The Journal of Systems and Software 84 (2011) 1757–1782

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Identification of extract method refactoring opportunities for the decomposition of methods

Nikolaos Tsantalis*, Alexander Chatzigeorgiou

Department of Applied Informatics, University of Macedonia, 156 Egnatia Str., 54006 Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 27 July 2010
Received in revised form 26 January 2011
Accepted 9 May 2011
Available online 14 May 2011

Keywords:
Extract Method refactoring
Program slicing
Module decomposition

ABSTRACT

The extraction of a code fragment into a separate method is one of the most widely performed refactoring activities, since it allows the decomposition of large and complex methods and can be used in combination with other code transformations for fixing a variety of design problems. Despite the significance of Extract Method refactoring towards code quality improvement, there is limited support for the identification of code fragments with distinct functionality that could be extracted into new methods. The goal of our approach is to automatically identify Extract Method refactoring opportunities which are related with the complete computation of a given variable (*complete computation slice*) and the statements affecting the state of a given object (*object state slice*). Moreover, a set of rules regarding the preservation of existing dependences is proposed that exclude refactoring opportunities corresponding to slices whose extraction could possibly cause a change in program behavior. The proposed approach has been evaluated regarding its ability to capture slices of code implementing a distinct functionality, its ability to resolve existing design flaws, its impact on the cohesion of the decomposed and extracted methods, and its ability to preserve program behavior. Moreover, precision and recall have been computed employing the refactoring opportunities found by independent evaluators in software that they developed as a golden set.

1. Introduction

According to several empirical studies procedures/modules with large size (Banker et al., 1993), high complexity (Gill and Kemerer, 1991), and low cohesion (Meyers and Binkley, 2007) require significantly more time and effort for comprehension, debugging, testing and maintenance. A solution to this kind of design problems is given by Extract Method refactoring (Fowler et al., 1999) which simplifies the code by breaking large methods into smaller ones and creates new methods which can be reused. However, existing IDEs and research approaches have focused on automating the extraction of statements which are indicated by the developer without providing support for the automatic identification of code fragments that could benefit from decomposition. Abadi et al. (2008) stressed the inadequate support that is offered by modern IDEs for various cases requiring the application of Extract Method refactoring.

Extract Method refactoring is employed for fixing several design flaws such as *Duplicated Code* (Fowler et al., 1999) where the same code structure existing in more than one place is extracted into a single method, *Long Method* (Fowler et al., 1999) where parts of a large and complex method having a distinct functionality are extracted into new methods, and *Feature Envy* (Fowler et al., 1999) where a part of a method using several data of another class is initially extracted into a new method and then moved to the class that it envies. Furthermore, in the study by Binkley et al. (2006), Extract Method refactoring transformations have been widely employed to enable the migration of object-oriented programs to the aspect-oriented paradigm. The wide use of Extract Method refactoring has also been evident in several empirical studies (Murphy et al., 2006; Murphy-Hill et al., 2009) that analyzed the refactoring operations performed by programmers using the Eclipse IDE.

Our approach covers the identification of refactoring opportunities which (a) extract the complete computation of a given variable (referred to as *complete computation slice*) into a new method, (b) extract the statements affecting the state of a given object (referred to as *object state slice*) into a new method. A complete computation slice is a slice that contains all the assignment statements of a given variable within the body of a method, while an object state slice is a slice that contains all the statements modifying the state of a given object (by method invocations through references pointing to this specific object) within the body of a method. It should be emphasized that object state slice has no relevance with the concept of *object slice* introduced by Liang and Harrold (1998) which is defined as "*the statements in the methods of a particular object that might affect the slicing criterion*". Fig. 1 illustrates two code examples for a complete computation slice and an object state slice, respectively. Our approach builds upon well established techniques such as

* Corresponding author. Tel.: +30 2310 891886; fax: +30 2310 891290.
E-mail address: nikos@java.uom.gr (N. Tsantalis).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.05.016

```java
public void translate(double dx, double dy){
  if (getParent() == null) {
    dy = TOP_GAPY - getBounds().getY();
  }
  else {
    double y = getBounds().getY() + dy;
    y = Math.max(y,
        getParent().getBounds().getMinY()
        - topHeight / 2);
    y = Math.min(y,
        getParent().getBounds().getMaxY()
        - topHeight / 2);
    dy = y - getBounds().getY();
  }
  super.translate(dx, dy);
}
```

(a)

```java
public void draw(Graphics2D g2) {
  super.draw(g2);
  Color oldColor = g2.getColor();
  g2.setColor(color);

  Shape path = getShape();
  g2.fill(path);
  g2.setColor(oldColor);
  g2.draw(path);

  Rectangle2D bounds = getBounds();
  GeneralPath fold = new GeneralPath();
  fold.moveTo((float)(bounds.getMaxX()
    - FOLD_X), (float)bounds.getY());
  fold.lineTo((float)bounds.getMaxX()
    - FOLD_X, (float)bounds.getY() + FOLD_X);
  fold.lineTo((float)bounds.getMaxX(),
    (float)(bounds.getY() + FOLD_Y));
  fold.closePath();
  oldColor = g2.getColor();
  g2.setColor(g2.getBackground());
  g2.fill(fold);
  g2.setColor(oldColor);
  g2.draw(fold);

  text.draw(g2, getBounds());
}
```

(b)

Fig. 1. (a) *complete computation slice* for variable `dy`. (b) *object state slice* for object reference `fold`.

program dependence graphs for the representation of dependences in methods and as a vehicle to perform slicing and block-based slicing to determine alternative regions to which a slice may expand. The contribution of the approach is the identification of behavior-preserving and meaningful refactoring opportunities in object-oriented code without human intervention, by combining a variety of techniques which improve the quality of the resulting slices.

The evaluation of the approach provides evidence that both complete computation and object state slices are able to capture code fragments implementing a distinct and independent functionality compared to the rest of the original method and thus lead to extracted methods with useful functionality.

In a previous work (Tsantalis and Chatzigeorgiou, 2009), we presented an approach for the identification of complete computation slices in object-oriented systems along with a set of rules for the preservation of program behavior after slice extraction.

The novelty of the current approach lies at the following points:

- It introduces the concept of object state slice as a means to capture code that modifies the state of a given object and proposes an algorithm for the identification of such slices.
- It proposes a set of additional rules that exclude refactoring opportunities corresponding to slices whose extraction could possibly cause a change in program behavior.
- It adopts a variety of program analysis techniques (such as alias analysis, polymorphic method call analysis) in order to improve the precision of the resulting slices.
- It supports the handling of branching statements (i.e. break, continue) within loops and switch statements, throw statements and try/catch blocks.
- The evaluation has been enriched and consists of two main parts:
  - ◦ Qualitative and quantitative evaluation on an open-source project, consisting of the following three sub-analyses:
    - (a) an independent assessment of the identified refactoring opportunities on a well-known open-source project regarding their soundness and usefulness,
    - (b) an investigation of the impact of the suggested refactorings on slice-based cohesion metrics and
    - (c) an investigation of the impact of the suggested refactorings on the external behavior of the program.
  - ◦ Evaluation of precision and recall against the findings of independent evaluators on projects developed by themselves.

The rest of the paper is organized as follows: Section 2 provides an overview of the related work. Section 3 presents a thorough analysis of the methodology for the identification of slice extraction refactoring opportunities. Section 4 presents the tool implementing the proposed methodology. The evaluation of the proposed approach is presented in Section 5. Finally, we conclude in Section 6.

2. Related work

The vast majority of the papers found in the literature of function extraction are based on the concept of program slicing. According to Weiser (1984), 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 dependences. 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 (Korel and Laski, 1988) 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 (Bergeretti and Carré, 1985) 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 (Harman et al., 2003) 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 (Horwitz et al., 1990) 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 (Tip, 1995; Binkley and Gallagher, 1996; Harman and Hierons, 2001).

A direct application of program slicing in the field of refactoring is *slice extraction*, which has been formally defined by Ettinger (2007) 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 (2007). 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 first approach for decomposing a procedure was proposed by Gallagher and Lyle (1991). They introduce the concept of *decomposition slice* as a slice that captures all computation on a given variable. The decomposition slice for a variable v is the union of the slices that result by using as seed statements in slicing criteria the statements that output variable v along with the last statement of the procedure. As output statement is considered a statement that prints or returns the value of a given variable. They also defined dependence relations between the resulting decomposition slices of a procedure. Two decomposition slices $S(\upsilon)$ and $S(w)$ are considered as *independent* if their intersection is empty ($S(\upsilon) \cap S(w) = \varnothing$). Decomposition slice $S(\upsilon)$ is considered as *strongly dependent* on $S(w)$ if $S(\upsilon)$ is a proper subset of $S(w)$, $S(\upsilon) \subset S(w)$. The dependence relationships between the decomposition slices are used to construct the *lattice* of decomposition slices, which can be considered as a directed graph where nodes represent the decomposition slices of a procedure and edges represent the strongly dependent relationships between them. Fig. 2b shows the lattice of decomposition slices for the code in Fig. 2a. The decomposition slices for the output variables of the code in Fig. 2a are the following: $S(c) = \{12, 13, 24\}$, $S(nc) = \{11, 12, 13, 14, 24\}$, $S(nl) = \{9, 12, 13, 15, 17, 18, 24\}$, $S(inword) = \{8, 12, 13, 15, 16, 20, 21, 24\}$, and $S(nw) = \{8, 10, 12, 13, 15, 16, 20, 21, 22, 24\}$.

As it can be observed in Fig. 2b, $S(c)$ is strongly dependent on all other decomposition slices and $S(inword)$ is strongly dependent on $S(nw)$. Tonella (2003) introduced the *concept lattice* of decomposition slices as an extension to decomposition slice graph (Gallagher and Lyle, 1991) in order to represent *weak inferences* (i.e. shared statements which are not decomposition slices) between decomposition slices. For example, statement 15 in Fig. 2a is shared by decomposition slices $S(nl)$, $S(inword)$ and $S(nw)$ but does not form a decomposition slice. Fig. 2c illustrates the concept lattice of decomposition slices for the code in Fig. 2a.

By examining Fig. 2c, it can be observed that by traversing the concept lattice from the bottom to the decomposition slice of a vari-

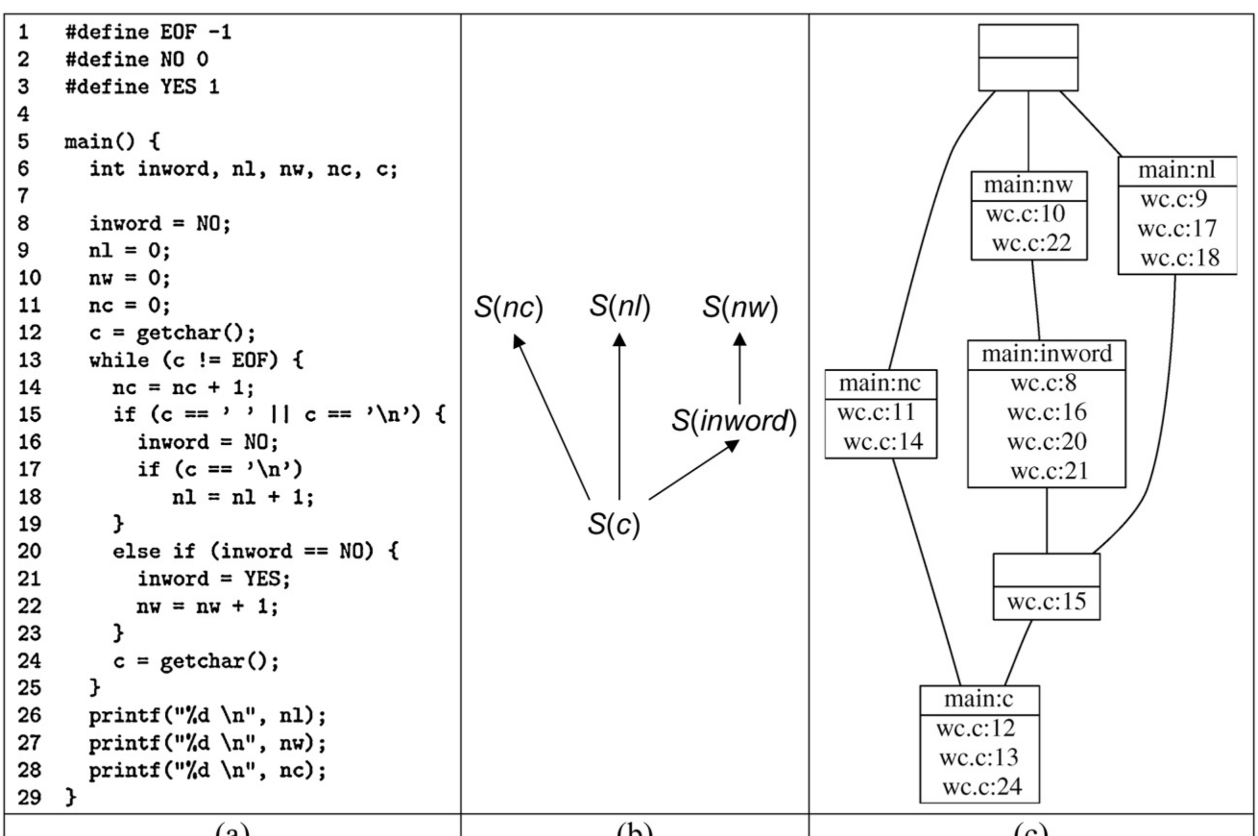


Fig. 2. (a) The code of a word counting program. (b) The lattice of decomposition slices according to Gallagher and Lyle (1991). (c) The concept lattice of decomposition slices according to Tonella (2003).

able v whose computation is intended to be extracted, the slice of variable v is the union of the statements in the traversed decomposition slices, while the statements that will be duplicated if the slice is extracted is the union of the statements in the traversed decomposition slices excluding the decomposition slice of variable v. The lattice of decomposition slices is used by Gallagher and Lyle (1991) in order to construct the *complement* of a decomposition slice (i.e. the statements that should remain in the original procedure after the extraction of the decomposition slice: the complement consists of statements that do not belong in the decomposition slice along with statements of the decomposition slice that have to be duplicated in the original procedure). Our approach in a similar manner computes the *indispensable* statements corresponding to a slice. The indispensable statements are statements that although belong to the slice, should not be removed from the original method to preserve the behavior of the remaining statements (i.e. statements not belonging to the slice). The indispensable statements along with the remaining statements correspond to the complement of a slice, as defined by Gallagher and Lyle (1991).

