A Lightweight Visualization of Interprocedural Data-Flow Paths for Source Code Reading

Takashi Ishio, Shogo Etsuda and Katsuro Inoue
Graduate School of Information Science and Technology
Osaka University
1-5 Yamadaoka, Suita, Osaka, Japan
{ishio, s-etuda, inoue}@ist.osaka-u.ac.jp

***Abstract*—To understand the behavior of a program, developers must read source code fragments in various modules. For developers investigating data-flow paths among modules, a call graph is too abstract since it does not visualize how parameters of method calls are related to each other. On the other hand, a system dependence graph is too fine-grained to investigate interprocedural data-flow paths. In this research, we propose an intermediate-level of visualization; we visualize interprocedural data-flow paths among method parameters and fields with summarized intraprocedural data-flow paths. We have implemented our visualization as an Eclipse plug-in for Java. The tool comprises a lightweight data-flow analysis and an interactive graph viewer using fractal value to extract a small subgraph of data-flow related to variables specified by a developer. A case study has shown our visualization enabled developers to investigate more data-flow paths in a fixed time slot. In addition, we report our lightweight data-flow analysis can generate precise data-flow paths for 98% of Java methods.**

***Index Terms*—data-flow analysis, static analysis, software visualization, program comprehension.**

I. Introduction

Many software developers spend much of their time to investigate source code [1]. Some estimate that understanding source code accounts for more than half of the total cost of maintenance [2]. Program understanding is difficult partly because a single functionality is implemented by a complex interaction of modules, e.g. methods and classes in Java. To investigate an interaction of modules, developers often search code fragments related to their current task, using explicit dependencies including control-flow and data-flow among modules [3].

While developers have to investigate data-flow paths, existing source code viewers focus on method call as a main relationship [4], [5], [6], [7]. To analyze interprocedural data-flow using these tools, developers have to manually investigate intraprocedural data-flow paths among method parameters and fields on source code. On the other hand, system dependence graph (SDG) [8] represents the details of control-flow and data-flow of a program. Although SDG includes sufficient data-flow information for developers, a SDG for a typical program is too large to visualize [9].

In this paper, we propose an intermediate-level data-flow visualization for developers to investigate interprocedural data-flow paths. We visualize data-flow paths among formal parameters and actual parameters of method invocation and field access. To exclude the implementation details of a method from a data-flow graph, we summarize intraprocedural data-flow paths between formal/actual parameters as direct edges between the vertices. The resultant data-flow graph shows how method parameters and fields are related to each other.

Our data-flow visualization tool comprises two components: data-flow analyzer and visualizer. The analyzer employs control-flow-insensitive data-flow analysis. In other words, we simply connect data-flow edges from all assignment statements of a variable to all reference statements of the variable. The analysis is applicable to a part of a system whose control-flow information is not available. In addition, the analysis is efficient, consistent with existing tools, and precise enough for developers to understand data-flow paths.

Our visualizer is an interactive graph viewer integrated into Eclipse. When a mouse button is clicked on a method name or a variable name in a text editor, inter-procedural data-flow paths related to the selected entity are visualized in the graph viewer. To keep the size of a visualized graph manageable, we have employed fractal value [10]. Our visualization tool automatically stops graph traversal at vertices which have a large number of data-flow edges so that developers can decide whether they continue investigating the data-flow paths or not.

We have evaluated the effectiveness of our tool for program understanding and the performance of the tool. In the experiment, we have assigned two program understanding tasks on JEdit, a Java text editor, to 16 participants. The result shows the participants using Eclipse enhanced with our data-flow graph view could more completely investigate data-flow paths than the participants using a regular Eclipse. Although our data-flow analysis may generate infeasible edges because of flow-insensitivity, we have found no problems caused by the infeasible data-flow edges. In addition, we have shown that only 1.9% of methods of 8 Java software include infeasible intraprocedural data-flow paths. This result shows that a lightweight data-flow analysis is practical enough for Java visualization tools.

This paper makes the following contributions:

- We have defined a flow-insensitive inter-procedural data-flow analysis. The analysis is an approximation of program slicing that is applicable to a part of a system whose control-flow information is not available. It is consistent with program analysis tools integrated in Eclipse JDT.

- We have proposed a visualization that focuses on inter-procedural data-flow paths. We have summarized intra-procedural data-flow paths to exclude the implementation details from a graph view.
- We have shown the effectiveness of the inter-procedural data-flow information in the experiment using a prototype working with Eclipse and an existing graph layout tool.

