Published in IET Software
Received on 3rd April 2012
Revised on 9th April 2013
Accepted on 9th April 2013
doi: 10.1049/iet-sen.2012.0058

Special Issue: 11th IEEE International Working Conference on Source Code Analysis and Manipulation

ISSN 1751-8806

Conflict-aware optimal scheduling of prioritised code clone refactoring

Minhaz Fahim Zibran, Chanchal Kumar Roy

Department of Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada S7N5C9
E-mail: minhaz.zibran@usask.caminhaz.zibran@gmail.com

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 of different priorities. Addressing all the 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 'maximise benefit and minimise refactoring effort'. However, the estimation of the efforts required to perform code clone refactoring is a challenging task. This study makes two contributions. First, the authors propose an effort model for the estimation of code clone refactoring efforts. Second, the authors propose a constraint programming (CP) approach for conflict-aware optimal scheduling of code clone refactoring. 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. The authors also quantitatively compared their refactoring scheduler with other well-known scheduling techniques such as the genetic algorithm, greedy approaches and linear programming. The authors' empirical study suggests that the proposed CP-based approach outperforms other approaches they considered.

1 Introduction

Code clone (duplicate or near-duplicate source code) is a well-known code smell [1, 2]. Programmers' copy–paste–modify practice is regarded as one of the main reasons for such 'intentional' clones that are beneficial in many ways [3] 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 that 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 because of a number of reasons. For example, the use of design patterns, frameworks and application programming interface (APIs) may result in unintentional code clones. Previous studies reported that software systems might have 9–17% [4] duplicated code, up to 50% [5].

Code clones may also be detrimental in many cases. Obviously, redundant code may inflate the codebase 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 minimised 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 prioritised refactoring schedule of the potential refactoring candidates (i.e. clones that are refactorable) 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 [1, 6] 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 minimised 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 maximised whereas the required effort is minimised [7].

Automated software refactoring is often performed with the aid of graph transformation tools [8], where the available refactorings are applied without having been optimally scheduled [9]. The application order of the semi-automated (or manual) refactorings is usually determined implicitly by human developers. However, developers' efforts may be inefficient and error-prone, especially for large systems. Although experienced developers may do it well, inexperienced ones 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 must 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 a 'non-deterministic polynomial-time (NP)-hard' problem [9–11] 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 'effort model' for estimating the developers' effort required to refactor code clones in procedural or object-oriented (OO) programs. Second, taking into account the 'effort model' and a wide variety of possible hard and soft constraints, we formulate the scheduling of code clone refactorings as a constraint satisfaction optimisation problem (CSOP) and solve it by applying constraint programming (CP) techniques that aims to maximise benefits (measured in terms of changes in the code/design quality metrics) while minimising refactoring efforts.

To the best of our knowledge, ours is the first effort model for refactoring OO 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 [12, 13].

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 [14], 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 organised 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 developer. 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 categorising definitions of code clone have been widely accepted today [4, 15–19].

Type-1 clones: Identical code fragments except for variations in white-spaces and comments are 'Type-1' clones.
Type-2 clones: Structurally/syntactically identical fragments except for variations in the names of identifiers, literals, types, layout and comments are called 'Type-2' clones.
Type-3 clones: 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 clones: 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, that is, each

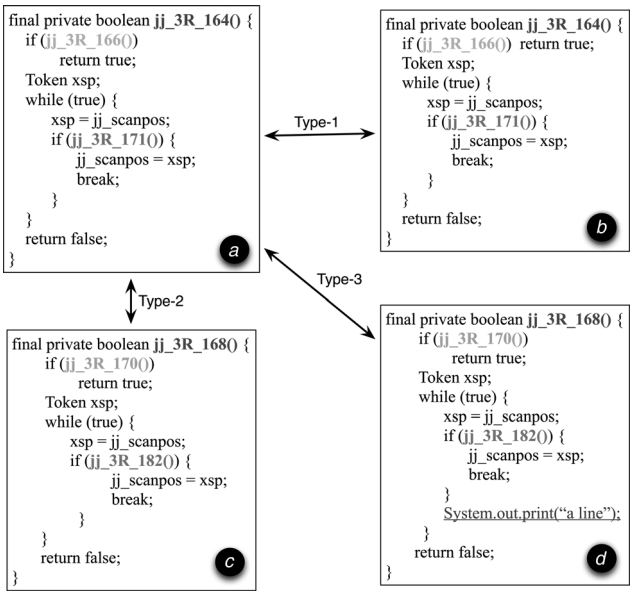

Fig. 1 *Examples of Type-1, Type-2 and Type-3 clones*

doi: 10.1049/iet-sen.2012.0058

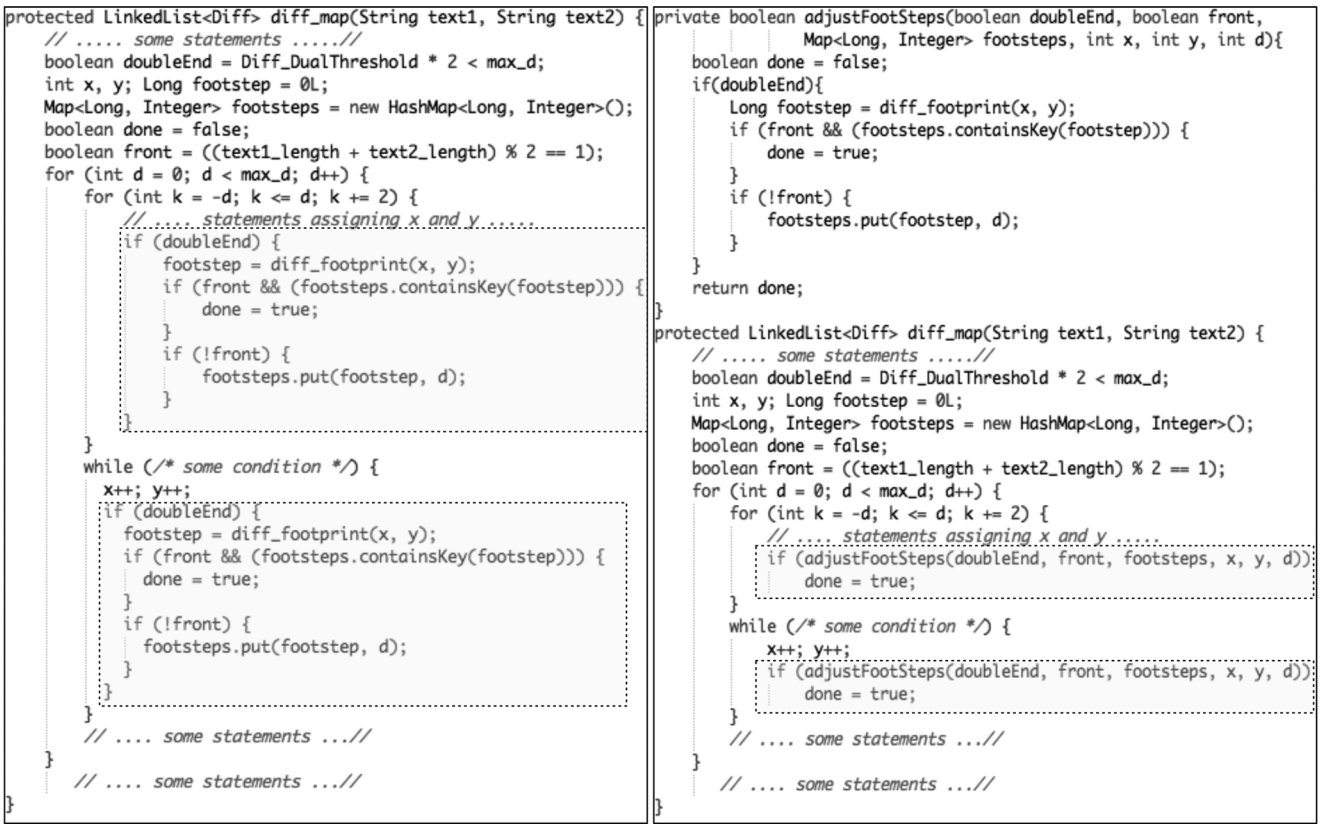

Fig. 2 *Example of clone refactoring in VisCad: the method on the top-right corner is extracted by generalising the clone-pairs (shaded blocks on the left)*

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. Owing 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. Fig. 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 and 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 Fig. 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.* [1] initially proposed a set of 72 software refactoring patterns and until recently the number has increased to 93 [6]. 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) [20]. Based on a survey of existing literature [11, 15, 21–24] and our experience, we find that among those general software refactoring patterns [1, 6], the following patterns are particularly suitable for code clone refactoring. Details about these refactoring patterns can be found elsewhere [1, 6].

