Identification of Extract Method Refactoring Opportunities

Nikolaos Tsantalis, Alexander Chatzigeorgiou
Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece
nikos@java.uom.gr, achat@uom.gr

Abstract

Extract Method has been recognized as one of the most important refactorings, since it decomposes large methods and can be used in combination with other refactorings for fixing a variety of design problems. However, existing tools and methodologies support extraction of methods based on a set of statements selected by the user in the original method. The goal of the proposed methodology is to automatically identify Extract Method refactoring opportunities and present them as suggestions to the designer of an object-oriented system. The suggested refactorings adhere to three principles: the extracted code should contain the complete computation of a given variable declared in the original method, the behavior of the program should be preserved after the application of the refactoring, and the extracted code should not be excessively duplicated in the original method. The proposed approach is based on the union of static slices that result from the application of a block-based slicing technique. The soundness of the identified refactoring opportunities has been evaluated by an independent designer on the system that he developed.

1. Introduction

Extract Method is considered as one of the most important refactorings, since it is often employed as a remedy for several design flaws such as Duplicated Code, Feature Envy, Long Method and Message Chains [9]. Moreover, it is usually used in combination with other core refactorings such as Move Method and Extract Class which are applicable only when method extraction has preceded. Method extraction has a positive effect on maintenance, since it simplifies the code by breaking large methods into smaller ones and creates new methods which can be reused.

The vast majority of the papers found in the literature of method extraction are based on the concept of *program slicing*. According to Weiser [23], a slice consists of all the statements in a program that may affect the value of a variable x at a specific point of interest p. The pair (p, x) is referred to as *slicing criterion*. In general, slices are computed by finding sets of directly or indirectly relevant statements based on control and data dependencies. After the original definition by Weiser, several notions of slicing have been proposed. Concerning the employment of runtime information, *static* slicing uses only statically available information to compute slices, while *dynamic* slicing [17] uses as input the values of variables for a specific execution of a program in order to provide more accurate slices. Concerning flow direction, in *backward* slicing a slice contains all statements and control predicates that may affect a variable at a given point, while in *forward* slicing [2] a slice contains all statements and control predicates that may be affected by a variable at a given point. Concerning syntax preservation, *syntax-preserving* slicing simplifies a program only by deleting statements and predicates that do not affect a computation of interest, while *amorphous* slicing [11] employs a range of syntactic transformations in order to simplify the resulting code. Concerning slicing scope, *intraprocedural* slicing computes slices within a single procedure, while *interprocedural* slicing [14] generates slices that cross the boundaries of procedure calls. Program slicing has several applications in various software engineering domains such as debugging, program comprehension, testing, cohesion measurement, maintenance and reverse engineering [22, 3, 10].

Static slicing of object-oriented programs has drawn considerable research interest as noted in the survey of Mohapatra et al. [21]. Larsen and Harrold [19] extended the *System Dependence Graph* (SDG) proposed by Horwitz et al. [14] to represent object-oriented programs. Each class in a system is represented by a *Class Dependence Graph* (ClDG) that captures the control and data dependence relationships that can be determined about a class without knowledge of calling environments. Each method in a ClDG is represented by a *Procedure Dependence*

1534-5351/09 $25.00 © 2009 IEEE
DOI 10.1109/CSMR.2009.23

Graph (or *Program Dependence Graph* - PDG) which was initially introduced by Ferrante et al. [8]. The computation of static interprocedural slices is performed using an efficient two-pass graph reachability algorithm. Chen and Xu [4] proposed a new approach to represent dependences for object-oriented programs that differs from the previous SDG representations, in the sense that it does not connect the PDGs of all methods with each other in order to construct the SDG (i.e. each PDG is an independent graph). They redefined the program dependence graph of a method as a directed graph where dependence edges are enriched with tags. The tags have the form (x, y), where x and y are variables, and are used to distinguish the different definitions and dependences in a statement. Using the redefined PDG of a method, they solved intra-method slicing as a graph reachability problem with tags (i.e. their approach checks not only edges but also the tags on these edges).

A direct application of program slicing in the field of refactorings is *slice extraction*, which has been formally defined by Ettinger [7] as the extraction of the computation of a set of variables *V* from a program *S* as a reusable program entity, and the update of the original program *S* to reuse the extracted slice. Within the context of slice extraction the literature can be divided into two main categories according to Ettinger [7]. In the first category belong the methodologies that extract slices based on a set of selected statements which are indicated by the user (*arbitrary method extraction*). In the second category belong the methodologies that extract slices based on a variable of interest at a specific program point which is indicated by the user.

The proposed methodology aims at automatically identifying Extract Method refactoring opportunities. To this end, it employs and extends a block-based slicing technique [20] in order to suggest slice extraction refactorings which contain the complete computation of a given variable, are behavior-preserving and result in code that is not excessively duplicated in the original and extracted method. Moreover, it has been implemented as an Eclipse plug-in that presents the slice extraction suggestions to the designer and applies the selected Extract Method refactorings on source code. The identified Extract Method refactoring opportunities have been evaluated by an independent designer for the system that he developed concerning their soundness and usefulness.

