Conflict-aware Optimal Scheduling of Prioritized Code Clone Refactoring

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

Abstract

Duplicated or similar source code, also known as code clones, are possible malicious ‘code smells’ that may need to be removed through refactoring to enhance maintainability. Among many potential refactoring opportunities, the choice and order of a set of refactoring activities may have distinguishable effect on the design/code quality measured in terms of software metrics. Moreover, there may be dependencies and conflicts among those refactorings. The high-level decision makers may also impose priorities on certain refactoring activities. Addressing all these conflicts, priorities, and dependencies, a manual formulation of an optimal refactoring schedule is very expensive, if not impossible. Therefore, an automated refactoring scheduler is necessary to maximize benefit and minimize refactoring effort. However, the estimation of the efforts required to perform code clone refactoring is a challenging task.

This paper makes two contributions. First, we propose an effort model for the estimation of code clone refactoring efforts. Second, we propose a constraint programming (CP) approach for conflict-aware optimal scheduling of code clone refactoring. We empirically evaluated of our effort model and the clone refactoring scheduler. A qualitative evaluation of the effort model from the developers perspective suggests that the model is complete and useful for code clone refactoring effort estimation. We also quantitatively compared our refactoring scheduler with other well known scheduling techniques such as the genetic algorithm, greedy approaches and linear programming. Our experiment suggests that the proposed CP-based approach outperforms other approaches we considered.
1 Introduction

Code clone (duplicate or near-duplicate source code) is a well-known code smell [12, 18]. Programmers’ copy-paste-modify practice is regarded as one of the main reasons for such intentional clones that are beneficial in many ways [19] during the development phase. For example, code cloning is a code reuse mechanism commonly adopted by developers for increased productivity. Cloning of existing code, which is already known to be flawless, might save the developers from probable mistakes they might have made if they had to implement the same from scratch. It also saves time and effort in devising the logic and typing the corresponding textual code. Code cloning may also help in decoupling classes or components and facilitate independent evolution of similar feature implementations. However, unintentional clones also appear due to a number of reasons. For example, the use of design patterns, frameworks, and similar APIs may result in unintentional code clones. Previous studies reported that software systems might have 9%-17% [42] duplicated code, up to 50% [30].

Code clones may also be detrimental in many cases. Obviously, redundant code may inflate the code base and may increase resource requirements. Such increases in resource requirements may be crucial for embedded systems and systems such as hand-held devices, telecommunication switches, and small sensor systems. 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 bugs. Nevertheless, in many cases, code clones are unavoidable or even desirable. Yet, to prevent code inflation and reduce maintenance cost, the number of code clones should be minimized by applying justified refactoring. However, refactoring is not free, rather it is often risky as it might introduce new bugs and hidden dependencies. Moreover, not all clones can always be feasible targets of refactoring. Therefore, it is important to have a prioritized refactoring schedule of the potential refactoring candidates (i.e., clones that are feasible and beneficial for refactoring) so that the maintenance engineers can focus on a short list of refactoring candidates considering the existing constraints, potential benefits, risks, and available resources.

There are many patterns [12, 13] for refactoring source code in general. Given a context,
a refactoring pattern describes a sequence of refactoring activities (i.e., modification operations) that can be performed to improve code/design quality. However, not all of the refactoring patterns are directly applicable to code clone refactoring. The applicability of a certain refactoring largely depends on the context (e.g., dependency of the code fragment under consideration with the rest of the source code). Therefore, for code clones, refactoring activities and the relevant contexts must to be identified in the first place. The consequences of clone refactoring (e.g., impact on the code/design quality) should also be taken into account. The effort required for applying certain refactoring on the 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 refactorings may result in distinguishable impact on the overall code quality. Moreover, there may be sequential dependencies and conflicts among the refactorings, which lead to the necessity that, from all refactoring candidates a subset of non-conflicting refactorings be selected and ordered for application, such that the quality of the codebase is maximized while the required effort is minimized [43].

Automated software refactoring is often performed with the aid of graph transformation tools [27], where the available refactorings are applied without having been optimally scheduled [23]. The application order of the semi-automated (or manual) refactorings is usually determined implicitly by human practitioners. However, human efforts may be inefficient and error-prone specially for large systems. While experienced engineers may do it well, inexperienced practitioners may build a poor/infeasible schedule. The challenge is likely to be more severe when 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 refactorings is a justified need. However, such a scheduling of code clone refactoring is an \textit{NP-hard} problem [5, 22, 23] and thus, the complexity of a problem instance grows exponentially for large systems having many code clones.

In this regard, this paper makes two contributions. First, we introduce an \textit{effort model} for estimating the effort required to refactor code clones in procedural or object-oriented (OO) programs. Second, taking into account the \textit{effort model} and a wide variety of possible hard and soft constraints, we formulate the scheduling of code clone refactorings as a constraint satisfaction optimization problem (CSOP) and solve it by applying Constraint Programming (CP) techniques that aims to maximize benefits (measured in terms of changes in the code/design quality metrics).
while minimizing refactoring efforts.

To the best of our knowledge, ours is the first refactoring effort model for object-oriented source code and we are the first to apply CP techniques in the scheduling of code clone refactorings. We choose to adopt CP for two main reasons. First, CP is a natural fit for solving CSOPs such as scheduling problems. Second, this technique integrates the strengths from both artificial intelligence (AI) and operations research (OR) and has been shown efficient in solving CSOPs [4, 47].

To evaluate the effectiveness of our scheduler and the code clone refactoring effort model, we also conduct an empirical study on six 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.

This research is a significant extension to our previous work [46], in which we introduced the clone refactoring effort model and our CP scheduler. The initial evaluation of the CP scheduler was based on an empirical study with four subject systems and comparison with greedy and manual scheduling approaches. We extended the work in several directions. First, we developed additional schedulers using genetic algorithm (GA) and Linear Programming (LP) techniques. Second, we compared our CP scheduler with the GA and LP schedulers. Third, we extended the empirical study with two more subject systems and two more developers.

The remainder of the paper is organized as follows. In Section 2, we describe the terminologies and concepts necessary to follow the paper. In Section 3, we identify the refactoring patterns that are suitable for code clone refactoring. In Section 4, we describe our clone refactoring effort model. Section 5 discusses how the effect of refactoring can be estimated by a software engineer. In Section 6, we describe the possible constraints on refactorings. In Section 7, we present our CSOP formulation of the refactoring scheduling problem. In Section 9, we illustrate our empirical study to evaluate our refactoring scheduler and the effort model. In Section 10, we discuss related work. Finally, in Section 11, we conclude the paper in our directions to future research.
2 Background

In this section, we describe the terminologies and background necessary to follow the remainder of the paper.

Similar or duplicated code fragments are known as code clones. Over more than a decade of research on code clones, the following categorizing definitions of code clone have been widely accepted today [20, 29, 32, 39, 42, 44].

**Type-1 Clone:** Identical code fragments except for variations in white-spaces and comments are *Type-1* clones.

**Type-2 Clone:** Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout, and comments are called *Type-2* clones.

**Type-3 Clone:** Code fragments that exhibit similarity as of *Type-2* clones and also allow further differences such as additions, deletions, or modifications of statements are known as *Type-3* clones.

**Type-4 Clone:** Code fragments that exhibit identical functional behaviour but implemented through different syntactic structure are known as *Type-4* clones.

The granularity of the code clones (fragments) can be at different levels, such as the entire function bodies (i.e., function clones), syntactic blocks (i.e., block clones), or contiguous sequences of arbitrary statements. The block clones also subsume the function clones, i.e., each function clone is also a block clone, as the entire function body can be regarded as a syntactic block. *Type-1* clones are also called *exact* clones, whereas, the *Type-2* and *Type-3* clones are also known as *near-miss* clones. Due to the semantic similarity rather than syntactic similarity, *Type-4* clones are also referred to as *semantic* clones. Our work deals with the *exact* (*Type-1*) and *near-miss* (*Type-2* and *Type-3*) block clones excluding the semantic (*Type-4*) clones, because the accurate detection of semantic (*Type-4*) clones is still an open problem.

Two code fragments that are clones to each other are called a *clone-pair*. A *clone-group* is a set of clone fragments such that any two members of the set is a clone-pair. Figure 1 presents examples of different types of function clone-pairs. In the figure, the code fragment(a) and fragment(b) form a *Type-1* clone-pair, fragment(a) and fragment(c) form a *Type-2* clone-pair while the fragment(a)
and fragment(d) form a Type-3 clone-pair. All the four code fragments can form a clone-group. Another example, is presented in Figure 2, where the two shaded blocks in the left form a block level exact clone-pair.

3 Clone Refactoring

There have been immense research in software refactoring in general. Fowler et al. [12] initially proposed a set of 72 software refactoring patterns and until recently the number has increased to 93 [13]. Those patterns of refactorings in general are meant to get rid of different types of code smells (including duplicated code) and to prevent software decay (i.e., loss of code/design quality) [1]. Based on a survey of existing literature [14, 15, 20, 22, 35, 41] and our experience, we find that among those general software refactoring patterns [12, 13], the following patterns are particularly suitable for code clone refactoring. Detail about these refactoring patterns can be found elsewhere [12, 13].

- **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 2 shows an example of the *extract method* refactoring.

- **Pull-up method (PM)** removes similar methods found in several classes by introducing a generalized method in their common superclass. Figure 3(a) demonstrates a *pull-up method* refactoring through a schematic diagram.

- **Extract superclass (ES)** introduces a new common superclass for two or more classes having similar methods, and then applies *pull-up method*. Extract superclass refactoring may be necessary when those classes do not already have a common superclass and those classes can be brought under a common superclass. Figure 3(b) presents an a schematic diagram demonstrating the *extract superclass* refactoring pattern.

- **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. Figure 3(c) demonstrates extract utility-class refactoring through a schematic diagram.

Note that a refactoring pattern is composed of a sequence of other refactoring patterns or low-level modification 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, which may be necessary to produce generalized blocks of code from near-miss (similar, but not exact duplicate) clones. For the purpose of formulation, we use the term refactoring operators to denote both the composite refactoring patterns and low-level modification operations.