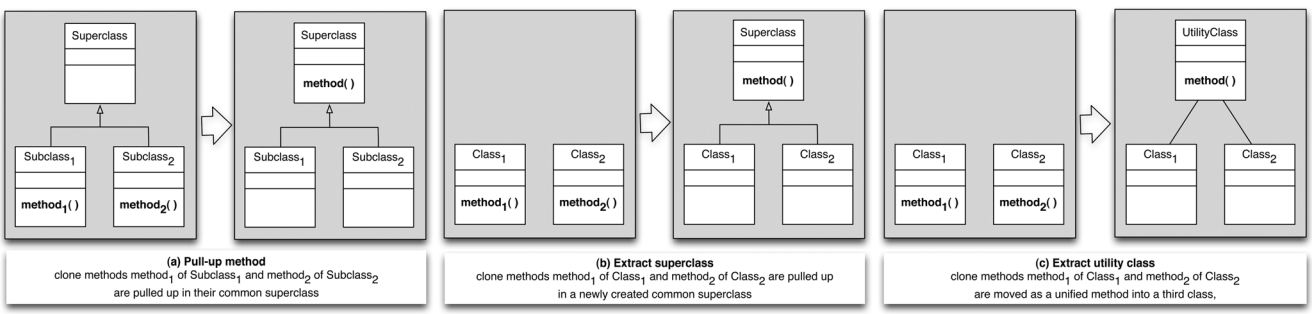

Fig. 3 *Different OO patterns for code clone refactoring*

• *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 generalised method. Fig. 2 shows an example of the EM refactoring.
• *Pull-up method (PM)* removes similar methods found in several classes by introducing a generalised method in their common superclass. Fig. 3*a* demonstrates a PM refactoring through a schematic diagram.
• *Extract superclass (ES)* introduces a new common superclass for two or more classes having similar methods and then applies PM. ES refactoring may be necessary when those classes do not already have a common superclass and those classes can be brought under a common superclass. Fig. 3*b* presents an a schematic diagram demonstrating the ES 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 generalising the similar methods that must be removed from those classes. Fig. 3*c* demonstrates EU refactoring through a schematic diagram.

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 generalised 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 blocks 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 formalised as:

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

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.

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

$$c_i \curvearrowright m_i \curvearrowright C_i \curvearrowright F_i \curvearrowright D_i, \quad \text{for object - oriented code}$$

$$c_i \curvearrowright m_i \curvearrowright F_i \curvearrowright D_i, \quad \text{for procedural code}$$

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

$$C(g) = \{C_i \quad | \forall c_i \in g, c_i \curvearrowright C_i\} \tag{1}$$

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

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 EM refactoring pattern on exact duplicate code clones will require less effort than that for applying it 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 OO source code. We have formulated this 'effort model' based on our understandings, developed from a survey of existing literature [1, 6, 20, 22, 25–27] and our experience in clone refactoring.

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.

4.1.1 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

$$\text{delegation}(c_i) = M_f(c_i) \cup M_r(c_i) \cup \{m_i\} \tag{2}$$

Hence, for understanding delegation concerning all the clone

doi: 10.1049/iet-sen.2012.0058

fragments in g, the set of methods required to examine, becomes

$$\text{Delegation}(g) = \bigcup_{c_i \in C(g)} \text{delegation}(c_i) \quad (3)$$

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

$$E_d(g) = \sum_{m \in \text{Delegation}(g)} \text{LOC}(m) \quad (4)$$

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

4.1.2 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 $C(g)$ [For 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 can be ignored, and those classes are considered to have no common superclass.]. 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 $E_h(g)$ required for this can be estimated as

$$E_h(g) = \sum_{C \in C_h(g)} \text{LOC}(C) \quad (5)$$

4.2 Effort for code modifications

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

4.2.1 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

$$E_t(g) = \sum_{i=1}^{k} \delta(t_i, t_i') \quad (6)$$

However, the state-of-the-art integrated development environments (IDEs) 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 μ. By default, $\mu = 1$, but the developer can set a different value to μ as appropriate.

4.2.2 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 β 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 β.

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

$$E_n(g) = |F_d(g) \cup F_h(g)| + |D_d(g) \cup D_h(g)| + \text{DCH}(g) + \text{DFH}(g) \quad (7)$$

where:

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

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

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

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

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

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

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 represent 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, $\partial(C_i, C_j) = \max\{\text{pathLength}(C_i, C_p), \text{pathLength}(C_j, C_p)\}$, where pathLength($C_i$, C_p) is measured as the number

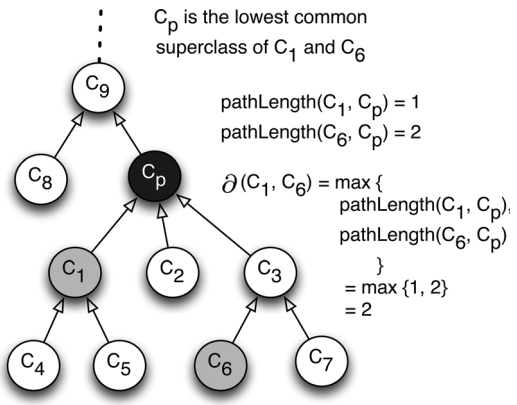

Fig. 4 *Computation of distance between classes*

of edges in the shortest path from C_i to C_p. In Fig. 4, the computation of distance between classes is illustrated with an example. More detail about DCH(g) can be found elsewhere [21]. DFH(g) is a similar metric that captures the 'dispersion of files' with $\delta(F_i, F_j)$ denoting the distance between files F_i and F_j in the file-system hierarchy.

4.4 Effort model

Based on the criteria discussed before, the total effort $E(g)$ needed to refactor clone-group g is estimated as

$$E(g) = w_d \times E_d(g) + w_h \times E_h(g) + w_t \times E_t(g) + w_r \times E_r(g) + w_n \times E_n(g) \quad (8)$$

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 developer may assign different weights to penalise 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 OO systems, these metrics can be supplemented by OO design quality models, such as quality model for object-oriented design (QMOOD) [28] or design quality metrics, such as the Chidamber–Kemerer [29] 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 [10, 11, 30–32].

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

$$\overline{Q}_r = \sum_{j=1}^{\eta} \vartheta_j \times \left(q_{j'} - q_j\right) \quad (9)$$

where ϑ_j is the weight on the jth quality attribute. By default $\vartheta_j = 1$, but the developers 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 OO source code. QMOOD is a prominent quality model for OO systems, which is used by other researchers [9–11]. 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 (9) may not be able to utilise the full benefit of the quality model, but it serves our purpose.

Table 1 QMOOD formula for quality attributes [28]

Attribute	Formula
reusability	$= -0.25 \times \text{DCC} + 0.25 \times \text{CAM} + 0.5 \times \text{CIS} + 0.5 \times \text{DSC}$
flexibility	$= 0.25 \times \text{DAM} - 0.25 \times \text{DCC} + 0.5 \times \text{MOA} + 0.5 \times \text{NOP}$
understandability	$= -0.33 \times \text{ANA} + 0.33 \times \text{DAM} - 0.33 \times \text{DCC} + 0.33 \times \text{CAM} - 0.33 \times \text{NOP} - 0.33 \times \text{NOM} - 0.33 \times \text{DSC}$
functionality	$= 0.12 \times \text{CAM} + 0.22 \times \text{NOP} + 0.22 \times \text{CIS} + 0.22 \times \text{DSC} + 0.22 \times \text{NOH}$
extendability	$= 0.5 \times \text{ANA} - 0.5 \times \text{DCC} + 0.5 \times \text{MFA} + 0.5 \times \text{NOP}$
effectiveness	$= 0.2 \times \text{ANA} + 0.2 \times \text{DAM} + 0.2 \times \text{MOA} + 0.2 \times \text{MFA} + 0.2 \times \text{NOP}$

Table 2 QMOOD metrics for design properties [28]

Design property	Metric	Description
design size	DSC	design size in classes
complexity	NOM	number of methods
coupling	DCC	direct class coupling
polymorphism	NOP	number of polymorphic methods
hierarchies	NOH	number of hierarchies
cohesion	CAM	cohesion among methods in class
abstraction	ANA	average number of ancestors
encapsulation	DAM	data access metric
composition	MOA	measure of aggregation
inheritance	MFA	measure of functional abstraction
messaging	CIS	class interface size

6 Refactoring constraints

Among the applicable refactorings, there may be conflicts and dependencies [26] besides their distinguishable impacts on the design quality. The application of one refactoring may cause the operands of other refactorings to disappear and thus may invalidate their applicability [10, 11, 26]. Besides such 'mutual exclusion' on conflicting refactorings, the application of one refactoring may also reveal new refactoring opportunities, as suggested by Lee *et al*. [11]. We understand that these are largely because of the composite structure of certain refactoring patterns, where one larger refactoring is composed of several smaller core refactorings [20]. For example, when ES refactoring is applied, PM is also applied as a part of it (Fig. 3*b*).