The rest of the paper is organized as follows: Section 2 provides an overview of the related work. Section 3 briefly presents the block-based slicing technique proposed by Maruyama [20], while Section 4 describes and resolves some flaws found in his approach concerning behavior preservation. Our methodology for extracting the complete computation of a variable is presented in Section 5 and is evaluated in Section 6. Finally, we conclude in Section 7.

2. Related work on slice extraction

Lakhotia and Deprez [18] proposed a transformation, called Tuck, which can be used to restructure a program by breaking its large functions into smaller ones. The tuck transformation consists of three steps: Wedge, Split, and Fold. The wedge is a program slice that contains all the statements that influence a given set of *seed* statements. The split transformation splits the original function into two single-entry, single-exit (SESE) regions, one containing all the computations relevant to the set of seed statements and the other containing all the remaining computations. The transformation introduces new variables or renames variables and composes the two new regions such that the overall computation remains unchanged. Finally, the fold transformation creates a function for the SESE region corresponding to the seed statements and replaces the statements by a call to this function.

Komondoor and Horwitz [16] proposed a methodology that takes as input the control flow graph of a procedure and a set of statements to be extracted (*marked* statements) and applies semantics-preserving transformations to make the marked statements form a contiguous, well-structured block that is suitable for extraction. The applied transformations are the reordering of unmarked statements in order to make the marked statements contiguous, the duplication of predicates in both the extracted and original procedure, the promotion of unmarked statements to the marked ones, and the special handling of exiting jumps such as return, break and continue statements.

Harman et al. [12] introduced a variation of the algorithm proposed by Komondoor and Horwitz [16] which is based on amorphous procedure extraction. Amorphous extraction relaxes the syntactic constraints of the original program in order to enable the application of simplifying transformations. However, it retains the requirement that the extracted program and the original must be semantically equivalent. The goal of the proposed variation is to minimize the need for statement promotion (i.e. when a statement which was not originally marked for extraction must be extracted to preserve the semantics of the program) in order to make the extraction process more precise.

The three aforementioned methodologies concern arbitrary slice extraction for procedural programming

languages. The methodology that follows concerns variable-based slice extraction for object-oriented programming languages.

Maruyama [20] simplified an interprocedural slicing algorithm proposed by Larsen and Harrold [19] by making it intraprocedural and then introduced the concept of block-based region into the resulting algorithm. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. Maruyama employed a block-partitioning algorithm [1] in order to decompose the control flow graph of a method into basic blocks. A block-based region for a given basic block B_n is the set of statements that are in reachable basic blocks from B_n. The approach of Maruyama is able to extract more than one slices for a given slicing criterion by using the appropriate block-based regions, compared to classic static slicing algorithms that extract only a single slice for a given slicing criterion by using the whole source method as target region.

Jiang et al. [15] performed an empirical study on six open-source projects in order to evaluate the splitability of procedures. Concerning the frequency of splitable procedures, they concluded that the majority of procedures are not splitable, while those which are splitable can be split into two or three subprocedures. Furthermore, they studied the *overlap* distribution of splitable procedures. Overlap is a measure of code duplication between the resulting subprocedures. The higher the overlap, the more cohesive the original procedure is, and therefore, less likely to be splitable.

3. Brief presentation of block-based slicing

The approach of Maruyama takes as input a slicing criterion (n, u) which consists of statement n belonging to method m and variable u that is defined or used inside n. The control flow graph of method m is constructed in order to decompose it into basic blocks. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. A block-partitioning algorithm [1] marks as *leader* nodes the first node, the join nodes, and the nodes that immediately follow a branch node in the control flow graph of the method. For each leader node, its basic block consists of itself and all subsequent nodes up to the next leader or the last node in the control flow graph. Figure 1 illustrates the control flow graph (decomposed into basic blocks) for method `statement()` used in a well-established refactoring example [9].

```
1  public String statement() {
2    double totalAmount = 0;
3    int frequentRenterPoints = 0;
4    Enumeration rentals = _rentals.elements();
5    String result = "Rental Record for "
                + getName() + "\n";
6    while(rentals.hasMoreElements()) {
7      Rental each = rentals.nextElement();
8      double thisAmount = each.getCharge();
9      if(each.getMovie().getPriceCode()
                == Movie.NEW_RELEASE
                && each.getDaysRented() > 1)
10       frequentRenterPoints += 2;
       else
11       frequentRenterPoints++;
12     result += "\t"
       + each.getMovie().getTitle() + "\t"
       + String.valueOf(thisAmount) + "\n";
13     totalAmount += thisAmount;
     }
14   result += "Amount owed is "
     + String.valueOf(totalAmount) + "\n";
15   result += "You earned "
     + String.valueOf(frequentRenterPoints)
     + " frequent renter points";
16   return result;
   }
```