For code clone refactoring, these refactoring operators will operate on groups of clone fragments (i.e., code segments that are clones of each other) 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 on which the refactoring operator \( op \) operates. 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 activity.

4 Estimation of Refactoring Effort

The effort required for code clone refactoring is likely to depend on the type of refactoring operators and operands. For example, 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 fragments that are scattered across different locations of the source code with respect to the file system or inheritance hierarchy may require relatively more effort than that for refactoring clones residing cohesively at a certain location of the source code. To address these issues, we propose a code clone refactoring effort model, which, to the best of our knowledge, is the first model for the estimation efforts needed to refactor code clones in procedural and
object-oriented source code. We have formulated this effort model based on our understandings
developed from a survey of existing literature [1, 8, 12, 13, 15, 24, 36] and our experience in the
area.

Let us consider a clone-group, \( g = \{c_1, c_2, c_3, \ldots, c_n\} \), 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,

\[
\begin{align*}
& c_i \maps m_i \maps C_i \maps F_i \maps D_i, & \text{for object-oriented code.} \\
& c_i \maps m_i \maps F_i \maps D_i, & \text{for procedural code.}
\end{align*}
\]

Where, the symbol \( \maps \) indicates a containment relationship. \( x \maps y \) means that \( x \) is contained in \( y \),
in other words, \( y \) contains \( x \). The relationship preserves transitive property, i.e., \( x \maps y \maps z \Rightarrow x \maps z \).

Thus, the set \( C(g) \) of all classes hosting the clone fragments in \( g \) can be defined as:

\[
C(g) = \{C_i \mid \forall c_i \in g, c_i \maps C_i\} \tag{1}
\]

We use this notation in subsequent sections of this paper, in particular, in formalizing the navigation effort in Section 4.3.

\subsection*{4.1 Context Understanding Effort}

The applicability of refactoring on certain code clones is largely dependent on the context. The context captures the relationship of a certain code fragment with the rest of the source code. Therefore, before refactoring, the developer must understand the context pertaining to the refactoring candidate at hand. Code clone refactoring may result in removal or relocation of code fragments that may span a block of code or an entire method/function or even an entire class. Such removal or relocation of code fragments may cause changes in the underlying inheritance hierarchy and method call-graph. Hence, for understanding the context and the possible impact of changes, the developer must examine the caller-callee delegation of methods and the inheritance hierarchy.

\textbf{Effort for Understanding Method Delegation}. A certain refactoring under consideration may cause the clone fragment to move to a different location (e.g., class, package). Such a
relocation may hinder the visibility of any methods to which the clone fragment refers. Moreover, if a function clone is relocated, all references to the original function must be updated accordingly. To understand the delegation of methods involving the concerned code fragment \( c_i \in g \), the developer must 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 must also 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 to be investigated for understanding the delegation effort concerning \( c_i \) is determined as:

\[
\text{delegation}(c_i) = M_f(c_i) \cup M_r(c_i) \cup \{m_i\}.
\]

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

\[
\text{Delegation}(g) = \bigcup_{c_i \in C(g)} \text{delegation}(c_i).
\]

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

\[
E_d(g) = \sum_{m \in \text{Delegation}(g)} \text{LOC}(m)
\]

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

**Effort for Understanding Inheritance Hierarchy.** Suppose that \( C_p(g) \) is the set of all lowest/closest common superclasses of all pairs of classes in \( C(g) \). The developer must also understand those classes in the inheritance hierarchy that have overridden (in subclasses) or referred 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 be examined for understanding the inheritance hierarchy concerning the code clones in \( g \) and the effort \( E_h(g) \)

---

1For any two Java classes \( C_i \) and \( C_j \) containing code clones \( c_i \) and \( c_j \) respectively, there may be at most one lowest common superclass, as Java does not support multiple inheritance. Any Java class is a subclass of the **Object** class. If this **Object** class is found to be the lowest common superclass of any pair of classes, this should be ignored, and those classes should be considered to have no common superclass.
required for this can be estimated as:

\[ E_h(g) = \sum_{C \in C_h(g)} LOC(C). \]  \hspace{1cm} (5)

### 4.2 Effort for Code Modifications

To perform refactoring on the target clones, the developer must edit at difference locations in the source code. The effort needed to perform such edits can be captured in terms of token modifications and code relocations.

**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:

\[ E_t(g) = \sum_{i=1}^{k} \delta(t_i, t'_i). \] \hspace{1cm} (6)

However, the state-of-the-art IDEs (Integrated Development Environments) such as Eclipse and NetBeans facilitate identifier/variable renaming simply by select-replace operations with the help of graphical user interfaces. Thus, with the availability of such tool support, the edit distance \( \delta(t_i, t'_i) \) can simply be replaced by a constant \( \mu \). By default, \( \mu = 1 \), but the software engineer can set a different value to \( \mu \) as appropriate.

**Code Relocation Effort.** When developers must 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 \( \beta \) is the set of all non-adjacent blocks of code that must be relocated to perform the refactoring. \( |\beta| \) denotes the number of blocks in the set \( \beta \).
4.3 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 effort, our 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

$$F_d(g) = \{ F_i \mid m_i \leftarrow F_i, m_i \in Delegation(g) \}$$

$$F_h(g) = \{ F_i \mid C_i \leftarrow F_i, C_i \in C_h(g) \}$$

$$D_d(g) = \{ D_i \mid F_i \leftarrow D_i, F_i \in F_d(g) \}$$

$$D_h(g) = \{ D_i \mid F_i \leftarrow D_i, F_i \in F_h(g) \}$$

$$DCH(g) = \max_{C_i, C_j \in C_h(g)} \{ \partial(C_i, C_j) \}$$

$$DFH(g) = \max_{F_i, F_j \in F_d(g) \cup F_h(g)} \{ \partial(F_i, F_j) \}$$

Here, $DCH(g)$ refers to the dispersion of class hierarchy with $\partial(C_i, C_j)$ denoting the distance between class $C_i$ and class $C_j$ in the inheritance hierarchy. The distance between any two classes $C_i$ and $C_j$ is computed based on an abstract directed graph where each node represents a class and their exists an edge between each superclass and its subclass. Let $C_p$ be the lowest common superclass of both $C_i$ and $C_j$. Then, $\partial(C_i, C_j) = \max\{pathLength(C_i, C_p), pathLength(C_j, C_p)\}$, where $pathLength(C_i, C_p)$ is measured as the number of edges in the shortest path from $C_i$ to $C_p$. In Figure 4, the computation of distance between classes is illustrated with an example. More detail about $DCH(g)$ can be found elsewhere [14]. $DFH(g)$ is a similar metric that captures the dispersion of files with $\partial(F_i, F_j)$ denoting 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 weights 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.

5 Prediction of Refactoring Effects

The expected benefit from code clone refactoring is the structural improvement in the source code, which should also enhance the software design quality. Obvious expected benefits include reduced source lines of code (SLOC), less redundant code, to name a few. 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 quality models, such as QMOOD [3] or design quality metrics, such as the Chidamber-Kemerer [6] metric suite. Quantitative or qualitative estimation of the effect of refactoring on the quality metrics can be possible before the actual application of the refactoring [5, 22, 33, 37, 38].

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 impact of a certain refactoring \( r \) in code/design quality can be assessed by comparing the quality attributes before and after performing that particular refactoring. Hence, the total gain (or loss) 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 \( \vartheta_j \) 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 design quality model for estimating the effect of refactoring on object-oriented source code. QMOOD is a prominent quality model for object-oriented systems, which is used by other researchers [5, 23, 22]. We choose QMOOD because this quality model has the advantage that it defines six high-level design quality attributes (Table 1) from the 11 lower
level structural property metrics (Table 2). Indeed, the sum of differences in Equation 9 may not be able to utilize the fullest benefit of the quality model, but it serves our purpose.

6 Refactoring Constraints

Among the applicable refactorings, there may be conflicts and dependencies [24] besides their distinguishable impacts on the design quality. The application of one refactoring may cause the operands of other refactorings disappear and thus invalidate their applicability [5, 22, 24]. Besides such mutual exclusion on conflicting refactorings, the application of one refactoring may also reveal new refactoring opportunities, as suggested by Lee et al. [22]. We understand that these are 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 (Figure 3(b)).

There may also be sequential dependencies between refactoring activities [22, 24]. Constraints of mutual inclusion may also arise when the application of one refactoring will necessitate the application of certain other refactorings [41]. Figure 5(a) presents an example of mutual inclusion constraint and Figure 5(b) demonstrates a mutual exclusion constraint with an example. 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 \( r_j \not\rightarrow r_i \).

**Definition 2 (Mutual exclusion)** Two refactorings \( r_i \) and \( r_j \) are said to be mutually exclusive, if both \( r_i \not\rightarrow r_j \) and \( r_i \not\leftrightarrow r_j \) holds. The mutual exclusion between \( r_i \) and \( r_j \) is denoted as \( r_i \not\leftrightarrow r_j \).

Thus, \( r_i \not\leftrightarrow r_j \) implies both \( r_j \not\rightarrow r_i \) and \( r_i \not\rightarrow r_j \).

**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 \( r_i \leftrightarrow r_j \).
The complete independence of \( r_i \) and \( r_j \) is expressed as \( r_i \perp r_j \). For further detail about the refactoring constraints with concrete examples, interested readers are referred to elsewhere [5, 22, 24, 41].

7 Formulation of Refactoring Schedule

Upon identification of all the hard and soft constraints pertaining to a scheduling problem instance, computing an optimal refactoring schedule to maximize code quality while minimizing efforts is a NP-hard problem [5, 22, 23]. Finding the optimum solution for a large instance of such a problem becomes practically too expensive (time consuming) and, thus, a feasible optimal (near-optimum) solution is desired. However, the problem is by nature a Constraint Satisfaction Optimization Problem (CSOP). A CSOP is a kind of problem characterized by a set of constraints that must be satisfied and among all the feasible solutions, the best possible solution is desired. A solution is said to be feasible if it satisfies all the constraints. A solution is evaluated better than another based on an objective function, which a solver strives to optimize (i.e., maximize or minimize).