There may also be 'sequential dependencies' between refactoring activities [11, 26]. Constraints of 'mutual inclusion' may also arise when the application of one refactoring will necessitate the application of certain other refactorings [24]. Fig. 5*a* presents an example of mutual inclusion constraint and Fig. 5*b* demonstrates a mutual exclusion constraint with an example. The organisation's management may also impose 'priorities' on certain refactoring activities [10], 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.

doi: 10.1049/iet-sen.2012.0058

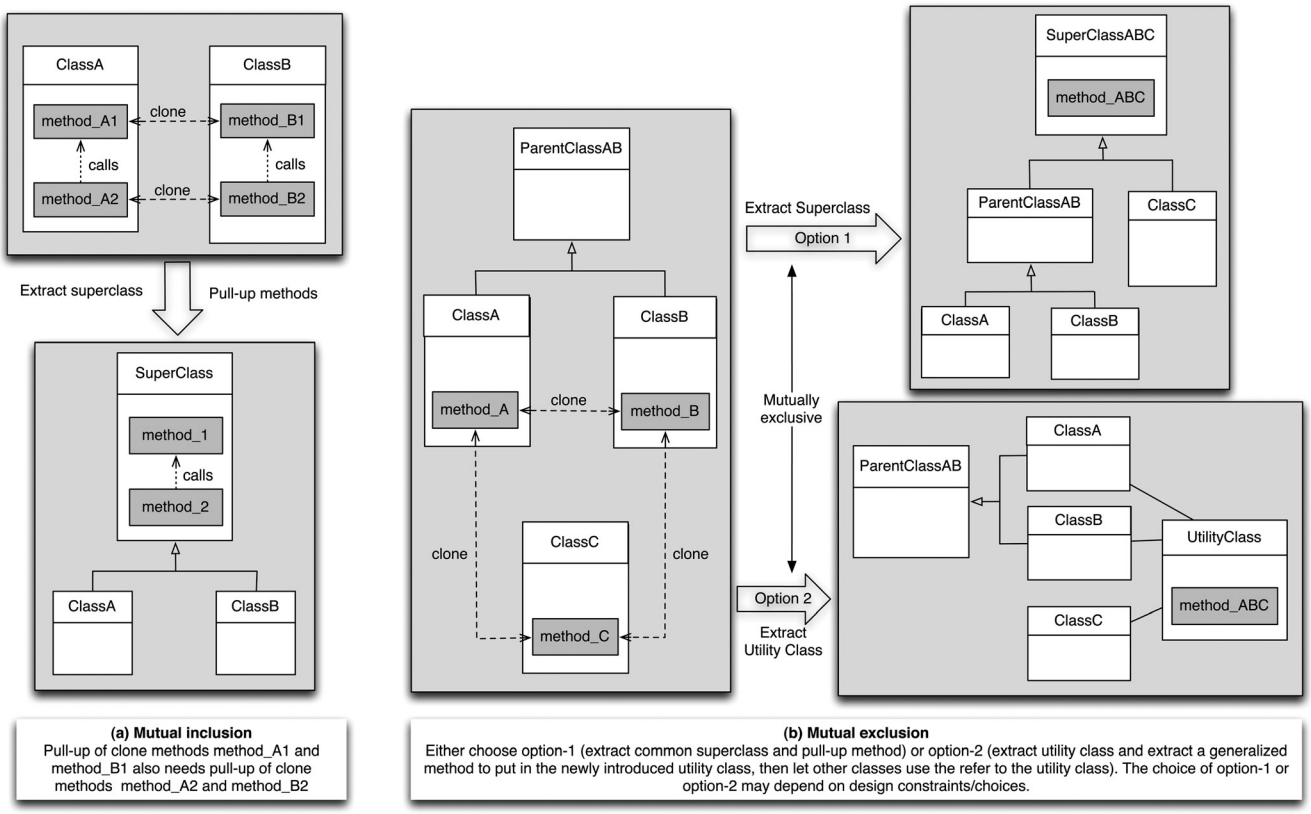

Fig. 5 *Mutual inclusion and mutual exclusion constraints on clone refactoring*

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

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

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 [10, 11, 24, 26].

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 maximise code quality while minimising efforts is a NP-hard problem [9–11]. 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 CSOP. A CSOP is a kind of problem characterised 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 optimise (i.e. maximise or minimise). We model the refactoring scheduling problem as a CSOP and solve it by applying a CP 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 $\mathcal{Y}_r$

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

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

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 $\mathcal{Z}$ to capture the constraints and sequential dependencies between distinct refactorings r_i and r_j:

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

$\mathcal{Z}_{i,j} = -\mathcal{Z}_{j,i}$ or $\mathcal{Z}_{i,j} = \mathcal{Z}_{j,i} = 1$, for all $\langle i, j \rangle$.

Let ρ_r be the priority on the refactoring r that operates on clone-group g_r. The CSOP formulation of the refactoring

scheduling problem can be defined as follows

$$\text{maximise} \quad \sum_{r\in\mathcal{R}} \mathcal{X}_r \rho_r \left(\overline{Q}_r - E(g_r)\right) \tag{10}$$

subject to (with $i \neq j$)

$$\mathcal{X}_r + \mathcal{Y}_r \neq 1, \quad \forall r \in \mathcal{R} \tag{11}$$

$$\mathcal{X}_{r_i} + \mathcal{X}_{r_j} = 2 \Rightarrow \mathcal{Y}_{r_i} \neq \mathcal{Y}_{r_j}, \quad \forall r_i, r_j \in \mathcal{R} \tag{12}$$

$$\mathcal{Z}_{i,j} - \mathcal{Z}_{j,i} > 0 \Rightarrow \mathcal{Y}_{r_i} < \mathcal{Y}_{r_j}, \quad \forall r_i, r_j \in \mathcal{R} \tag{13}$$

$$\mathcal{Z}_{i,j} - \mathcal{Z}_{j,i} < 0 \Rightarrow \mathcal{Y}_{r_i} > \mathcal{Y}_{r_j}, \quad \forall r_i, r_j \in \mathcal{R} \tag{14}$$

$$|\mathcal{Z}_{i,j}| = 1 \Rightarrow \mathcal{X}_{r_i} + \mathcal{X}_{r_j} \leq 1, \quad \forall r_i, r_j \in \mathcal{R} \tag{15}$$

$$|\mathcal{Z}_{i,j}| = 2 \Rightarrow \left(\mathcal{X}_{r_i} + \mathcal{X}_{r_j}\right) \text{ modulo } 2 = 0, \quad \forall r_i, r_j \in \mathcal{R} \tag{16}$$

$$\sum_{r\in\mathcal{R}} \mathcal{X}_r \leq \mathcal{M} \tag{17}$$

Here, (10) is the objective function for maximising the code quality and minimising the refactoring effort while rewarding refactoring activities having higher priorities. One of the product term in the objective function is $\mathcal{X}_r$, which is equal to 1 only for selected refactorings, and for all other refactorings $\mathcal{X}_r$ 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 $\mathcal{X}_r$ and $\mathcal{Y}_r$ are kept consistent as their values are assigned. If the refactoring r is not selected (i.e. $\mathcal{X}_r = 0$), then $\mathcal{Y}_r$ must also be 0, to denote that the refactoring r is not assigned any position in the sequence of the scheduled refactorings. If the refactoring r is selected (i.e. $\mathcal{X}_r = 1$), then $\mathcal{Y}_r$ must not be zero, that is, $\mathcal{X}_r + \mathcal{Y}_r \neq 1$.