The major difference of our approach with decomposition slicing is related with the selection of the seed statements which are required to derive the computation of a given variable. The decomposition slicing technique uses as seed statements the statements that output the variable under consideration along with the last statement of the procedure. As a result, the selected seed statements may include code not dealing with the computation of the variable under consideration (e.g. a print or a return statement does not contribute to the computation of the variable of interest). Moreover, they may lead to the inclusion of additional irrelevant statements in the resulting slices due to the use of multiple variables within the seed statements (a statement that prints or returns an expression involving multiple variables, e.g. `return x+y;`). On the other hand, our approach uses as seed statements the statements where the variable under consideration is defined, leading to slices that contain the pure computation of the variable under consideration (i.e. it does not include the computation of variables that are completely irrelevant to the variable of interest).

Cimitile et al. (1996) proposed a specification driven slicing process for identifying reusable functions based on the precondition and postcondition of a given function. Initially, a symbolic execution technique is used to recover the preconditions for the execution of each statement and predicate existing within the body of the function. Eventually, the statements whose preconditions are equivalent to the pre- and post-conditions of the function serve as candidate entry and exit points of the computed slice (i.e. a pair of statements restricting the expansion of the slice within their boundaries). This approach requires to associate the data of the function's specification with the program variables, to define the set of output variables of the function and to provide invariant assertions that cannot be automatically derived in order to operate.

Lanubile and Visaggio (1997) introduced the notion of *transform* slicing as a method for extracting reusable functions. A transform slice includes the statements which contribute directly or indirectly to the transformation of a set of input variables into a set of output variables. The computation of a transform slice is similar to the computation of a static backward slice with the difference that it expands until the statements that define values for the input variables are included in the slice. Transform slicing uses output statements as seeds for the slicing criteria, or the last program statement if it is not possible to find an output statement in the proper place. This approach requires domain knowledge regarding conceptually simple tasks which are performed in the system (functional abstractions) along with their input and output data, so that the user can choose the right slice among candidates resulting from transform slicing.

Kang and Bieman (1998) proposed the *input–output dependence graph* (IODG) as a means to model and visualize the dependency relationships between inputs and outputs of a module. Based on the IODG representation of a module they defined the design-level cohesion (DLC) measure which provides an objective criterion for evaluating and comparing alternative design structures. Moreover, the DLC measure can be used as a criterion to determine whether or not a given module should be redesigned or restructured. Based on the IODG representation and the DLC measure they defined eight basic restructuring operations (i.e. module decomposition and composition operations) and described a process for applying the restructuring operations to improve design of system modules. The restructuring process of this approach requires specifying expected marginal DLC levels of the examined modules, decomposing the IODG of the poorly designed modules in appropriate partitions exhibiting higher DLC level, and locating unnecessarily decomposed modules based on the IODG visualization and coupling, size, and/or reuse measures.

Lakhotia and Deprez (1998) 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 the statements in the smallest single-entry, single-exit (SESE) region including all the seed statements. The split transformation splits the original function into two 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 so 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. A major limitation of the approach is that the tuck transformation requires as external input a set of seed statements, and a foldable subgraph (i.e. a subgraph where there is no edge from its exit node to any node of the subgraph) containing the seed statements. Furthermore, the evaluation performed by Komondoor and Horwitz (2003) has shown that the performance of the approach was poor on a dataset of "difficult" cases because it promotes statements in a non-intelligent manner (i.e. copies/moves unnecessary code to the extracted function) and does not handle exiting jumps.

Komondoor and Horwitz (2000) proposed an algorithm for reordering a given set of control flow graph nodes so that they can be extracted into a procedure while preserving semantics by taking as input a set of nodes chosen for extraction. Their approach is based on a polygraph that represents ordering constraints imposed by data flow, def-order, anti, and output dependences. The acyclic graphs defined by the polygraph are examined whether they form extractable pieces of code. This approach does not allow the duplication of any predicate node and does not handle exiting jumps.

Komondoor and Horwitz (2003) proposed an algorithm 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. This approach does not allow the duplication of assignment statements and loop predicates leading to missed extraction opportunities in favor of low code duplication.

Harman et al. (2004) introduced a variation of the algorithm proposed by Komondoor and Horwitz (2003) 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) and predicate duplication in order to make the extraction process more precise.

Jiang et al. (2008) 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. They concluded that the splitability of a procedure depends on the inter-dependency between its subprocedures. The higher the inter-dependency of subprocedures, the more statements they share with each other, and splitting generates a larger amount of duplicated code. Finally, the empirical results have shown a strong correlation between procedure size and splitability in the case of 2-way splitable procedures.

The aforementioned methods concern only procedural programming languages and thus they do not take into account important issues regarding object-oriented programming languages, such as: (a) the fact that in contrast to primitive variables, some variables can be references to objects and therefore it is possible to change their state (by modifying their field values) apart from their value, which in turn may affect the correctness of the resulting slices, and (b) the fact that beyond slices containing the computation of a primitive variable, there exist slices containing the "computation" of an object (i.e. the statements that affect the state of an object).

Maruyama (2001) simplified an interprocedural slicing algorithm proposed by Larsen and Harrold (1996) 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 in order to decompose the control flow graph of a method into basic blocks and form several block-based regions used for restricting the expansion of a slice within their boundaries. In this way it is possible to extract more than one slice 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 entire source method as region. Although the approach of Maruyama was the first to cover slice extraction in object-oriented programming languages, it suffers from several limitations. It does not handle behavior preservation issues that can be raised from duplication of statements. It does not guarantee that the complete computation of the variable indicated by the user will be extracted as a separate method. Finally, it does not support extraction opportunities which are related with objects but only with variables of primitive type.

All of the aforementioned approaches require external input in terms of seed statements, input/output variables, or seed variables in order to operate. Although this feature makes them more general and flexible, it restricts their degree of automation due to their dependence on human intervention and expertise. Our approach automatically determines the required parameters for the extraction of a slice, since its goal is to identify and suggest all feasible and behavior preserving refactoring opportunities being present within a given method.

3. Methodology

The proposed methodology handles two main categories of Extract Method refactoring opportunities. The first category refers to variables (having primitive data types or being object references) whose value is modified by assignment statements throughout the body of the original method. The second category refers to object references (which are local variables or fields of the class containing the original method) pointing to objects whose state is affected by method invocations throughout the body of the original method. It should be noted that the state of an object reference is affected by method invocations that modify the value of at least one of its attributes. In the first case, the goal is to extract the complete computation of a given variable (*complete computation slice*), while in the second case, the goal is to extract all the statements modifying the state of a given object (*object state slice*) within the scope of the original method. The aforementioned goals ensure at a certain degree that the extracted code will exhibit useful functionality. To achieve these goals our approach employs the union of static slices by different means according to the specific needs of each category. According to De Lucia et al. (2003) the unions of static slices which rely on slicing algorithms that do preserve a subset of the direct data and control dependence relations of the original program are valid slices.

3.1. Construction of the program dependence graph

Our approach employs the *program dependence graph* (PDG) in order to represent the methods under examination. The program dependence graph was initially introduced by Ferrante et al. (1987) in order to represent control and data flow dependences between the operations of a procedure. The nodes of a PDG represent the statements of the corresponding procedure. Each node has a set of *defined* variables which consists of the variables whose value is modified by an assignment, and a set of *used* variables which consists of the variables whose value is used at the corresponding statement. A control dependence edge from node p to node q denotes that the execution of statement q depends on the control conditions of statement p. The sets of defined and used variables are employed to compute data dependences between the statements throughout the procedure control flow. A data dependence edge from node p to node q due to variable x denotes that statement p defines variable x, statement q uses variable x and there exists a control flow path from statement p to q without an intervening definition of x.

Later on, Horwitz et al. (1990) introduced the *System Dependence Graph* (SDG) in order to represent procedure calls between PDGs and face the problem of interprocedural slicing (i.e. slicing that crosses the boundaries of procedure calls). A procedure call is represented using a *call-site* node, while the information transfer is represented using four different kinds of *parameter* nodes. The PDGs are connected using three kinds of edges, namely *call*, *parameter-in* and *parameter-out* edges. Larsen and Harrold (1996) extended the *System Dependence Graph* (SDG) proposed by Horwitz et al. (1990) to represent object-oriented programs. They introduced the *Class Dependence Graph* (ClDG) to represent the methods and instance variables belonging to a class. Additionally, they proposed ways to represent inherited methods, class instantiations and polymorphic method calls. Liang and Harrold (1998) improved the aforementioned approach by providing a way to distinguish data members for different objects instantiated from the same class.

Since our approach aims at extracting intraprocedural slices (i.e. slices that extend within the boundaries of a method) as new sepa-

```java
public class Customer {
  private String _name;
  private Vector _rentals = new Vector();

  public Customer (String name) {
    _name = name;
  }
  public void addRental(Rental arg) {
    _rentals.addElement(arg);
  }
1 public static void main(String args[]) {
2   Customer customer =
      new Customer("customer");
3   Movie movie =
      new Movie("title",Movie.NEW_RELEASE);
4   Rental rental = new Rental(movie, 3);
5   customer.addRental(rental);
  }
}
```

```java
public class Vector<E> extends AbstractList<E> {
  protected Object[] elementData;
  protected int elementCount;
  protected int capacityIncrement;

  public synchronized void addElement(E obj) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = obj;
  }
  private void ensureCapacityHelper(int minCapacity){
    int oldCapacity = elementData.length;
    if (minCapacity > oldCapacity) {
      Object[] oldData = elementData;
      int newCapacity = (capacityIncrement > 0)
        ? (oldCapacity + capacityIncrement)
        : (oldCapacity * 2);
      if (newCapacity < minCapacity) {
        newCapacity = minCapacity;
      }
      elementData =
        Arrays.copyOf(elementData, newCapacity);
    }
  }
}
```

Fig. 3. Code example to demonstrate the handling of method invocations.

rate methods, we have adopted the PDG representation which does not include any method call representation elements. However, the information regarding the state of the objects being referenced inside the body of a method is crucial for the formation of precise and correct slices, as well as the preservation of program behavior after code extraction. The state of an object can be modified or accessed by invoked methods which modify or access the fields of this object inside their body. These methods can be invoked directly by using the object reference as invoker, or indirectly by passing the object reference as parameter to another method which in turn uses this object reference as invoker.

Let us assume that statement *s* inside the body of method *m* invokes a method through object reference *r* or passes object reference *r* as parameter to a method. A partial call graph is recursively generated starting from method *m* that includes only the method calls which are associated with object reference *r* (i.e. methods which are actually invoked through the original reference *r* or the original reference *r* is passed as parameter to them). While the partial call graph is constructed, the fields which are modified or used inside the body of each visited method are added to the sets of defined and used variables of statement *s*, respectively. These fields are represented as composite variables (i.e. variables consisting of more than one parts), where the last part is the name of the corresponding field and the initial part is the actual reference through which the field was modified or accessed.

In the code example of Fig. 3, statement 5 of method `main` invokes method `addRental` through object reference `customer` and passes as parameter to the invoked method object reference `rental`. The partial call graph corresponding to this method invocation is shown in Fig. 4. At each method node in the call graph the sets of defined and used variables are shown, where the formal parameters have been replaced with the actual parameters (e.g. in method `addElement` of class `Vector`, formal parameter `obj` has been replaced with actual parameter `rental`) and `this` reference has been replaced with the actual invoker reference (e.g. in method `addRental` of class `Customer`, `this` reference has been replaced with the actual invoker reference `customer`). The sets of defined and used variables for statement 5 are derived from the union of the defined and used variable sets, respectively, for each method in the call graph. For example, the set of defined variables for statement 5 is {`customer._rentals.modCount`, `customer._rentals.elementCount`, `customer._rentals.elementData`} and is derived by the union of the sets of defined variables for node `Vector::addElement` and `Vector::ensureCapacityHelper` in the call graph shown in Fig. 4.

The computation of data dependences in the PDG of method *m* takes also into account the composite variables which are related with the state of object references existing in the body of *m*. These additional data dependences allow the formation of more precise and correct slices and at the same time enable the extraction of code that affects the state of a given object reference.

Our approach adopts a variety of code analysis techniques in order to further increase the precision and correctness of the resulting slices.

a *Alias analysis* (Ohata and Inoue, 2006): An *alias relationship* exists between two references when they refer to the same object in memory during program execution. The set of references in which each element pair satisfies an alias relationship is called an *alias set*. Alias analysis is a method for extracting alias sets by static code analysis. Alias analysis techniques are mainly divided into two categories, namely *flow insensitive* where the execution order of statements is not taken into account and *flow sensitive* where the execution order of statements is taken into account. Flow sensitive techniques follow the control flow of a program in order to determine alias relationships and as a result they can extract more accurate alias relations compared to flow insensitive approaches. Landi et al. (1993) have introduced the concept of *reaching alias sets* in order to compute flow sensitive alias relationships. A reaching alias set for a given statement is a collection of alias sets which apply just before the execution of this statement. For example, in the code of Fig. 5 the reaching alias set for both statements 5 and 6 is ⟨`a`, `b`⟩, since after the execution of statement 4 references `a` and `b` point to the same object in memory. Our approach handles the existence of a reaching alias set *RASet* for statement *s* in the following way:

For each composite variable in the sets of defined and used variables of statement *s* whose first part is a reference *r* belong-

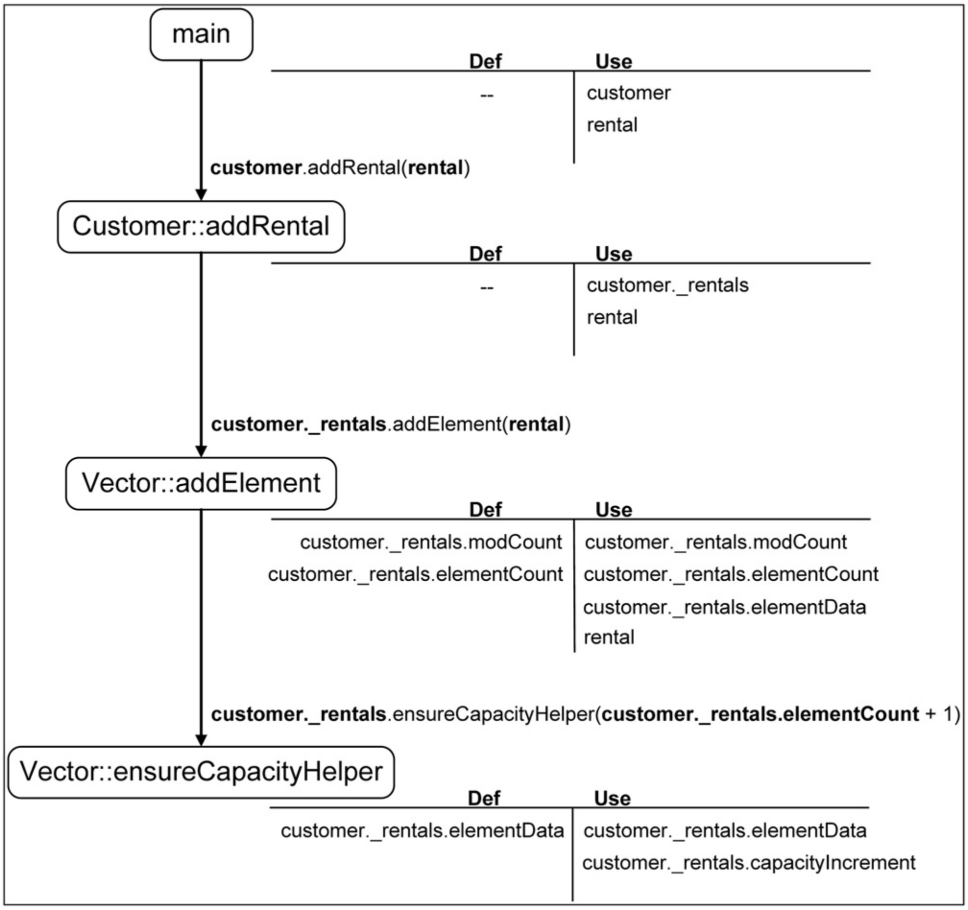


Fig. 4. Call graph for statement 5 of method `main` in Fig. 3, along with the sets of defined and used variables for each visited method. (The actual references being used as invokers or being passed as arguments are highlighted with bold fonts.)

ing to an alias set *A* of *RASet*, an additional number of composite variables is added (to the set of defined or used variables, respectively) which is equal to the number of references belonging to alias set *A* (excluding *r*) by replacing the first part of the composite variable with each one of the aliases of reference *r*.

