II).

TABLE I
QMOOD formula for quality attributes [3]

Attribute		Formula
Reusability	=	$-0.25 \times$ DCC $+ 0.25 \times$ CAM $+ 0.5 \times$ CIS $+0.5 \times$ DSC
Flexibility	=	$0.25 \times$ DAM $- 0.25 \times$ DCC $+ 0.5 \times$ MOA $+0.5 \times$ NOP
Understandability	=	$-0.33 \times$ ANA $+ 0.33 \times$ DAM $- 0.33 \times$ DCC $+0.33 \times$ CAM $- 0.33 \times$ NOP $- 0.33 \times$ NOM $-0.33 \times$ DSC
Functionality	=	$0.12 \times$ CAM $+ 0.22 \times$ NOP $+ 0.22 \times$ CIS $+0.22 \times$ DSC $+ 0.22 \times$ NOH
Extendability	=	$0.5 \times$ ANA $- 0.5 \times$ DCC $+ 0.5 \times$ MFA $+0.5 \times$ NOP
Effectiveness	=	$0.2 \times$ ANA $+ 0.2 \times$ DAM $+ 0.2 \times$ MOA $+0.2 \times$ MFA $+ 0.2 \times$ NOP

TABLE II
QMOOD metrics for design properties [3]

Design Property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

V. Refactoring Constraints

Among the applicable refactoring activities, there may be conflicts and dependencies [21]. The application of one refactoring may cause elements of other refactorings disappear, and thus invalidate their applicability [5], [20], [21]. Besides such *mutual exclusion* on conflicting refactorings, the application of one refactoring may also reveal new refactoring opportunities, as suggested by Lee et. al. [20]. We understand that this is largely due to the composite structure of certain refactoring patterns, where one larger refactoring is composed of several smaller core refactorings [1]. For example, when *extract superclass* refactoring is applied, *pull-up method* is also applied as a part of it. In other words, *pull-up method*, at times, may require extraction of new superclass.

There may also be *sequential dependencies* between refactoring activities [20], [21]. Constraints of *mutual inclusion* may also arise when the application of one refactoring will necessitate the operation of certain other refactorings [31]. Moreover, the organization's management may also impose *priorities* on certain refactoring activities [5], for example, lower priorities on refactoring clones in the critical parts of the system. We identify such priorities as soft constraints in addition to the following three types of hard constraints.

Definition 1 (Sequential dependency): Two refactorings r_i and r_j are said to have sequential dependency, if r_i cannot be applied after r_j. This is denoted as, $r_j \nrightarrow r_i$ or $r_i \nleftarrow r_j$.

Definition 2 (Mutual exclusion): Two refactorings r_i and r_j are said to be mutually exclusive, if both $r_i \nrightarrow r_j$ and $r_i \nleftarrow r_j$ holds. The mutual exclusion between r_i and r_j is denoted as, $r_i \nleftrightarrow r_j$ or $r_j \nleftrightarrow r_i$.

Definition 3 (Mutual inclusion): Two refactorings r_i and r_j are said to be mutually inclusive, if r_i is applied, r_j must also be applied before or after r_i, and vice versa. This is denoted as $r_i \leftrightarrow r_j$ or $r_j \leftrightarrow r_i$.

The complete independence of r_i and r_j is expressed as $r_i \perp r_j$ or $r_j \perp r_i$. For further detail about the refactoring constraints with concrete examples, interested readers are referred to elsewhere [5], [20], [21], [31].

VI. Formulation of Refactoring Schedule

Upon identification of all the hard and soft constraints pertaining to a scheduling problem instance, it becomes very difficult to compute an optimal refactoring schedule aiming to maximize code quality while minimizing efforts. Such a problem is known to be NP-hard [5], [19], [20]. Finding the optimum solution for such problems practically becomes too expensive (time consuming), and thus, a feasible optimal (near-optimum) solution is desired. However, the problem is by nature a CSOP. We thus model the problem as a CSOP and solve it by applying *constraint programming* technique, which no one attempted before.

Having identified the set $\mathcal{R}$ of all potential refactoring activities, we define two decision variables $\mathcal{X}_r$ and $\mathcal{Y}_r$, such that,

$$\mathcal{X}_r = \begin{cases} 0 & \text{if } r \in \mathcal{R} \text{ is not chosen} \\ 1 & \text{if } r \in \mathcal{R} \text{ is chosen} \end{cases}$$

$$\mathcal{Y}_r = \begin{cases} 0 & \text{if } r \in \mathcal{R} \text{ is not chosen} \\ k & \text{if } r \in \mathcal{R} \text{ is chosen as the } k^{th} \text{ activity} \end{cases}$$

where $1 \leq k \leq |\mathcal{R}|$.