The remainder of the paper is structured as follows. In Section II, we present related work. In Section III, we describe our data-flow analysis method. The result of experiments is shown in Section IV. Section V presents the conclusions and future work of our research.

II. Related Work

A. Software Visualization for Program Comprehension

There are two approaches to software visualization for program comprehension: one starts from an overview of a program and the other starts from an implementation detail [11]. Our approach and several tools are involved in the latter approach.

Method call is important to investigate source code fragments. Fluid Source Code View [6] is a source code viewer that helps developers understanding an interprocedural control-flow path. While a regular source code editor such as Eclipse shows each file in a split view, this tool shows multiple methods of multiple classes in a single view by inlining called methods into the body of their callers. This tool is good for investigating the detailed behavior of a method that strongly depends on other methods. On the other hand, this approach does not support backward traversal of method calls to investigate how a method depends on its callers.

The relationships among source code fragments should be visualized so that developers can select an appropriate source code location to be investigated. Code Bubbles [4] is a unified viewer of source code and its related documents. The tool focuses on the user interface based on the bubble metaphor. The tool shows a number of source code fragments and their method call relationships so that developers can keep the progress of their investigation tasks. On the other hand, the tool does not analyze the implementation details of each code fragment. To investigate data-flow paths, developers must open and read source code fragments. Our research visualizes summarized data-flow paths in source code so that developers can choose source code fragments to be investigated and ignore irrelevant source code.

A visualization tool should reflect the structure of source code fragments. DA4Java [7] shows an overview of Java source code as a nested graph. Vertices in the graph represent source code entities which are packages, classes, methods and fields. Edges in the graph represent class inheritance/subtyping, method calls and field access. DA4Java represents a class as a node which contains vertices representing methods and fields belonging to the class. We have followed the idea of the nested visualization for data-flow graph visualization.

B. Program Slicing

Program slicing [12] is a well-known technique to extract a program slice, or a set of program statements related to slicing criteria selected by a developer. A program slice is computed by backward traversal of System Dependence Graph (SDG) from slicing criteria [8].

SDG is a directed graph whose vertices represent statements of a program. Its directed edges represent data and control dependencies. A data dependency is a relation between an assignment and a reference of a variable. When all of the following conditions are satisfied, we say that a data dependency from statement s_1 to statement s_2 exists:

1) s_1 assigns a value to v, and
2) s_2 refers to v, and
3) At least one execution path from s_1 to s_2 without re-defining v exists.

The third condition depends on a control-flow graph of a method containing s_1 and s_2.

A control dependency is a relation between a conditional statement and a controlled statement. We say that a control dependency from statement s_1 to statement s_2 exists if:

1) s_1 is a conditional predicate, and
2) The result of s_1 determines whether s_2 is executed or not.

The definition of control dependence relation also depends on a control-flow graph.

Program slicing is effective to investigate the detailed behavior of a program for debugging [13]. SDG is also employed to support source code reading, e.g. to locate features in source code [5] and to search similar code fragments [14].

Although SDG includes sufficient data-flow information for developers, there are still two challenges to applying program slicing to program comprehension tasks. Firstly, program slicing is not always applicable since the complete set of source code for a system is not always available. For example, components of web applications may be controlled by an external framework. In this research, we approximate program slicing using a flow-insensitive analysis so that we can analyze a part of a system without control-flow information. Secondly, visualizing a SDG for a typical program is difficult because of its large amount of vertices [9]. In this research, we visualize only statements related to interprocedural data-flow such as method invocations and field access. In addition to a reduced data-flow graph, we introduce an interactive viewer to visualize a subgraph closely related to a selected variable.

C. Lightweight Analysis

Our approach is an approximation of data dependence analysis. Jász [15] proposed Static Execute After/Before dependencies as another approximation of data dependencies. The approach is a control-flow-based approximation without data-flow analysis, while our approach is a data-flow analysis without control-flow analysis. The control-flow-based approximation is effective for data-flow paths involved in a sequential procedure. However, the approach is not good at handling a

```
static int max ( int x, int y ) {
    int result = y;
    if ( x > y )
        result = x;
    return result;
}
```

Fig. 1. An example procedure

message loop involved in GUI and server applications, since such a message loop connects control-flow paths among all functionalities invoked by messages. Our analysis can extract data-flow paths in such applications.