Equation (12) enforces that no two refactorings are scheduled at the same point in the sequence. Equations (13) and (14) impose the sequential dependency constraints (i.e. $r_i \nrightarrow r_j$) on feasible schedules. Mutual exclusion (i.e. $r_i \nleftrightarrow r_j$) and mutual inclusion (i.e. $r_i \leftrightarrow r_j$) constraints are enforced by (15) and (16), respectively. Equation (17) specifies that maximum $\mathcal{M}$ number of refactorings can be chosen for scheduling. By default $\mathcal{M} = |\mathcal{R}|$, but $\mathcal{M}$ can be set to a lower integer when a schedule of a certain number of refactoring activities is desired, because of limitation of time, resource or the like.

7.1 Illustrative example

Now, with an example, we further illustrate our formulation, especially the constraint matrix $\mathcal{Z}$. Consider a set of five refactorings $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$ having constraints as follows:

(i) $r_2 \nleftrightarrow r_4$ (i.e. r_2 and r_4 are mutually exclusive),
(ii) $r_4 \nrightarrow r_1$ (i.e. r_1 cannot be applied after r_4),
(iii) $r_5 \nrightarrow r_3$ (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 $\langle r_i, r_j \rangle \in \mathcal{R}$ 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 $\mathcal{Z}$ is shown in Table 3. The empty cells in the table are filled up will zeros, which we omitted here for the purpose of better readability.

Table 3 Constraint matrix $\mathcal{Z}$ representing the constraints among the refactorings in $\mathcal{R}$

	r_1	r_2	r_3	r_4	r_5
r_1	–	–	–	+3	–
r_2	–	–	–	1	–
r_3	–	–	–	–	+2
r_4	−3	1	–	–	–
r_5	–	–	−2	–	–

The constraint (i) is enforced by (15). Here, $\mathcal{Z}_{2,4} = \mathcal{Z}_{4,2} = 1$. If both r_2 and r_4 are selected then $\mathcal{X}_{r_1} + \mathcal{X}_{r_2} = 1 + 1 = 2$, which violates the constraint in (15).

The constraint (ii) is imposed by (13) and (14). Here, $\mathcal{Z}_{1,4} = +3$ and $\mathcal{Z}_{4,1} = -3$ and thus, $\mathcal{Z}_{1,4} - \mathcal{Z}_{4,1} = 6$, which is higher than zero. Hence, (13) imposes that $\mathcal{Y}_{r_1} < \mathcal{Y}_{r_4}$, and thus r_1 preceedes r_4 in the schedule (if both are selected). Again, with respect to (14), $\mathcal{Z}_{4,1} - \mathcal{Z}_{1,4} = -6$, which is less than zero and hence $\mathcal{Y}_{r_4} > \mathcal{Y}_{r_1}$ is imposed. Thus, (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, $\mathcal{Z}_{3,5} - \mathcal{Z}_{5,3} = 4$ and $\mathcal{Z}_{5,3} - \mathcal{Z}_{3,4} = -4$, 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 (16). According to our current example, $|\mathcal{Z}_{i,j}| = |\mathcal{Z}_{j,i}| = 2$. Hence, (16) ensures that the remainder of $(\mathcal{X}_{r_3} + \mathcal{X}_{r_5})$ divided by 2 must be equal to 0, which is possible only if $\mathcal{X}_{r_3} = \mathcal{X}_{r_5} = 1$ or $\mathcal{X}_{r_3} = \mathcal{X}_{r_5} = 0$, 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 optimisation programming language (OPL) [Optimisation programming language (OPL) is a relatively new modelling language for combinatorial optimisation that simplifies the formulation and solution of optimisation problems.]. For OPL programming, we used the IBM ILOG CPLEX Optimisation Studio 12.2 IDE under an academic license. The IDE can be integrated with CPLEX Solver and CP Optimiser, which are IBM's optimisation engines for solving optimisation problems modelled in LP and CP, respectively.

CP combines techniques from AI and OR and it has been shown to be effective in solving combinatorial optimisation problems, especially in the area of scheduling and planning [12, 13]. Over the past decade, a separate conference series [International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR).] 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

doi: 10.1049/iet-sen.2012.0058

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 Optimiser from inside the IBM ILOG CPLEX Optimisation 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 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 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 LP, GA and three variants of greedy algorithms for optimising the automated scheduling of code clone refactorings. Further details 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 [Software Research Lab, Department of Computer Science, University of Saskatchewan, Canada, http://www.cs.usask.ca/research/research_groups/selab]. The subject systems and their sizes in terms of 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 effectiveness 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 through post-study questionnaires.

Table 4 Software systems subject to the empirical study

Subject systems	SLOC	Description
DomClone	2239	a domain information-based clone analysis (prediction) tool
mutation framework	2901	ongoing extended implementation of the mutation framework proposed by Roy and Cordy [16]
LIME [7]	3494	a source code comparison engine
SimCad [18]	3771	a clone detection tool
gCad [17, 33]	4563	a clone genealogy extractor
VisCad [34]	9323	a tool for analysis and visualisation of code clones

Typically, it is difficult and risky for developers to refactor a source code with which they are not familiar [9]. However, it is the developers who are likely to know the best about the critical parts of the projects that 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 should 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 catalogue of common software refactoring patterns [6] 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, and no automated tool exists to compute them. 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 [35] for detecting near-miss 'block' clones of at least five lines in pretty-printed format. We used the 'blind-rename' option of NiCad with unique percentage of items threshold (UPIT) 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 normalised 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 analysed the detected clones and rearranged the groups when necessary, based on the suitability for refactoring within context according to their understanding. For the analysis, the developers used VisCad [34], a code clone analysis and visualisation tool developed in our research lab. For each of the systems, the number of clone-groups and the number of distinct clone blocks involved in those groups are presented in Table 5, which the developers identified as the potential candidates for refactoring.

Having the code clones organised 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 Fig. 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 [The code is originally a part of diff-match-patch, an open-source library (available at http://code.google.com/p/google-diff-match-patch/) that VisCad uses internally. We deliberately chose to present this simple example, so that anyone can easily follow and verify.]. The developers chose to refactor them by applying the 'EM' 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 three 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. Although 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 a 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 compute the total number of classes or methods (before and after the refactoring) in the system, they just estimated the changes in the number of classes or methods. For example, the refactoring scenario presented in Fig. 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 clone-groups that 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.

Table 5 Code clones in the systems under study

Subject systems	Clone groups	Clone fragments	Total refactorings
DomClone	21	56	77
mutation framework	21	62	72
LIME	20	55	67
SimCad	16	42	64
gCad	28	91	93
VisCad	57	136	166

Table 6 Example of operations and efforts for EM

Operations for EM	Efforts
produce signature of the target method	15
copy clone fragment to the body of target method	1
perform necessary modifications in the body	5
replace clone fragments by calls to the extracted method	2
total efforts	23

9.3 Data normalisation

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

doi: 10.1049/iet-sen.2012.0058

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 optimised towards three dimensions: minimising the refactoring efforts, maximising the refactoring benefits and maximising 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 normalised 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, norm(v_i) denotes the normalised value of v_i. The set S can be the set of values for all the refactorings along any of the three dimensions. The normalisation actually brings the magnitudes of all those values between 0 and 1 (i.e. $0 < |\text{norm}(v_i)| \leq 1$) and thus minimises the inter-dimension influence of the magnitudes, whereas still preserving the relative ratios of magnitudes within dimensions. The emphasis on the efforts compared with the qualities can be tweaked by setting higher or lower weights as described in (8).

For priorities, before applying the aforementioned normalisation, we carried out an additional normalisation phase by adding +6 to each priority values. Thus, we removed any priority ≤ 0. This removal was necessary as in our objective function, the difference between quality and effort is multiplied by priorities and we must ensure 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 normalised data and organised 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 GA that was used by other researchers [10, 11]. In the third phase, we compared the CP scheduling with a manual approach. 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 Sections 4 and 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 normalised 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 Optimisation 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 normalised data for each of the subject system were fed to our scheduler as described in Sections 7 and 8. The scheduler, upon obtaining the data in valid OPL format, applies constraint propagation and domain reduction techniques [13] to generate the optimal solution as instructed.