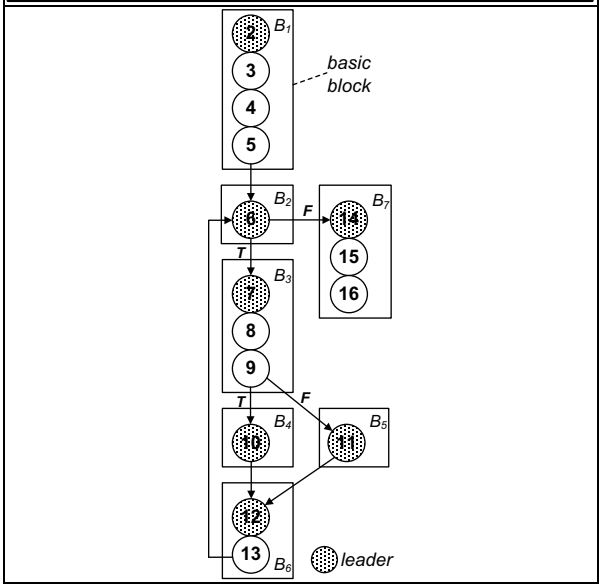

Figure 1: Original method `statement()` and the corresponding control flow graph.

As a next step, the program dependence graph of method m is constructed, containing control and data dependence edges between the statements of m. The set of boundary blocks *Blocks(n)* is computed for statement n of the slicing criterion. Considering that statement n belongs to basic block B, the set of boundary blocks is the intersection of the forward reachable blocks from B and the dominated blocks by node r, which is the node that directly dominates the leader node of B (a block is considered dominated by r if there exists a transitive control dependence from r to

this block). For each boundary block $B_n \in Blocks(n)$, a subgraph of the program dependence graph is constructed containing only the nodes belonging to the *block-based region* of B_n. The block-based region $R(B_n)$ for boundary block B_n is the set of nodes that are in reachable basic blocks from B_n. The block-based slice $S_B(n, u, B_n)$ for slicing criterion (n, u) and boundary block B_n is the set of statements that may affect the computation of variable u at statement n (backward slice), extracted from the program dependence subgraph corresponding to $R(B_n)$. The set of remaining statements is $U_B = N(m) \setminus S_B$, where $N(m)$ is the set of all statements inside method m. Along with the block-based slice, the set of variables that should be passed as parameters to the extracted method and the set of indispensable statements I_B (i.e. statements that should not be removed from the original method in order to preserve its behavior) are calculated. The set of statements that should eventually be removed from method m after slice extraction is calculated as $R_B = N(m) \setminus \{U_B \cup I_B\}$. Finally, the invocation of the extracted method is placed exactly before the leader node of block B_n in the original method.

The block-based slice extraction approach proposed by Maruyama is able to produce more than one slice extraction suggestions for a given slicing criterion according to the regions that result from the boundary blocks. As a result, the designer has more options concerning the scope of the code that will be extracted.

4. Improvements concerning behavior preservation

Maruyama claimed that his slice extraction technique is behavior-preserving based on the proof by Horwitz et al. [13] showing that if the program dependence graphs of two programs are isomorphic then the programs are strongly *equivalent*. Within the context of slice extraction, equivalent means that every variable in the original method has the same value with the corresponding variable in either the extracted or the remaining method (i.e. the original method after slice extraction) after the execution of the original and the remaining method. However, it is rather questionable whether the Equivalence Theorem [13] can be directly applied (without being extended) to slice extraction.

4.1. Handling of method invocations changing the state of objects

In object-oriented programming languages the invocation of a method can change the state of the object being referenced. This change in object state may in turn affect the execution of the code that follows in a method. Obviously, the duplication of such method invocations in both the remaining and the extracted method may not preserve the behavior of the code. To support our argument, two slice extraction examples taken from [20] will be demonstrated. Both examples concern the extraction of code from the method shown in Figure 1 using the same slicing criterion (10, `frequentRenterPoints`) but different block-based regions. The set of boundary block for statement 10 is $Blocks(10) = \{B_1, B_2, B_3, B_4\}$, and as a result, four block-based slices can be derived from this slicing criterion. Figure 2 shows the remaining and the extracted method when block-based slice $S_B(10,$ `frequentRenterPoints`$, B_2)$ is used.

```
1   public String statement() {
2     double totalAmount = 0;
3     int frequentRenterPoints = 0;
4     Enumeration rentals = _rentals.elements();
5     String result = "Rental Record for "
                + getName() + "\n";
      frequentRenterPoints =
        getFrequentRenterPoints(
        frequentRenterPoints, rentals);
6     while(rentals.hasMoreElements()) {
7       Rental each = rentals.nextElement();
8       double thisAmount = each.getCharge();
12      result += "\t"
        + each.getMovie().getTitle() + "\t"
        + String.valueOf(thisAmount) + "\n";
13      totalAmount += thisAmount;
      }
14    result += "Amount owed is "
      + String.valueOf(totalAmount) + "\n";
15    result += "You earned "
      + String.valueOf(frequentRenterPoints)
      + " frequent renter points";
16    return result;
    }

    private int getFrequentRenterPoints(
              int frequentRenterPoints,
              Enumeration rentals) {
6     while(rentals.hasMoreElements()) {
7       Rental each = rentals.nextElement();
9       if(each.getMovie().getPriceCode()
                == Movie.NEW_RELEASE
                && each.getDaysRented() > 1)
10        frequentRenterPoints += 2;
        else
11        frequentRenterPoints++;
      }
      return frequentRenterPoints;
    }
```