We model the refactoring scheduling problem as a CSOP and solve it by applying a constraint programming technique, which no one reported to have applied before.

Having identified the set \( \mathcal{R} \) of potential refactoring activities, we define two decision variables \( \mathcal{X}_r \) and \( \mathcal{Y}_r \):

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

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

where \( 1 \leq k \leq |\mathcal{R}| \). Thus, \( \mathcal{X}_r \) captures whether a refactoring \( r \) is included in the schedule and \( \mathcal{Y}_r \) captures the order of refactoring \( r \) in the selected schedule of refactorings.

We also define a \(|\mathcal{R}| \times |\mathcal{R}|\) constraint matrix \( Z \) to capture the constraints and sequential
dependencies between distinct refactorings \( r_i \) and \( r_j \):

\[
Z_{i,j} = \begin{cases} 
0 & \text{if } r_i \perp r_j \\
1 & \text{if } r_i \leftrightarrow r_j \\
+2 & \text{if } r_j \rightarrow r_i \text{ and } r_i \leftrightarrow r_j \\
-2 & \text{if } r_i \rightarrow r_j \text{ and } r_i \leftrightarrow r_j \\
+3 & \text{if } r_j \rightarrow r_i \text{, but neither } r_i \leftrightarrow r_j \text{ nor } r_i \leftrightarrow r_j \\
-3 & \text{if } r_i \rightarrow r_j \text{, but neither } r_i \leftrightarrow r_j \text{ nor } r_i \leftrightarrow r_j 
\end{cases}
\]

\( Z_{i,j} = -Z_{j,i} \) or \( Z_{i,j} = Z_{j,i} = 1 \), for all \( \langle i, j \rangle \).

Let \( \rho_r \) 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.

\[
\text{maximize } \sum_{r \in R} X_r \rho_r (Q_r - E(g_r)) \tag{10}
\]

subject to (with \( i \neq j \)),

\[
X_r + Y_r \neq 1, \quad \forall r \in \mathcal{R} \tag{11}
\]

\[
X_{r_i} + X_{r_j} = 2 \Rightarrow Y_{r_i} \neq Y_{r_j}, \quad \forall r_i, r_j \in \mathcal{R} \tag{12}
\]

\[
Z_{i,j} - Z_{j,i} > 0 \Rightarrow Y_{r_i} < Y_{r_j}, \quad \forall r_i, r_j \in \mathcal{R} \tag{13}
\]

\[
Z_{i,j} - Z_{j,i} < 0 \Rightarrow Y_{r_i} > Y_{r_j}, \quad \forall r_i, r_j \in \mathcal{R} \tag{14}
\]

\[
|Z_{i,j}| = 1 \Rightarrow X_r + X_r \leq 1, \quad \forall r_i, r_j \in \mathcal{R} \tag{15}
\]

\[
|Z_{i,j}| = 2 \Rightarrow (X_{r_i} + X_{r_j}) \text{ modulo } 2 = 0, \quad \forall r_i, r_j \in \mathcal{R} \tag{16}
\]

\[
\sum_{r \in \mathcal{R}} X_r \leq \mathcal{M} \tag{17}
\]

Here, Equation 10 is the objective function for maximizing the code quality and minimizing the refactoring effort while rewarding refactoring activities having higher priorities. One of the product term in the objective function is \( X_r \), which is equal to 1 only for selected refactorings, and for all other refactorings \( X_r \) equals to 0. Thus, the objective function takes into account the priority, quality, and efforts pertaining to only the selected refactorings.
Equation 11 ensures that the decision variables $X_r$ and $Y_r$ are kept consistent as their values are assigned. If the refactoring $r$ is not selected (i.e., $X_r = 0$), then $Y_r$ must also be 0, to denote that the refactoring $r$ is not assigned any position in the sequence of the scheduled refactorings. On the other hand, if the refactoring $r$ is selected (i.e., $X_r = 1$), then $Y_r$ must not be zero, i.e., $X_r + Y_r \neq 1$.

Equation 12 enforces that no two refactorings are scheduled at the same point in the sequence. Equation 13 and Equation 14 impose the sequential dependency constraints (i.e., $r_i \rightarrow r_j$) on feasible schedules. Mutual exclusion (i.e., $r_i \leftrightarrow r_j$) and mutual inclusion (i.e., $r_i \leftrightarrow r_j$) constraints are enforced by Equation 15 and Equation 16 respectively. Equation 17 specifies that maximum $M$ number of refactorings can be chosen for scheduling. By default $M = |R|$ but $M$ 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.

### 7.1 An Illustrative Example

Now, with an example, we further illustrate our formulation, especially the constraint matrix $Z$.

Consider a set of five refactorings $R = \{r_1, r_2, r_3, r_4, r_5\}$ having constraints as follows:

(i) $r_2 \leftrightarrow r_4$ (i.e., $r_2$ and $r_4$ are mutually exclusive)

(ii) $r_1 \rightarrow r_4$ (i.e., $r_1$ cannot be applied after $r_4$)

(iii) $r_5 \rightarrow r_3$ (i.e., $r_3$ cannot be applied after $r_5$)

(iv) $r_3 \leftrightarrow r_5$ (i.e., $r_3$ and $r_5$ are mutually inclusive)

Other than the above mentioned constraints, any two refactorings $\langle r_i, r_j \rangle \in R$ are independent (i.e., $r_i \perp r_j$). The constraint(iii) and constraint(iv) above jointly enforces that if either of $r_3$ and $r_5$ is selected, the other refactoring must also be selected and then $r_3$ must also be scheduled before $r_5$. According to the constraint-specifications, a valid constraint matrix $Z$ becomes as shown in Table 3. The empty cells in the table are filled up with zeros, which we omitted here for the purpose of better readability.

The constraint(i) is enforced by Equation 15. Here, $Z_{2,4} = Z_{4,2} = 1$. If both $r_2$ and $r_4$ are selected then $X_{r_2} + X_{r_4} = 1 + 1 = 2$, which violets the constraint in Equation 15.
The constraint(ii) is imposed by the Equation 13 and Equation 14. Here, \( Z_{1,4} = +3 \) and \( Z_{4,1} = -3 \) and thus, \( Z_{1,4} - Z_{4,1} = 6 \), which is higher than zero. Hence, Equation 13 imposes that \( \bar{Y}_{r_1} < \bar{Y}_{r_4} \), and thus \( r_1 \) precedes \( r_4 \) in the schedule (if both are selected). Again, with respect to Equation 14, \( Z_{4,1} - Z_{1,4} = -6 \), which is less than zero and hence \( \bar{Y}_{r_4} > \bar{Y}_{r_1} \) is imposed. Thus, Equation 14, ensures that \( r_4 \) follows \( r_1 \) in the schedule (if both are selected).

The constraint(iii) is satisfied in the same way the constraint(ii) is satisfied. Although in this case, \( Z_{3,5} - Z_{5,3} = 4 \) and \( Z_{5,3} - Z_{3,4} = -4 \), the evaluation of negativity works the same way as does for satisfying the constraint(ii).

Finally, the mutual inclusion in constraint(iv) is enforced by Equation 16. According to our current example, \( |Z_{i,j}| = |Z_{j,i}| = 2 \). Hence Equation 16 ensures that the remainder of \( (X_{r_3} + X_{r_5}) \) divided by 2 must be equal to zero, which is possible only if \( X_{r_3} = X_{r_5} = 1 \) or \( X_{r_3} = X_{r_5} = 0 \), that means if both or neither of \( r_3 \) and \( r_5 \) are selected. Thus the constraint of mutual inclusion is satisfied.

8 Implementation

Based on the CSOP formulation of the scheduling problem, we developed a CP model using \textit{OPL} (Optimization Programming Language)\(^2\). For OPL programming, we used the \textit{IBM ILOG CPLEX Optimization Studio 12.2 IDE} under an academic license. The IDE can be integrated with CPLEX Solver and CP Optimizer, which are IBM’s optimization engines for solving optimization problems modeled in LP and CP respectively.

Constraint programming (CP) combines techniques from AI and OR, and it has been shown to be effective in solving combinatorial optimization problems, especially in the area of scheduling and planning [4, 47]. Over the past decade, a separate conference series\(^3\) is held to host research to integrate and combine AI and OR techniques in CP. CP allows a more natural and flexible way to express objective functions and constraints where the functions and equations do not necessarily have to be strictly linear. Based on the CSOP formulation of the scheduling problem, we also developed a CP model of the problem and for solving it, invoked the CP Optimizer from inside

\(^2\)OPL (Optimization Programming Language) is a relatively new modeling language for combinatorial optimization that simplifies the formulation and solution of optimization problems.

\(^3\)International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR).
the IBM ILOG CPLEX Optimization Studio 12.2 IDE.

The CP technique works as follows. Given a set of variables with their domains and a set of constraints on those variables, first the domains of the variables are identified. Then, based on the given constraints, the domains of the concerned variables are modified. When a variables domain is modified, the effects of this modification are then propagated through constraint propagation to any constraint involving that variable. For each constraint, domain reduction detects inconsistencies among the domains of variables pertinent to that constraint and removes inconsistent values.

When a particular variable’s domain becomes empty, it may be determined that the constraint cannot be satisfied and backtracking may occur undoing an earlier choice. CP repeatedly applies constraint propagation and domain reduction algorithms to make the domain of each variable as small as possible while keeping the entire system consistent. To find the optimal solution, the CP technique may need to explore, in the worst case, all the feasible solutions and compare them based on the objective function’s values.

For the purpose of evaluation, we also implemented Linear Programming (LP), genetic algorithm (GA), and three variants of greedy algorithms for optimizing the automated scheduling of code clone refactorings. Further detail of about the LP, GA, and greedy scheduling techniques are presented in Section 9.4.

9 Empirical Evaluation

To evaluate our refactoring scheduler and the effort model, we conducted an empirical study on refactoring six software systems developed (or under development) in our software research lab. The subject systems and their sizes in terms of source lines of code (SLOC) are described in Table 4. All the subject systems shown in Table 4 are written in Java and the sizes of the systems in terms of SLOC exclude the comments and blank lines.