We also define a $|\mathcal{R}| \times |\mathcal{R}|$ constraint matrix $\mathcal{Z}$ to capture the constraints and sequential dependencies between refactorings r_i and r_j, such that,

$$\mathcal{Z}_{ij} = \begin{cases} 0 & \text{if } r_i \perp r_j \\ 1 & \text{if } r_i \nleftrightarrow r_j \\ +2 & \text{if } r_j \nrightarrow r_i \text{ and } r_i \leftrightarrow r_j \\ -2 & \text{if } r_i \nrightarrow r_j \text{ and } r_i \leftrightarrow r_j \\ +3 & \text{if } r_j \nrightarrow r_i, \text{ but neither } r_i \leftrightarrow r_j \text{ nor } r_i \nleftrightarrow r_j \\ -3 & \text{if } r_i \nrightarrow r_j, \text{ but neither } r_i \leftrightarrow r_j \text{ nor } r_i \nleftrightarrow r_j \end{cases}$$

Also note that, $\mathcal{Z}_{ij} = -\mathcal{Z}_{ji}$ or $\mathcal{Z}_{ij} = \mathcal{Z}_{ji} = 1$, for all $\langle i, j \rangle$.

Let, ρ_r be the priority on the refactoring r that operates on clone-group g_r. The CSOP formulation of the refactoring scheduling problem can be defined as follows.

$$\text{maximize} \quad \sum_{r \in \mathcal{R}} \mathcal{X}_r \rho_r (\overline{Q}_r - E(g_r)) \tag{1}$$

subject to,

$$\mathcal{X}_r + \mathcal{Y}_r \neq 1, \qquad \forall\, r \in \mathcal{R} \tag{2}$$

$$\mathcal{X}_{r_i} + \mathcal{X}_{r_j} = 2 \Rightarrow \mathcal{Y}_{r_i} \neq \mathcal{Y}_{r_j}, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{3}$$

$$\mathcal{Z}_{ij} - \mathcal{Z}_{ji} > 0 \Rightarrow \mathcal{Y}_{r_i} < \mathcal{Y}_{r_j}, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{4}$$

$$\mathcal{Z}_{ij} - \mathcal{Z}_{ji} < 0 \Rightarrow \mathcal{Y}_{r_i} > \mathcal{Y}_{r_j}, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{5}$$

$$|\mathcal{Z}_{ij}| = 1 \Rightarrow \mathcal{X}_{r_i} + \mathcal{X}_{r_j} \leq 1, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{6}$$

$$|\mathcal{Z}_{ij}| = 2 \Rightarrow (\mathcal{X}_{r_i} + \mathcal{X}_{r_j})\ \mathit{modulo}\ 2 = 0, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{7}$$

$$\sum_{r \in \mathcal{R}} \mathcal{X}_r \leq \mathcal{M} \tag{8}$$

Here, Equation 1 is the objective function for maximizing the code quality and minimizing the refactoring effort while rewarding refactoring activities having higher priorities. Equation 2 ensures that the decision variables $\mathcal{X}_r$ and $\mathcal{Y}_r$ are kept consistent as their values are assigned. Equation 3 enforces that no two refactorings are scheduled at the same point in the sequence. Equation 4 and Equation 5 impose the sequential dependency constraints (i.e., $r_i \nrightarrow r_j$) on feasible schedules. Mutual exclusion (i.e., $r_i \nleftrightarrow r_j$) and mutual inclusion (i.e., $r_i \leftrightarrow r_j$) constraints are enforced by Equation 6 and Equation 7 respectively. Equation 8 specifies that maximum $\mathcal{M}$ number of refactorings can be chosen for scheduling. By default $\mathcal{M} = |\mathcal{R}|$, but $\mathcal{M}$ can be set to a lower integer when a schedule of a certain number of refactoring activities is desired, due to limitation of time, resource, or the like.

We implemented the CSOP model applying constraint programming using *OPL* (Optimization Programming Language) in the `IBM ILOG CPLEX Optimization Studio 12.2`. Constraint programming is a recent methodology that combines techniques from AI and OR, and it has been proved to be very effective in solving combinatorial optimization problems, specially in the area of scheduling and planning [4], [35]. Over the past decade, a separate conference series[1] is held in quest of the scopes to integrate and combine AI and OR techniques in CP.