Nguyen [16] has proposed a flow-insensitive data-flow analysis for mining source code patterns. The analysis constructs a directed acyclic graph named *groum* whose nodes represent method calls and field access in a Java method. A data dependency edge between two nodes is generated if the two nodes share at least a common variable. Since a groum ignores data-flow paths caused by loop structures (e.g. `while` or `for` statements) it is not directly applicable to program comprehension.

III. Data-Flow Visualization

We propose a data-flow visualization for developers to investigate interprocedural data-flow paths. To exclude the implementation details of a method, we summarize intraprocedural data-flow paths, while we keep interprocedural data-flow paths. We have defined Variable Data-Flow Graph (VDFG) as a Java program model. Our visualization tool is an interactive viewer for VDFG working with Eclipse JDT.

Our data-flow analysis is control-flow-insensitive, context-insensitive and object-insensitive. It should be noted that our data-flow analysis is consistent with program analysis features provided by Eclipse JDT. For example, "Mark Occurrences" feature shows all assignment and reference statements of a selected variable. This feature is a control-flow-insensitive approximation of data dependency. Eclipse JDT also provides a cross-reference feature for method call and field access. The cross-reference tool depends on class hierarchy; it is context-insensitive and object-insensitive.

A. Variable Data-Flow Graph

We have defined VDFG as a directed graph to represent data-flow paths in a Java program. VDFG includes both variables and instructions as vertices since variables are important to investigate data-flow paths. Our analysis is control-flow insensitive; therefore, we do not construct control-flow graphs for a target program. To approximate data and control dependencies without control-flow graphs, we have defined approximated data and control dependencies as follows.

- If a statement s_1 assigns a value to v and another statement s_2 refers to v, then s_2 depends on s_1 via v.
- A statement is controlled by its enclosing control statement such as `if` and `while`.

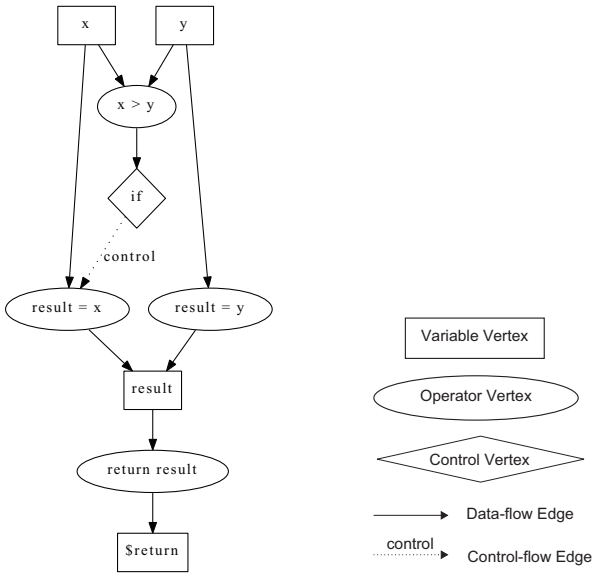


Fig. 2. The variable data-flow graph of the procedure max in Figure 1

Compared with data dependency for program slicing described in Section II-B, we have removed the third condition. The difference may generate infeasible data dependency but miss no traditional data dependency. Approximated control dependency is based on syntax tree instead of control-flow graph.

To describe VDFG, we use an example code fragment shown in Figure 1. The code fragment is a method named `max`. The method takes two parameters `x` and `y` and returns the larger one. Figure 2 shows the VDFG of the method.

VDFG comprises three kinds of vertices and two kinds of edges as follows.

- **A variable vertex** corresponds to a variable or a literal. Variables include a local variable, a formal parameter of a method (formal-in), a return value of a method (formal-out), an actual parameter of a method call (actual-in), a return value of a method call (actual-out), an instance variable (field), a class variable (static field) and a literal. In Figure 2, a rectangle indicates a variable vertex. Vertices `x`, `y` and `result` respectively correspond to the local variables "`int x`", "`int y`" and "`int result`". Vertex "`$return`" represents the return value of the method.
- **An operator vertex** corresponds to an operator of an expression. An operator vertex has one or more incoming data-flow edges from vertices corresponding to operands. If the resultant value of an operator is assigned to a variable or used by another operator, the operator vertex has an outgoing data-flow edge. In Figure 2, ellipse vertices represent operators. For example, "`x > y`" is a comparison operator takes as inputs two edges from `x` and `y` and give the resultant value to the following `if` statement.

  We treat method calls, field access and array access as

special operators whose operands, e.g. a receiver object and parameters, are represented by variable vertices.