Figure 2: Slice extraction using block-based slice $S_B(10,$ `frequentRenterPoints`$, B_2)$

As it can be observed from Figure 2, after the execution of the extracted method `getFrequentRenterPoints()` the Enumeration `rentals` will not have any more elements to provide,

since the while loop inside the extracted method has already iterated over all the elements of the enumeration. As a result, the while loop that follows inside method `statement()` will not be executed, since the invocation of method `hasMoreElements()` will return false. Obviously, in this case the behavior of the code is not preserved after slice extraction. The reason causing the change of behavior is that the invocation of method `nextElement()` in statement 7 affects the internal state of object `rentals` and at the same time statement 7 is duplicated in both the remaining and the extracted method. An alternative slice extraction using block-based slice S_B(10, *frequentRenterPoints*, B_1) is shown in Figure 3.

```
1   public String statement() {
      int frequentRenterPoints =
        getFrequentRenterPoints();
2     double totalAmount = 0;
4     Enumeration rentals = _rentals.elements();
5     String result = "Rental Record for "
                + getName() + "\n";
6     while(rentals.hasMoreElements()) {
7       Rental each = rentals.nextElement();
8       double thisAmount = each.getCharge();
12      result += "\t"
        + each.getMovie().getTitle() + "\t"
        + String.valueOf(thisAmount) + "\n";
13      totalAmount += thisAmount;
      }
14    result += "Amount owed is "
      + String.valueOf(totalAmount) + "\n";
15    result += "You earned "
      + String.valueOf(frequentRenterPoints)
      + " frequent renter points";
16    return result;
    }

    private int getFrequentRenterPoints() {
3     int frequentRenterPoints = 0;
4     Enumeration rentals = _rentals.elements();
6     while(rentals.hasMoreElements()) {
7       Rental each = rentals.nextElement();
9       if(each.getMovie().getPriceCode()
                == Movie.NEW_RELEASE
                && each.getDaysRented() > 1)
10        frequentRenterPoints += 2;
        else
11        frequentRenterPoints++;
      }
      return frequentRenterPoints;
    }
```

Figure 3: Slice extraction using block-based slice S_B(10, *frequentRenterPoints*, B_1)

As it can be observed from Figure 3, the slice extraction based on basic block B_1, where slicing covers the whole source method, preserves the behavior of the code in contrast with the slice extraction based on basic block B_2. The reason causing the preservation of behavior is that apart from statement 7, the declaration of variable `rentals` (statement 4) is also duplicated in both the remaining and the extracted method. As a result, the while loops in the remaining and the extracted method iterate over two different `Enumeration` references derived from the same `Vector` object (field `_rentals`).

To overcome this problem in behavior preservation, the duplicated statements (i.e. the statements belonging to the intersection of slice and indispensable statements, $S_B \cap I_B$) are examined whether they contain method invocations that their duplication in the remaining and the extracted method might change the behavior of the code. In general, the invoked methods in duplicated statements should not modify the attributes of the class to which they belong, since such modifications change the state of the objects. On the other hand, the invocation of methods that simply access the attributes of the class to which they belong or do not access any attributes at all is not possible to change the behavior of the code. It should be emphasized that the examination of invoked methods is recursive. This means that if a method being examined contains other method invocations the corresponding methods should be also examined.

Consequently, in the case where a duplicated statement contains a method invocation that modifies the state of an object, the corresponding block-based slice is rejected. An exception applies to method invocations which are invoked through a reference whose declaration is also included in the duplicated statements (as happens in the slice extraction example of Figure 3).

4.2. Handling of anti-dependencies

Another case that may cause change in behavior is when a statement of the slice anti-depends on a statement that remains in the original method. An anti-dependency exists from statement A to statement B (or statement B anti-depends on A), when statement A uses the value of a variable that is later modified at statement B. Figure 4 shows an example of code containing anti-dependencies and the corresponding control flow graph decomposed into basic blocks.

Let us consider that slicing criterion (8, x) is used for the code of Figure 4. The set of boundary blocks for statement 8 is *Blocks*(8) = {B_1, B_3}, and as a result, two block-based slices can be derived from this slicing criterion. The first block-based slice is S_B(8, x, B_1) = {7, 8} and the second is S_B(8, x, B_3) = {7, 8}. Although, the two block-based slices consist of the same statements, their extraction is completely different as shown in Figure 5.