VII. Case Study

To evaluate our refactoring scheduler and the effort model, we conducted a case study on refactoring four software systems developed (or under development) in our software research lab[2]. The subject systems and their sizes in terms of source lines of code (SLOC) are described in Table III. All the subject systems shown in Table III are written in Java and the sizes of the systems in terms of SLOC exclude the comments and blank lines.

In particular, the case study was designed to address the following two research questions:

RQ1: Given a set of refactoring activities and a set of constraints for them, can our refactoring scheduler effectively compute conflict-free optimal scheduling of refactorings?

[1]International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)

[2]Software Research Lab, Department of Computer Science, University of Saskatchewan, Canada

TABLE III
SOFTWARE SYSTEMS SUBJECT TO THE CASE STUDY

Subject Systems	SLOC	Description
Mutation Framework	2901	Ongoing extended implementation of the mutation framework proposed by Roy and Cordy [23]
LIME [33]	3494	A source code comparison engine
gCad [25]	4563	A clone genealogy extractor
VisCad [2]	9323	A tool for analysis and visualization of code clones

TABLE IV
CODE CLONES IN THE SYSTEMS UNDER STUDY

Subject Systems	Clone Groups	Clone Fragments	Total Refactorings
Mutation Framework	21	62	72
LIME	20	55	67
gCad	28	91	93
VisCad	57	136	166

RQ2: Is the code clone refactoring effort model (described in Section III) useful in capturing and estimating the efforts required for performing the refactorings?

Typically, it is difficult and risky for the practitioners to refactor a codebase that they are not familiar with [19]. Rather, it is the developers, who are likely to know the best about the critical parts of the projects they develop, and thus they can better assess both the efforts and effects of refactoring, and prudently assign priorities on certain refactoring candidates. Therefore, to evaluate our refactoring scheduler, we chose projects (Table III) developed in our own research lab. This not only facilitated manual verification for correctness but also reduced the evaluation cost.

At the beginning of the case study, we described to the developers the objectives of the study, and provided them our refactoring effort model, as well as an initial list of refactoring operators that can be used for code clone refactoring. Then we demonstrated a catalog of common software refactoring patterns [10] to them, and showed them how some of those can be applied for code clone refactoring. We also described the QMOOD design property metrics to them, and upon discussion, came to a consensus to use the first six metrics (Table II) in our study. We all agreed that the rest of the metrics were too difficult to estimate through qualitative analysis, and even if attempted, possibly those would not provide meaningful values. Hence, we assumed that code clone refactoring will not affect those metrics, and the total gain in code/design quality (Section IV) was computed having values of changes in those metrics set to zero. It should be noted that all the developers were graduate students pursuing research in the area of software clones, and thus possess some knowledge and expertise in code clone analysis for refactoring.

A. Clone Detection

The first and foremost activity towards code clone refactoring is the detection of code clones from the underlying codebase. We used NiCad-2.6.3 [7] for detecting near-miss (code clones that are not exact duplicates, but share certain level of similarities) *block* clones of at least five lines in pretty-printed format. We used the 'blind-rename' option of NiCad having UPIT set to 30%. The 'blind-rename' option instructs NiCad to normalize the code snippets by renaming the identifiers before the comparison of code fragments. UPIT (Unique Percentage of Items Threshold) is a size-sensitive dissimilarity threshold, that NiCad uses to find near-miss code clones. For example, if UPIT is set to 0% without the 'renaming' option, NiCad detects only the exact clones; if the UPIT is 30% having the 'renaming' option set, NiCad detects two code fragments as clones if at least 70% of the normalized pretty-printed text lines are identical (i.e., if they are at most 30% different). NiCad reports code clones clustered into clone-groups based on their similarity.

TABLE V
EXAMPLE OF OPERATIONS AND EFFORTS FOR EXTRACT METHOD

Operations for extract method	Efforts
Produce signature of the target method	15
Copy clone fragment to the body of target method	1
Perform necessary modifications in the body	5
Replace clone fragments by calls to the extracted method	2
Total effort	23

B. Data Acquisition

The results of clone detection from the four subject systems were provided to the concerned developers, who then further analyzed the detected clones and re-arranged the groups when necessary, based on the suitability for refactoring within the context. For the analysis, the developers used VisCad [2], a code clone analysis and visualization tool developed in our research lab. For each of the systems, the number of clone-groups and the number of distinct blocks of code involved in those groups are presented in Table IV, which the developers identified as the potential candidates for refactoring.