- **A control vertex** corresponds to a control statement such as `if` and `while`. A control vertex has an incoming data-flow edge representing the conditional expression and outgoing control edges to call-site vertices and operator vertices controlled by the condition. In Figure 2, assignment "`result = x;`" is controlled by the enclosing `if` statement. Therefore, vertex "`if`" has a control edge to operator vertex "`result = x`".
- **A data-flow edge** is an edge representing a data-flow. An intraprocedural data-flow edge connects between a variable vertex and an operator vertex. An interprocedural data-flow edge connects a data-flow for a method parameter or a field. In this paper, a data-flow edge is represented by a solid arrow without a label.
- **A control-flow edge** is an edge from a control vertex to an operator vertex. This edge represents an approximated control dependency. In this paper, a control edge is represented by a dotted arrow.

A VDFG is constructed by the following steps.

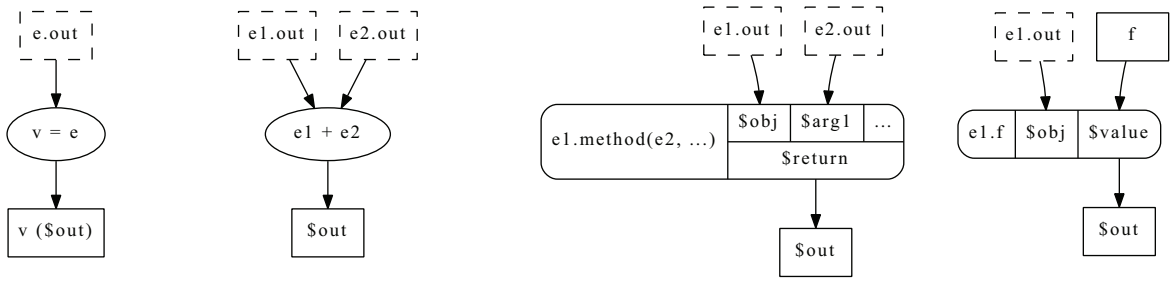
1) Create a variable vertex for each variable declaration.
2) Translate each statement into vertices and edges.

The first step processes fields, local variables and literals. All instance fields and class fields in the target program are translated into variable vertices. Each field corresponds to a single vertex, since our analysis is object-insensitive. For each method, local variables are translated into variable vertices. Literals, `this` and the return value (formal-out) of a method are also regarded as local variables of the method.

The second step is a rule-based transformation of abstract syntax tree (AST). Figure 3 shows four rules frequently used in the transformation. A rule specifies AST nodes to be translated and a set of vertices connected by edges generated for the AST nodes. A rule includes two special vertices; "`e.out`" is a placeholder for a sub-expression, "`$out`" is a vertex that replaces such a placeholder, respectively. For example, statement "`result = x;`" is an assignment statement that matches rule (a). The rule indicates that an assignment statement is represented by three vertices and two edges. We create a new operator vertex with label "`result = x`" and connect a data-flow edge from the vertex to variable vertex "`result`". In this expression, the right hand side `e` is a reference to variable "x". Hence, the placeholder "`e.out`" is replaced with variable vertex "x". If the right hand side of the assignment were an expression "`x + y`", the operator vertex for the assignment would have an incoming edge from another operator vertex "`x + y`" generated by rule (b).

Rules to process major Java language constructs are listed as follows.

- A compound operator is decomposed to primitive operators. A pre-increment operator and a post-increment operator such as "x++" and "`--x`" are regarded as statements "`x = x + 1`" and "`x = x - 1`". A compound assignment operator, e.g. "+=", is regarded as a combination of a simple assignment operator and a regular binary operator. For example, "`a += 2;`" is the same as "`a = a + 2;`".
- Control statements are translated into control edges. `if`, `while`, `for`, `do-while` and `switch` statements are simply translated into control vertices. A control vertex has an incoming data-flow edge from its control predicate and outgoing edges to all operator vertices in the controlled block. Statements `break` and `continue` are ignored.
- A method call expression is translated into a set of method call vertices. We create a method call vertex for each method that may be invoked by the method call according to dynamic binding. Dynamic binding is resolved by Class Hierarchy Analysis [17].
  Each method call vertex comprises a single operator vertex and a set of variable vertices representing actual parameters as shown in rule (c) of Figure 3. We connect interprocedural data-flow edges from each of actual-in parameter vertices to its corresponding formal-in parameter vertex. A data-flow edge for a return value is connected from the formal-out vertex of a called method to the actual-out vertex of a call site.
  According to the above rule, a `static` method without parameters cannot have edges to the callers. To solve the problem, we add a pseudo parameter "`called`" to a method to visualize a link from a call-site. Since the pseudo vertices are added for visualization, they have no effect on graph traversal.
- A return statement "`return e;`" in a method is regarded as an assignment from the expression `e` to the formal-out vertex of the method.
- A constructor of an object is regarded as a method call taking as input a new object. We represent the new object initialized by the constructor call by a pseudo literal "`new C`". The literal is necessary because a constructor of class C may be called to initialize an instance of a subclass of C.
- A field access instruction is regarded as similar to a method call. As shown in rule (d) of Figure 3, a field reference vertex has two parameters. One of the parameters is an actual-in vertex that specifies an owner of a field, the other is an actual-out vertex that receives a value from the field, respectively. A field assignment has two actual-in vertices; one specifies an object and the other specifies a value for a field. Since our analysis is object-insensitive, actual-in vertices to specify objects are not connected to any other vertices.
- An array access is also regarded as a method invocation. An array read instruction is translated into an operator vertex associated with two actual-in vertices and an actual-out vertex. The two actual-in vertices represent an array instance and an index, the actual-out vertex represents a value, respectively. An array write instruction is translated into an operator vertex associated with three actual-in vertices that represent an array instance, an