In particular, we designed the study 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? The effective-
ness of the technique is measured by quantitative comparison with other techniques such as GA and LP. The conflict-freeness is verified by manually checking for any constraint violation in the schedules computed by the scheduler.

**RQ2:** Is the code clone refactoring effort model (described in Section 4) useful in capturing and estimating the efforts required for performing the refactorings? We address this exploratory research questions by qualitative analysis of the observation on developers during the study and the developers’ feedback at post-study questionnaire.

Typically, it is difficult and risky for software engineers to refactor a source code with which they are not familiar [23]. However, it is the programmers, who are likely to know the best about the critical parts of the projects they develop and thus can better assess both the efforts and effects of refactoring and prudently assign priorities on certain refactoring candidates. While the developers’ ability to assess the refactoring efforts and effect for large and complex projects can still remain unreliable, for smaller systems, their ability can be fairly reliable. Therefore, to evaluate our refactoring scheduler, we chose small projects (Table 4) developed in our own research lab. The use of in-house software systems in the study not only facilitated manual verification for correctness but also reduced the evaluation cost.

At the beginning of the study, we described to the developers the objectives of the study and provided them with our refactoring effort model, as well as an initial list of refactoring operators that can be used for code clone refactoring. Then, we explained a catalog of common software refactoring patterns [13] to them and showed them how some of those can be applied for code clone refactoring. We also described the QMOOD quality attributes to them and, upon discussion, came to a consensus to use the first six metrics (Table 2) in our study. We all agreed that the rest of the metrics were too difficult to estimate through subjective investigation. Hence, we ignored the effect of code clone refactoring on those metrics. To ignore them, the total gain in code/design quality (Section 5) was computed having values of changes in those metrics set to zero. 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.
9.1 Clone Detection

The first and foremost activity towards code clone refactoring is the detection of code clones from the source code. We used NiCad-2.6.3 [7] for detecting near-miss block clones of at least five lines in pretty-printed format. We used the ‘blind-rename’ option of NiCad with UPIT (Unique Percentage of Items Threshold) set to 30%. The ‘blind-rename’ option instructs NiCad to ignore the differences in the names of identifies/variables during comparison of the code fragments. UPIT is a size-sensitive dissimilarity threshold that sets NiCad’s sensitivity to differences in the code fragments during the detection of near-miss code clones. For example, if UPIT is set to 0% without the ‘renaming’ option, NiCad detects only exact clones (code clones that have identical program text but may have variations in layouts); 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). In the detection of code clones, NiCad also ignores the comments in the source code and reports code clones clustered into clone-groups based on their similarity.

9.2 Data Acquisition

The results of clone detection from the six 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 context according to their understanding. 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 (i.e., clone fragments) involved in those groups are presented in Table 5, 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 clone-group (refactoring candidate). The identification of a 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 right-most column of Table 5 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 2, we present a clone-pair (shaded blocks on the left) with partial context (surrounding code). The example is an excerpt from the source code of VisCad\(^5\). The developers chose to refactor them by applying the *extract method* operation. The developers recorded further fine grained operations and required efforts in order, as shown in Table 6. As explained by the developers, the effort for producing the method signature was estimated by twice (for type and name) the number of parameters 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.

As far as we are concerned, there is no existing tool for calculating refactoring efforts and ours is the first conceptual model for this purpose, 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 that 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. We used our observations and the developers’ feedback for subjective evaluation of our effort model.

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 2), 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.

\(^5\)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/](http://code.google.com/p/google-diff-match-patch/)) that VisCad uses internally. We deliberately chose to present this simple example, so that anyone can easily follow and verify.
(before and after the refactoring) in the system, they just had to estimate the changes in the number of classes or methods. For example, the refactoring scenario presented in Figure 2 causes the complexity (number of methods) to 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 that 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 assigned the lowest priority to them for examining how our scheduler handles them in the scheduling process.

The developers’ estimation of refactoring efforts, effects, and the assignment of priorities are then used to compute refactoring schedules in the evaluation of our code clone refactoring scheduler.

9.3 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 4.0 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:

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

where, \( \text{norm}(v_i) \) 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., \( 0 < |\text{norm}(v_i)| \leq 1 \)) and thus minimizes the inter-dimension influence of the magnitudes, while still preserving the relative ratios of magnitudes within dimensions. The emphasis on the efforts compared to the qualities can be tweaked by setting higher or lower weights as described in Equation 8.

For priorities, before applying the aforementioned normalization, we carried out an additional normalization phase by adding +6 to each priority values. Thus we got rid of any priority less than or equal to zero. This was necessary as in our objective function, the difference between quality and effort is multiplied by priorities, and we need to make sure that the multiplication negatives or multiplication by zero priority do not take place.

9.4 Schedule Generation

For each of the systems subject to our study, we enumerated all the refactorings, accumulated them with the normalized data and organized them in an appropriate OPL format to feed to the schedulers for automated computation of the refactoring schedules.

We evaluated our CP scheduling approach in four phases. In the first phase, we compared our CP scheduler with three variants of a greedy algorithm. The second phase compared our CP approach with genetic algorithm (GA) that was used by other researchers [5, 22] in the past. In the third phase, we compared the CP scheduling with a manual approach. And finally, the fourth phase compared the CP technique with the LP approach. In our study, we used the default settings in the estimation of total effort and quality gain, as described in Section 4 and Section 5. For each of the subject systems, we first computed the refactoring schedule using our CP approach and then applied GA, greedy, manual, and LP approaches (described later) to compute schedules for the same set of refactorings.
The normalized data for each of the subject systems were separately fed to each of the schedulers. All the schedulers were executed on an Apple “MacBookPro5,5” computer with Intel Core 2 Duo (2.26 GHz) processor and 4 GB primary memory (RAM). The CP, LP, and greedy schedulers operated inside the IBM ILOG CPLEX Optimization Studio 12.2 IDE running on Windows XP operating system. All the IDE parameters were set to the defaults. The GA scheduler operated on the same computer but on a Mac OS 10.6.8 operating environment.

9.4.1 CP Scheduling

For each of the subject systems, the normalized data for each of the subject system were fed to our scheduler as described in Section 7 and Section 8. The scheduler, upon obtaining the data in valid OPL format, applies constraint propagation and domain reduction techniques [47] to generate the optimal solution as instructed.

9.4.2 LP Scheduling

Linear programming (LP) is a mathematical programming technique for solving optimization problems. Over the past few decades, LP has been widely used in the operations research (OR) community for dealing with optimization problems. The basic idea is to formulate the problem as a linear programming problem and solve it using linear programming algorithms such as the simplex method, ellipsoid method, and interior-point techniques [40]. A linear programming problem is a mathematical formulation of an optimization problem defined in terms of an objective function and a set of constraints. The objective function is a linear function of variables whose values are unknown and the set of constraints consists of linear equalities and linear inequalities. The requirement of the linearity of the objective function and the constraint equations as well as the solution technique are the most obvious traits that make LP distinct from CP. On the basis of the CSOP formulation described in Section 7, we implemented an LP model of the scheduling problem and invoked the CPLEX Solver for solving the LP model. The CP implementation differs from the CP implementation in two ways: first, in the LP implementation all the constraints had to be expressed in terms of strictly linear equations, whereas in CP we used CP-specific OPL statements all of which were not necessarily expressed in terms of linear equations. Second, the CP and LP implementations included different instructions to explicitly specify whether to invoke
the CP Solver or the LP Solver of the IBM ILOG CPX Optimization Studio 12.2.

To compute scheduling of refactorings for each of the subject systems, we invoked our LP scheduler, which applied the mixed integer linear programming (MILP) technique for computing the schedules. MILP is a kind of LP, where the variables can hold integer or floating point values only. Our LP scheduler, in consultation with the CPLEX Solver, invokes the branch-and-cut algorithm, which in turn applies the simplex algorithm to solve a series of relaxed LP subproblems and gradually converge to a strictly optimal solution. The simplex algorithm operates by repeatedly applying linear algebraic techniques to solve systems of linear equations. Further detail about the branch-and-cut and simplex algorithms can be found elsewhere [40, 47].

9.4.3 GA Scheduling

Genetic algorithm (GA) is a kind of evolutionary algorithm from the field of artificial intelligence (AI) for solving optimization problems. In GA, a candidate solution is encoded as a sequence of values, called a chromosome, and a set of candidate solutions is called a population. The algorithm iterates over generations to evolve a population towards better solutions through a number of operations such as crossover and mutation. A fitness function is used to guide the evolution towards optimality. In our study, the objective of the GA was set to select the best subset having maximum $M$ members from the set $R$ of all potential refactorings (recall from Section 7) for each of the subject systems.

Encoding. For each system subject to our study, all the candidate refactorings were enumerated with integers 1 through $|R|$. Having such an enumeration, a common approach is to represent the problem as a binary Knapsack [5] problem and encode a solution as a binary string (Figure 6) of length $|R|$. A bit in the string is 1, if and only if the corresponding refactoring is selected in the schedule. However, such an encoding scheme cannot deal with order dependencies among the refactorings.

To capture the order dependencies, we devised a different encoding scheme. A candidate solution was encoded in a chromosome $\zeta$ as a sequence of $M$ integers, having their positions indexed with 1 through $M$, as shown in Figure 7.

Having $\zeta[i]$ denoting the chosen refactoring at index $i$, the encoding scheme signifies as follows.
• $\zeta[i] = 0$ implies that no refactoring is selected at the index $i$.

• $\zeta[i] = r_k$ implies that $r_k$ is the $i^{th}$ refactoring in the schedule represented by $\zeta$.

• $\zeta[i] < \zeta[j]$ and $\zeta[i] \neq 0 \neq \zeta[j]$ means that the refactoring at index $i$ is scheduled before the refactoring at index $j$.

• A solution encoded by the chromosome $\zeta$ is feasible, if any two chosen refactorings $\zeta[i]$ and $\zeta[j]$ satisfy all the hard constraints. Otherwise, the solution is infeasible.