9.4.2 LP scheduling: LP is a mathematical programming technique for solving optimisation problems. Over the past few decades, LP has been widely used in the OR community for dealing with optimisation problems. The basic idea is to formulate the problem as a LP problem and solve it using LP algorithms, such as the simplex method, ellipsoid method and interior-point techniques [36]. An LP problem is a mathematical formulation of an optimisation 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 were 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 CPEX Optimisation 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 [13, 36].

9.4.3 GA scheduling: GA is a kind of evolutionary algorithm from the field of AI for solving optimisation problems. In GA, a candidate solution is encoded as a sequence of values, called a 'chromosome'; 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 $\mathcal{M}$ members from the set $\mathcal{R}$ of all potential refactorings (recall from Section 7) for each of the subject systems.

Fig. 6 *Traditional encoding of a solution in a binary string*

Fig. 7 *Our encoding of a solution in a chromosome*

Encoding: For each system subject to our study, all the candidate refactorings were enumerated with integers 1 through $|\mathcal{R}|$. Having such an enumeration, a common approach is to represent the problem as a binary Knapsack [10] problem and encode a solution as a binary string (Fig. 6) of length $|\mathcal{R}|$. 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 ζ as a sequence of $\mathcal{M}$ integers, having their positions indexed with 1 through $\mathcal{M}$, as shown in Fig. 7.

Having $\zeta[i]$, denoting the chosen refactoring at index i, the encoding scheme is 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 ith refactoring in the schedule represented by ζ.
- $\zeta[i] < \zeta[j]$ 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 ζ 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 ζ must not select the same refactoring more than once. That is, $\zeta[i] \neq \zeta[j]$ must hold if $\zeta[i] \neq 0$.

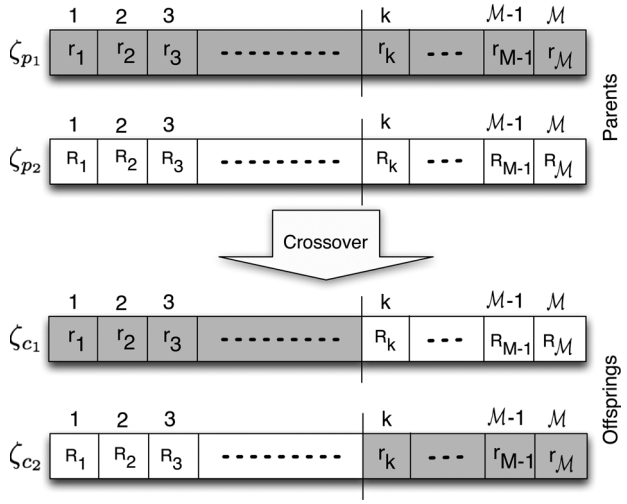

Fig. 8 *Crossover operation*

Crossover operation: The crossover operation, as shown in the Fig. 8, randomly selects two chromosomes ζ_{p_1} and ζ_{p_2} as parents, an index k as the point of crossover, and creates two offsprings ζ_{c_1} and ζ_{c_2} as follows

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

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

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

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

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

Algorithm 1

Require:
  $R \leftarrow \{1, 2, 3, \ldots, |\mathcal{R}|\}$ // $\mathcal{M}$ is the length of a chromosome
  procedure CREATEPOPULATION(P) // enumerations over all refactorings $\mathcal{M} \leq |\mathcal{R}|$
    $S_p \leftarrow \{\}$ // a set to accommodate all solutions
    while $|S_p| < P$ **do**
      $x \leftarrow$ RandomSolution( )
      $S_p \leftarrow S_p \cup \{x\}$
    end while
    return S_p
  end procedure
  procedure RANDOMSOLUTION
    $\zeta[\,] \leftarrow \{0, 0, \ldots, 0\}$ // array of $\mathcal{M}$ zeros
    $S_r \leftarrow R \cup \{0\}$ // $\mathcal{R}$ is the set of all refactorings
    for $k \leftarrow 0$ to $\mathcal{M} - 1$ **do**
      $r \leftarrow$ randomly chosen, $r \in S_r$
      $\zeta[k] \leftarrow r$
      if $r \neq 0$ **then**
        $S_r \leftarrow S_r - \{r\}$
      end if
    end for
    return ζ
  end procedure

Fig. 9 *Algorithm for creating population*

doi: 10.1049/iet-sen.2012.0058

Algorithm 2

$g \leftarrow 0$ // generation set to 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
  $d \leftarrow max_{population} - |S_p|$
  if d is odd **then**
    $d \leftarrow d + 1$
  end if
  $S_n \leftarrow \{\}$
  for $i \leftarrow 1$ to $\frac{d}{2}$ **do**
    $\{\zeta_{p_1}, \zeta_{p_2}\} \leftarrow$ random chromosomes from S_p
    $\{\zeta_{c_1}, \zeta_{c_2}\} \leftarrow \text{CrossOver}(\zeta_{p_1}, \zeta_{p_2})$
    $S_n \leftarrow S_n \cup \{\zeta_{c_1}, \zeta_{c_2}\}$
  end for
  for $\zeta \in S_n$ **do**
    $m \leftarrow \text{MutateChance}(\zeta)$ // chance of ζ to be mutated
    if $m \geq mutationRate$ **then**
      $\zeta \leftarrow \text{Mutation}(\zeta)$ // mutate ζ
    end if
  end for
  $S_p \leftarrow S_p \cup S_n$
  $t \leftarrow \text{elapsedTime}()$ // time duration of evolution
until $(g \geq max_{generation})$ or $(t \geq max_{time})$
return ζ_{best}

Fig. 10 *Description of genetic algorithm*

the population participate in crossover operation leaving 20% survivors (unchanged chromosomes).

Mutation operation: The mutation operation on a chromosome ζ selects a random index k and replaces the refactoring $\zeta[k]$ by another randomly selected refactoring $r \in \mathcal{R}$ that is not already in the chromosome. Mathematically, $\zeta[i] \neq r, \forall i \in \{1, 2, 3, \ldots, \mathcal{M}\}$ holds before the mutation.

Fitness function: The fitness function is defined by the objective function described by (10) in Section 7. The fitness function determines how good (fit) a solution is, as represented by a chromosome.

Population generation: The GA 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 (see Fig. 9).

Genetic evolution: The algorithm evolves according to the description in Algorithm 2 (see Fig. 10). The GA and its evolution is characterised by a number of parameters. We executed the GA 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 GA. The chosen values for the parameters, as presented in Table 7, are consistent with general recommendations [37]. For each of the subject systems, we executed the GA scheduler five times. At each run, the scheduler executed for an evolution time of 10 000 s (i.e. 2.78 h) and we kept 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).

Table 7 Parameters for GA

Parameter	Value
population size	$\simeq 1.0 \times$ choromosome-length
mutation rate	$\simeq 1.0\%$
crossover rate	$\simeq 80\%$
elitism rate	$\simeq 30\%$

9.4.4 Greedy scheduling: Recall that we identified three dimensions (i.e. effort, quality and priority) for optimising scheduling of code clone refactorings. Thus, we implemented (using OPL) three variants of a greedy algorithm, each aiming to optimise along one of the dimensions (i.e. optimisation criteria) disregarding the other two. The prime objective of the 'Greedy^{e}' approach is to compute schedules by minimising refactoring effort whereas the 'Greedy^{p}' and 'Greedy^{q}' approaches aim to maximise 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 optimisation 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 (see Fig. 10).

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.