In the example of Fig. 5, the additional composite variables that were added in the sets of defined and used variables are highlighted in rectangles. In this way, it is ensured that in the case of an alias relationship all statements affecting the state of the same object in memory will be extracted together regardless of the actual references through which the methods changing the object's state are invoked.

```
public class Buffer {
  private String s = "";

  public void append(String s) {
    this.s += s;
  }

1 public void method() {
2   Buffer a = new Buffer();
3   Buffer b;
4   b = a;                Def        Use
5   a.append("a");   a.s, b.s   a.s, b.s, a
6   b.append("b");   b.s, a.s   b.s, a.s, b
  }
}
```

Fig. 5. Code example containing an alias relationship between references `a` and `b`. (The composite variables that were added in the sets of defined and used variables due to the existence of alias set ⟨`a`, `b`⟩ are highlighted in rectangles.).

b *Polymorphic method call analysis* (Larsen and Harrold, 1996; Liang and Harrold, 1998): A polymorphic method call occurs when an abstract method is invoked through a reference of abstract type. Usually, the actual subclass type of the reference can be determined only at runtime. When the type of the caller reference cannot be statically determined, all concrete implementations of the abstract method are visited in the respective call graph. In this way, it is ensured that the state information associated with the caller reference covers all possible subclass types that the reference may obtain at runtime.

c *Handling of branching statements directly in the PDG* (Ball and Horwitz, 1993; Kumar and Horwitz, 2002): Unstructured control flow is achieved in Java by three kinds of branching statements. The `break` statement terminates the innermost loop and transfers the control flow to the statement following the innermost loop. The `continue` statement skips the current iteration of the innermost loop and transfers the control flow to the evaluation expression that controls the innermost loop. Finally, the `return` statement exits from the current method and transfers the control flow to the point where the method was invoked. In general, the problem caused by branching statements is that they cannot be included in slices, thus affecting slice correctness. The reason behind the non-inclusion of branching statements in slices is that they do not form control or data dependences with other statements in the traditional PDG of a method.

Ball and Horwitz (1993) proposed the *augmented* program dependence graph (APDG) as a means to handle properly the branching statements being present in a PDG. As a first step, the augmented control flow graph (ACFG) is constructed to represent the branching statements as pseudo-predicates. A pseudo-predicate node has two outgoing edges where the one (labeled as true) goes to the target of the jump and the other one (labeled as false) goes to the statement that would follow the branching statement if no branching occurred. As a second step, the APDG is constructed based on the ACFG by adding to the branching statements appropriate outgoing control dependences. The target nodes of these outgoing control dependences are the statements that follow the branching statement within the body of the innermost loop (and are directly control dependent on the innermost loop) and the innermost loop itself. The handling of branching statements as pseudo-predicates in the ACFG affects the way that block-based regions are formed in our approach, since block-partitioning depends on branching nodes (i.e. nodes having two or more outgoing flow edges) as explained in Section 3.2. As a result, our approach adds the required control dependences directly on the PDG without constructing the intermediate ACFG.

Regarding the special case of `break` statements within the body of a `switch` statement, Kumar and Horwitz (2002) proposed the pseudo-predicate PDG (PPDG) which is also constructed based on the ACFG. In this approach, the `switch case` statements are handled as pseudo-predicates in the ACFG where the control flow labeled as true goes to the statement that follows the `switch case` and the control flow labeled as false goes to the `default` case of the `switch` statement (or the first statement following the `switch` if no `default` case is present). In the PPDG, a `switch case` statement has outgoing control dependences to the statements that follow it (and are directly control dependent on the `switch` statement) before the next `break` statement. Again, our approach adds the required control dependences directly on the PDG without constructing the intermediate ACFG.

d *Handling of try/catch blocks and throw statements directly in the PDG* (Allen and Horwitz, 2003): Try/catch blocks are used in Java as a means to handle exceptions caused at runtime. The `try` block contains code that could throw an exception, while the `catch` blocks contain code that is directly executed when an exception is thrown in the body of the `try` block. Each `catch` block is responsible for handling a specific exception type. A try/catch block may also have a `finally` block (apart from `catch` blocks) which always executes when the `try` block exits. Allen and Horwitz (2003) extended the System Dependence Graph (SDG) to support slicing programs with exceptions by treating `try` and `throw` statements as pseudo-predicate nodes in the CFG. A `try` node (in the CFG) has an outgoing edge (labeled as true) to the first statement inside the try/catch block and an outgoing edge (labeled as false) to the first statement that follows the last catch block. In the construction of the PDG, a `try` node has outgoing control dependences to the statements inside the try/catch block that may throw an exception (such statements can be either `throw` statements or statements containing method invocations whose declaration throws an exception).

Throw statements are special statements which are used for creating and throwing exception objects. Exception types are divided into *checked* exceptions which must be explicitly handled by a `catch` block or propagated up the call stack of methods (java.lang.Exception subclasses), and *unchecked* exceptions which do not have this requirement (java.lang.RuntimeException subclasses). Similarly to branching statements, `throw` statements do not form control or data dependences with other statements in the traditional PDG of a method. Allen and Horwitz (2003) also treat `throw` statements and statements containing method invocations that throw an exception as pseudo-predicate nodes in the CFG. A node throwing an exception (in the CFG) has an outgoing edge (labeled as true) to the `catch` block that handles the thrown exception and an outgoing edge (labeled as false) to the statement that would follow the statement causing the exception if no exception occurred. In the construction of the PDG, a node throwing an exception has outgoing control dependences to the statements that follow it within the body of the `try` block (if the node throwing an exception is placed within a `try` block) or the statements that follow it within the body of the method (if the node throwing an exception is not placed within a `try` block and the method has a `throws` clause for the corresponding exception to its declaration).

The handling of `try` and `throw` statements as pseudo-predicates in the CFG affects the way that block-based regions are formed in our approach, since block-partitioning depends on branching nodes. As a result, our approach adds the required control dependences directly on the PDG.

3.2. Block-based slicing

Traditional intraprocedural slicing algorithms use the entire method body as a region where the slice may expand starting from the statement of the slicing criterion. However, within the context of slice extraction, where the goal is to extract the resulting slice as a new separate method, the extraction of a slice that expands throughout the entire method body is not always feasible. Maruyama introduced the concept of block-based slicing (Maruyama, 2001) as a means for producing more than one slice for a given slicing criterion. This is achieved by constructing block-based regions within the body of a method, which can be used to restrict the expansion of a slice within their boundaries. In our approach, block-based slicing helps to determine regions of the original method where slices starting from statements that belong to different blocks and concern the computation of the same variable can be extracted together as a union.

The block-based regions of method m can be determined employing the following steps.

3.2.1. Decomposition of control flow graph into basic blocks

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 (Aho et al., 1986) marks as *leader* nodes the first node, the join nodes (i.e. the nodes which have two or more incoming flow edges), and the nodes that immediately follow a branch node (i.e. a node which has two or more outgoing flow edges) 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. Fig. 6 illustrates the control flow graph (decomposed into basic blocks) for method `statement()` used in a well-established refactoring example (Fowler et al., 1999).

3.2.2. Determination of reachable blocks

Maruyama defined as *reachable blocks* for basic block B, $Reach(B)$, the set of blocks that can be reached from B on the control flow graph without traversing loopback edges. For example, the reachable blocks for basic block B_3 in the control flow graph of Fig. 6 is the set $Reach(B_3) = \{B_3, B_4, B_5, B_6\}$, since the loopback edge from statement 13 to statement 6 is excluded from being traversed.

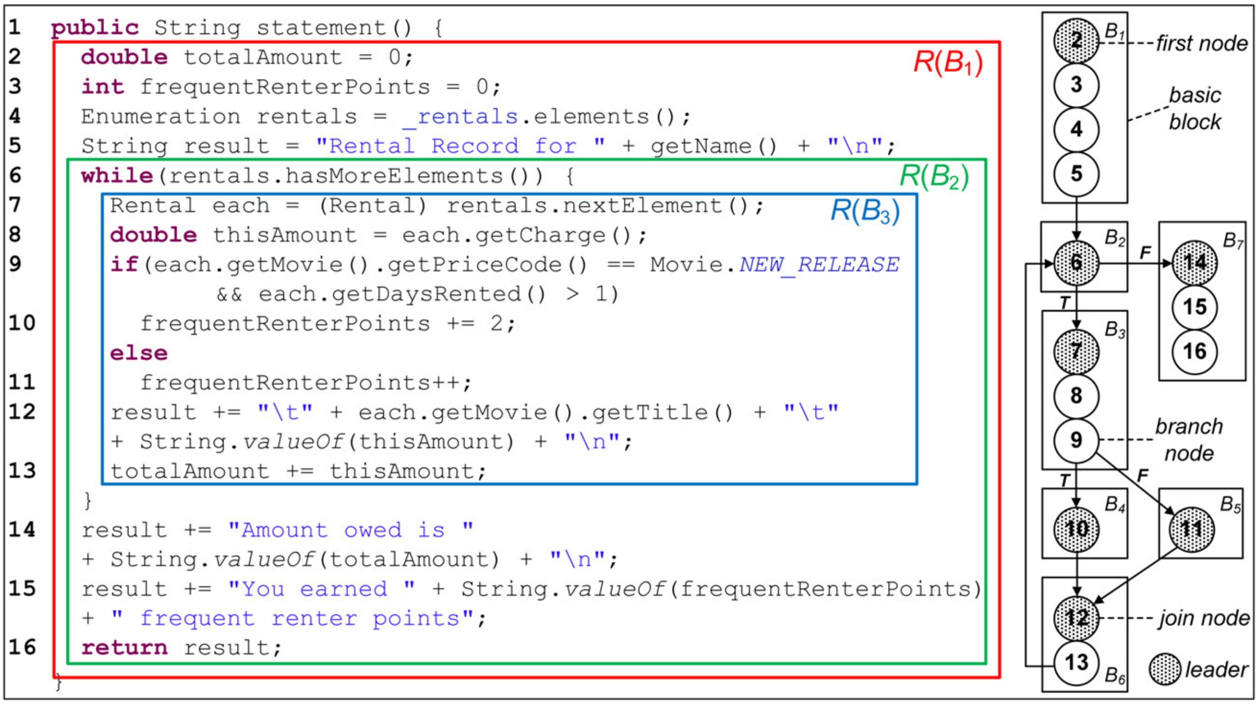


Fig. 6. Method `statement()` and its corresponding control flow graph.

3.2.3. Construction of control dependence graph and determination of dominated blocks

Next, the *control dependence graph* (i.e. the program dependence graph containing only control dependence edges) of method m is constructed. Fig. 7 shows the control dependence graph of method `statement()` decomposed into basic blocks (block-based CDG). The control dependence graph actually represents the nesting of statements inside a method (assuming that the code does not include unstructured control flow or exception flow).

Assuming that node r is the node that directly dominates the leader node of basic block B, Maruyama defined as *dominated blocks* for basic block B, $Dom(B)$, the set of blocks that are dominated by node r (a block is considered dominated by r if there exists a transitive control dependence from r to this block). For example, the leader node of block B_3 (node 7) is directly dominated by node 6 in the control dependence graph of Fig. 7. As a result, the dominated blocks for basic block B_3 are the blocks that are dominated by node 6, namely $\{B_3, B_4, B_5, B_6\}$. It should be mentioned that the notion of dominance in control dependence graphs is different from dominance in control flow graphs.

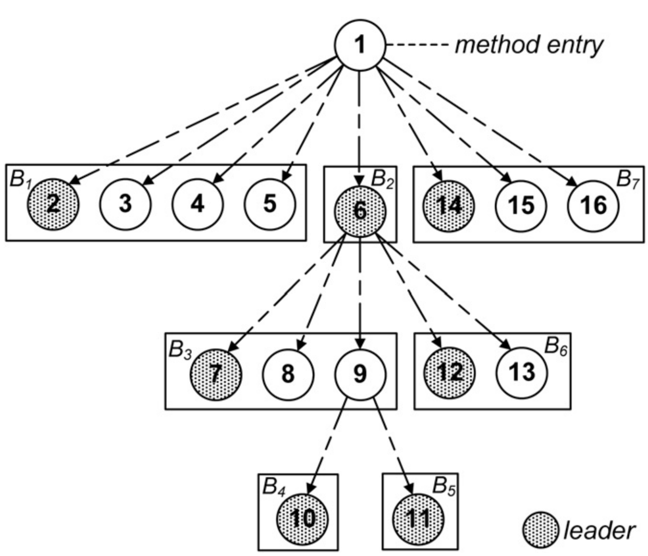


Fig. 7. Control dependence graph of method `statement()`.

3.2.4. Computation of boundary blocks

The sets of reachable and dominated blocks are used to compute the set of *boundary blocks* for statement n, $Blocks(n)$, in the following way:

For each basic block B of method m compute the sets of blocks $Reach(B)$ and $Dom(B)$.
If the basic block of statement n is contained in set $Reach(B) \cap Dom(B)$, then block B is added to the set of boundary blocks for statement n.

For example, the boundary blocks for statement 8 in Fig. 6, which belongs to basic block B_3, is the set $Blocks(8) = \{B_1, B_2, B_3\}$, since block B_3 is contained in the intersection of reachable and dominated blocks for basic blocks B_1, B_2 and B_3.