Having the code clones organized into groups, the developers carried out further qualitative analysis to determine the strategies for refactoring each of those refactoring candidates. The identification of refactoring strategy in particular involved finding the appropriate refactoring operations, their order of application, and mutual dependencies (if any). For each of the clone-groups chosen for refactoring, the developers wrote down the sequence of operations that they would perform to refactor that clone-group. In determining the operations, the developers were free to choose any operations beyond the list of refactoring operators they were initially provided. The rightmost column of Table IV presents the total number of refactoring activities identified for each of the subject systems. The developers also noted down any restrictions in the ordering of the operations that must be followed to successfully refactor a clone-group. Any such ordering restrictions between clone-groups were recorded as well.

As an example, in Figure 1, we present a clone-pair (shaded blocks on the left) with partial context (surrounding code).

The example is excerpted from the source code of VisCad[3]. The developers chose to refactor them applying the *extract method* operation. The developers recorded further fine grained operations and required efforts in order, as shown in Table V. As explained by the developers, the effort for producing the method signature was estimated by twice the number of parameters (type and name) to the method, plus one each for method name, return type, and access modifier. Code modification effort was estimated by the number of words (tokens) added, deleted or modified.

For estimation of the refactoring efforts, we relied on the developers' opinions, and wanted to see to what extent our effort model was useful in estimating the effort of code clone refactoring. The developers were instructed to estimate efforts required for each refactoring activity they identified, or for each of the clone-groups as a whole they chose to refactor. Though they were provided the refactoring effort model, they were free to apply their own understanding and analytical evaluations for the efforts estimation. As the developers estimated refactoring efforts, at times, we observed and communicated with them to understand how they were estimating the efforts for refactoring.

In the estimation of quality gains expected from the refactorings of code clones, we again relied on the developers' judgements, which we feel is important in this context. Using the QMOOD design property metrics (Table II), it was relatively easy for the developers to estimate the quality gain expected from the refactoring of a clone-group. For example, to estimate the change in *design size* or *complexity*, the developers did not need to compute the total number of classes or methods (before and after the refactoring) in the system, they just had to estimate the changes in the number of classes or methods. For instance, the refactoring example presented in Figure 1 causes the complexity (number of methods) increase by one, and all other QMOOD design property metrics under consideration remain unaffected.

Next, the developers were instructed to assign non-zero priorities between -5 (the lowest priority) and $+5$ (the highest priority) to certain clone-groups they considered important in terms of the necessity and risks involved in refactoring them. The priorities were set to $+1$ for those clone-groups, which were left unassigned by the developers. For each of the systems, the developers identified some intentional code clones in particular parts of the systems. They considered some of them to be critical, and preferred not to take the risk of refactoring them. Taking the developers' opinions into consideration, we could have excluded those from our study. Instead, we had the lowest priority assigned to them for examining how our scheduler handles them in the scheduling process.

[3]The code is originally a part of `diff-match-patch`, an open-source library (available at http://code.google.com/p/google-diff-match-patch/) that VisCad makes use of. We deliberately chose to present this simple example, so that anyone can easily follow and verify.

C. Data Normalization

As described before, both the estimation of expected change in code/design quality and the refactoring efforts, as well as the priorities were sometimes set on refactoring of clone-groups as a whole. Thus, in situations, where the developers made those estimations for refactoring an entire clone-group, we equally distributed those estimations to all the refactoring operations involved in refactoring that particular clone-group.

Recall that, the scheduling of code clone refactoring activities can be optimized towards three dimensions: minimizing the refactoring efforts, maximizing the refactoring benefits, and maximizing the satisfaction of priorities. However, the ranges of values obtained along those dimensions were different. For example, the priorities ranged between $+5$ and -5, whereas the values of total refactoring efforts varied between 04 and 47. To prevent our scheduler getting biased towards any of the individual dimensions, we first normalized the values obtained for all the three dimensions using the following procedure.

Let, $S = \{v_1, v_2, v_3, \ldots, v_n\}$ be a set of values, then

$$norm(v_i) = \frac{v_i}{max\{|v_1|, |v_2|, |v_3|, \ldots, |v_n|\}}, \quad \forall v_i \in S$$

where, $norm(v_i)$ denotes the normalized value of v_i. The set S can be the set of values for all the refactorings along any of the three dimensions. The normalization actually brings the magnitudes of all those values between zero and one (i.e., $0 < |norm(v_i)| \leq 1$), and thus discards the inter-dimension influence of the magnitudes, while still preserving the relative ratios of magnitudes within dimensions.