Table 8 Comparison of automated scheduling approaches

Subject systems	Scheduling approaches	Values at dimensions			Quality effort	$P \times (Q - E)$	Refac. chosen
		Prior.	Effort	Quality			
mutation framework	Greedy^{p}	20.06	21.94	18.53	−3.41	−68.40	40
	Greedy^{e}	9.63	6.06	10.04	3.98	38.33	20
	Greedy^{q}	18.16	21.82	19.64	−2.18	−39.59	42
	GA[a]	21.27	19.99	18.46	−1.53	−32.54	36
	LP	9.34	7.86	11.48	3.62	33.81	20
	CP	9.34	7.86	11.48	3.62	33.81	20
LIME	Greedy^{p}	22.42	21.12	19.93	−1.19	−26.68	47
	Greedy^{e}	13.00	8.28	13.61	5.33	69.29	33
	Greedy^{q}	16.29	23.49	26.07	2.58	42.03	51
	GA	10.17	15.71	15.21	−0.50	−5.09	33
	LP	11.04	12.32	16.12	3.80	41.95	33
	CP	11.04	12.32	16.12	3.80	41.95	33
SimCad	Greedy^{p}	27.42	25.23	16.82	−8.41	−230.60	52
	Greedy^{e}	13.23	7.12	13.7	6.58	87.05	25
	Greedy^{q}	23.57	24.64	30.18	5.54	130.58	51
	GA[a]	19.33	17.18	20.95	3.77	72.87	32
	LP	12.78	8.99	18.96	9.97	127.42	25
	CP	12.78	8.99	18.96	9.97	127.42	25
gCad	Greedy^{p}	19.65	21.62	20.00	−1.62	−31.83	41
	Greedy^{e}	9.61	9.53	11.57	2.04	19.60	28
	Greedy^{q}	12.05	23.48	25.98	2.50	30.13	44
	GA[a]	25.18	26.12	20.86	−5.26	−132.45	45
	LP	6.70	15.19	17.99	2.80	18.73	28
	CP	6.70	15.19	17.99	2.80	18.73	28
VisCad	Greedy^{p}	36.14	32.57	25.71	−6.86	−247.92	66
	Greedy^{e}	16.12	18.63	13.20	−5.43	−87.53	40
	Greedy^{q}	29.02	33.81	34.32	0.51	14.80	72
	GA[a]	45.03	42.57	38.09	−4.48	−201.73	83
	LP	15.02	16.20	22.32	6.12	91.92	41
	CP	15.33	15.78	21.90	6.12	93.82	40
DomClone	Greedy^{p}	37.64	28.06	23.77	−4.29	−161.48	62
	Greedy^{e}	18.79	7.54	10.98	3.44	64.64	33
	Greedy^{q}	33.12	25.62	28.93	3.31	109.63	56
	GA[a]	26.64	24.02	23.23	−0.79	−21.05	36
	LP	19.14	13.33	23.35	10.02	191.78	35
	CP	19.49	12.57	22.41	9.84	191.78	33

Here, Greedy^{p}, approach greedy towards priority satisfaction; Greedy^{e}, approach greedy towards effort minimisation; Greedy^{q}, approach greedy towards quality gain
[a]The computed schedule was infeasible, $P \times (Q–E)$ = Priotity × (Quality – Effort)

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 to be handled by the manual approach. With the same goal

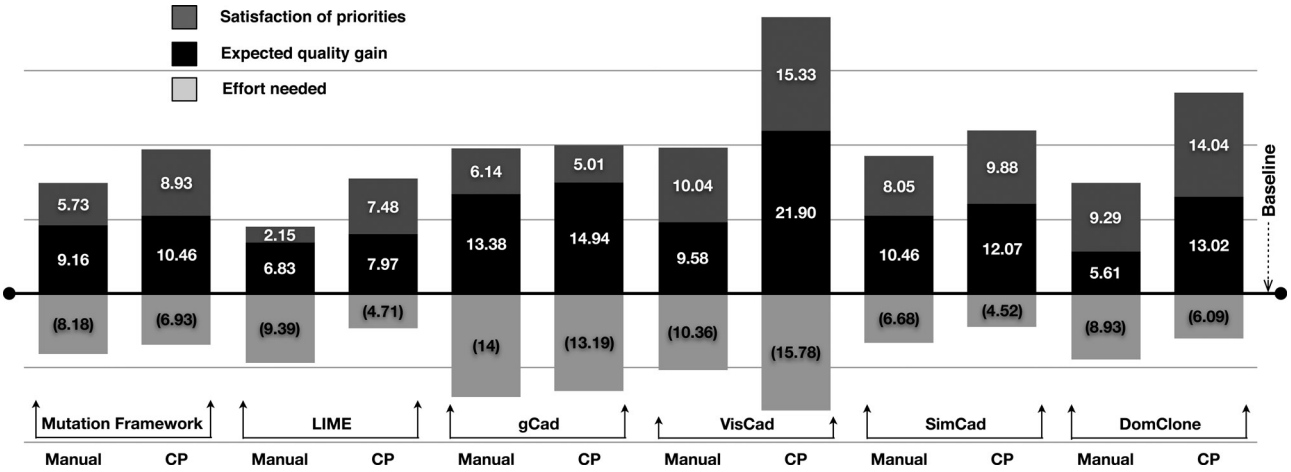

Fig. 11 *Automated CP against manual scheduling*

doi: 10.1049/iet-sen.2012.0058

(i.e. to schedule roughly 25% of all the refactorings), we executed our CP scheduler.

The purpose of the manual approach was to confirm that manually solving a CSOP can be difficult and the solution obtained from manual scheduling can be worse than an automated technique, such as CP. The objective was to compare the produced schedules, 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 optimisation dimensions namely the satisfaction of 'priorities' $\left(\sum_{r\in\mathcal{R}} \mathcal{X}_r \rho_r\right)$, required 'effort' $\left(\sum_{r\in\mathcal{R}} \mathcal{X}_r \underline{E}(g_r)\right)$, and expected gain in software 'quality' $\left(\sum_{r\in\mathcal{R}} \mathcal{X}_r \overline{Q}_r\right)$, obtained from our CP scheduling and manual scheduling, are presented in Fig. 11. For an effective schedule, the values for expected quality gain and satisfaction of priorities are expected to be high whereas 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 optimisation 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 (10) optimises along these three dimensions. From our observations during the study and the developers' feedback, as well as the results presented in Table 8 and Fig. 11, 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 Fig. 11, for four of the subject systems (i.e. Mutation Framework, LIME, SimCad and DomClone), the refactoring schedules generated by our CP approach are consistently lower whereas the expected quality gain and satisfaction of priorities are consistently higher compared with those for the manually computed schedules. For gCad, both the CP and manual approaches performed almost equally well. In the case of VisCad, the schedule computed by CP demands higher refactoring effort compared with 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 optimisation dimensions (i.e. effort, quality and priority). Again, the efforts are expected to be low whereas 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. (10)) compared with 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 visible 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 minimise the risks of refactorings, as seen in the right-most column of Table 9, the optimal schedule obtained from our scheduler always includes the least number of refactorings, compared with those from the GA and greedy schedulings.

As expected, our CP scheduler always outperformed manual scheduling for all the six subject systems (Fig. 11). The superiority in the optimality of the schedules (in terms of efforts, quality gain and priorities) obtained from our CP and LP schedulers, compared with manual scheduling, gradually increased as the sizes of the systems and the number of candidate refactorings increased. Our CP scheduler took no >7 s in computing any of the refactoring schedules presented in this paper, whereas, for manual

Table 9 Time and memory comparison of CP and LP scheduling

Subject systems	Scheduling approaches	Resource consumpsion	
		Time, s	Memory
mutation framework	LP	14.33	7.39 MB
	CP	0.14	5.9 Mb
*LIME	LP	185.69	6.45 MB
	CP	0.20	5.2 Mb
*SimCad	LP	54.09	12.40 MB
	CP	0.97	9.4 Mb
*gCad	LP	58.08	12.40 MB
	CP	0.94	9.4 Mb
*VisCad	LP	313.78	37.81 MB
	CP	6.02	27.6 Mb
*DomClone	LP	396.02	8.29 MB
	CP	0.80	6.7 Mb

Here, MB = Megabyte; Mb = Megabit.

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 >2 h. 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 2.75 h 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 9 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 optimisation problems with many constrains can easily become 'GA-hard' [38, 39], because crossover and mutation, the core operations of GA are based on random selections, which do not guarantee for constraint satisfaction or optimisation. Thus, traditional GA may work for optimisation problems with a 'few' constraints, but GA approaches do not seem to work well for CSOPs [39]. Above all, GA approaches are by nature time-consuming and memory intensive.