3.2.5. Determination of block-based regions

Based on the definition of reachable blocks, Maruyama defined as *block-based region* $R(B_n)$ for boundary block B_n the set of nodes which belong to $Reach(B_n)$. Fig. 6 depicts the statements that belong to regions $R(B_1)$, $R(B_2)$ and $R(B_3)$, respectively. In terms of program dependency, a block-based region can be considered as a subgraph of the program dependence graph of method m which contains as dependence edges only the edges that start from and also end in nodes of the region. It should be noted that a loop-carried data dependence belongs to the region subgraph, if additionally the loop node through which the dependence is carried belongs to the nodes of the region. Formally, the edges belonging to region $R(B)$ is the set

$$E_B(R(B)) = \{p \to_c q \in E(m) | p, q \in R(B)\} \cup \{p \to_d q \in E(m) | p, q \in R(B)\}$$
$$\cup \{p \to_{d(l)} q \in E(m) | l, p, q \in R(B)\}$$

where $E(m)$ is the set of all edges in the PDG of method m,

$p \to_c q$ denotes a control dependence edge from node p to node q,
$p \to_d q$ denotes a loop-independent data dependence edge from node p to node q, and
$p \to_{d(l)} q$ denotes a loop-carried data dependence edge from node p to node q which is carried by loop l.

Assuming that slicing criterion (n, u) is given, which consists of statement n belonging to method m and variable u that is defined or used in statement n, the block-based regions in which a slice can be computed are the regions of the boundary blocks for statement n, $Blocks(n)$. For example, the block-based regions for slicing criterion (8, `thisAmount`) are $R(B_1)$, $R(B_2)$ and $R(B_3)$, since the boundary blocks for statement 8 is the set $Blocks(8) = \{B_1, B_2, B_3\}$.

3.3. Algorithms for the identification of Extract Method refactoring opportunities

Our approach provides two main algorithms for the identification of Extract Method refactoring opportunities. The first algorithm identifies refactoring opportunities where the complete computation of a local variable or parameter (*complete computation slice*) can be extracted, meaning that the resulting slice will contain all the assignment statements modifying the value of the local variable. The second algorithm identifies refactoring opportunities where all the statements affecting the state of an object (*object state slice*) can be extracted. The object reference can be a local variable which is declared inside the body of the original method, a parameter of the original method, or a field of the class containing the original method. Both algorithms do not require any user input (i.e. selection of statements or variables) in order to operate.

3.3.1. Identification of complete computation 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 whose value is modified by at least one assignment statement, covering the complete computation of the corresponding variable. The algorithm consists of the following steps:

1. Identify the set of local variables V which are declared inside method m.
2. For each variable $v \in V$ identify the set of seed 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 block-based slice $S_B(c, v, B_n)$. Block-based slice $S_B(c, v, B_n)$ is the set of statements that may affect the computation of variable v at statement c (backward slice), extracted from the program dependence subgraph corresponding to region $R(B_n)$.
6. For each $B_n \in Blocks(C)$ the union of slices $US_B(C, v, B_n) = \bigcup_{c \in C} S_B(c, v, B_n)$ is a slice that covers the complete computation of variable v within the region $R(B_n)$.

This algorithm produces for each variable v declared inside method m, a number of slices which is equal to the size of $Blocks(C)$, where C is the set of statements containing an assignment of variable v. The application of the algorithm will be demonstrated on a well-established refactoring teaching example (Demeyer et al., 2005). Fig. 8 illustrates method `printDocument()` and its control flow graph decomposed into basic blocks.

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

a. The assignment statements of variable `author` are statements 11 and 20 (underlined in the code of Fig. 8).
b. 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 Fig. 8).
c. The intersection of the two sets of boundary blocks is $Blocks(\{11, 20\}) = \{B_1\}$ and as a result only block-based region $R(B_1)$ can be used as region for the union of the resulting static slices.
d. 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.
e. The union of the static slices is $US_B(\{11, 20\}, \texttt{author}, B_1) = \{2, 4, 5, 6, 7, 8, 9, 10, 11, 19, 20\}$.

3.3.2. Identification of object state slices

The proposed algorithm takes as input a method m and returns a set of slice extraction refactoring suggestions for each reference inside method m pointing to an object whose state is affected by at least one statement containing an appropriate method invocation or a direct field modification (in the case encapsulation is violated). The algorithm consists of the following steps:

1. Identify the set of object references R existing inside method m. These references are local variables, parameters of m, or fields of the class containing m having a non-primitive type.
2. For each object reference $r \in R$ identify the set of fields F_r which are modified through reference r by method invocations (or direct field modifications) inside the body of m. This is achieved by searching in the defined variables of each statement for composite variables having reference r as first part.
3. For each field $f \in F_r$ identify the set of seed statements C_f within the body of m that contain f in their set of defined variables. These statements along with variable f form a set of slicing criteria (c, f), where $c \in C_f$.
4. For each statement $c \in C_f$ compute the set of boundary blocks $Blocks(c)$.
5. Calculate the common boundary blocks for the statements in each set C_f (referring to defined variable f) as $Blocks(C_f) = \bigcap_{c \in C_f} Blocks(c)$.
6. Calculate the common boundary blocks for all $Blocks(C_f)$, $\forall f \in F_r$ (referring to object reference r) as $Blocks(r) = \bigcap_{f \in F_r} Blocks(C_f)$.
7. For each slicing criterion (c, f), where $c \in C_f$, $f \in F_r$ and boundary block $B_n \in Blocks(r)$ compute the block-based slice $S_B(c, f, B_n)$. Block-based slice $S_B(c, f, B_n)$ is the backward slice extracted from the program dependence subgraph corresponding to region $R(B_n)$.

```
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);
    }
```

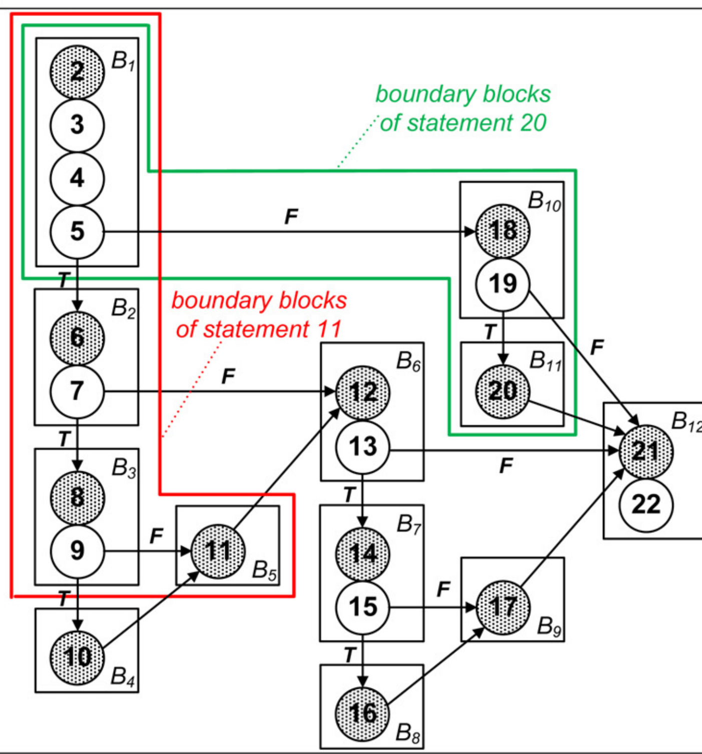


Fig. 8. Method `printDocument()` and the corresponding control flow graph.

8. For each $B_n \in Blocks(r)$ the union of slices for field f is $US_B(C_f, f, B_n) = \bigcup_{c \in C_f} S_B(c, f, B_n)$.
9. For each $B_n \in Blocks(r)$ the union of slices for reference r $US_B(r, B_n) = \bigcup_{f \in F_r} US_B(C_f, f, B_n)$ is a slice that contains all the statements in method m affecting the state of the object referenced by r.

This algorithm produces for each reference r, a number of slices which is equal to the size of $Blocks(r)$. The application of the algorithm will be demonstrated on a real example taken from an open-source project, namely Violet 0.16 (Horstmann, 2006). Fig. 9 illustrates method `removeSelected()` and its control flow graph decomposed into basic blocks.

Assume that the statements affecting the state of the object referenced by field `graph` are intended to be extracted as a separate method. The algorithm is applied as follows:

a. The set of fields F_{graph} which are modified through reference `graph` consists of the following composite variables:
   1. graph.nodesToBeRemoved.elementData
   2. graph.nodesToBeRemoved.size
   3. graph.nodesToBeRemoved.modCount
   4. graph.needsLayout
   5. graph.edgesToBeRemoved.elementData
   6. graph.edgesToBeRemoved.size
   7. graph.edgesToBeRemoved.modCount
b. Fields 1–3 are defined at statement 6, while fields 4–7 are defined at statements 6 and 8. The resulting slicing criteria are eleven in total, based on the following sets of seed statements:
   1. $C_{\text{graph.nodesToBeRemoved.elementData}} = \{6\}$
   2. $C_{\text{graph.nodesToBeRemoved.size}} = \{6\}$
   3. $C_{\text{graph.nodesToBeRemoved.modCount}} = \{6\}$
   4. $C_{\text{graph.needsLayout}} = \{6, 8\}$
   5. $C_{\text{graph.edgesToBeRemoved.elementData}} = \{6, 8\}$
   6. $C_{\text{graph.edgesToBeRemoved.size}} = \{6, 8\}$
   7. $C_{\text{graph.edgesToBeRemoved.modCount}} = \{6, 8\}$
c. The sets of boundary blocks for statements 6 and 8 are $Blocks(6) = \{B_1, B_2, B_3, B_4\}$ and $Blocks(8) = \{B_1, B_2, B_3, B_5, B_6\}$, respectively (as shown in the control flow graph of Fig. 9).
d. The resulting intersections of basic blocks are:
   1. $\text{Blocks}(C_{\text{graph.nodesToBeRemoved.elementData}}) = \{B_1, B_2, B_3, B_4\}$
   2. $\text{Blocks}(C_{\text{graph.nodesToBeRemoved.size}}) = \{B_1, B_2, B_3, B_4\}$
   3. $\text{Blocks}(C_{\text{graph.nodesToBeRemoved.modCount}}) = \{B_1, B_2, B_3, B_4\}$
   4. $\text{Blocks}(C_{\text{graph.needsLayout}}) = \{B_1, B_2, B_3\}$
   5. $\text{Blocks}(C_{\text{graph.edgesToBeRemoved.elementData}}) = \{B_1, B_2, B_3\}$
   6. $\text{Blocks}(C_{\text{graph.edgesToBeRemoved.size}}) = \{B_1, B_2, B_3\}$
   7. $\text{Blocks}(C_{\text{graph.edgesToBeRemoved.modCount}}) = \{B_1, B_2, B_3\}$
e. The final intersection of basic blocks is $Blocks(\texttt{graph}) = \{B_1, B_2, B_3\}$ and as a result block-based regions $R(B_1)$, $R(B_2)$ and $R(B_3)$ can be used as regions for the union of the resulting static slices.
f. In this code example, the resulting slices are the same for all slicing criteria. More specifically, $S_B(c, f, B_1) = \{2, 3, 4, 5, 6, 7, 8\}$, $S_B(c, f, B_2) = \{3, 4, 5, 6, 7, 8\}$ and $S_B(c, f, B_3) = \{4, 5, 6, 7, 8\}$, where $f \in F_{\text{graph}}$ and $c \in C_f$.
g. Consequently, the resulting unions of slices are also the same for all fields belonging to F_{graph}. More specifically, $US_B(C_f, f, B_1) = \{2, 3, 4, 5, 6, 7, 8\}$, $US_B(C_f, f, B_2) = \{3, 4, 5, 6, 7, 8\}$ and $US_B(C_f, f, B_3) = \{4, 5, 6, 7, 8\}$, where $f \in F_{\text{graph}}$.
h. Finally, the unions of slices for reference `graph` are $US_B(\texttt{graph}, B_1) = \{2, 3, 4, 5, 6, 7, 8\}$, $US_B(\texttt{graph}, B_2) = \{3, 4, 5, 6, 7, 8\}$ and $US_B(\texttt{graph}, B_3) = \{4, 5, 6, 7, 8\}$, respectively.

As it can be observed in the code of method `removeSelected()` in Fig. 9, statements 2–8 exhibit a distinct functionality compared to the rest of the statements, which is related with the removal of the selected nodes and edges from the graph object corresponding to field `graph`.

3.3.3. Determination of indispensable statements and parameters of the extracted method

Indispensable statements are statements that belong to a given slice but should not be removed from the original method after slice extraction to assure that the original method remains operational (i.e. are statements required by the statements that remain in the original method in order to operate correctly).

Maruyma's formalization of indispensable statements (Maruyama, 2001) concerns a single slice derived from a single criterion (statement, variable) within a given region. We have extended the formalization to handle the union of several slices derived from multiple criteria within a given region. The philosophy behind the determination of indispensable statements is in both cases the same: the slices for the remaining statements (i.e. method statements not belonging to the slice intended to be extracted) are computed. The statements which are common among the aforementioned slices and the slice to be extracted are the indispensable statements. The determination of indispensable statements can be formalized as follows:

Let $N(m)$ be the set of all nodes and $E(m)$ the set of all edges in the PDG of method m. Let S_B be a block-based slice resulting from the region of boundary block B, $R(B)$. Let U_B be the set of remaining nodes after the nodes of S_B are removed from $N(m)$, $U_B = N(m) \backslash S_B$.

Let N_{CD} be the set of nodes belonging to S_B on which nodes belonging to U_B are control dependent (i.e. there exists a control dependence edge from a node in S_B to a node in U_B).

$N_{CD}(S_B, U_B) = \{p \in N(m) | p \rightarrow_c q \in E(m) \wedge p \in S_B \wedge q \in U_B\}$, where $p \rightarrow_c q$ denotes a control dependence from node p to node q.

Let N_{DD} be the set of nodes belonging to S_B on which nodes belonging to U_B are data dependent (i.e. there exists a data dependence edge from a node in S_B to a node in U_B) due to a variable other than the variable of the slicing criterion.

$N_{DD}(S_B, U_B, v) = \{p \in N(m) | p \rightarrow_d^u q \in E(m) \wedge u \neq v \wedge p \in S_B \wedge q \in U_B\}$, where $p \rightarrow_d^u q$ denotes a data dependence from node p to node q due to variable u and v is the variable of the slicing criterion.

Control indispensable nodes I_{CD} are the nodes of the slices that result using (p, u, B) as slicing criteria, where $p \in N_{CD}(S_B, U_B)$ and u belongs to the used variables of node p.

$$I_{CD}(S_B, U_B) = \left\{ q \in N(m) | q \in S_B(p, u, B) \wedge p \in N_{CD}(S_B, U_B) \wedge u \in Use(p) \right\}$$

Data indispensable nodes I_{DD} are the nodes of the slices that result using (p, u, B) as slicing criteria, where $p \in N_{DD}(S_B, U_B, v)$ and u belongs to the defined variables of node p.