• A solution encoded by the chromosome $\zeta$ must not select the same refactoring more than once. That is, $\zeta[i] \neq \zeta[j]$ must hold if $\zeta[i] \neq 0$.

Crossover Operation. The crossover operation, as shown in the Figure 8, randomly selects two chromosomes $\zeta_{p_1}$ and $\zeta_{p_2}$ as parents, an index $k$ as the point of crossover, and creates two offsprings $\zeta_{c_1}$ and $\zeta_{c_2}$ as follows,

$\zeta_{c_1}[i] = \zeta_{p_1}[i]$, for $i \in \{1, 2, 3, \ldots, k-1\}$

$\zeta_{c_1}[j] = \zeta_{p_2}[j]$, for $j \in \{k, k+1, k+2, \ldots, M\}$

$\zeta_{c_2}[i] = \zeta_{p_2}[i]$, for $i \in \{1, 2, 3, \ldots, k-1\}$

$\zeta_{c_2}[j] = \zeta_{p_1}[j]$, for $j \in \{k, k+1, k+2, \ldots, M\}$

A configurable parameter crossover rate defines what proportion of the population of chromosomes in a certain generation will participate in crossover operation to produce offsprings. A crossover rate of 80% indicates that 80% of the population participate in crossover operation leaving 20% survivors (unchanged chromosomes).

Mutation Operation. The mutation operation on a chromosome $\zeta$ selects a random index $k$ and replaces the refactoring $\zeta[k]$ by another randomly-selected refactoring $r \in R$ that is not already in the chromosome. Mathematically, $\zeta[i] \neq r, \forall i \in \{1, 2, 3, \ldots, M\}$ holds before the mutation.
**Fitness Function.** The fitness function is defined by the objective function described by the Equation 10 in Section 7. The fitness function determines how good (fit) a solution is, as represented by a chromosome.

**Population Generation.** The genetic algorithm begins with an initial population and evolves from generation to generation. A population of \( P \) distinct solutions is randomly created using the procedure described in Algorithm 1.

### Algorithm 1 : Algorithm for Creating Population

**Require:**

\[
R \leftarrow \{1, 2, 3, \ldots, |\mathcal{R}|\} \\
\text{// enumerations over all refactorings } \mathcal{M} \leq |\mathcal{R}|
\]

**procedure** CREATEPOPULATION(\( P \))

\[
S_p \leftarrow \{} \\
\text{// a set to accommodate all solutions}
\]

while \( |S_p| < P \) do

\[
x \leftarrow \text{RandomSolution( )} \\
S_p \leftarrow S_p \cup \{x\}
\]

end while

return \( S_p \)

end procedure

**procedure** RandomSolution( )

\[
\zeta[\cdot] \leftarrow \{0, 0, \ldots, 0\} \\
\text{// array of } \mathcal{M} \text{ zeros}
\]

\[
S_r \leftarrow R \cup \{0\} \\
\text{// } \mathcal{R} \text{ is the set of all refactorings}
\]

for \( k \leftarrow 0 \text{ to } \mathcal{M} - 1 \) do

\[
r \leftarrow \text{randomly chosen, } r \in S_r \\
\zeta[k] \leftarrow r \\
\text{if } r \neq 0 \text{ then}
\]

\[
S_r \leftarrow S_r - \{r\}
\]

end if

end for

return \( \zeta \)

end procedure

**Genetic Evolution.** The algorithm evolves according to the description in Algorithm 2. The genetic algorithm and its evolution is characterized by a number of parameters. We executed the genetic algorithm several times with different combinations of parameters. After tuning with different combinations, we chose the combination that yielded the best performance (i.e., highest fitness) of the genetic algorithm. The chosen values for the parameters, as presented in Table 7, are consistent with general recommendations [25]. For each of the subject systems, we executed the GA scheduler for five times. At each run, the scheduler executed for 10,000 seconds (i.e., 2.78
hour) and we obtained the best (fittest) solution produced in the five runs. This run-time is much higher than those required for other scheduling approaches (e.g., CP, LP).

Algorithm 2 : Genetic Algorithm

```plaintext
// generation set to 0
\( g \leftarrow 0 \)
\( \zeta_{best} \leftarrow \emptyset \)  // null array
\( S_p \leftarrow \text{CreatePopulation}(max_{population}) \)

repeat
  \( g \leftarrow g + 1 \)
  \( S_p \leftarrow \text{FilterInfeasibles}(S_p) \)  // drop infeasible solutions
  \( S_p \leftarrow \text{Sort}(S_p) \)  // sort in descending order of fitness
  \( \zeta_{best} \leftarrow \text{First}(S_p) \)  // record the best solution
  \( S_p \leftarrow \text{DropPoors}(S_p) \)  // drop poor sol. w.r.t. elitism rate
  \( d \leftarrow \text{max}_{population} - |S_p| \)
  if \( d \) is odd then
    \( d \leftarrow d + 1 \)
  end if
  \( S_n \leftarrow \{ \} \)
  for \( i \leftarrow 1 \) to \( \frac{d}{2} \) do
    \( \{ \zeta_{p1}, \zeta_{p2} \} \leftarrow \text{random chromosomes from } S_p \)
    \( \{ \zeta_{c1}, \zeta_{c2} \} \leftarrow \text{CrossOver}(\zeta_{p1}, \zeta_{p2}) \)
    \( S_n \leftarrow S_n \cup \{ \zeta_{c1}, \zeta_{c2} \} \)
  end for
  for \( \zeta \in S_n \) do
    \( m \leftarrow \text{MutateChance}(\zeta) \)  // chance of \( \zeta \) to be mutated
    if \( m \geq \text{mutationRate} \) then
      \( \zeta \leftarrow \text{Mutation}(\zeta) \)  // mutate \( \zeta \)
    end if
  end for
  \( S_p \leftarrow S_p \cup S_n \)
  \( t \leftarrow \text{elapsedTime()} \)  // time duration of evolution
until \((g \geq \text{max}_{generation}) \) or \((t \geq \text{max}_{time}) \)
return \( \zeta_{best} \)
```

9.4.4 Greedy Scheduling

Recall that we identified three dimensions (i.e., effort, quality, and priority) for optimizing scheduling of code clone refactorings. Thus, we implemented (using OPL) three variants of a greedy algorithm, each aiming to optimize along one of the dimensions (i.e., optimization criteria) disregarding the other two. The prime objective of the Greedy\(_e\) approach is to compute schedules by minimizing refactoring effort while the Greedy\(_p\) and Greedy\(_q\) approaches aim to maximize the satisfaction of priorities and quality gain, respectively. The general greedy scheduling algorithm
can be described in terms of a few simple steps. First, all the refactorings are sorted in the descending order of the optimization criteria. Then, refactorings are chosen one by one from the top of the sorted list as long as the new candidate does not conflict with any of the already chosen refactorings.

Intuitively, the minimum refactoring effort (i.e., zero effort) can be achieved by scheduling no refactoring at all. Therefore, in the application of the approach greedy towards refactoring efforts, we must set a minimum number of refactorings that must be scheduled. To keep the approach greedy towards refactoring efforts comparable with our CP technique, the minimum number of refactorings was set 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 8.

9.4.5 Manual Scheduling

In the third phase of the evaluation, our goal was set to schedule roughly 25% of the total number of refactorings for each of the subject systems. The developers of the concerned systems were instructed to manually (or, in the way they would do it without help from any automated scheduler) produce a schedule as best as they could. Manually solving a CSOP such as scheduling of code clone refactoring is a time-consuming and difficult task, especially for medium to large problem instances. Therefore, we chose to schedule 25% of the total number of refactorings to keep the problem instance small enough that can be handled by the manual approach. With the same goal (i.e., to schedule roughly 25% of all the refactorings), we executed our CP scheduler.

The purpose of the manual approach was confirm that manually solving a CSOP can be difficult and the solution obtained from manual scheduling can be worse than an automated technique such CP. The objective was to compare the produce schedules and compare them, given a set of constraints, estimation of refactoring, efforts, effects, and priorities, it was not necessary to actually carry out the those refactorings in the subject systems.

9.5 Findings

The values along the three optimization dimensions namely the satisfaction of priorities ($\sum_{r \in R} x_r \rho_r$), required effort ($\sum_{r \in R} x_r E(g_r)$), and expected gain in software quality ($\sum_{r \in R} x_r Q_r$), obtained
from our CP scheduling and manual scheduling, are presented in Figure 9. For an effective schedule, the values for expected quality gain and satisfaction of priorities are expected to be high while the values for the efforts needed are expected to be low. Hence, in the figure the heights of the bars above the baseline (round-ended line) are expected to be high while the heights of the bars below the baseline are expected to be as low as possible.

Table 8 presents values along all the three optimization dimensions obtained by separately running all the automated schedulers for each of the subject systems in our study. Recall that the objective function as stated in Equation 10 optimizes along these three dimensions. From our observations during the study and the developers’ feedback, as well as the results presented in Table 8 and Figure 9, we can answer 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.

As seen from Figure 9, for four of the subject systems (i.e., Mutation Framework, LIME, Sim-Cad, and DomClone), for the refactoring schedules generated by our CP approach are consistently lower while the expected quality gain and satisfaction of priorities are consistently higher compared to those for the manually computed schedules. For gCad, both the CP and manual approaches performed almost equally well. In case of VisCad, the schedule computed by CP demands higher refactoring effort compared to the manually computed schedule; however, the expected quality gain and satisfaction of priorities for the CP schedule are much higher than those for the schedule computed by manual approach. Overall, it can be said that the CP approach outperforms the manual approach.