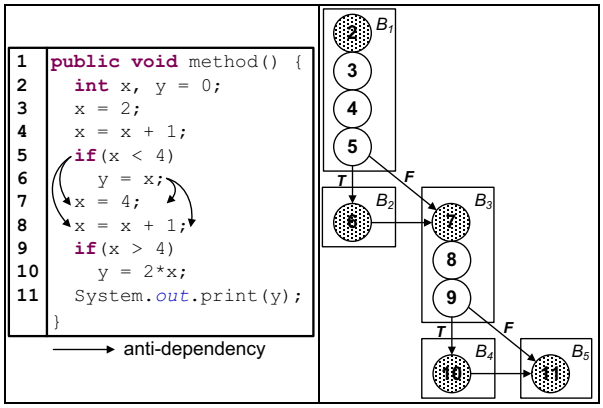

Figure 4: A method containing anti-dependencies and the corresponding control flow graph.

```
1  public void method() {
2    int x = x(),
         y = 0;
3    x = 2;
4    x = x + 1;
5    if(x < 4)
6      y = x;
9    if(x > 4)
10     y = 2*x;
11   System.out.print(y);
   }

   private int x() {
7    int x = 4;
8    x = x + 1;
     return x;
   }
```

a) $S_B(8, x, B_1)$

```
1  public void method() {
2    int x, y = 0;
3    x = 2;
4    x = x + 1;
5    if(x < 4)
6      y = x;
     x = x();
9    if(x > 4)
10     y = 2*x;
11   System.out.print(y);
   }

   private int x() {
7    int x = 4;
8    x = x + 1;
     return x;
   }
```

b) $S_B(8, x, B_3)$

Figure 5: Extraction using different block-based slices

As it can be observed from Figure 5a the behavior of the code is not preserved when block-based slice $S_B(8, x, B_1)$ is used, since the final value of variable y is equal to 3 (in the original method the final value of variable y is equal to 10). On the other hand, as it can be observed from Figure 5b the behavior of the code is preserved when block-based slice $S_B(8, x, B_3)$ is used, since the final value of variable y is equal to 10 as happens in the original method. The reason causing this change in behavior is that block-based region $R(B_1)$, where $S_B(8, x, B_1)$ is calculated, contains anti-dependencies from statements 5 and 6 to the slice statements (statements 7 and 8), while block-based region $R(B_3)$, where $S_B(8, x, B_3)$ is calculated, does not contain any anti-dependencies to the slice statements. In general, the invocation of the extracted method should not be placed before statements that the slice statements anti-depend on. Consequently, in the case where a block-based region contains an anti-dependency from a statement in the remaining method to a non-duplicated slice statement, the corresponding block-based slice is rejected.

5. Extraction of the complete computation of a variable

An important principle of the proposed methodology is that the slice extraction refactorings should cover the complete computation of the variable corresponding to the slicing criterion. In other words, the slices which are computed for a specific variable should contain all the assignment statements that modify the value of this variable in the original method. The application of a backward static slicing algorithm on a slicing criterion does not guarantee that the computed slice will contain all the assignment statements corresponding to the variable of the slicing criterion, since there may not exist a backward path of control and data flow dependencies passing from all the assignments of the variable. As a solution to the problem of obtaining the complete computation for a given variable, we propose an algorithm employing the union of the static slices that result when each assignment statement corresponding to the variable of interest is used as slicing criterion. According to De Lucia et al. [5] the approaches relying on slicing algorithms that do preserve a subset of the direct data and control dependence relations of the original program (such as the algorithm employed by Maruyama) produce unions of static slices which are valid slices.

The proposed algorithm takes as input a method declaration m and returns a set of slice extraction refactoring suggestions for each variable declared inside method m, covering the complete computation of the corresponding variable. The algorithm consists of the following steps:

1. Identify the set of variables V which are declared inside method m.
2. For each variable $v \in V$ identify the set of statements C which contain an assignment of variable v. These statements along with variable v form a set of slicing criteria (c, v), where $c \in C$.
3. For each statement $c \in C$ compute the set of boundary blocks $Blocks(c)$.
4. Calculate the common boundary blocks for the statements in set C as $Blocks(C) = \bigcap_{c \in C} Blocks(c)$.
5. For each slicing criterion (c, v), where $c \in C$, and boundary block $B_n \in Blocks(C)$ compute the set of slice statements $S_B(c, v, B_n)$ and the set of removable statements $R_B(c, v, B_n)$.
6. For each $B_n \in Blocks(C)$ the union of slice statements is $US_B(C, v, B_n) = \bigcup_{c \in C} S_B(c, v, B_n)$ and

the union of removable statements is $UR_B(C, v, B_n) = \bigcup_{c \in C} R_B(c, v, B_n)$.

The unions of slice and removable statements for a given block form a candidate slice extraction suggestion. The final set of candidate slice extraction suggestions are examined against a set of rules in order to assure that the code extracted after the application of a suggestion preserves behavior and is functionally useful.

Concerning behavior preservation the rules are:

A1. The union of slice statements US_B should not contain `break`, `continue`, or `return` statements. These statements constitute unstructured control flow and their extraction will change the behavior of the remaining method.

A2. The statements which are duplicated in both the remaining and the extracted method should not contain method invocations that modify the state of objects (as explained in Section 4.1).