9.5.2 CP against LP: As can be seen in Table 9, 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 could 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 questions about the developers' background, 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 IQ_5 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 minimise their effects. Recall that the objective of our empirical study was 2-folds: first, to evaluate our refactoring effort model

Table 10 Developers feedback on the Likert scale questions

Questions	Choice of answers and number of developers' responses in brackets				
IQ_1	very easy (0)	easy (0)	somewhat difficult (0)	difficult (2)	very difficult (4)
IQ_2	provided no assistance (0)	somewhat useful (0)	quite useful (1)	very useful (3)	it was a necessity (2)

doi: 10.1049/iet-sen.2012.0058

and, second, to evaluate our CP scheduler. Hence, we organise 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 [40].

In our study, we relied on the developers' qualitative evaluations in the estimation of both refactoring efforts and effects. There is a possibility to question the individual developer's ability to correctly estimate those following our effort model. This work is based on our initial proof-of-concept proposal [41], 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, allowed the developers to estimate the refactoring efforts and effects with a higher probability of accuracy compared with 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. Owing to the unavailability of any baseline approach or benchmark data, it was not possible 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 manual) in terms of optimisation values along the three dimensions (i.e. effort, quality and priority satisfaction) as well as in terms of run-time and memory consumption. We manually investigated all the refactoring schedules obtained from our CP scheduler and confirmed correctness (i.e. feasiblity) 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 optimal schedule, because the code clone refactoring scheduling problem is NP-hard [9–11]. However, we made efforts to challenge the optimality of the produced solutions by attempting to further increase the objective values by means of pseudorandom 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' [42].

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 use does not affect the functionality of our refactoring scheduler. Rather, our observations and the developers' expressions towards the need for realisation 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 (9). The weighted sum of differences in (9) might have not been able to capture the full benefits of the quality model. These are also threats to the study. Although 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 with 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 produced infeasible solutions. Our choice and implementation of the mutation and crossover operators may seem to be responsible. To minimise 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 [38, 39], as discussed in Section 9.5.1.

9.6.3 External validity: External validity questions the generalisability 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 PhD 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 minimise the threats to construct validity. One of the participants of the study had 5 years of industry experience and another had >2 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 minimised by increasing the number and sizes of the subject systems, choosing both industrial and open-source software for study, and involving developers with diverse levels of expertise (i.e. beginner, intermediate and 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 [http://www.cs.usask.ca/faculty/croy/]. The data, the OPL implementation of our CP and LP scheduler, as well as the Java implementation of GA are also made available online [http://usask.ca/minhaz.zibran/pages/projects.html] for the interested parties. Therefore it should be possible to replicate the study.

10 Related work

Much research has been conducted towards effective identification and removal of different types of code smells from the source code. Our work is focused on scheduling of 'code clone' refactoring; we confine our discussion to those work that deal with scheduling of refactoring this particular code smell.

The work of Bouktif *et al*. [10], Lee *et al*. [11] and Liu *et al*. [9] closely relate to ours. Bouktif *et al*. [10] formulated the refactoring problem as a constrained 'Knapsack problem' and applied a GA to obtain an optimal solution. However, they ignored the constraints that might exist among the refactorings. Lee *et al*. [11] applied 'ordering messy GA (OmeGA)', whereas Liu *et al*. [9] 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 a CP approach, which we have shown to be better than theirs. Second, we took into account a wide category of refactoring constraints and dimensions of optimisations, some of which they ignored, as summarised in Table 11. Although Bouktif *et al*. [10] 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. Our effort model is applicable not only to procedural but also to OO source code, as it takes into account diverse categories of efforts covering the constructs of an OO system.

Table 11 Comparison of code clone refactoring schedulers

	Bouktif *et al*. [10]	Lee *et al*. [11]	Liu *et al*. [9]	Our scheduler
approach	GA	OmeGA	heuristic	CP
refactoring effort	√	–	–	√
quality gain	√	√	√	√
sequential dependency	–	√	√	√
mutual exclusion	–	√	√	√
mutual inclusion	–	–	–	√
priorities satisfaction	–	–	–	√

O'Keeffe and Ó Cinnéide [43] conducted an empirical comparison of 'simulated annealing', 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 techniques [12] 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 LP techniques in the scheduling of code clone refactorings. 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 GA-hard [38, 39], as discussed in Section 9.5.1.

A number of methodologies [15, 23, 24, 44, 45] and metric-based tools such as CCShaper [22] and Aries [21] have been proposed for semi-automated extraction of code clones as refactoring candidates. Several tools, such as Libra [46] and CnP [47], 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 OO 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 effort model for refactoring OO source code and our CP approach is a technique that no one else in the past reported to have applied in this context. Combining the strengths from both AI and OR, the CP approach has been shown to be effective in solving scheduling problems [12, 13]. Our CP scheduler computes the conflict-free schedule making optimal balance among the three optimisation

doi: 10.1049/iet-sen.2012.0058

dimensions: minimised refactoring effort, maximised 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, GA, LP 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 on the subject systems. In the future, we plan to carry out a more structured, large-scale user study where we will provide the developers with a tool implementation of our effort model. Then, we will compare the tool estimation with the actual efforts that the developers must put for performing those refactorings by hand. In the estimation of the effect of clone refactoring, we will use the QMOOD quality model with its full strength. We will also experiment with varying weight factors in our parameterised model and observe the impact of those variations on our approach. We will also compare our CP-based approach with different variations of GA (e.g. NSGA-II, Pareto-GA and OmeGA) and other evolutionary algorithms such as Artificial Bee Colony, Ant Colony Optimisation and Particle Swarm Optimisation. Our immediate future plan also includes the evaluation of our scheduler in a larger context involving both diversified open-source and industrial software systems written in different programming languages and, finally, the integration of a smart scheduler with the code clone management tool [7, 19] that we have been developing.

12 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.