For all the subject systems, as seen in Table 8, our CP scheduler and the LP scheduler compute the optimal refactoring schedule by efficiently balancing the three optimization dimensions (i.e., effort, quality, and priority). Again, the efforts are expected to be low while the quality gain and priorities are expected to be as high as possible. For some of the smaller systems (Mutation Framework, LIME, and gCad), the greedy approaches, especially the approach greedy towards refactoring efforts, closely competes with our CP approach. For Mutation Framework and LIME, the approach greedy towards efforts can be perceived (according to the third column from the
right) to have performed even slightly better than our CP scheduler. Again, for all the systems, the priorities and quality gain for the schedules computed by the approach greedy towards priorities are consistently higher than those for the schedules computed by CP approach. According to the second column from the right, the approach greedy towards priorities closely competes with our CP scheduler for smaller systems such as LIME and SimCad, whereas for gCad, the approach greedy towards priorities is found to have performed better than CP. However, the required efforts for those schedules (computed by the approach greedy towards priorities) are also consistently much higher (two or three times for most of the systems) than those for the schedules computed by the CP approach. These much lower efforts can make the CP scheduler preferable.

Other than those few cases discussed above, for all the systems the CP and LP schedulers are found to have significantly outperformed the other techniques. We also found that the schedules generated by the CP approach exhibited higher values of the objective-function (i.e., Equation 10) compared to those computed by the greedy approaches. As the sizes of the systems in terms of SLOC and the number of candidate refactorings increases, the CP and LP schedulers outperform the greedy schedulers, which is vivid for the largest systems, VisCad and DomClone. Overall, the CP and LP schedulers perform better than the greedy schedulers, or at least as good as those schedulers.

The risks of refactorings can be best estimated through subjective analysis by the individuals who are familiar with the underlying source code. 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, the CP and LP schedulers also minimize the risks of refactorings, as seen in the right-most column of Table 8, the optimal schedule obtained from our scheduler always includes the least number of refactorings, compared to those from the GA and greedy scheduling.

As expected, our CP scheduler always outperformed manual scheduling for all the six subject systems (Figure 9). The superiority in the optimality of the schedules (in terms of efforts, quality gain, and priorities) obtained from our CP and LP schedulers 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 seven 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. Recall that at each run the GA scheduler executed for more than two hours. Thus, in terms of run-time, the CP approach outperforms the manual, greedy, and GA approaches.

9.5.1 Special Note on GA

The performance of GA scheduling is found to be worse than all other automated scheduling techniques in our study. Within the 200 seconds evolution time, GA was able to produce feasible solution for only LIME. For all other systems, the results of GA scheduling presented in the Table 8 correspond to infeasible solutions. Given the refactoring scheduling problem instances for those subject systems, during our study, we found that the GA technique executed for hours and found solutions, which were not feasible solution. An explanation to this observation can be the fact that optimization problems with many constrains can easily become GA-hard [10, 11], because crossover and mutation, the core operations of GA are based on random selections, which do not guarantee for constraint satisfaction or optimization. Thus, traditional GA may work for optimization problems with a few constraints, but GA approaches do not seem to work well for CSOPs [11]. Above all, GA approaches are by nature time consuming and memory intensive.

9.5.2 CP vs. LP

As can be seen in Table 8, the results for both CP and LP are identical for all the subject systems, except for VisCad and DomClone. This observation means that, for each of those systems, both CP and LP produced equally optimal solutions and, thus, in terms of the quality (e.g., optimality) of the solutions both CP and LP performed equally well.

For VisCad and DomClone, although the optimal schedules computed by CP and LP are different, the values of the objective functions were found to be equal: 8.364 for VisCad and 6.03 for DomClone. There were more than one equally optimal solutions, which allowed the CP and LP schedulers to choose different solutions with the same objective values. However, the CP scheduler picked the solutions with the number of chosen refactoring less than that of the LP scheduler. Thus, our CP scheduler mitigates the risk of refactoring better than the LP scheduler. However, the subtle difference may not be statistically significant and a larger study can verify this phenomenon with statistical confidence.
In Table 9, we present the time and memory consumption of both the CP and LP schedulers in the computation of optimal scheduling of clone refactorings for each of the subject systems. As can be observed in the table, the time and memory consumption of our CP scheduler is significantly less than that of the LP scheduler. Therefore, we can conclude that the CP approach outperforms the LP approach in terms of both run-time and memory consumption.

**Answer to RQ2:** Yes. The code clone refactoring effort model (described in Section 4) is useful in capturing and estimating the efforts required for performing the refactorings.

During the study, we observed the developers as they were manually estimating the efforts required to refactor the code clones at hand and assigning priorities to the candidate clones. We encouraged them to *think aloud* so that we were able to capture information about what and how they were thinking as well as what kind of difficulties they were facing. As they completed their part, we collected feedback from them using questionnaire with a fixed set of both close and open-ended questions. Other than the developers’ background related queries, the questionnaire included the following questions:

- **IQ$_1$:** How difficult was it to use the effort model in manually estimating the required efforts for refactoring the clones?

- **IQ$_2$:** To what extent the effort model appeared useful to you in the estimation of the refactoring efforts?

- **IQ$_3$:** Is there anything that you think is missing and should be included in the effort model?

- **IQ$_4$:** Is there anything that you suggest to exclude from the effort model?

- **IQ$_5$:** Any other comments about the effort model?

The questions IQ$_1$ and IQ$_2$ were Likert scale questions. The possible answers to these questions and the developers’ feedbacks are presented in Table 10. The questions IQ$_3$, IQ$_4$, and IQ$_5$ were open-ended questions. None of the developers responded to IQ$_3$ and IQ$_4$, which hints the completeness of our effort model, at least from those developers’ perspective. Only one of them
responded to IQ5 with a concise comment saying, “Tool [support] needed [for calculation of such effort estimation].”

Upon collection of the developers’ feedback through the questionnaire, we further conducted a focus group discussion session with all the developers to obtain their opinions about usefulness and potential improvements of our effort model. During the focus group session, all the developers indicated that the model was useful and it guided them in the estimation of the efforts. One of the developers further expressed that he would not have any clue about how to estimate efforts without the help of the effort model. All the developers proposed that an automated tool, offering accurate calculations according to the model, would be necessary to use the effort model more accurately. Our observations of 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 refactorings. However, everyone agreed that the effort model and the priority scheme in combination were effective in capturing both the efforts and the risks.

9.6 Threats to Validity

In this section, we point to the possible threats to the validity of our work and how we addressed those threats to minimize their effects. Recall that the objective of our empirical study was twofolds: first, to evaluate our refactoring effort model and, second, to evaluate our CP scheduler. Hence, we organize the discussion of the threats along these two perspectives.

9.6.1 Construct Validity

Construct validity questions the correctness of the design of the study in terms of whether the data collection and operational measures are used correctly to reflect the concepts studied. Ensuring construct validity is typically challenging for studies involving human developers [28].

In our 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 following our effort model. This work is based on our initial proof-of-concept proposal [45], 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 smaller systems, the developers can be expected
to do a fair job in estimating the efforts and risks involved in refactoring the system. Hence, we
intentionally chose in-house systems and the concerned developers for our study, which may appear
as a bias. In practical settings, it is often likely that refactorings, especially during the development
phase, will be performed by the concerned developers who are familiar with the source code.
Thus, our choice of in-house systems and their developers rather imitates the practical settings.
Moreover, the subject systems, being in-house and fairly small, enabled the developers to estimate
the refactoring efforts and effects with a higher probability of accuracy compared to that if we had
used large open-source subject systems. Still, we manually verified each developer’s refactoring
solutions by performing a detailed inspection of the code clones and surrounding source code in
the subject systems. We did not rely only on observing the developers while they were estimating
the refactoring efforts. Through a questionnaire, we also collected the developers’ feedback about
the effort model and further verified their feedback in a focus group session. As such, we have a
high confidence in the validity of the evaluation.

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 is the question for evaluating the
scheduler. Due to the unavailability of any baseline approach or benchmark data, it was not possi-
ble to evaluate our scheduler in terms of precision (specificity) and recall (sensitivity). Therefore,
we chose to compare our CP scheduler with other approaches (i.e., greedy, GA, LP, and man-
ual) in terms of optimization values along the three dimensions (i.e., effort, quality, and priority
satisfaction) as well as in terms of runtime and memory consumption. We manually investigated
all the refactoring schedules obtained from our CP scheduler and confirmed correctness (i.e., feas-
sibility) in terms of constraint satisfaction. The choice of in-house systems and their developers
also facilitated manual investigation of constraints satisfaction and optimality of the refactoring
schedules computed by our scheduler.

Manual verification of optimality was difficult as it was difficult to manually produce an op-
timal schedule, because the code clone refactoring scheduling problem is NP-hard [5, 23, 22].
However, we made efforts to challenge the optimality of the produced solutions by attempting to
further increase the objective values by means of pseudo-random replacement of refactorings in the computed schedule. As we were not successful in that endeavour, we became convinced that our CP scheduler indeed produced optimal solution.

Given that the problem is well-defined, the mathematical foundation behind the LP technique guarantees to identify the optimum solution and so a head-to-head comparison between the schedules produced by the CP and LP techniques enables an automated approach for mathematical verification of the optimality of the schedules computed by the CP scheduler. As the schedules obtained from the CP scheduler were identical (or having same objective values) to those computed by the LP scheduler, we can confidently conclude that our CP scheduler indeed produced the optimum refactoring schedules for each of the subject systems in our study.

9.6.2 Internal Validity

Internal validity is mostly concerned with the “possible errors in our algorithm implementations and measurement tools that could affect outcomes” [26].

Our refactoring effort model requires 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 that 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 for realization of the effort model in a software tool further indicate the necessity and effectiveness of our effort model.

In the estimation of the impact of refactoring on code/design quality, we used the six of the QMOOD design property metrics and ignored the rest. Moreover, the impact of clone refactoring was estimated in accordance with Equation 9. The weighted sum of differences in Equation 9 might have not been able to capture the full benefits of the quality model. These are also threats to the study. While the choice of the design property metrics does affect the estimation of refactoring effects it does not affect the estimation of refactoring efforts based on our effort model. Indeed, the inclusion of all the metrics may affect the scheduling approaches and produce different schedules, but we see no reason why this may degrade the performance of our CP scheduler compared to the others. Nevertheless, carrying out a follow-up study including all the QMOOD design property
metrics can be worthwhile, which we plan to do in the future.