D. Schedule Generation

The normalized data were fed to our scheduler implemented in OPL. The data and the OPL implementation of our scheduler are made available online[4] for the interested parties. In our study, we used the default settings in the estimation of total effort and quality gain, as described in Section III and Section IV. The scheduler, upon obtaining the data in valid OPL format, applies constraint propagation and domain reduction techniques [35] to generate the optimal solution as instructed. The scheduler was run on the `IBM ILOG CPLEX Optimization Studio 12.2` IDE having all its parameters set to the defaults. The experiments were executed on Windows XP operating system running on an Apple MacBookPro5,5 computer with Intel Core 2 Duo (2.26 GHz) processor.

We evaluated our scheduler in two phases. In the first phase, we compared our CP optimization with the greedy algorithm. For each of the subject systems, we first computed the refactoring schedule using our CP approach. Then we computed schedules applying approaches greedy towards each of the three dimensions (i.e., effort, quality, and priority) as described earlier. Intuitively, the minimum refactoring effort (i.e., zero effort) can be achieved by scheduling no refactoring

[4]http://usask.ca/~minhaz.zibran/pages/research.html

TABLE VI
COMPARISON OF CP AND GREEDY SCHEDULING APPROACHES

Subject systems	Scheduling approaches	Values at dimensions			Refac. chosen
		Prior.	Effort	Quality	
Mutation Framework	$Greedy^p$	20.06	21.94	18.53	40
	$Greedy^e$	9.63	6.06	10.04	20
	$Greedy^q$	18.16	21.82	19.64	42
	CP	9.34	7.86	11.48	20
LIME	$Greedy^p$	22.42	21.12	19.93	47
	$Greedy^e$	13.00	8.28	13.61	33
	$Greedy^q$	16.29	23.49	26.07	51
	CP	11.04	12.32	16.12	33
gCad	$Greedy^p$	19.65	21.62	20.00	41
	$Greedy^e$	9.61	9.53	11.57	28
	$Greedy^q$	12.05	23.48	25.98	44
	CP	6.69	15.19	17.99	28
VisCad	$Greedy^p$	36.14	32.57	25.71	66
	$Greedy^e$	16.12	18.63	13.20	40
	$Greedy^q$	29.02	33.81	34.32	72
	CP	15.33	15.78	21.90	40

Here, $Greedy^p$ = approach greedy towards priority satisfaction
$Greedy^e$ = approach greedy towards effort minimization
$Greedy^q$ = approach greedy towards quality gain

at all. Thus, in the application of the approach greedy towards refactoring efforts, we set a minimum number of refactorings that must be scheduled, which was equal to the number of refactorings scheduled by our CP scheduler. The values along all the three dimensions obtained from these scheduling approaches are presented in Table VI.

In the second phase of the evaluation, our goal was set to schedule roughly the 25% of the total number of refactorings for each of the subject systems. The developers of the concerned systems were instructed to do it manually (or, in the way they would do it without help from any automated scheduler). With the same goal, we executed our refactoring scheduler. The values along the three optimization dimensions, obtained from our CP scheduling and manual scheduling, are presented in Figure 2.

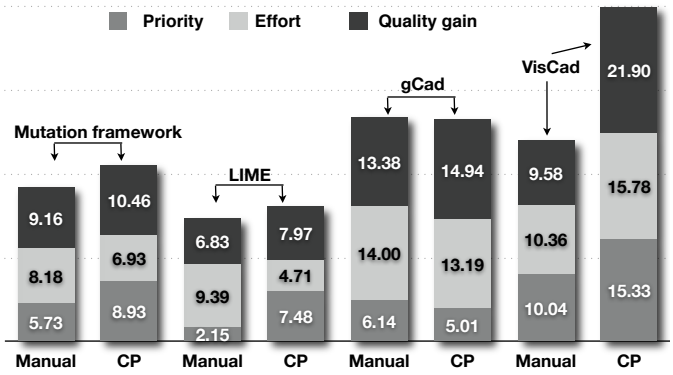


Fig. 2. Automated CP vs. manual scheduling

E. Findings

From our observations during the case study and the developers feedback, as well as the results presented in Table VI and Figure 2, we now can confidently answer the the two research questions formulated before.

> **Answer to RQ1:** Yes. Given a set of refactoring activities and constraints among them, our refactoring scheduler can effectively compute a conflict-free optimal schedule of refactorings.

For all the subject systems, as seen in Table VI, our CP scheduler computes the optimal refactoring schedule by efficiently making balance among the three optimization dimensions (i.e., effort, quality, and priority). For the smaller systems (Mutation Framework, LIME, and gCad) the greedy approaches, especially the approach greedy towards refactoring efforts, closely competes with our CP approach. As the sizes of the systems in terms of SLOC and the number of candidate refactorings increases, our CP scheduler outperforms the greedy schedulers, which is vivid for the largest system, VisCad.