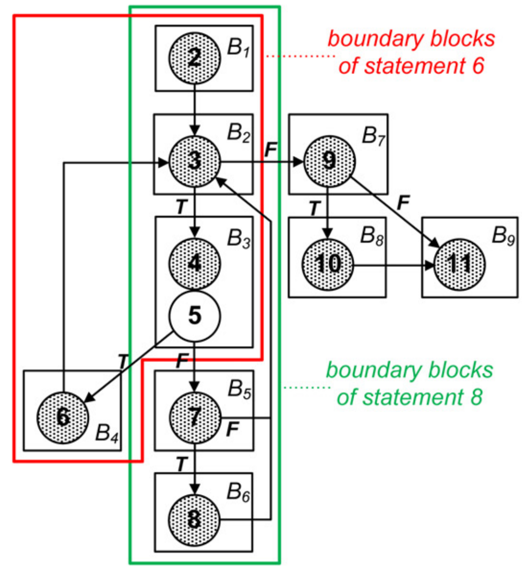
$$I_{DD}(S_B, U_B, v) = \left\{ q \in N(m) | q \in S_B(p, u, B) \wedge p \in N_{DD}(S_B, U_B, v) \wedge u \in Def(p) \right\}$$

Eventually, the indispensable nodes I_B is the set resulting from the union of I_{CD} and I_{DD} sets ($I_B = I_{CD} \cup I_{DD}$). Indispensable nodes will be duplicated in both the original and the extracted method after slice extraction, while the set of nodes that can be actually removed from the original method is $S_B \backslash I_B$ and the set of nodes that actually remain in the original method is $U_B \cup I_B$.

The parameters of the extracted method are the variables of the original method for which a data dependence exists from the set of remaining nodes U_B to the set of slice nodes S_B. Formally, $P(S_B, U_B) = \{u \in V(m) | p \rightarrow_d^u q \in E(m) \wedge p \in U_B \wedge q \in S_B\}$, where $V(m)$ is the set of variables which are declared within the body of method m (including the parameters of method m).

```
public class GraphPanel extends JPanel {
  private Graph graph;
  private Set selectedItems;

1 public void removeSelected() {
2   Iterator iter = selectedItems.iterator();
3   while (iter.hasNext()) {
4     Object selected = iter.next();
5     if (selected instanceof Node) {
6       graph.removeNode((Node) selected);
      }
7     else if (selected instanceof Edge) {
8       graph.removeEdge((Edge) selected);
      }
    }
9   if (selectedItems.size() > 0)
10    setModified(true);
11  repaint();
  }
}
```




```
public abstract class Graph {
  private ArrayList nodes;
  private ArrayList edges;
  private transient ArrayList nodesToBeRemoved;
  private transient ArrayList edgesToBeRemoved;
  private transient boolean needsLayout;

  public void removeNode(Node n) {
    if (nodesToBeRemoved.contains(n)) return;
    nodesToBeRemoved.add(n);
    // notify nodes of removals
    for (int i = 0; i < nodes.size(); i++) {
      Node n2 = (Node) nodes.get(i);
      n2.removeNode(this, n);
    }
    for (int i = 0; i < edges.size(); i++) {
      Edge e = (Edge)edges.get(i);
      if (e.getStart() == n || e.getEnd() == n)
        removeEdge(e);
    }
    needsLayout = true;
  }
  public void removeEdge(Edge e) {
    if (edgesToBeRemoved.contains(e)) return;
    edgesToBeRemoved.add(e);
    for (int i = nodes.size() - 1; i >= 0; i--) {
      Node n = (Node)nodes.get(i);
      n.removeEdge(this, e);
    }
    needsLayout = true;
  }
}
```

Fig. 9. Method `removeSelected()` and the corresponding control flow graph.

3.4. Rules regarding behavior preservation and usefulness of the extracted functionality

The slices resulting from the algorithms of Section 3.3 are examined against a set of rules that exclude refactoring opportunities corresponding to slices whose extraction could possibly cause a change in program behavior. The rules are preventive in the sense that they prescribe conditions that should not hold in order to obtain extractable slices which preserve program behavior. Moreover, there is a category of rules which are used to reject some extreme cases of slices that lead to extracted methods with limited usefulness in terms of functionality.

```
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 = (Rental) 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 = (Rental) rentals.nextElement();
9      if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
              && each.getDaysRented() > 1)
10       frequentRenterPoints += 2;
       else
11       frequentRenterPoints++;
     }
     return frequentRenterPoints;
   }
```

Fig. 10. Slice extraction using block-based slice $S_B(10,$ `frequentRenterPoints`$, B_2)$ causing change in behavior.

3.4.1. Duplication of statements affecting the state of an object

In object-oriented code the invocation of a method may 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. The duplication of such method invocations in both the remaining and the extracted method may not preserve the behavior of the program, since a duplicated statement is executed twice (i.e. once in the remaining method and once in the extracted method). To support our argument, two slice extraction examples based on the code of Fig. 6 will be demonstrated. Both examples concern the extraction of code from method `statement` using the same slicing criterion (10, `frequentRenterPoints`) but different block-based regions. The set of boundary blocks for statement 10 is $Blocks(10) = \{B_1, B_2, B_3, B_4\}$ (the layout of blocks is shown in Fig. 6), and as a result, four block-based slices can be derived from this slicing criterion. Fig. 10 shows the remaining and the extracted method when block-based slice $S_B(10,$ `frequentRenterPoints`$, B_2)$ is used.

As it can be observed in Fig. 10, 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 program is not preserved by 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 reference `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 Fig. 11.

As it can be observed in Fig. 11, the slice extraction based on basic block B_1, where slicing covers the entire source method, preserves the behavior of the program 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 object reference `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`).

Rule 1: The duplicated statements (i.e. the statements belonging to the set of indispensable nodes I_B) should not contain composite field variables (i.e. composite variables whose first part is an object reference existing in the original method and last part is a field) in their set of defined variables. From this rule are excluded the local object references whose declaration statement is also included to the duplicated statements. Formally, the rule is expressed as:

$$\left\{p \in I_B | o.f \in Def(p)\right\} \backslash \left\{p \in I_B | o.f \in Def(p) \wedge \exists q \in I_B | o \in Decl(q)\right\} = \varnothing$$

$Def(p)$ denotes the set of variables which are defined at statement p, $Decl(q)$ denotes the set of variables which are declared at statement q and $o.f$ denotes a composite variable whose first part is object reference o and last part is field f.

```
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 = (Rental) 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 = (Rental) rentals.nextElement();
9       if(each.getMovie().getPriceCode() == Movie.NEW_RELEASE
               && each.getDaysRented() > 1)
10        frequentRenterPoints += 2;
        else
11        frequentRenterPoints++;
      }
      return frequentRenterPoints;
    }
```

Fig. 11. Slice extraction using block-based slice $S_B(10,$ `frequentRenterPoints`, $B_1)$.

3.4.2. Duplication of statements containing a class instance creation

In the same manner that a statement causing a change in the state of an object can be duplicated, a statement creating an object may also be duplicated. Let us assume that a statement initializing or assigning reference r with a class instantiation (i.e. $r = new\ Type()$) is duplicated in both the original and the extracted method. Then each reference r (one being in scope within the original method and the other within the extracted method) will be referring to a different object in memory. As a result, the existence of non-duplicated statements affecting the state of the reference existing in the original method or the extracted method would cause an inconsistency of state between the two references. Such an inconsistency could in turn affect statements depending on reference r, causing a change in the behavior of the program.

Rule 2: A duplicated statement (i.e. a statement belonging to the set of indispensable nodes I_B) initializing or assigning object reference r with a class instantiation, should not have a data dependence due to variable r within the block-based region $R(B)$ of slice S_B that ends in a statement of the removable nodes $S_B \backslash I_B$. Formally, the rule is expressed as:

$$\left\{p \rightarrow^{r} q \in D_B(R(B)) | p \in I_B \wedge q \in S_B \backslash I_B \wedge r \in Inst(p)\right\} = \varnothing$$

where $D_B(R(B)) = \{p \rightarrow_d^u q | p,\ q \in R(B)\} \cup \{p \rightarrow_{d(l)}^u q | l,\ p,\ q \in R(B)\}$, $p \rightarrow_d q$ denotes a loop-independent data-dependence edge from node p to node q, $p \rightarrow_{d(l)} q$ denotes a loop-carried data-dependence edge from node p to node q which is carried by loop l and $Inst(p)$ denotes the set of object references which are instantiated at statement p.

3.4.3. Preservation of existing anti-dependences

Another case that may cause change in behavior is the existence of an anti-dependence between a statement that remains in the original method and a statement belonging to the slice statements that will be removed from the original method. An anti-dependence (Komondoor and Horwitz, 2000) exists from statement p to statement q (or statement q anti-depends on p) due to variable x, when there is a control flow path starting from statement p that uses the value of x and ending to statement q that modifies the value of x (regardless of any intermediate statements that may use the value of variable x). Just like data flow dependences, anti-dependences can be either loop carried (i.e. carried by a specific loop) or loop independent. Fig. 12 shows an example of code containing a loop carried anti-dependence which is carried by the `while` loop in statement 7 (this example is exactly the same with the one used in the previous sections, with the only difference that the declaration of variable `thisAmount` has been placed outside the `while` loop in order to make reasonable the extraction of its computation over the entire method body). As it can be observed, the value of variable `thisAmount` is used at statement 14 and in the next iteration of the `while` loop its value is modified at statement 9.

Let us consider that slicing criterion (9, `thisAmount`) is used for the code of Fig. 12. The set of boundary blocks for statement 9 is $Blocks(9) = \{B_1, B_2, B_3\}$ (the layout of blocks is shown in Fig. 6), and as a result, three block-based slices can be derived from this slicing criterion. The slice corresponding to the block-based region of block B_1 (region $R(B_1)$ contains all the statements of the original method) is $S_B(9,$ `thisAmount`, $B_1) = \{4, 5, 7, 8, 9\}$ and is extracted as shown in Fig. 13.

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

---> loop-carried anti-dependence

Fig. 12. Code example containing a loop carried anti-dependence.

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

    private double getThisAmount() {
4     double thisAmount = 0;
5     Enumeration rentals = _rentals.elements();
7     while (rentals.hasMoreElements()) {
8       Rental each = (Rental) rentals.nextElement();
9       thisAmount = each.getCharge();
      }
      return thisAmount;
    }
```

Fig. 13. Extraction of slice $S_B(9, \texttt{thisAmount}, B_1)$ causing change in behavior.

As it can be observed in Fig. 13, the behavior of the program is not preserved by the extraction of block-based slice S_B(9, `thisAmount`, B_1), since the extracted method returns the amount of charge corresponding to the last element of Vector `_rentals`. As a result, the value of variable `thisAmount`, which is used in statements 13 and 14 in the original method, is correct only in the last iteration of the `while` loop inside the original method. Obviously, the final values of variables `result` and `totalAmount` are affected due to the incorrect value of variable `thisAmount` at each iteration. The reason causing this change in behavior is that the anti-dependence that initially existed in the original method is altered after slice extraction, since the statement from which it started remains in the original method while the statement to which it ended is moved to the extracted method thus affecting their order of execution.

Rule 3: There should not exist an anti-dependence (due to variable u) within the block-based region $R(B)$ of slice S_B starting from a statement p of the remaining nodes $U_B \cup I_B$ and ending to a statement q of the removable nodes $S_B \backslash I_B$, without the presence of a data dependence due to variable u within the block-based region $R(B)$ between a statement k of the remaining nodes and statement p. Formally, the rule is expressed as:

$$\left\{p \rightarrow q \in A_B(R(B)) | p \in U_B \cup I_B \wedge q \in S_B \backslash I_B\right\} \backslash$$
$$\left\{p \rightarrow q \in A_B(R(B)) | p \in U_B \cup I_B \wedge q \in S_B \backslash I_B \wedge \exists k \rightarrow_d^u p | k \in R(B) \wedge k \in U_B \cup I_B\right\} = \varnothing$$

where $A_B(R(B)) = \{p \rightarrow_a^u q | p, q \in R(B)\} \cup \{p \rightarrow_{a(l)}^u q | l, p, q \in R(B)\}$, $p \rightarrow_a q$ denotes a loop-independent anti-dependence edge from node p to node q, $p \rightarrow_{a(l)} q$ denotes a loop-carried anti-dependence edge from node p to node q which is carried by loop l and $k \rightarrow_d^u p$ denotes a data dependence from node k to node p due to variable u.

The exception regarding the presence of a data dependence between statement k (defining variable u) and p (using variable u) of the remaining nodes is motivated by the fact that the definition of u in statement k kills any previous definition, such as the one in statement q which after the slice extraction would be placed before k and p.

3.4.4. Preservation of existing output-dependences

Another case that may cause change in behavior is the existence of an output-dependence between a statement that remains in the original method and a statement belonging to the slice statements that will be removed from the original method. An output-dependence (Komondoor and Horwitz, 2000) exists from statement p to statement q (or statement q is output-dependent on p) due to variable x, when there is a control flow path starting from statement p that modifies the value of x and ending to statement q that also modifies the value of x (regardless of any intermediate statements that may use the value of variable x). Fig. 14 shows an example of code containing two output-dependences from statement 3 to statements 9 and 13 (this example is taken from class ChartPanel in JFreeChart project).

Let us consider that the complete computation of variable `drawWidth` is intended to be extracted by using the entire method body as region. Based on our approach, statements 3, 9 and 13 will be used as seed statements and the slice resulting from the union of the corresponding slices is US_B({3, 9, 13}, `drawWidth`, B_1) = {3, 7, 9, 11, 13}. The set of statements that should be duplicated is I_B = {3, 7, 11}. Statements 7 and 11 should be duplicated due to the remaining statements within their bodies, leading eventually to the duplication of statement 3. It becomes obvious, that if the extracted method is invoked at the beginning of method `paintComponent()`, statements 7 and 11 will be correctly executed, since the value of variable `drawWidth` will be the same as the original program (due to the duplication of statement 3). However, the value of variable `drawWidth` will not be the same as the original program at statement 15, due to its redefinition at statement 3 after its initial definition at the beginning of the method through the invocation of the extracted method. The reason causing this change in behavior is that the output-dependences that initially existed in the original method are altered after slice extraction, since the statement from which they started remains in the original method while the statements to which they ended are moved to the extracted method.

It should be emphasized that in the code of Fig. 14 there also exist anti-dependences (namely 7 → 9, 7 → 13, 8 → 9, 11 → 13 and 12 → 13) which are not being preserved by slice extraction. However, these cases of anti-dependences do not activate the rule of Section 3.4.3 due to the presence of duplicated statement 3 that kills the initial definition of variable `drawWidth` at the beginning of the method through the invocation of the extracted method.

Rule 4: There should not exist an output-dependence (due to variable u) within the block-based region $R(B)$ of slice S_B starting from a statement p of the remaining nodes $U_B \cup I_B$ and ending to a statement q of the removable nodes $S_B \backslash I_B$. Formally, the rule is expressed as:

$$\left\{p \rightarrow q \in O_B(R(B)) | p \in U_B \cup I_B \wedge q \in S_B \backslash I_B\right\} = \varnothing$$

where $O_B(R(B)) = \{p \rightarrow_o^u q | p, q \in R(B)\} \cup \{p \rightarrow_{o(l)}^u q | l, p, q \in R(B)\}$, $p \rightarrow_o q$ denotes a loop-independent output-dependence edge from node p to node q, and $p \rightarrow_{o(l)} q$ denotes a loop-carried output-dependence edge from node p to node q which is carried by loop l.

3.4.5. Rules regarding the usefulness of the extracted code in terms of functionality

The goal of the rules defined in this section is to prevent some extreme cases of slices from being suggested as refactoring opportunities. These rules are related with the extent of the slice compared to the number of seed statements and the size of the original method, the degree of code duplication and the variable which is returned by the original method.

a. The number of statements in the union of slice statements US_B should be greater than the number of seed statements used in slicing criteria. In the case where the number of statements in US_B is equal to the number of seed statements used in 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 for the computation of a given variable (or by the statements affecting the state of a given object). This means that a slice should consist of two statements at minimum, if we assume that a single seed statement is used.
b. The number of statements in the union of slice statements US_B should not be equal to the number of statements in the original method. In such a case the extracted method would be exactly the same as the original method.
c. The statements which are duplicated in both the original and the extracted method should not contain all the seed statements used in slicing criteria. If all the seed statements used in slicing criteria were duplicated, then the computation of a given variable (or the statements affecting the state of a given object) would exist in both the original and the extracted method making the extraction redundant.
d. The variable which is returned by the original method should be excluded from slice extraction. If the computation of a given variable (or the statements affecting the state of a given object) that is returned by the original method was extracted, then the extracted method would contain a significant portion of the