In our study, we found that the GA approach did not perform well and kept producing infeasible solutions. Our choice and implementation of the mutation and crossover operators may seem to be responsible for this. To minimize this threat, we tweaked those operators in several ways and tuned the parameters to the GA algorithm in separate runs. Then, we chose the best combination of parameters to use in our study. We believe that the reason to the poor performance of the GA approach was that the large set of constraints actually made the problem GA-hard [10, 11], as discussed in Section 9.5.1.

9.6.3 External Validity

External validity questions the generalizability of the results of a study across different experimental settings with larger population not considered in the study.

The six subject systems used in our study are in-house and small to medium in size. All the six respective developers are graduate students; among them two are Ph.D. students and the rest are M.Sc. students at the end of their program. It is arguable that the population is not large enough and subject systems of the study are not representatives of industrial or open-source systems while the developers may not represent the industrial practitioners. Thus, our study may be subject to threats to external validity. However, the choice of in-house systems and their developers helped us to minimize the threats to construct validity. One of the participants of the study had five years of industry experience and another had more than two years experience of working as a developer in software industry. Thus, the group of developers participated in our study represents a sample of programmers with different levels of expertise. Therefore, we believe that our study achieves an acceptable level of external validity. The threats to external validity can be further minimized by increasing the number and sizes of the subject systems, choosing both industrial and open-source software for study, and involving software developers with diverse levels of expertise (i.e., beginner, intermediate, expert).

9.6.4 Reliability

The methodology of the study including the procedure for data collection are documented in this paper. The NiCad clone detector as well as the in-house software systems used in our study are
available online\(^6\). The data, the OPL implementation of our CP and LP scheduler, as well as the Java implementation of genetic algorithm are also made available online\(^7\) for the interested parties. Therefore, it should be possible to replicate the study.

10 Related Work

Until recently, much research has been conducted towards effective identification and removal of different types of code smells from the source code. Because our work is focused on scheduling of code clone refactoring, we confine our discussion to those work that deal with scheduling of refactoring this code smell.

The work of Bouktif et al. [5], Lee et al. [22], and Liu et al. [23] closely relate to ours. Bouktif et al. [5] formulated the refactoring problem as a constrained Knapsack problem and applied a genetic algorithm (GA) to obtain an optimal solution. However, they ignored the constraints that might exist among the refactorings. Lee et al. [22] applied ordering messy GA (OmeGA), whereas Liu et al. [23] used a heuristic algorithm to schedule refactoring of code bad smells in general. Both those studies took into account conflicts and sequential dependencies among the refactorings, but missed the constraints of mutual inclusion and refactoring efforts. Our work differs from all those work in two ways. First, for computing the refactoring schedule, we applied constraint programming approach, which we have shown to be better than 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 11. Although Bouktif et al. [5] proposed a small effort model for code clone refactoring, their model was for procedural code only, which considers only the method call-chain and token modification efforts in terms of edit distance. On the contrary, our effort model is applicable to not only procedural but also object-oriented source code, as ours take into account diverse categories of efforts covering the constructs of an object-oriented paradigm.

O’Keeffe et al. [21] 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. However, we used CP, which combines the strengths of both AI and OR tech-

\(^6\)http://www.cs.usask.ca/faculty/croy/
\(^7\)http://usask.ca/~minhaz.zibran/pages/projects.html
niques [4] and thus led to our belief that CP would be a better choice for solving such scheduling problems. Indeed, from our empirical study, we found that the CP approach outperformed both GA and Linear Programming (LP) techniques in the scheduling of code clone refactoring. In our case, GA did not perform well because the refactoring scheduling problem that we have addressed is much stricter with a wide range of hard constraints that might have made the problem \textit{GA-hard} [10, 11], as discussed in Section 9.5.1.

A number of methodologies [9, 20, 34, 35, 41] and metric-based tools such as \textsc{CCShaper} [15] and \textsc{Aries} [14] have been proposed for semi-automated extraction of code clones as refactoring candidates. Several tools, such as \textsc{Libra} [16] and \textsc{CnP} [17], 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.

11 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 object-oriented and procedural 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 object-oriented source code, and our CP approach is a technique that no one else 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 shown to be effective in solving scheduling problems [4, 47]. 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 an empirical study with six in-house software systems and their developers. Through comparison with greedy, genetic algorithm (GA), linear programming, and manual approaches, we showed that our CP scheduler outperformed those techniques.
Our refactoring effort model was also found by the developers to be useful for estimating the efforts required for code clone refactoring. Indeed, the evaluation of the effort model is based on a pilot study with a few developers where the developers did not actually apply the refactorings of the subject systems. In the future, we plan to carry out a more structured user study in a larger scale where we will provide the developers with a tool implementation of our effort model. Then we will compare the tool’s estimation with the actual efforts the developers need to put for performing those refactorings by hand. Our immediate future plan also 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 [43, 44] that we have been developing.

Acknowledgments: The authors acknowledge the contributions of Ripon Saha, Muhammad Asaduzzaman, Sharif Uddin, Saidur Rahman, Manishankar Mondal, and Mohammad Khan for participating in the study to empirically evaluate our code clone refactoring scheduler and the effort model. This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC) and the Walter C. Sumner Memorial Foundation.

References


[33] Sahraoui, H., Godin, R., and Miceli, T.: ‘Can metrics help to bridge the gap between the improvement of OO design quality and its automation?’, Proc. 16th IEEE Int. Conf. on Software Maintenance (ICSM), San Jose, California, USA, October 2000, pp. 154–162.


Figure 1: Examples of Type-1, Type-2, and Type-3 clones
Figure 2: 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)
Figure 3: Different object-oriented patterns for code clone refactoring

(a) Pull-up method
- Clone methods method\(_1\) of Subclass\(_1\) and method\(_2\) of Subclass\(_2\) are pulled up in their common superclass.

(b) Extract superclass
- Clone methods method\(_1\) of Class\(_1\) and method\(_2\) of Class\(_2\) are pulled up in a newly created common superclass.

(c) Extract utility class
- Clone methods method\(_1\) of Class\(_1\) and method\(_2\) of Class\(_2\) are moved as a unified method into a third class.
$C_p$ is the lowest common superclass of $C_1$ and $C_6$.

$\text{pathLength}(C_1, C_p) = 1$
\[\text{pathLength}(C_6, C_p) = 2\]

$\partial(C_1, C_6) = \max \{\text{pathLength}(C_1, C_p), \text{pathLength}(C_6, C_p)\}$
\[= \max \{1, 2\} = 2\]

Figure 4: Computation of distance between classes
Figure 5: Mutual inclusion and mutual exclusion constraints on clone refactoring
Figure 6: Traditional encoding of a solution in a binary string
Figure 7: Our encoding of a solution in a chromosome
Figure 8: Crossover operation
Figure 9: Automated CP vs. manual scheduling
Table 1: QMOOD formula for quality attributes [3]

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reusability</td>
<td>$-0.25 \times DCC + 0.25 \times CAM + 0.5 \times CIS + 0.5 \times DSC$</td>
</tr>
<tr>
<td>Flexibility</td>
<td>$0.25 \times DAM - 0.25 \times DCC + 0.5 \times MOA + 0.5 \times NOP$</td>
</tr>
<tr>
<td>Understandability</td>
<td>$-0.33 \times ANA + 0.33 \times DAM - 0.33 \times DCC + 0.33 \times CAM - 0.33 \times NOP - 0.33 \times NOM - 0.33 \times DSC$</td>
</tr>
<tr>
<td>Functionality</td>
<td>$0.12 \times CAM + 0.22 \times NOP + 0.22 \times CIS + 0.22 \times DSC + 0.22 \times NOH$</td>
</tr>
<tr>
<td>Extendability</td>
<td>$0.5 \times ANA - 0.5 \times DCC + 0.5 \times MFA + 0.5 \times NOP$</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>$0.2 \times ANA + 0.2 \times DAM + 0.2 \times MOA + 0.2 \times MFA + 0.2 \times NOP$</td>
</tr>
</tbody>
</table>
Table 2: QMOOD metrics for design properties [3]

<table>
<thead>
<tr>
<th>Design Property</th>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design size</td>
<td>DSC</td>
<td>Design size in classes</td>
</tr>
<tr>
<td>Complexity</td>
<td>NOM</td>
<td>Number of methods</td>
</tr>
<tr>
<td>Coupling</td>
<td>DCC</td>
<td>Direct class coupling</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>NOP</td>
<td>Number of polymorphic methods</td>
</tr>
<tr>
<td>Hierarchies</td>
<td>NOH</td>
<td>Number of hierarchies</td>
</tr>
<tr>
<td>Cohesion</td>
<td>CAM</td>
<td>Cohesion among methods in class</td>
</tr>
<tr>
<td>Abstraction</td>
<td>ANA</td>
<td>Average number of ancestors</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>DAM</td>
<td>Data access metric</td>
</tr>
<tr>
<td>Composition</td>
<td>MOA</td>
<td>Measure of aggregation</td>
</tr>
<tr>
<td>Inheritance</td>
<td>MFA</td>
<td>Measure of functional abstraction</td>
</tr>
<tr>
<td>Messaging</td>
<td>CIS</td>
<td>Class interface size</td>
</tr>
</tbody>
</table>
Table 3: Constraint matrix $Z$ representing the constraints among the refactorings in $\mathcal{R}$

<table>
<thead>
<tr>
<th></th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>$r_4$</th>
<th>$r_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_1$</td>
<td></td>
<td></td>
<td>+3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_2$</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+2</td>
</tr>
<tr>
<td>$r_4$</td>
<td>−3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>−2</td>
</tr>
</tbody>
</table>
Table 4: Software systems subject to the empirical study