The risks of refactorings can be best estimated by subjective analysis by the individuals who are familiar with the underlying codebase. Quantitative measurement of such risks would be very difficult, if not impossible. However, the risks of refactorings can be expected to be positively proportional to the number of refactorings. In this sense, our CP scheduler also minimizes the risks of refactorings, as seen in the rightmost column of Table VI, the optimal schedule obtained from our scheduler always includes the least number of refactorings, compared to those from the greedy scheduling.

As expected, our CP scheduler always outperformed manual scheduling for all the four subject systems (Figure 2). The superiority in the optimality of the schedules (in terms of efforts, quality gain, and priorities) obtained from our CP scheduler compared to manual scheduling, gradually increased as the sizes of the systems and the number of candidate refactorings increased. Our CP scheduler took no more than five seconds in computing any of the refactoring schedules presented in this paper, whereas, for manual scheduling the developers had to spend several hours depending on the number of refactoring candidates and the constraints involved.

> **Answer to RQ2:** Yes. The code clone refactoring effort model (described in Section III) is useful in capturing and estimating the efforts required for performing the refactorings.

Regarding the refactoring effort model, the developers' direct feedback was that the model was useful, and it guided them in the estimation of the efforts. Moreover, the developers expressed that an automated tool offering accurate calculations according to the model, would be of immense help in this regard. Our observations on the developers (while they were estimating the refactoring efforts), also support this proposition. Some of the developers argued that the effort model was useful for quantitative estimation of refactoring efforts, but it alone could not capture the risks involved in code clone refactoring. However, everyone agreed in the matter that the effort model and the priority scheme in combination were effective in capturing both the efforts and the risks. The first author of this paper also worked as one of the developers and experienced that the model was effective in general.

F. Threats to the Validity

In the case study, we relied on the qualitative evaluations of the developers in the estimation of both refactoring efforts and effects. There is a possibility to question on the individual developer's ability to correctly estimate those. However, our refactoring scheduler (the primary contribution of this paper) is independent of how the refactoring data are obtained. Given a set of refactorings along with their mutual constraints and priorities, as well as, the estimation of refactoring efforts and changes in code/design quality, how effectively our scheduler can compute the optimal schedule, becomes the question for evaluation. In this regard, we compared our CP scheduler with the greedy approaches, as well as the manual procedure, and our scheduler was found to have outperformed both those techniques. We manually investigated all the refactoring schedules obtained from our CP scheduler, and confirmed correctness in terms of constraint satisfaction and optimality.

This work is a significant extension to our recently accepted initial concept [34], where the reviewers suggested to evaluate our refactoring effort model from the developers' perspective. We also understand that manual scheduling of refactorings may be too difficult for large systems, but for much smaller systems, the developers can be expected to efficiently estimate the efforts and risks involved in refactoring the system. Also note that, the purpose of the case study was not only to evaluate our refactoring scheduler, but also the refactoring effort model. Hence, we intentionally chose the in-house systems and the concerned developers for our case study. The subject systems being in-house and fairly small, enabled the developers to estimate the refactoring efforts and effects with high probability of accuracy. The same fact also facilitated manual investigation of constraints satisfaction and optimality of the refactoring schedules computed by our scheduler.

Our refactoring effort model suggests some fine grained computations (e.g., token modification efforts in terms of edit distances), which were not possible for the developers to perform by hand. For this reason, during estimation of the refactoring efforts, the developers used our effort model as a guideline, and followed this as much as it was feasible. However, this does not affect the functionality of our refactoring scheduler. Rather, our observations and the developers' expressions towards the need of tool support for the effort model further validate its necessity and effectiveness.

Though in this study, we intentionally depended on the manual estimation of refactoring efforts and effects, we also showed how these can be automatically computed making use of our effort model and a design quality metric suite.

VIII. Related Work

Until recently, many research have been conducted towards effective identification and removal of code smells from the codebase. Fowler et. al. [9] introduced a catalog of 72 refactoring patterns, and till date, the number has increased to 93 [10]. Since our work is focused on scheduling of *code clone* refactoring, we confine our discussion on those work that deal with scheduling of refactoring this code smell.