```
1   public void paintComponent(Graphics g) {
      //create object Rectangle2D available
2     boolean scale = false;
3     drawWidth = available.getWidth();
4     drawHeight = available.getHeight();
5     this.scaleX = 1.0;
6     this.scaleY = 1.0;

7     if (drawWidth < this.minimumDrawWidth) {
8       this.scaleX = drawWidth / this.minimumDrawWidth;
9       drawWidth = this.minimumDrawWidth;
10      scale = true;
      }
11    else if (drawWidth > this.maximumDrawWidth) {
12      this.scaleX = drawWidth / this.maximumDrawWidth;
13      drawWidth = this.maximumDrawWidth;
14      scale = true;
      }
      //compute drawHeight
15    Rectangle2D chartArea = new Rectangle2D.Double(
                   0.0, 0.0, drawWidth, drawHeight);
    }
```

output-dependence

Fig. 14. Code example containing output-dependences.

functionality of the original method and to a large extent would serve the same purpose.

3.5. Limitations

Our approach employs block-based regions as a means to demarcate the boundaries of slice expansion. This strategy enables the extraction of slices which in some cases would not be feasible if the entire method body was used as region. However, the boundaries of a block-based region are not always ideal as scope for slice extraction. Clearly, there are cases where it would be preferable to employ boundaries other than those implied by the block-based regions. This could avoid the inclusion of additional variable computations leading to the duplication of the corresponding statements. On the other hand, the exploration of all possible boundaries would introduce a significant computational cost and would drastically increase the number of reported refactoring opportunities.

The proposed rules regarding behavior preservation may be too strong on some cases causing the rejection of certain valid opportunities. For example, duplicating the invocation of a setter method with the same argument would be safe, since the value of the corresponding field (and eventually the state of the corresponding object) would remain the same even if the setter method would be invoked twice after slice extraction. However, the determination of whether the duplication of statements affecting the state of an object eventually changes the behavior of code requires extensive semantic analysis which is not covered by the employed techniques. As a result, the rule of Section 3.4.1 can be regarded as a worst-case rule.

Our approach does not handle labeled `break` and `continue` statements which have a similar functionality with the `goto` statements used in older programming languages. The reason is that such statements affect drastically the ordinary control flow of programs and eventually the formation of block-based regions which are essential in our approach. However, their use is generally discouraged since they violate the structured programming principles and are rarely used in modern programs (Gellerich et al., 1996; Stamelos et al., 2002).

Finally, our approach does not handle `return` statements, since the operation of a `return` statement is directly associated with the method that it belongs to, and thus a `return` statement cannot be copied to another method. As a result, if a `return` statement has a direct or indirect incoming control dependence from a statement belonging to a given slice, then this slice is rejected from being suggested as a refactoring opportunity.

4. JDeodorant eclipse plug-in

The proposed methodology has been implemented as an Eclipse plug-in (JDeodorant, 2010) that identifies Extract Method refactoring opportunities on Java projects, highlights the code fragments suggested to be extracted (by indicating with green color the statements that will be moved to the extracted method and with red color the statements that will be duplicated in both the original and the extracted method) and automatically applies on source code the refactorings which are eventually approved by the user. In order to control the number and the quality of the identified refactoring opportunities being reported, JDeodorant offers a preference page where the user can define various threshold values regarding the following properties:

- The minimum size (in number of statements) that a method should consist of in order to be examined for potential refactoring opportunities.
- The minimum number of statements that a slice should consist of in order to be reported as a refactoring opportunity.
- The maximum number of duplicated statements (between the original and the extracted method) that the extraction of a slice may introduce in order to be reported as a refactoring opportunity.

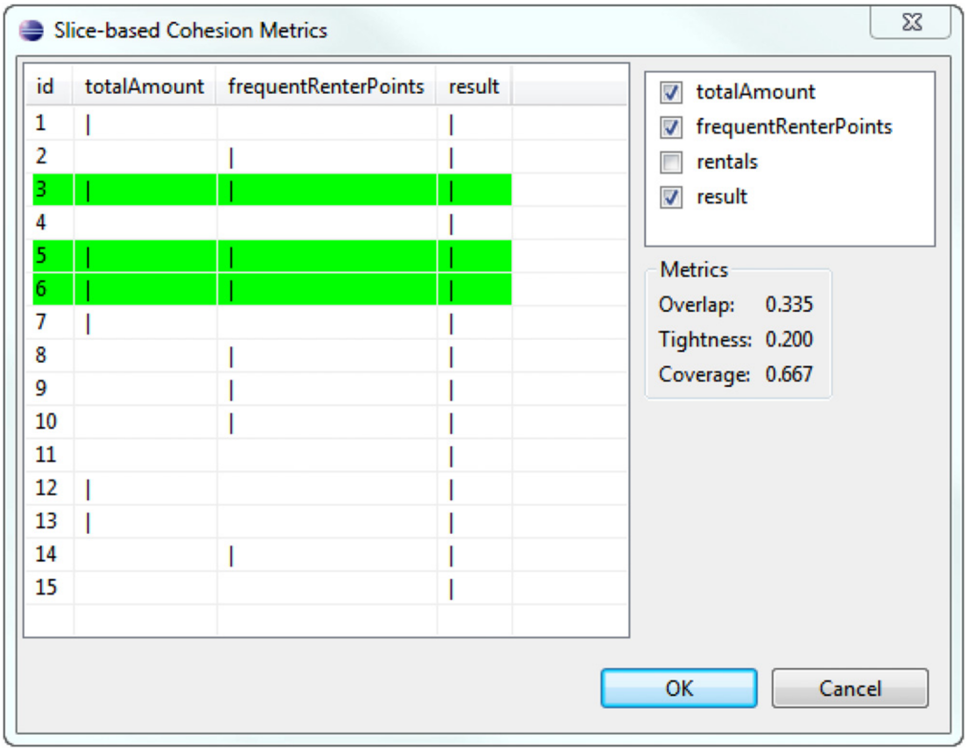


Fig. 15. JDeodorant calculator for slice-based cohesion metrics.

- The maximum ratio of duplicated to extracted statements (ranging over the interval [0, 1]) that should apply for a slice extraction refactoring in order to be reported.

In order to support the user in assessing the cohesion of a given method (i.e. the degree of interdependence among the statements required for the computation of the variables declared inside a method), JDeodorant offers a flexible calculator for slice-based cohesion metrics (Ott and Thuss, 1993). The calculator automatically computes the backward slices for all the local variables whose scope is the block corresponding to the method body, constructs the slice profile (Ott and Thuss, 1989) of the examined method and highlights the statements which are common to all computed slices, as shown in Fig. 15. The user has the ability to exclude from the slice profile of the examined method any variables which cannot be considered as output variables (i.e. variables playing an auxiliary role in the computation of other variables and whose computation is not intended to be extracted in a separate method) in order to improve the accuracy of the calculated slice-based cohesion metrics. Fig. 15 shows the slice profile and the calculated slice-based cohesion metrics, namely overlap, tightness and coverage for the method of Fig. 6. As it can be observed, variable `rentals` has been excluded from the slice profile, since it plays an auxiliary role in the computation of the other variables.

Finally, JDeodorant sorts the identified refactoring opportunities according to their effectiveness as measured by the *duplication ratio* (i.e. the ratio of the number of statements that will be duplicated after the extraction of a slice to the number of statements which are going to be extracted). First, the identified slice extraction opportunities are grouped according to the variable or object reference that they concern (i.e. a slice that can be extracted using more than one block-based regions is considered as a single refactoring opportunity) in order to present relevant refactoring opportunities in a consecutive way. The resulting groups are sorted according to the average duplication ratio of the refactoring opportunities belonging to each group in ascending order. In the case where two groups have an average duplication ratio equal to zero (i.e. none of the slice extraction opportunities belonging to the groups causes duplication of statements), the groups are sorted according to the maximum number of statements that can be extracted by the refactoring opportunities belonging to each group in descending order. The reasoning behind this sorting mechanism is that the extraction of slices causing significant duplication should be less preferred, since such slices are generally cohesive with the method from which they are extracted.

5. Evaluation

The evaluation of the proposed methodology consists of two main parts. The first part concerns the evaluation on an open-source project and includes an independent assessment of the identified refactoring opportunities regarding their soundness and usefulness, an investigation of the impact of the suggested refactorings on slice-based cohesion metrics and finally an investigation of the impact of the suggested refactorings on the external behavior of the program. The second part concerns the evaluation of the identified refactoring opportunities against those identified by independent evaluators on software that they developed.

5.1. Qualitative and quantitative evaluation on an open-source project

The criteria for selecting an appropriate project for the evaluation of the proposed methodology are the following:

(a) The source code of the project should be publicly available, since JDeodorant performs source code analysis in order to identify refactoring opportunities. Furthermore, source code availability will make possible the reproduction of the experimental results.
(b) The project should be large enough in order to present a sufficient number of refactoring opportunities.

Table 1
Independent assessment of the identified refactoring opportunities.

Package org.jfree.chart	Number of refactoring opportunities				
	Question a		Question b		
	Identified	Having distinct functionality	Removing duplicated code	Decomposing a complex method	Constituting a feature envy case
Complete computation	11	7	2[a]	5	1[b]
Object state	53	50	13	6	0
Total	64	57	15	11	1

[a] Two complete computation slices have been commented as both decomposing a complex method and removing duplicated code.
[b] One complete computation slice has been commented as both decomposing a complex method and constituting a feature envy case.

(c) The project should exhibit high test coverage to make feasible the examination of behavior preservation after the application of the identified refactoring opportunities.

The project which has been selected is JFreeChart. It is a rather mature open-source chart library which has been constantly evolving since 2002. Version 1.0.0 consists of 771 classes and 95K lines of source code (as measured by sloccount), while its average test coverage is 63.7% (as measured by EclEmma code coverage tool).

5.1.1. Independent assessment

To consider an approach for identifying refactoring opportunities successful, it must be able to suggest refactorings which preserve program behavior, are conceptually sound and useful, and have a positive impact on certain quality metrics. The conceptual soundness and usefulness of the refactoring opportunities can only be assessed by human expertise. To this end, an independent expert was asked to express his opinion on the refactoring opportunities that were identified in package org.jfree.chart of JFreeChart project. The independent designer had significant experience in software design (he has been working for more than 13 years as a telecommunications software designer) and deep knowledge of object-oriented design principles. More specifically, the independent designer had to answer the following questions for each identified refactoring opportunity, which also form the research questions in this part of the evaluation:

(a) Does the code fragment suggested to be extracted as a separate method have a distinct and independent functionality compared to the rest of the original method? If yes, describe its functionality by providing the name of the extracted method. If no, provide the reason for which the refactoring suggestion is not acceptable.
(b) Does the application of the suggested refactoring solve an existing design flaw (e.g. by decomposing a complex method, removing a code fragment that is duplicated among several methods, or extracting a code fragment suffering from Feature Envy)?

Therefore, the hypothesis being examined can be stated as:

> "the identified refactoring opportunities concern code fragments having distinct and independent functionality and their application resolves an existing design flaw".

Package org.jfree.chart (excluding its sub-packages) consists of 18 classes, 301 methods with body and 4564 lines of source code. It is actually the core package of JFreeChart library, since it is responsible for generating all supported chart types. In order to obtain meaningful refactoring suggestions we have excluded from the report the methods having less than 10 statements and the refactoring opportunities corresponding to slices with less than 4 statements by activating the appropriate property thresholds. The activated threshold, which is related with the size of the methods being examined for the identification of refactoring opportunities, reduced the number of analyzed methods from 301 to 51, from which 39 presented at least one refactoring opportunity (64 refactoring opportunities in total). The results of the evaluation are summarized in Table 1. The identified slice extraction opportunities have been grouped according to the variable or object reference that they concern (i.e. a slice that can be extracted using more than one block-based regions is considered as one refactoring opportunity).

As it can be observed in Table 1, the independent designer reported that 57 out of 64 (89%) identified refactoring opportunities correspond to code fragments having a distinct functionality compared to the rest of the original method. The independent designer disapproved 7 out of 64 (11%) identified refactoring opportunities for the following reasons:

- The code fragment suggested to be extracted did not have an obvious functionality and thus the extracted method would not have a clear purpose. (2/7)
- The code fragment suggested to be extracted had a trivial functionality and thus the extracted method would be useless. (1/7)
- The code fragment suggested to be extracted covered a large portion of the original method and thus the remaining functionality in the original method would be very limited after its extraction. (2/7)
- The code fragment suggested to be extracted shared several statements with other slices in the original method and thus its extraction would cause significant code duplication between the remaining and the extracted method. (2/7)

Furthermore, the independent designer reported that 27 out of 64 (42%) identified refactoring opportunities actually resolved (or in some cases helped to resolve) an existing design flaw. More specifically, 15 refactoring opportunities were utilized to remove three groups of duplicated code. The largest group of duplicated code consists of 11 cases that were extracted into a single method. Finally, 11 refactoring opportunities were utilized to decompose complex methods and one refactoring opportunity resulted in an extracted method suffering from Feature Envy that should be further moved to the envied class.

Since the proposed methodology employs rules to avoid changes in program behavior and non-useful slice extraction opportunities, Table 2 lists the number of slices (along with the percentage over the total number of slices) that have been rejected by each rule. It should be emphasized that some slices have been rejected by more than one rule.

As it can be observed in Table 2, about 20–30% of the constructed complete computation and object state slices are finally accepted and presented as refactoring opportunities. Furthermore, the effect of behavior preservation rules is much more intense on object state slices compared to complete computation slices.

Table 2
Number of slices rejected by each rule.

Rules	Description	Complete computation	Object state
Behavior preservation	3.4.1 Duplication of statements affecting the state of an object	12 (6.5%)	182 (34.9%)
	3.4.2 Duplication of statements containing a class instance creation	7 (3.8%)	102 (19.6%)
	3.4.3 No preservation of existing anti-dependences	18 (9.7%)	24 (4.6%)
	3.4.4 No preservation of existing output-dependences	27 (14.6%)	79 (15.2%)
Usefulness	3.4.5.a Slice statements are equal to seed statements	27 (14.6%)	85 (16.3%)