<table>
<thead>
<tr>
<th>Subject</th>
<th>SLOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomClone</td>
<td>2,239</td>
<td>A domain information-based clone analysis (prediction) tool</td>
</tr>
<tr>
<td>Mutation Framework</td>
<td>2,901</td>
<td>Ongoing extended implementation of the mutation framework proposed by Roy and Cordy [29]</td>
</tr>
<tr>
<td>LIME [43]</td>
<td>3,494</td>
<td>A source code comparison engine</td>
</tr>
<tr>
<td>SimCad [39]</td>
<td>3,771</td>
<td>A clone detection tool</td>
</tr>
<tr>
<td>gCad [31, 32]</td>
<td>4,563</td>
<td>A clone genealogy extractor</td>
</tr>
<tr>
<td>VisCad [2]</td>
<td>9,323</td>
<td>A tool for analysis and visualization of code clones</td>
</tr>
</tbody>
</table>
Table 5: Code clones in the systems under study

<table>
<thead>
<tr>
<th>Subject Systems</th>
<th>Clone Groups</th>
<th>Clone Fragments</th>
<th>Total Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomClone</td>
<td>21</td>
<td>56</td>
<td>77</td>
</tr>
<tr>
<td>Mutation Framework</td>
<td>21</td>
<td>62</td>
<td>72</td>
</tr>
<tr>
<td>LIME</td>
<td>20</td>
<td>55</td>
<td>67</td>
</tr>
<tr>
<td>SimCad</td>
<td>16</td>
<td>42</td>
<td>64</td>
</tr>
<tr>
<td>gCad</td>
<td>28</td>
<td>91</td>
<td>93</td>
</tr>
<tr>
<td>VisCad</td>
<td>57</td>
<td>136</td>
<td>166</td>
</tr>
</tbody>
</table>
Table 6: Example of operations and efforts for extract method

<table>
<thead>
<tr>
<th>Operations for extract method</th>
<th>Efforts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Produce signature of the target method</td>
<td>15</td>
</tr>
<tr>
<td>Copy clone fragment to the body of target method</td>
<td>1</td>
</tr>
<tr>
<td>Perform necessary modifications in the body</td>
<td>5</td>
</tr>
<tr>
<td>Replace clone fragments by calls to the extracted method</td>
<td>2</td>
</tr>
<tr>
<td>Total efforts</td>
<td>23</td>
</tr>
</tbody>
</table>
Table 7: Parameters for genetic algorithm

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>$\approx 1.0 \times$ chromosome-length</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>$\approx 1.0%$</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>$\approx 80%$</td>
</tr>
<tr>
<td>Elitism rate</td>
<td>$\approx 30%$</td>
</tr>
</tbody>
</table>
Table 8: Comparison of automated scheduling approaches

<table>
<thead>
<tr>
<th>Subject systems</th>
<th>Scheduling approaches</th>
<th>Values at dimensions</th>
<th>Quality − Effort</th>
<th>P × (Q − E)</th>
<th>Refac. chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation Framework</td>
<td>Greedy(^p)</td>
<td>20.06</td>
<td>21.94</td>
<td>18.53</td>
<td>-3.41</td>
</tr>
<tr>
<td></td>
<td>Greedy(^e)</td>
<td>9.63</td>
<td>6.06</td>
<td>10.04</td>
<td>3.98</td>
</tr>
<tr>
<td></td>
<td>Greedy(^q)</td>
<td>18.16</td>
<td>21.82</td>
<td>19.64</td>
<td>-2.18</td>
</tr>
<tr>
<td></td>
<td>GA(^*)</td>
<td>21.27</td>
<td>19.99</td>
<td>18.46</td>
<td>-1.53</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>9.34</td>
<td>7.86</td>
<td>11.48</td>
<td>3.62</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>9.34</td>
<td>7.86</td>
<td>11.48</td>
<td>3.62</td>
</tr>
<tr>
<td>LIME</td>
<td>Greedy(^p)</td>
<td>22.42</td>
<td>21.12</td>
<td>19.93</td>
<td>-1.19</td>
</tr>
<tr>
<td></td>
<td>Greedy(^e)</td>
<td>13.00</td>
<td>8.28</td>
<td>13.61</td>
<td>5.33</td>
</tr>
<tr>
<td></td>
<td>Greedy(^q)</td>
<td>16.29</td>
<td>23.49</td>
<td>26.07</td>
<td>3.77</td>
</tr>
<tr>
<td></td>
<td>GA(^*)</td>
<td>10.17</td>
<td>15.71</td>
<td>15.21</td>
<td>-0.50</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>11.04</td>
<td>12.32</td>
<td>16.12</td>
<td>3.80</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>11.04</td>
<td>12.32</td>
<td>16.12</td>
<td>3.80</td>
</tr>
<tr>
<td>SimCad</td>
<td>Greedy(^p)</td>
<td>27.42</td>
<td>25.23</td>
<td>16.82</td>
<td>-8.41</td>
</tr>
<tr>
<td></td>
<td>Greedy(^e)</td>
<td>13.23</td>
<td>7.12</td>
<td>13.7</td>
<td>6.58</td>
</tr>
<tr>
<td></td>
<td>Greedy(^q)</td>
<td>23.57</td>
<td>24.64</td>
<td>30.18</td>
<td>5.54</td>
</tr>
<tr>
<td></td>
<td>GA(^*)</td>
<td>19.33</td>
<td>17.18</td>
<td>20.95</td>
<td>3.77</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>12.78</td>
<td>8.99</td>
<td>18.96</td>
<td>9.97</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>12.78</td>
<td>8.99</td>
<td>18.96</td>
<td>9.97</td>
</tr>
<tr>
<td>gCad</td>
<td>Greedy(^p)</td>
<td>19.65</td>
<td>21.62</td>
<td>20.00</td>
<td>-1.62</td>
</tr>
<tr>
<td></td>
<td>Greedy(^e)</td>
<td>9.61</td>
<td>9.53</td>
<td>11.57</td>
<td>2.04</td>
</tr>
<tr>
<td></td>
<td>Greedy(^q)</td>
<td>12.05</td>
<td>23.48</td>
<td>25.98</td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td>GA(^*)</td>
<td>25.18</td>
<td>26.12</td>
<td>20.86</td>
<td>-5.26</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>6.70</td>
<td>15.19</td>
<td>17.99</td>
<td>2.80</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>6.70</td>
<td>15.19</td>
<td>17.99</td>
<td>2.80</td>
</tr>
<tr>
<td>VisCad</td>
<td>Greedy(^p)</td>
<td>36.14</td>
<td>32.57</td>
<td>25.71</td>
<td>-6.86</td>
</tr>
<tr>
<td></td>
<td>Greedy(^e)</td>
<td>16.12</td>
<td>18.63</td>
<td>13.20</td>
<td>-5.43</td>
</tr>
<tr>
<td></td>
<td>Greedy(^q)</td>
<td>29.02</td>
<td>33.81</td>
<td>34.32</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>GA(^*)</td>
<td>45.03</td>
<td>42.57</td>
<td>38.09</td>
<td>-4.48</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>15.02</td>
<td>16.20</td>
<td>22.32</td>
<td>6.12</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>15.33</td>
<td>15.78</td>
<td>21.90</td>
<td>6.12</td>
</tr>
<tr>
<td>DomClone</td>
<td>Greedy(^p)</td>
<td>37.64</td>
<td>28.06</td>
<td>23.77</td>
<td>-4.29</td>
</tr>
<tr>
<td></td>
<td>Greedy(^e)</td>
<td>18.79</td>
<td>7.54</td>
<td>10.98</td>
<td>3.44</td>
</tr>
<tr>
<td></td>
<td>Greedy(^q)</td>
<td>33.12</td>
<td>25.62</td>
<td>28.93</td>
<td>3.31</td>
</tr>
<tr>
<td></td>
<td>GA(^*)</td>
<td>26.64</td>
<td>24.02</td>
<td>23.23</td>
<td>-0.79</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>19.14</td>
<td>13.33</td>
<td>23.35</td>
<td>10.02</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>19.49</td>
<td>12.57</td>
<td>22.41</td>
<td>9.84</td>
</tr>
</tbody>
</table>

Here, Greedy\(^p\) = approach greedy towards priority satisfaction,
Greedy\(^e\) = approach greedy towards effort minimization,
Greedy\(^q\) = approach greedy towards quality gain,
*the computed schedule was infeasible

\[ P \times (Q − E) = Priorit \times (Quality − Effort) \]
Table 9: Time and memory comparison of CP and LP scheduling

<table>
<thead>
<tr>
<th>Subject systems</th>
<th>Scheduling approaches</th>
<th>Resource Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time (sec.)</td>
</tr>
<tr>
<td>Mutation Framework</td>
<td>LP</td>
<td>14.33</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>0.14</td>
</tr>
<tr>
<td>LIME</td>
<td>LP</td>
<td>185.69</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>0.20</td>
</tr>
<tr>
<td>SimCad</td>
<td>LP</td>
<td>54.09</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>0.97</td>
</tr>
<tr>
<td>gCad</td>
<td>LP</td>
<td>58.08</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>0.94</td>
</tr>
<tr>
<td>VisCad</td>
<td>LP</td>
<td>313.78</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>6.02</td>
</tr>
<tr>
<td>DomClone</td>
<td>LP</td>
<td>396.02</td>
</tr>
<tr>
<td></td>
<td>CP</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Here, MB = Megabyte, Mb = Megabit
### Table 10: Developers feedback on the Likert scale questions

<table>
<thead>
<tr>
<th>Ques.</th>
<th>Choice of answers and number of developers’ responses in brackets</th>
</tr>
</thead>
<tbody>
<tr>
<td>$IQ_1$</td>
<td>very easy (0)</td>
</tr>
<tr>
<td>$IQ_2$</td>
<td>provided no assistance (0)</td>
</tr>
</tbody>
</table>
Table 11: Comparison of code clone refactoring schedulers

<table>
<thead>
<tr>
<th></th>
<th>Bouktif et al. [5]</th>
<th>Lee et al. [22]</th>
<th>Liu et al. [23]</th>
<th>Our Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approach</td>
<td>GA</td>
<td>OmeGA</td>
<td>Heuristic</td>
<td>CP</td>
</tr>
<tr>
<td>Refactoring effort</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Quality gain</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Sequential dependency</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Mutual inclusion</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Priorities satisfaction</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>