The work of Bouktif et. al. [5], Lee et. al. [20], and Liu et. al. [19] closely relate to ours. Bouktif et. al. [5] formulated the refactoring problem as a constrained *Knapsack problem* and applied a metaheuristic *genetic algorithm* (GA) to obtain an optimal solution. However, they ignored the constraints that might exist among the refactorings. Lee et. al. [20] applied *ordering messy GA (OmeGA)*, whereas, Liu et. al. [19] used a heuristic algorithm to schedule refactoring of code bad smells in general. Both of those work took into account the conflict and sequential dependencies among the refactoring activities, but missed the constraints of mutual inclusion and refactoring efforts. Our work significantly differs from all those work in two ways. First, for computing the refactoring schedule, we applied constraint programming approach, which is different from theirs. Second, we took into account a wide category of refactoring constraints and dimensions of optimizations, some of which they ignored, as summarized in Table VII. Although Bouktif et. al. [5] proposed a brief effort model for code clone refactoring, their model was for procedural code only, whereas, our effort model is applicable to not only procedural but also to OO source code.

O'Keeffe et. al. [18] conducted an empirical comparison of *simulated annealing* (SA), *GA* and *multiple ascent hill-climbing* techniques in scheduling refactoring activities in five software systems written in Java. They reported that among those AI techniques, the hill-climbing approach performed the best. However, we used CP, which combines the strengths of both AI and OR techniques [4], and thus led to our belief that CP would be a better choice for solving such scheduling problems. Therefore, we opted out of the immediate comparison of our CP approach against those AI techniques. Nonetheless, such a comparative evaluation could be an interesting study, which we might do in future as well.

A number of methodologies [8], [17], [27], [28], [31] and metric based tools such as `CCShaper` [12] and `Aries` [11] have been proposed for semi-automated extraction of code clones as refactoring candidates. Several tools, such as `Libra` [13] and `CnP` [14], have been developed for providing support for simultaneous modification of code clones. Our work is neither on finding potential clones for refactoring nor on providing editing support to apply refactorings. Rather, we focus on efficient scheduling of those refactoring candidates, which is missing in those tools. However, the metrics used in those tools can be exploited to estimate the refactoring effort and expected gain.

IX. Conclusion and Future Work

In this paper, we presented our work towards conflict-aware optimal scheduling of code clone refactorings. To estimate the refactoring effort, we proposed an effort model for refactoring code clones in OO source code. Moreover, the risks of refactoring are captured in a priority scheme. Considering a diverse category of refactoring constraints, we modelled the scheduling of code clone refactoring as a CSOP, and implemented the model using the CP technique. To the best of our knowledge, ours is the first refactoring effort model for

TABLE VII
COMPARISON OF SOFTWARE REFACTORING SCHEDULERS

Refactoring scheduler	Scheduling approach	Refactoring effort	Quality gain	Sequential dependency	Mutual exclusion	Mutual inclusion	Priorities satisfaction
Bouktif et. al. [5]	GA	√	√				
Lee et. al. [20]	OmeGA		√	√	√		
Liu et. al. [19]	Heuristic		√	√	√		
Our Scheduler	CP	√	√	√	√	√	√

OO code corpus, and our CP approach is a unique technique that no one in the past reported to have applied in this regard. Having been equipped with the strengths from both AI and OR, the CP approach has been proved to be very effective in solving scheduling problems [4], [35]. Our CP scheduler computes the conflict-free schedule making optimal balance among the three optimization dimensions: minimized refactoring effort, maximized quality gain, and satisfaction of higher priorities.

To evaluate our approach, we conducted a case study with four in-house software systems and their developers. Through comparison with greedy and manual approaches, we showed that our CP scheduler outperformed those techniques. Our refactoring effort model was also found to be useful for estimating the efforts required for code clone refactoring. Our immediate future plan includes the evaluation of our scheduler in larger context involving both diversified open-source and industrial software systems written in different programming languages, and finally integration of a smart scheduler with the code clone management tool [33], we have been developing.

Acknowledgments: The authors acknowledge the contributions of Ripon Saha, Muhammad Asaduzzaman, Sharif Uddin, and Mohammad Khan for participating in the case study and helping in the evaluation of our code clone refactoring scheduler and the effort model. The authors also thank Nils Göde for his valuable suggestions and comments on this work, who also encouraged us to conduct the study with in-house software systems and their developers.

REFERENCES

[1] D. Advani, Y. Hassoun, and S. Counsell. Understanding the complexity of refactoring in software systems: a tool-based approach. *Intl. J. Gen. Sys.*, 35(3): 329–346, 2006.

[2] M. Asaduzzaman, C. K. Roy, and K. Schneider. VisCad: Flexible Code Clone Analysis Support For NiCad. In *IWSC*, 2 pp., 2011 (to appear).