	3.4.5.b Slice is equal to method body	0(0%)	7 (1.3%)
	3.4.5.c All seed statements are duplicated	8 (4.3%)	78(15%)
	3.4.5.d Variable or object reference associated with the slice is returned by the original method	8 (4.3%)	12 (2.3%)
User threshold	Slice size is less than 4 statements	96 (51.9%)	147 (28.2%)
Accepted slices		36 (19.5%)	154 (29.6%)
Total number of slices		185	521

5.1.2. Impact on slice-based cohesion metrics

The empirical study of Meyers and Binkley (2007) has shown that slice-based metrics can be used to quantify the deterioration that accompanies software evolution and measure the progress of a reengineering effort. To provide an estimate of the improvement in terms of cohesion introduced by the decomposition of methods, we have measured the slice-based cohesion of the original method (before slice extraction), the remaining method (after slice extraction) and the extracted method for the refactoring opportunities that the independent designer has agreed on. The hypothesis being examined in this part of the evaluation is that

> "the application of the identified refactoring opportunities improves the cohesion of the affected code".

Ott and Thuss (1993) were the first that formally defined a set of quantitative metrics in order to estimate the level of cohesion in a module. The defined cohesion metrics were based on *slice profiles* (Ott and Thuss, 1989) which constitute a convenient representation for revealing slice patterns within a module. Let V_M be the set of variables used by module M and V_O be a subset of V_M containing only the output variables of M. As output variables are considered: (a) the variable which is returned by M, (b) the global variables which are modified by M, and (c) the parameters which are passed by reference and are modified by M. Finally, let SL_i be the slice obtained for variable $v_i \in V_O$ and SL_{int} be the intersection of SL_i over all $v_i \in V_O$. The tightness, overlap and coverage of module M are defined as:

$$Tightness(M) = \frac{|SL_{\text{int}}|}{length(M)}, \quad Overlap(M) = \frac{1}{|V_O|}\sum_{i=1}^{|V_O|}\frac{|SL_{\text{int}}|}{|SL_i|},$$

$$Coverage(M) = \frac{1}{|V_O|}\sum_{i=1}^{|V_O|}\frac{|SL_i|}{length(M)}$$

Tightness expresses the ratio of the number of statements which are common to all slices over the module length, while overlap expresses the average ratio of the number of statements which are common to all slices to the size of each slice. The higher the tightness and overlap of a module is, the more cohesive the module is. Obviously, in modules with high tightness or overlap the number of duplicated statements between the remaining and the extracted method will be large after the extraction of a slice. On the other hand, coverage expresses the average slice size over the module length and thus is not directly associated with the degree of common statements among the slices. However, a high value of coverage, which can be achieved when the slices extend over a large portion of the module, indirectly indicates the existence of several common statements among the slices.

In the Java programming language only a single variable can be returned by a given method, since the parameters are passed by value and thus their initial value is not possible to change during the execution of the method. Obviously, using a single variable (i.e. the returned variable) in the slice profile of a method would result in artificially high values of slice-based cohesion metrics which would not sufficiently reveal the actual cohesion. To overcome this problem, we have considered as output variables all the variables whose scope is the block corresponding to the body of the method under examination, since these variables could be potentially returned at the end of the method. Furthermore, the considered output variables which are simply accessed and not modified within the body of the method are excluded from the slice profile.

Table 3 shows the average change of slice-based cohesion metrics caused by the application of the Extract Method refactorings which have been approved by the independent expert (presented by slice type and in total). More specifically, the values of the third column have been calculated as the average difference between the remaining method (i.e. the original method after the application of the refactoring) and the original method. This expresses the (expected) improvement in cohesion of the original method that

Table 3
Average change of slice-based cohesion metrics.

Slice type	Metric	Remaining – original	Extracted	(Extracted + remaining)/2 – original
Complete computation	Overlap	+0.177	0.995	+0.305
	Tightness	–0.038	0.989	+0.304
	Coverage	–0.123	0.995	+0.135
Object state	Overlap	+0.302	0.876	+0.302
	Tightness	+0.222	0.803	+0.322
	Coverage	0	0.905	+0.110
Total	Overlap	+0.287	0.891	+0.303
	Tightness	+0.190	0.827	+0.319
	Coverage	–0.015	0.917	+0.113

Table 4
Number of slices rejected by each rule for the methods associated with unit tests.

Rules	Description	Complete computation	Object state
Behavior preservation	3.4.1 Duplication of statements affecting the state of an object	6 (7.7%)	41 (18.4%)
	3.4.2 Duplication of statements containing a class instance creation	0(0%)	44 (19.7%)
	3.4.3 No preservation of existing anti-dependences	12 (15.4%)	8 (3.6%)
	3.4.4 No preservation of existing output-dependences	1 (1.3%)	2 (0.9%)
Usefulness	3.4.5.a Slice statements are equal to seed statements	14 (17.9%)	45 (20.2%)
	3.4.5.b Slice is equal to method body	0(0%)	0(0%)
	3.4.5.c All seed statements are duplicated	5 (6.4%)	24 (10.7%)
	3.4.5.d Variable or object reference associated with the slice is returned by the original method	7(9%)	0(0%)
User threshold	Slice size is less than 4 statements	55 (70.5%)	65 (29.1%)
Accepted slices		7(9%)	109 (48.9%)
Total number of slices		78	223

has been refactored. The fourth column indicates the average metric values for the extracted methods which have been created after the application of the refactorings. Finally, the values of the fifth column have been calculated as the average difference between the average metric value for the extracted and the remaining method (i.e. the changed/created methods after the application of the refactoring) and the original method (i.e. the method existing before the application of the refactoring). This expresses the (expected) improvement in the average cohesion of the two resulting methods (remaining and extracted).

As it can be observed in the third and fifth columns of Table 3, the improvement of slice-based cohesion metrics can be considered significant by taking into account that their values range over the [0, 1] interval. Deterioration is observed in the average difference of coverage between the remaining and the original method (third column) for the cases that resulted from the extraction of complete computation slices. On the other hand, the average difference of coverage between the remaining and the original method (third column) for the cases that resulted from the extraction of object state slices is zero. Finally, as it can be observed in the fourth column of Table 3, the slice-based cohesion metrics for the extracted methods exhibit significantly high average values (especially for the methods that resulted from the extraction of complete computation slices) indicating that the corresponding complete computation and object state slices constitute strongly cohesive code fragments.

5.1.3. Impact on program behavior

To assess the impact of the identified refactoring opportunities on program behavior we have applied the corresponding refactoring transformations on source code using the JDeodorant tool and run the JUnit tests of the project under examination in order to find out whether the applied refactorings caused test errors. The hypothesis being examined in this part of the evaluation is that

> "the application of the identified refactoring opportunities does not modify program behavior".

From the 39 methods presenting at least one refactoring opportunity in package org.jfree.chart of JFreeChart project, 21 were actually associated with unit tests with an average test code coverage equal to 87% (as measured by EclEmma code coverage tool). The average test code coverage percentage can be considered sufficiently high in order to assess the preservation of program behavior after the application of the refactorings.

In total, 41 refactoring opportunities were identified for the 21 methods being tested in package org.jfree.chart of JFreeChart project. After the application of each refactoring all unit tests of the project were executed in order to examine whether the applied refactoring caused test errors. All of the applied refactorings passed the tests successfully without causing any test failure. Therefore, we can conclude with a relative certainty that the defined behavior preservation rules have successfully excluded refactoring opportunities that could possibly cause a change in program behavior.

Table 4 lists the number of slices (along with the percentage over the total number of slices) that have been rejected by each rule for the 21 methods associated with unit tests in package org.jfree.chart of JFreeChart project. It should be emphasized that some slices have been rejected by more than one rule.

5.2. Evaluation of precision and recall against the findings of independent evaluators

The goal of this part of the evaluation is to employ the refactoring opportunities found by independent evaluators in selected pieces of software that they developed. The opportunities that have been identified by the evaluators have been considered as a golden set (True Occurrences – TO), allowing the extraction of the precision and recall of our approach.

The two evaluators that participated in this study are PhD candidates, having significant experience in object-oriented design, while the analyzed projects have been developed within the context of their research. The PhD students were unfamiliar with the techniques and the underlying philosophy of our identification approach. The first analyzed project is WikiDev 2.0 (Fokaefs et al., 2010), which is a tool that adopts a wiki-based architecture for integrating information feeds from a variety of tools that software-team members use for design, development and communication. WikiDev is the result of 2 years of development at the Service Systems Research Group, in the Department of Computing Science at the University of Alberta, Canada. The second project is SelfPlanner 1.5.2 (Refanidis and Alexiadis, 2008), which is an intelligent Web-based calendar application that plans the tasks of a user employing an adaptation of the Squeaky Wheel Optimization framework. It is the outcome of 3 years of development in the Artificial Intelligence Group at the Department of Applied Informatics, University of Macedonia, Greece.

Since the evaluation of the entire project would require a prohibitive amount of time and effort by the evaluators, the analysis has been restricted to a number of selected methods presenting at least one refactoring opportunity (based on the findings of the proposed approach) and having varying number of statements.

The hypothesis being tested in this part of the evaluation can be stated as:

> "The findings of the proposed approach match the refactoring opportunities identified by human expertise to a large extent".

The task assigned to the evaluators was to manually identify Extract Method refactoring opportunities for the set of selected methods in their projects, respectively. A secondary task was to apply the corresponding refactorings either manually or by exploit-

Table 5
Precision and Recall of the proposed approach for project WikiDev 2.0.

Method	Cases found by the evaluator (method name – line numbers)	Ident. time (m:s)	Appl. time (m:s)	TO	#Cases found by the tool	TP	FN	FP	Precision (%)	Recall (%)
#1 clustering. Hierarchical::clustering	clusters (26–31) newDistances (50–81) newDistances (84–155)	3:22	3:00	3	4	2	1	2	50.0	66.7
#2 clustering. MatrixOperator::getFiedlerVector	L (25–38) sortedEigenValues (47–60) minIndex (62–72) eigenVectors (73–78)	2:00	1:25	4	2	1	3	1	50.0	25.0
#3 clustering. SammonsProjection::calculateDistanceMatrix	No opportunities found	1:24	–	0	1	0	0	1	0.0	N/A
#4 RelationshipMiner::getRelationships	initializeDocuments (100–137) calculateTfidf (139–171) mineRelationships (173–226)	1:40	1:18	3	1	1	2	0	100.0	33.3
#5 ClusteringMain::main	totalClusters (53, 67–74)[a] finalClusters (66–74) coords (60, 76–81)[a] writeClusterInDB (85–101)	2:38	1:54	4	3	1	3	2	33.3	25.0
#6 DataManager::getArtifactByTypeAndID	initArtifact (255–275) initSpecialArtifact (278–314) ticket (290–297) wiki (309–313) communication (299–306)	1:05	1:15	5	3	3	2	0	100.0	60.0
#7 RelationshipMiner::relateChangeSetToTicket	ids (295–310)	1:00	0:20	1	3	1	0	2	33.3	100.0
#8 city3d.Layout::getLayout	coords (34–42) range (73–80)	1:15	0:35	2	3	2	0	1	66.7	100.0
#9 city3d.Layout::cityBlockInitialization	cityBlocks (97–104)	0:30	0:18	1	1	1	0	0	100.0	100.0
#10 city3d.IndustrialLayout::printCityBlocks	line (354–365)	1:00	0:22	1	3	1	0	2	33.3	100.0
#11 city3d.Layout::pushBuildings	blockR (124–131)[b] subblockCenters (132–140)	0:27	0:56	2	1	1	1	0	100.0	50.0
#12 city3d.CityLayout::printCityBlocks	line (211–232)	0:08	0:20	1	3	1	0	2	33.3	100.0
#13 clustering. MatrixOperator::getConnectedComponents	component (269–277)	1:30	0:30	1	2	0	1	2	0.0	0.0
#14 clustering. SammonsProjection::iterate	No opportunities found	1:00	–	0	1	0	0	1	0.0	N/A
		Overall							50.0	63.3

All class names are preceded by package "ca.ualberta.cs.serl.wikidev."

[a] The evaluator moved the declaration of the variable of interest in order to make the statements consecutive.

[b] The initial selection of statements by the evaluator (118–131) could not be extracted, because it contained the computation of three variables. As a result, the evaluator reduced the number of selected statements (124–131).

Table 6
Precision and Recall of the proposed approach for project SelfPlanner 1.5.2.

Method	Cases found by the evaluator (method name – line numbers)	Ident. time (m:s)	Appl. time (m:s)	TO	#Cases found by the tool	TP	FN	FP	Precision (%)	Recall (%)
#1 app.domain.ManualPanel::redrawView	yearViewPanel (144, 165–168, 186–187)[a]	0:18	1:50	1	2	1	0	1	50.0	100.0
#2 app.domain.TemplatePanel::setTemplates	calcTemplates (301–318)	0:36	1:34	1	3	1	0	2	33.3	100.0
#3 app.domain.TableSquareHolder::mouseDragged	square (179–199)	0:15	0:32	1	2	1	0	1	50.0	100.0
#4 app.domain.DayPanel::initSquares	daylight (65–76)[b]	0:14	1:20	1	1	0	1	1	0.0	0.0
#5 data.Domain::clone	addClone (781–794)	0:06	0:56	1	1	0	1	1	0.0	0.0
#6 data.TaskManager::getPeriodicPartsOf	calcPeriods (409–412)	0:08	0:50	1	2	1	0	1	50.0	100.0
#7 data.TaskManager::clone	calcPastSolutions (238–243)	0:37	0:26	1	2	1	0	1	50.0	100.0
#8 data.TaskManager::sortTasks	min (487–495)	0:24	0:38	1	1	1	0	0	100.0	100.0
#9 app.HFTimeControl::addHours	daylight (76–85)[b]	0:06	1:20	1	1	0	1	1	0.0	0.0
#10 app.PeriodicTaskListPanel::setPeriodicParts	periodicPartNames (273–298)	0:38	0:34	1	1	1	0	0	100.0	100.0
#11 app.MFrame_Task::okButton_actionPerformed	validationOfTask (242–422)[c] newTask (404–419)	1:29	2:10	2	1	1	1	0	100.0	50.0
#12 app.EditLocClassPanel::save	createLocationClass (131–151)	0:26	0:58	1	2	1	0	1	50.0	100.0
#13 app.PeriodicPanel::getPrefs	period (302, 311–316)	0:24	0:42	1	1	1	0	0	100.0	100.0
#14 app.MFrame_QuickIns::okBut_actionPerformed	domain (131–185)	1:30	1:48	1	2	1	0	1	50.0	100.0
		Overall							52.4	75.0

All class names are preceded by package “gr.uom.csse.ai.myplanner.”

[a] The evaluator selected non-consecutive statements scattered through different cases of a switch statement.