A3. The statements that belong to the union of slice statements US_B and are not duplicated in both the remaining and the extracted method should not have incoming anti-dependencies from statements that do not belong to the union of removable statements UR_B and are inside the region of the corresponding block $B_n \in Blocks(C)$ (as explained in Section 4.2).

Concerning the functional usefulness of the extracted code the rules are:

B1. The variable which is returned by the original method should be excluded from slice extraction. If the complete computation of the variable being returned by the original method was extracted, then the extracted method would essentially have the functionality of the original method.

B2. The number of statements in the union of slice statements US_B should be greater than the number of statements used as slicing criteria ($|US_B| > |C|$). In the case where the number of statements in US_B is equal to the number of statements used as slicing criteria (this is actually the minimum number of statements that can be extracted), the extracted code would be algorithmically trivial, since no additional statements are required to compute the value of variable v.

B3. The statements which are duplicated in both the remaining and the extracted method should not contain all the statements used as slicing criteria. If all the statements used as slicing criteria were duplicated, then the computation of variable v would exist in both the remaining and the extracted method making the extraction redundant.

The application of the proposed algorithm will be demonstrated on a well-established refactoring teaching example [6]. Figure 6 illustrates method `printDocument()` and its control flow graph decomposed into basic blocks.

```
1  public void printDocument(Packet document) {
2    String author = "Unknown";
3    String title = "Untitled";
4    int startPos = 0, endPos = 0;
5    if (document.message_.startsWith("!PS")) {
6      startPos = document.message_.indexOf("author:");
7      if (startPos >= 0) {
8        endPos = document.message_.indexOf(
         ".", startPos + 7);
9        if (endPos < 0)
10         endPos = document.message_.length();
11       author = document.message_.substring(
         startPos + 7, endPos);
       }
12     startPos = document.message_.indexOf("title:");
13     if (startPos >= 0) {
14       endPos = document.message_.indexOf(
         ".", startPos + 6);
15       if (endPos < 0)
16         endPos = document.message_.length();
17       title = document.message_.substring(
         startPos + 6, endPos);
       }
     } else {
18     title = "ASCII DOCUMENT";
19     if (document.message_.length() >= 16)
20       author = document.message_.substring(8, 16);
     }
21   System.out.println(author);
22   System.out.println(title);
}
```

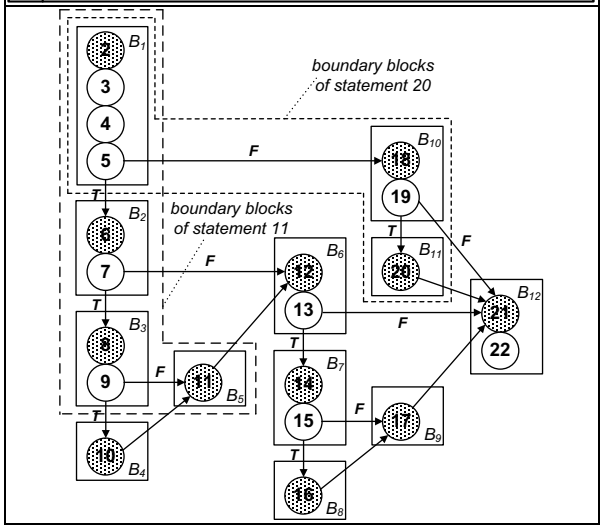

Figure 6: Method `printDocument()` and the corresponding control flow graph.

Assume that the computation of variable `author` is intended to be extracted as a separate method. The algorithm is applied as follows:

1. The assignment statements of variable `author` are statements 11 and 20 (underlined in the code of Figure 6).

2. The sets of boundary blocks for statements 11 and 20 are $Blocks(11) = \{B_1, B_2, B_3, B_5\}$ and $Blocks(20) = \{B_1, B_{10}, B_{11}\}$, respectively (as shown in the control flow graph of Figure 6).
3. The intersection of the two sets of boundary blocks is $Blocks(\{11, 20\}) = \{B_1\}$ and as a result only the union of static slices for basic block B_1 can be computed.
4. The block-based static slices for statements 11 and 20 are $S_B(11, \texttt{author}, B_1) = \{2, 4, 5, 6, 7, 8, 9, 10, 11\}$ and $S_B(20, \texttt{author}, B_1) = \{2, 5, 19, 20\}$, respectively. The sets of statements that should be removed after each slice is extracted are $R_B(11, \texttt{author}, B_1) = \{2, 6, 7, 8, 9, 10, 11\}$ and $R_B(20, \texttt{author}, B_1) = \{2, 19, 20\}$.
5. The union of the static slices is $US_B(\{11, 20\}, \texttt{author}, B_1) = \{2, 4, 5, 6, 7, 8, 9, 10, 11, 19, 20\}$ and the union of removable statements is $UR_B(\{11, 20\}, \texttt{author}, B_1) = \{2, 6, 7, 8, 9, 10, 11, 19, 20\}$.