13 References

1 Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: 'Refactoring: improving the design of existing code' (Addison Wesley Professional, 1999)
2 Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: 'Do code clones matter?'. Proc. 31st Int. Conf. Software Engineering (ICSE), Vancouver, BC, Canada, May 2009, pp. 485–495
3 Kapser, C., Godfrey, M.W.: 'Cloning considered harmful' considered harmful: patterns of cloning in software', *Empir. Softw. Eng.*, 2008, **13**, (6), pp. 645–692
4 Zibran, M., Saha, R., Asaduzzaman, M., Roy, C.: 'Analyzing and forecasting near-miss clones in evolving software: an empirical study'. Proc. 16th IEEE Int. Conf. Engineering of Complex Computer Systems (ICECCS), Las Vegas, Nevada, USA, April 2011, pp. 295–304
5 Rieger, M., Ducasse, S., Lanza, M.: 'Insights into system-wide code duplication'. Proc. 11th IEEE Working Conf. Reverse Engineering (WCRE), Delft, The Netherlands, November 2004, pp. 100–109
6 Fowler, M.: 'Refactoring catalog', http://refactoring.com/catalog/, accessed March 2012
7 Zibran, M., Roy, C.: 'Towards flexible code clone detection, management, and refactoring in IDE'. Proc. Fifth Int. Workshop of Software Clones (IWSC), Honolulu, Hawaii, USA, May 2011, pp. 75–76
8 Pérez, J., Crespo, Y., Hoffmann, B., Mens, T.: 'A case study to evaluate the suitability of graph transformation tools for program refactoring', *Int. J. Softw. Tools Technol. Transf.*, 2010, **12**, pp. 183–199
9 Liu, H., Li, G., Ma, Z., Shao, W.: 'Conflict-aware schedule of software refactorings', *IET Softw.*, 2008, **2**, (5), pp. 446–460
10 Bouktif, S., Antoniol, G., Neteler, M., Merlo, E.: 'A novel approach to optimize clone refactoring activity'. Proc. Eighth Annual Conf. Genetic and Evolutionary Computation (GECCO), Seattle, Washington, USA, July 2006, pp. 1885–1892
11 Lee, S., Bae, G., Chae, H.S., Bae, D., Kwon, Y.R.: 'Automated scheduling for clone-based refactoring using a competent GA', *Softw. Pract. Exper.*, 2010, **41**, (5), pp. 521–550
12 Barták, R.: 'Constraint programming: in pursuit of the holy grail'. Proc. Week of Doctoral Students (WDS), Part IV (invited lecture), Prague, Czech Republic, June 1999, pp. 555–564
13 Zibran, M.: 'A multi-phase approach to university course timetabling'. MSc thesis, Department of Mathematics and Computer Science, University of Lethbridge, Canada, September 2007, pp. 1–125
14 Zibran, M., Roy, C.: 'A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring'. Proc. 11th IEEE Int. Working Conf. Source Code Analysis and Manipulation (SCAM), Williamsburg, Virginia, USA, September 2011, pp. 105–114
15 Kodhai, E., Vijayakumar, V., Balabaskaran, G., Stalin, T., Kanagaraj, B.: 'Method level detection and removal of code clones in C and Java programs using refactoring', *Int. J. Comput. Commun. Inf. Syst. (IJCCIS)*, 2010, **2**, (1), pp. 93–95
16 Roy, C., Cordy, J.: 'A mutation/injection-based automatic framework for evaluating clone detection tools'. Proc. IEEE Int. Conf. Software Testing, Verification, and Validation Workshops (ICSTW), Denver, Colorado, USA, April 2009, pp. 157–166
17 Saha, R., Roy, C., Schneider, K.: 'An automatic framework for extracting and classifying near-miss clone genealogies'. Proc. 27th IEEE Int. Conf. Software Maintenance (ICSM), Williamsburg, Virginia, USA, September 2011, pp. 293–302
18 Uddin, S., Roy, C., Schneider, K., Hindle, A.: 'On the effectiveness of Simhash for detecting near-miss clones in large scale software systems'. Proc. 18th IEEE Working Conf. Reverse Engineering (WCRE), Lero, Limerick, Ireland, October 2011, pp. 13–22
19 Zibran, M., Roy, C.: 'IDE-based real-time focused search for near-miss clones'. Proc. 27th ACM Symp. Applied Computing (SAC), Riva del Garda, Trento, Italy, March 2012, pp. 1235–1242
20 Advani, D., Hassoun, Y., Counsell, S.: 'Understanding the complexity of refactoring in software systems: a tool-based approach', *Int. J. Gen. Syst.*, 2006, **35**, (3), pp. 329–346
21 Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: 'ARIES: refactoring support tool code clone'. Proc. Third Workshop on Software Quality (3-WoSQ), St. Louis, Missouri, USA, July 2005, pp. 1–4
22 Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: 'Refactoring support based on code clone analysis'. Product Focused Software Process Improvement (PROFES), Springer, Berlin, Heidelberg Press, 2004, *LNCS*, **3009**, pp. 220–233
23 Schulze, S., Kuhlemann, M.: 'Advanced analysis for code clone removal'. Proc. GI-Workshop on Software Reengineering (WSR), Bad-Honnef, Germany, May 2009, pp. 10–12
24 Yoshida, N., Higo, Y., Kamiya, T., Kusumoto, S., Inoue, K.: 'On refactoring support based on code clone dependency relation'. Proc. 11th IEEE Int. Software Metrics Symp. (METRICS), Como, Italy, September 2005, pp. 16–25
25 DeLine, R., Venolia, G., Rowan, K.: 'Software development with code maps', *ACM Commun.*, 2010, **53**, (8), pp. 48–54
26 Mens, T., Taentzer, G., Runge, O.: 'Analysing refactoring dependencies using graph transformation', *J. Softw. Syst. Model.*, 2007, **6**, (3), pp. 269–285
27 Sillito, J., Murphy, G., Volder, K.: 'Asking and answering questions during a programming change task', *IEEE Trans. Softw. Eng.*, 2008, **34**, (4), pp. 434–451
28 Bansiya, J., Davis, C.: 'A hierarchical model for object-oriented design quality assessment', *IEEE Trans. Softw. Eng.*, 2002, **28**, (1), pp. 4–17
29 Chidamber, S., Kemerer, C.: 'A metric suite for object-oriented design', *IEEE Trans. Softw. Eng.*, 1994, **25**, (5), pp. 476–493
30 Sahraoui, H., Godin, R., Miceli, T.: 'Can metrics help to bridge the gap between the improvement of OO design quality and its automation?' Proc. 16th IEEE Int. Conf. Software Maintenance (ICSM), San Jose, California, USA, October 2000, pp. 154–162
31 Simon, F., Steinbrucker, F., Lewerentz, C.: 'Metrics based refactoring'. Proc. Fifth European Conf. Software Maintenance and Reengineering (CSMR), Lisbon, Portugal, March 2001, pp. 30–38
32 Tahvildari, L., Kontogiannis, K.: 'A metric-based approach to enhance design quality through meta-pattern transformations'. Proc. Seventh

European Conf. Software Maintenance and Reengineering (CSMR), Benevento, Italy, March 2003, pp. 183–192
33 Saha, R., Asaduzzaman, M., Zibran, M., Roy, C., Schneider, K.: 'Evaluating code clone genealogies at release level: an empirical study'. Proc. 10th IEEE Int. Working Conf. Source Code Analysis and Manipulation (SCAM), Timisoara, Romania, September 2010, pp. 87–96
34 Asaduzzaman, M., Roy, C., Schneider, K.: 'VisCad: flexible code clone analysis support for NiCad'. Proc. Fifth Int. Workshop of Software Clones (IWSC), Waikiki, Honolulu, Hawaii, USA, May 2011, pp. 77–78
35 Cordy, J., Roy, C.: 'The NiCad clone detector'. Proc. 19th IEEE Int. Conf. Program Comprehension (ICPC), tool demo, Kingston, Ontario, Canada, June 2011, pp. 219–220
36 Winston, W.: 'Operations research applications and algorithms' (Duxbury Press, Belmont, CA, USA, 1994, 3rd edn.)
37 Obitko, M.: 'Introduction to genetic algorithms', A tutorial on genetic algorithm, http://www.obitko.com/tutorials/genetic-algorithms/recommendations.php, accessed March 2012
38 Davidor, Y.: 'Epistasis variance: a viewpoint on GA-hardness'. Proc. First Workshop on the Foundations of Genetic Algorithms (FOGA), Bloomington, Indiana, USA, July 1990, pp. 23–35
39 Eiben, A., Raue, P., Ruttkay, Z.: 'Solving constraint satisfaction problems using genetic algorithms'. Proc. First IEEE Conf. Evolutionary Computation, Orlando, Florida, USA, June 1994, pp. 542–547
40 Robillard, M., Coelho, W., Murphy, G.: 'How effective developers investigate source code: an exploratory study', *IEEE Trans. Softw. Eng.*, 2004, **30**, (12), pp. 889–903
41 Zibran, M., Roy, C.: 'Conflict-aware optimal scheduling of code clone refactoring: a constraint programming approach'. Proc. (Student Symp. of the 19th IEEE Int. Conf. Program Comprehension (ICPC), Kingston, Ontario, Canada, June 2011, pp. 266–269
42 Orso, A., Shi, N., Harrold, M.: 'Scaling regression testing to large software systems', *SIGSOFT Softw. Eng. Notes*, 2004, **29**, (6), pp. 241–251
43 O'Keeffe, M., Ó Cinnéide, M.: 'Search-based refactoring: an empirical study', *J. Softw. Maint. Evol. Res. Pract.*, 2008, **20**, (1), pp. 345–364
44 Ducasse, S., Rieger, M., Golomingi, G.: 'Tool support for refactoring duplicated OO code'. Proc. Object-Oriented Technology (ECOOP'99 Workshop Reader), number 1743 in *LNCS*, Springer-Verlag Press, 1999, pp. 2–6
45 Schulze, S., Kuhlemann, M., Rosenmüller, M.: 'Towards a refactoring guideline using code clone classification'. Proc. Second Workshop on Refactoring Tools (WRT), Nashville, Tennessee, USA, October 2008, pp. 6:1–6:4
46 Higo, Y., Ueda, Y., Kusumoto, S., Inoue, K.: 'Simultaneous modification support based on code clone analysis'. Proc. 14th Asia Pacific Software Engineering Conf. (APSEC), Nagoya, Aichi, Japan, December 2007, pp. 262–269
47 Hou, D., Jablonski, P., Jacob, F.: 'CnP: towards an environment for the proactive management of copy-and-paste programming'. Proc. 17th IEEE Int. Conf. Program Comprehension (ICPC), Vancouver, BC, Canada, May 2009, pp. 238–242

doi: 10.1049/iet-sen.2012.0058