[3] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Engg.*, 28(1): 4–17, 2002.

[4] R. Barták. Constraint programming: In pursuit of the holy grail. In *WDS* (invited lecture), 10 pp., 1999.

[5] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A Novel Approach to Optimize Clone Refactoring Activity. In *GECCO*, July 8 –12, 2006.

[6] S. Chidamber and C. Kemerer. A metric suite for object-oriented design. *IEEE Trans. Softw. Engg.*, 25(5): 476–493, 1994.

[7] J. R. Cordy and C. K. Roy. The NiCad Clone Detector. In *ICPC*, 2 pp., 2011 (tool demo to appear).

[8] S. Ducasse, M. Rieger, and G. Golomingi. Tool Support for Refactoring Duplicated OO Code. In *WOOT*, pp. 177–178, 1999.

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

[10] M. Fowler. Refactoring Catalog, http://refactoring.com/catalog/, (last access: 12 April, 2011).

[11] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring Support Tool Code Clone. In *3-WoSQ*, pp. 1 – 4, 2005.

[12] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring Support Based on Code Clone Analysis. *PROFES, LNCS 3009*, pp. 220–233, Springer-Verlag, 2004.

[13] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous Modification Support based on Code Clone Analysis. In *APSEC*. pp. 262–269, 2007.

[14] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC*, pp. 238–242, 2009.

[15] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner. Do Code Clones Matter? In *ICSE*, pp. 485-495, 2009.

[16] C. Kapser and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Emp. Soft. Engg.* 13(6): 645–692, 2008.

[17] E. Kodhai, V. Vijayakumar, G. Balabaskaran, T. Stalin, and B.Kanagaraj. Method Level Detection and Removal of Code Clones in C and Java Programs using Refactoring. In *IJJCET*, pp. 93 – 95, 2010.

[18] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.: Res. Pract.*, 20: 345 – 364, 2008.

[19] H. Liu, G. Li, Z. Ma, and W. Shao. Conflict-aware schedule of software refactorings. *IET Softw.*, 2(5): 446–460, 2008.

[20] S. Lee, G. Bae, H. S. Chae, and D. Bae, and Yong Rae Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Softw. Pract. Exper.*, Wiley Online Library, 2010.

[21] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *J. Softw. and Syst. Modeling*, 6(3): 269–285, 2007.

[22] J. Pérez, Y. Crespo, B. Hoffmann, and Tom Mens. A case study to evaluate the suitability of graph transformation tools for program refactoring. *Intl. J. Softw. Tools Tech. Transfer*, 12: 183–199, 2010.

[23] C. K. Roy and J. R. Cordy. A Mutation/Injection-based Automatic Framework for Evaluating Clone Detection Tools. In *Mutation*, pp. 157–166, 2009.

[24] M. Rieger, S. Ducasse, and M. Lanza. Insights into System-wide Code Duplication. In *WCRE*, pp. 100–109, 2004.

[25] R. K. Saha, C. K. Roy, and K. A. Schneider. An Automatic Framework for Extracting and Classifying Near-Miss Clone Genealogies. In *ICSM*, 10 pp., 2011 (submitted for review).

[26] H. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of OO design quality and its automation?. In *ICSM*, pp. 154–162, 2000.

[27] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a Refactoring Guideline Using Code Clone Classification. In *WRT*, pp. 6:1–6:4, 2008.

[28] S. Schulze and M. Kuhlemann. Advanced Analysis for Code Clone Removal. In *WSR*, 2009.

[29] F. Simon, F. Steinbrucker, and C. Lewerentz. Metrics based refactoring. In *CSMR*, pp. 30–38, 2001.

[30] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *CSMR*, pp. 183–192, 2003.

[31] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *METRICS*, 10 pp., 2005.

[32] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy. Analyzing and Forecasting Near-miss Clones in Evolving Software: An Empirical Study. In *ICECCS*, 10 pp., 2011 (to appear).

[33] M. F. Zibran and C. K. Roy. Towards Flexible Code Clone Detection, Management, and Refactoring in IDE. In *IWSC*, 2 pp., 2011 (to appear).

[34] M. F. Zibran and C. K. Roy. Conflict-aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach. In *ICPC* (Student Symposium), 4 pp., 2011 (to appear).

[35] M. F. Zibran. A Multi-phase Approach to University Course Timetabling. *M.Sc. Thesis*, Department of Mathematics and Computer Science, University of Lethbridge, Canada, 125 pp., 2007.