The extracted method regarding the computation of variable `author` is shown in Figure 7.

```
1   public void printDocument(Packet document) {
      String author = getAuthor(document);
3     String title = "Untitled";
4     int startPos = 0, endPos = 0;
5     if (document.message_.startsWith("!PS")) {
12      startPos = document.message_.indexOf("title:");
13      if (startPos >= 0) {
14        endPos = document.message_.indexOf(
          ".", startPos + 6);
15        if (endPos < 0)
16          endPos = document.message_.length();
17        title = document.message_.substring(
          startPos + 6, endPos);
        }
      } else {
18      title = "ASCII DOCUMENT";
      }
21    System.out.println(author);
22    System.out.println(title);
    }
    private String getAuthor(Packet document) {
2     String author = "Unknown";
4     int startPos = 0, endPos = 0;
5     if (document.message_.startsWith("!PS")) {
6       startPos = document.message_.indexOf("author:");
7       if (startPos >= 0) {
8         endPos = document.message_.indexOf(
          ".", startPos + 7);
9         if (endPos < 0)
10          endPos = document.message_.length();
11        author = document.message_.substring(
          startPos + 7, endPos);
        }
      } else {
19      if (document.message_.length() >= 16)
20        author = document.message_.substring(8, 16);
      }
      return author;
    }
```

Figure 7: Extraction of the computation of variable `author` as a separate method.

6. Evaluation

To evaluate the proposed methodology an independent designer assessed the soundness of the slice extraction refactoring opportunities that were identified for the system that he developed. The examined software project is an emulator of a telephone exchange, where the user can insert definition commands (e.g. define a connection and assign a subscriber number to it) and emulate calls between subscribers. It has been implemented in Java and consists of 61 classes, 144 methods with body (excluding abstract methods) and 4100 lines of code. The reasons for selecting the specific project are:

- It is a rather mature project which has been constantly evolving for more than 3 years. Moreover, it has been subject to continuous adaptive maintenance due to constant requirement changes.
- It has been designed and developed by a single person. Therefore, the independent designer had complete and deep knowledge of the system's architecture.
- The independent designer is an experienced telecommunications software designer with knowledge of object-oriented design principles that enabled him to assess the slice extraction refactoring opportunities and provide valuable feedback.

The identified slice extraction refactoring opportunities for the examined project along with the opinion of the independent designer are shown in Table 1. The first column contains the method in which the corresponding refactoring opportunity is identified and the second column contains the variable whose computation is suggested to be extracted. The results are sorted in ascending order according to the ratio of duplicated statements to extracted statements (as shown in the third column of Table 1), which expresses the percentage of slice statements that will be duplicated in both the remaining and the extracted method if the corresponding refactoring is applied. This ratio ranges over the interval [0, 1] and takes a value equal to zero when none of the extracted statements is duplicated (best case), and a value equal to one when all the extracted statements are duplicated in the original method (worst case). In the case where two or more slice extraction refactoring suggestions correspond to a number of duplicated statements which is equal to zero, then they are sorted in descending order according to the number of extracted statements. The importance of code duplication in slice extraction

has been also emphasized in the empirical study of Jiang et al. [15], where the definition of procedure splitability depends directly on the degree of code duplication between the resulting subprocedures.

Table 1: Identified slice extraction refactoring opportunities for the examined project.

original method	variable name	duplicated/ extracted	designer's opinion
validateParameters	parName	0/2	D_1
getAccessRef	ai	1/3	D_1
getSNBforAccessID	ai	1/3	D_1
parseCommandLine	parameterName	2/4	A
parseCommandLine	parameterValue	2/4	A
parseCommandLine	commandName	2/4	A
execute	id	8/11	A
execute	snb	8/11	A
parseCommandLine	commandName	6/8	D_2

A: agreement
D_1: disagreement due to small number of extracted statements
D_2: another block-based slice for the same variable is more preferred

As it can be observed from Table 1, method `parseCommandLine()` offers the largest number of slice extraction refactoring opportunities, since it is a rather complex method consisting of 30 statements (the average number of statements inside the methods of the examined project is approximately 5). This is in agreement with the empirical results by Jiang et al. [15] which have shown a strong correlation between procedure size and splitability. More specifically, for variable `commandName` two slice extraction refactoring opportunities are offered based on two different basic blocks. The independent designer preferred the slice which had a method segment as block-based region over the slice which had the whole method as target region, since the former slice extraction has a lower ratio of duplicated to extracted statements (2/4) compared to the latter slice extraction (6/8). This case clearly exhibits the advantage of block-based slicing over classic techniques that use the whole method as slicing region, since the latter would not be able to capture the slice which was eventually chosen by the independent designer.

The small number of slice extraction refactoring opportunities that were identified can be attributed to two reasons. First of all, the examined project proved to be a well-designed system that primarily consists of short methods without complex computations that do not offer decomposition opportunities. The second reason is that the applied rules reduced significantly the number of identified refactoring opportunities. Table 2 contains the number of candidate slice extraction suggestions that were rejected by each rule.