[b] The evaluator reported a case of duplicated code.

[c] The initial selection of statements by the evaluator (242–383) could not be extracted, because it contained statements nested at different levels. As a result, the evaluator expanded the number of selected statements.

ing the Extract Method refactoring feature of the employed IDE (Eclipse 3.6). For each identified refactoring opportunity, the evaluator indicated the involved statements and a name for the extracted method indicating its functionality. During this process, one of the authors recorded the reported results by the evaluators and kept track of the exact time required for the identification of the refactoring opportunities in each method and the application of the corresponding refactorings. The author also recorded cases where the application of an identified refactoring by the evaluators was infeasible.

The measures required for the classification of the refactoring opportunities identified by our approach are defined as follows:

- True Positive (TP): A refactoring opportunity identified by the independent expert, and also by the proposed technique.
- False Positive (FP): A refactoring opportunity identified by the proposed technique, but not by the independent expert.
- False Negative (FN): A refactoring opportunity identified by the independent expert, but not by the proposed technique.

The results for this part of the evaluation are shown in Tables 5 and 6 for each project, respectively. For each identified refactoring opportunity by the evaluators the line numbers of the involved statements are given within parentheses. Consecutive statements are indicated with a dash between the first and the last line number. The line numbers for non-consecutive statements are separated with commas.

A first observation that can be made from the results shown in Tables 5 and 6 is that both evaluators were able to mainly identify refactoring opportunities concerning consecutive statements. Furthermore, none of the refactoring opportunities identified by the evaluators caused any duplication of statements between the remaining and the extracted method. These results indicate that a human-guided identification process can reveal only relatively trivial refactoring opportunities. Furthermore, both evaluators made selections of statements which either could not be extracted (case #11 in Table 5 and case #11 in Table 6) or required slight code modifications (case #5 in Table 5) in order to make their extraction feasible. Considering also the time required for performing identification and application activities, it becomes evident that the manual selection of statements for extraction can be a rather time-consuming and error-prone process, since it requires detailed program analysis and understanding. As a result, software maintainers could greatly benefit from semi-automated approaches like ours which identify feasible and behavior preserving refactoring opportunities and leave the decision of applying them or not on human judgment and expertise. In conclusion, our approach demonstrated a precision of 51% and a recall of 69% on average, showing that it has the ability to identify refactoring opportunities that are usually found by human experts.

5.3. Threats to validity

All types of evaluation that have been presented in the previous subsections suffer from the usual threat to external validity in the sense that a limited number of projects and evaluators have been employed. This threat limits the ability to claim that the proposed approach will be effective in other experimental settings; however, it has been partially alleviated by the fact that three projects from different domains and three different evaluators, respectively, have been employed. The availability of the proposed approach in the form of an Eclipse plug-in provides the possibility to easily extend the evaluation on other projects.

A threat to construct validity is related to the underlying philosophy for identifying cohesive code fragments having a distinct functionality. Slicing may be an ideal way for extracting the computation of a variable as a separate method; however, there might be other ways to split a method (e.g., based on conceptual criteria). In any case, the examination of program dependences is a reliable way of finding related statements within the body of a method.

6. Conclusions

The proposed approach aims at automatically identifying Extract Method refactoring opportunities which are related with the complete computation of a given variable (complete computation slice) and the statements affecting the state of a given object (object state slice). The aforementioned types of slices aim to capture code fragments implementing a distinct and independent functionality compared to the rest of the original method. Furthermore, the approach proposes a set of rules that exclude refactoring opportunities corresponding to slices whose extraction could possibly cause a change in program behavior.

The evaluation has shown that the proposed methodology is able to capture slices of code implementing a distinct and independent functionality compared to the rest of the original method and thus lead to extracted methods with useful functionality. At the same time, the identified refactoring opportunities can help significantly to resolve existing design flaws by decomposing complex methods, removing duplicated code among several methods and extracting code fragments suffering from Feature Envy. Furthermore, the identified refactoring opportunities have a positive impact on the cohesion of the decomposed methods and lead to highly cohesive extracted methods. An evaluation based on unit testing has shown that the defined behavior preservation rules can successfully exclude refactoring opportunities that could possibly cause a change in program behavior. Finally, the comparison of the refactoring opportunities identified by independent evaluators to the findings of our approach revealed a satisfactory level of precision and recall.

The proposed technique could be applied to search-based refactoring approaches (O'Keeffe and Ó Cinnéide, 2008; Harman and Tratt, 2007; Qayum and Heckel, 2009) which treat the problem of improving the design of an object-oriented system as a search problem in the space of alternative designs. The goal of these approaches is to find a sequence of refactoring transformations leading to the optimal design in terms of a fitness function (which is used to rank the alternative designs). An indicative implementation of a search-based approach based on genetic algorithms would be to consider the statements of a method as a chromosome where the value of each gene represents a method that the corresponding statement should be placed. The slice formation algorithms and behavior preservation rules of the proposed technique can be employed to form the initial population as well as to guide the genetic operators (crossover/mutation) producing the next generation population of valid chromosomes (i.e. solutions representing feasible and behavior preserving Extract Method refactorings). The selection process can be guided by a fitness function combining a set of slice-based cohesion and complexity metrics.

Acknowledgement

This work has been funded by the Research Committee of the University of Macedonia.

References

Abadi, A., Ettinger, R., Feldman, Y.A., 2008. Re-approaching the refactoring rubicon. In: Proceedings of the Second ACM Workshop on Refactoring Tools.
Aho, V., Sethi, R., Ullman, J.D., 1986. Compilers: Principles, Techniques, and Tools. Addison-Wesley.

Allen, M., Horwitz, S., 2003. Slicing Java programs that throw and catch exceptions. In: Proceedings of the 2003 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation , pp. 44–54.
Ball, T., Horwitz, S., 1993. Slicing programs with arbitrary control flow. In: Proceedings of the First International Workshop on Automated and Algorithmic Debugging , pp. 206–222.
Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D., 1993. Software complexity and maintenance costs. Communications of the ACM 36 (11), 81–94.
Bergeretti, J.-F., Carré, B.A., 1985. Information-flow and data-flow analysis of while-programs. ACM Transactions on Programming Languages and Systems 7 (1), 37–61.
Binkley, D., Gallagher, K.B., 1996. Program slicing. Advances in Computers, 43.
Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P., 2006. Tool-supported refactoring of existing object-oriented code into aspects. IEEE Transactions on Software Engineering 32 (9), 698–717.
Cimitile, A., De Lucia, A., Munro, M., 1996. A specification driven slicing process for identifying reusable functions. Software Maintenance: Research and Practice 8, 145–178.
De Lucia, A., Harman, M., Hierons, R., Krinke, J., 2003. Unions of slices are not slices. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering , pp. 363–367.
Demeyer, S., Van Rysselberghe, F., Girba, T., Ratzinger, J., Marinescu, R., Mens, T., Du Bois, B., Janssens, D., Ducasse, S., Lanza, M., Rieger, M., Gall, H., El-Ramly, M., 2005. The LAN-simulation: a refactoring teaching example. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution , pp. 123–134.
Ettinger, R., 2007. Refactoring via program slicing and sliding. Ph.D. Dissertation. University of Oxford.
Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9 (3), 319–349.
Fokaefs, M., Tansey, B., Ganev, V., Bauer, K., Stroulia, E., 2010. WikiDev 2.0: facilitating software development teams. In: Proceedings of the Fourteenth European Conference on Software Maintenance and Reengineering.
Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.
Gallagher, K.B., Lyle, J.R., 1991. Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17 (8), 751–761.
Gellerich, W., Kosiol, M., Ploedereder, E., 1996. Where does GOTO go to? Lecture Notes in Computer Science 1088, 385–395.
Gill, G.K., Kemerer, C.F., 1991. Cyclomatic complexity density and software maintenance productivity. IEEE Transactions on Software Engineering 17 (12), 1284–1288.
Harman, M., Binkley, D., Danicic, S., 2003. Amorphous program slicing. The Journal of Systems and Software 68 (1), 45–64.
Harman, M., Binkley, D., Singh, R., Hierons, R.M., 2004. Amorphous procedure extraction. In: Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation , pp. 85–94.
Harman, M., Hierons, R.M., 2001. An overview of program slicing. Software Focus 2 (3), 85–92.
Harman, M., Tratt, L., 2007. Pareto optimal search based refactoring at the design level. In: Proceedings of the Ninth Annual Conference on Genetic and Evolutionary Computation , pp. 1106–1113.
Horstmann, C.S., 2006. Object-Oriented Design and Patterns, second ed. Wiley.
Horwitz, S., Reps, T.W., Binkley, D., 1990. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12 (1), 26–60.
JDeodorant, 2010. Available from: http://www.jdeodorant.com.
Jiang, T., Harman, M., Hassoun, Y., 2008. Analysis of procedure splitability. In: Proceedings of the Fifteenth Working Conference on Reverse Engineering , pp. 247–256.
Kang, B.-K., Bieman, J.M., 1998. Using design abstractions to visualize, quantify, and restructure software. The Journal of Systems and Software 42 (2), 175–187.
Komondoor, R., Horwitz, S., 2000. Semantics-preserving procedure extraction. In: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages , pp. 155–169.
Komondoor, R., Horwitz, S., 2003. Effective, automatic procedure extraction. In: Proceedings of the Eleventh IEEE International Workshop on Program Comprehension , pp. 33–42.
Korel, B., Laski, J., 1988. Dynamic program slicing. Information Processing Letters 29 (3), 155–163.
Kumar, S., Horwitz, S., 2002. Better slicing of programs with jumps and switches. In: Proceedings of the Fifth International Conference on Fundamental Approaches to Software Engineering , pp. 96–112.
Lakhotia, A., Deprez, J.-C., 1998. Restructuring programs by tucking statements into functions. Information and Software Technology 40 (11–12), 677–690.
Landi, W., Ryder, B.G., Zhang, S., 1993. Interprocedural modification side effect analysis with pointer aliasing. In: Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation , pp. 56–67.
Lanubile, F., Visaggio, G., 1997. Extracting reusable functions by flow graph-based program slicing. IEEE Transactions on Software Engineering 23 (4), 246–259.
Larsen, L., Harrold, M.J., 1996. Slicing object-oriented software. In: Proceedings of the Eighteenth International Conference on Software Engineering , pp. 495–505.
Liang, D., Harrold, M.J., 1998. Slicing objects using system dependence graphs. In: Proceedings of the Fourteenth IEEE International Conference on Software Maintenance , pp. 358–367.
Maruyama, K., 2001. Automated method-extraction refactoring by using block-based slicing. In: Proceedings of the Symposium on Software Reusability , pp. 31–40.
Meyers, T.M., Binkley, D., 2007. An empirical study of slice-based cohesion and coupling metrics. ACM Transactions on Software Engineering and Methodology 17 (1), Article 2.
Murphy, G.C., Kersten, M., Findlater, L., 2006. How are Java software developers using the eclipse IDE? IEEE Software 23 (4), 76–83.
Murphy-Hill, E., Parnin, C., Black, A.P., 2009. How we refactor, and how we know it. In: Proceedings of the Thirty-First International Conference on Software Engineering , pp. 287–297.
Ohata, F., Inoue, K., 2006. JAAT: Java alias analysis tool for program maintenance activities. In: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing , pp. 232–244.
O'Keeffe, M., Ó Cinnéide, M., 2008. Search-based refactoring for software maintenance. The Journal of Systems and Software 81 (4), 502–516.
Ott, L.M., Thuss, J.J., 1989. The relationship between slices and module cohesion. In: Proceedings of the Eleventh International Conference on Software Engineering , pp. 198–204.
Ott, L.M., Thuss, J.J., 1993. Slice-based metrics for estimating cohesion. In: Proceedings of the First International Software Metrics Symposium , pp. 71–81.
Qayum, F., Heckel, R., 2009. Local search-based refactoring as graph transformation. In: Proceedings of the First International Symposium on Search Based Software Engineering , pp. 43–46.
Refanidis, I., Alexiadis, A., 2008. SelfPlanner: planning your time! In: ICAPS 2008 Workshop on Scheduling and Planning Applications.
Stamelos, I., Angelis, L., Oikonomou, A., Bleris, G.L., 2002. Code quality analysis in open source software development. Information Systems Journal 12, 43–60.
Tip, F., 1995. A survey of program slicing techniques. Journal of Programming Languages 3 (3), 121–189.
Tonella, P., 2003. Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Transactions on Software Engineering 29 (6), 495–509.
Tsantalis, N., Chatzigeorgiou, A., 2009. Identification of extract method refactoring opportunities. In: Proceedings of the Thirteenth European Conference on Software Maintenance and Reengineering , pp. 119–128.
Weiser, M., 1984. Program slicing. IEEE Transactions on Software Engineering 10 (4), 352–357.

Nikolaos Tsantalis received the BS, MS and PhD degrees in applied informatics from the University of Macedonia, Greece, in 2004, 2006 and 2010, respectively. He is currently a Postdoctoral Fellow at the Department of Computing Science, University of Alberta, Canada. His research interests include design pattern detection, identification of refactoring opportunities, and design evolution analysis. He is a member of the IEEE and the IEEE Computer Society.

Alexander Chatzigeorgiou is an assistant professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom, Greece, as a telecommunications software designer. His research interests include object-oriented design, software maintenance, and metrics. He is a member of the IEEE.