Table 2: Rejected candidate slice extraction suggestions by each rule for the examined project.

rule	#instances
A1	0
A2	0
A3	18
B1	10
B2	6
B3	4
accepted	9
total	47

The total number of candidate slice extraction suggestions before the examination of the rules is 47. Moreover, 8 out of 47 candidate slice extraction suggestions resulted from the union of two slices, while the rest 39 resulted from a single slice. As a result, a block-based slicing approach that does not take into account behavior preservation and code duplication issues and does not employ the union of slices in order to extract the complete computation of variables, would result in 55 suggestions (39*1 + 8*2) for the examined project. On the other hand, a designer that is assisted by the proposed approach has to inspect significantly less refactoring suggestions and does not have to thoroughly examine the code resulting after the application of a refactoring concerning behavior preservation issues.

7. Conclusions

The proposed methodology aims at automatically identifying Extract Method refactoring opportunities that lead to the decomposition of complex methods. The key contributions of the proposed approach are that it employs the union of static slices in order to extract the complete computation of a given variable declared inside a method and it proposes a set of rules that preserve the behavior of the code after slice extraction and prevent the excessive duplication of code in the original and extracted method.

Evaluation has been performed by an independent designer who assessed the soundness and usefulness of the slice extraction refactoring opportunities that were identified for the system that he developed. The results of the evaluation indicated that the methodology is able to identify slice extraction refactorings which decompose complex methods, create new methods with useful functionality and preserve the behavior of the code. However, there is a clear need to extend the evaluation on more systems from different domains in order to further improve the effectiveness of the methodology.

8. References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] J.-F. Bergeretti, and B.A. Carré, "Information-flow and data-flow analysis of while-programs," ACM Transactions on Programming Languages and Systems, vol. 7, no. 1, pp. 37-61, 1985.

[3] D. Binkley, and K. B. Gallagher, "Program Slicing," Advances in Computers, vol.43, 1996.

[4] Z. Chen, and B. Xu, "Slicing object-oriented Java programs," ACM SIGPLAN Notices, vol. 36, no. 4, pp. 33-40, April 2001.

[5] A. De Lucia, M. Harman, R. Hierons, and J. Krinke, "Unions of Slices are not Slices," 7th European Conference on Software Maintenance and Reengineering (CSMR'03), pp. 363-367, March 2003.

[6] S. Demeyer, F. Van Rysselberghe, T. Girba, J. Ratzinger, R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall and M. El-Ramly, “The LAN-simulation: A Refactoring Teaching Example”, 8th International Workshop on Principles of Software Evolution (IWPSE'05), pp. 123-134, September 5-6, 2005.

[7] R. Ettinger, "Refactoring via Program Slicing and Sliding," Ph.D. dissertation, University of Oxford, United Kingdom, 2007.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and Its Use in Optimization," ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, July 1987.

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

[10] M. Harman, and R. M. Hierons, "An Overview of Program Slicing," Software Focus, vol. 2, no. 3, pp. 85-92, 2001.

[11] M. Harman, D. Binkley, and S. Danicic, "Amorphous Program Slicing," Journal of Systems and Software, vol. 68, no. 1, pp. 45-64, 2003.

[12] M. Harman, D. Binkley, R. Singh, and R. M. Hierons, "Amorphous Procedure Extraction," 4th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'04), pp. 85-94, 2004.

[13] S. Horwitz, J. Prins, and T. Reps, "On the Adequacy of Program Dependence Graphs for Representing Programs," 15th Annual ACM Symposium on Principles of Programming Languages (POPL'88), pp. 146-157, 1988.

[14] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," ACM Transactions on Programming Languages and Systems, vol. 12, no. 1, pp. 26-60, 1990.

[15] T. Jiang, M. Harman, Y. Hassoun, "Analysis of Procedure Splitability," 15th Working Conference on Reverse Engineering (WCRE'08), pp. 247-256, 2008.

[16] R. Komondoor, and S. Horwitz, "Effective, Automatic Procedure Extraction," 11th IEEE International Workshop on Program Comprehension (IWPC'03), 2003.

[17] B. Korel, and J. Laski, "Dynamic program slicing," Information Processing Letters, vol. 29, no. 3, pp. 155-163, 1988.

[18] A. Lakhotia, and J.-C. Deprez, "Restructuring Programs by Tucking Statements into Functions," Information and Software Technology, vol. 40, no. 11-12, pp. 677-690, 1998.

[19] L. Larsen, and M. J. Harrold, "Slicing object-oriented software," International Conference on Software Engineering (ICSE'96), pp. 495-505, 1996.

[20] K. Maruyama, "Automated Method-Extraction Refactoring by Using Block-Based Slicing," Symposium on Software Reusability (SSR'01), pp.31-40, 2001.

[21] D.P. Mohapatra, R. Mall, and R. Kumar, "An Overview of Slicing Techniques for Object-Oriented Programs," Informatica, vol. 30, no. 2, pp. 253-277, 2006.

[22] F. Tip, "A Survey of Program Slicing Techniques," Journal of Programming Languages, vol. 3, no. 3, pp. 121-189, 1995.

[23] M. Weiser, "Program Slicing," IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357, 1984.