(a) Assignment "v=e" (b) Binary operator "e1+e2" (c) Method invocation "e1.method(e2, ...)" (d) Field reference "e1.f"

Fig. 3. VDFG construction rules

```
Code 1:            Code 2:
  int X;             int X;
  int Y;             int Y;
  int Z;             int Z;
  X = Y;             Y = Z;
  Y = Z;             X = Y;
```

VDFG for Code 1 and Code 2:

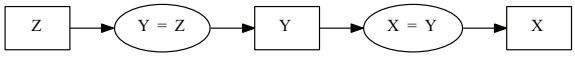


Fig. 4. Two code fragments that result in the same VDFG

index and a value. An instruction to obtain the length of an array (`array.length`) is translated into an operator vertex that takes as input an array instance and outputs a value.

Our analysis does not support several aspects of Java language as follows.

- We do not analyze library classes whose source code is not available. A method call to a library class is not connected to the called methods.
- The statements in a `catch` block are simply translated into a VDFG without a special rule.
  We ignored data-flow paths for an exception object from a method call or a `throw` statement to a `catch` block, although we could represent such a data-flow in VDFG. Developers who investigate exceptional control-flow can use another visualization tool such as Flow View specialized for exception handling [18].
- We did not take multi-threaded execution into account as regular control-flow paths. VDFG includes data dependencies between threads if the threads communicate by variables. A `synchronized` block is represented as a control vertex that takes as input an expression and controls statements in the block.

After a Java program is translated into a VDFG, we add summary edges that represent intraprocedural data-flow paths as direct edges. To visualize relationships between method parameters and fields, we compute all data-flow paths from formal-in and actual-out vertices to formal-out and actual-in vertices. The former (formal-in and actual-out) vertices are entry points of data-flow from the outside of a method. The latter (formal-out and actual-in) vertices are exit points of the method. If a transitive path of intraprocedural data-flow edges exists between vertices, we add a summary data-flow edge between the vertices. If no data-flow path exist between the vertices but a path including control-flow edges exists, we add a summary control-flow edge between the vertices.

The rule-based transformation uses only ASTs of source files, a list of variable declarations, a class hierarchy tree and a call graph for the whole program. While control-flow insensitivity reduces time and memory cost for the analysis, control-flow insensitivity may extract infeasible paths. An example is shown in Figure 4. On the left side, there is no data dependency between those two assignments "X=Y" and "Y=Z". The new value of X has no effect on the second statement. On the right side, the code has a data dependency because the new value of Y is used by the second statement. In our approach, both code result in the same graph including two data-flow edges from Z to Y and from Y to X. For the left code fragment, a transitive path from Z to X is infeasible. This is a shortcoming of our approach.

B. Graph Traversal and Visualization

Using VDFG, we can extract data-flow paths related to a variable selected by a developer. As SDG-based program slicing [8] constructs a system dependence graph and uses it to compute multiple slices, we construct a VDFG for a Java program and use it to extract multiple subgraphs for visualization. Given a variable selected by a developer, we execute backward traversal from the vertex corresponding to the variable in order to extract a set of variables and statements that affect the variable. Similarly, forward traversal extracts a set of variables and statements affected by the selected variable.

Since a VDFG represents the whole Java program, a simple application of graph traversal results in a too large subgraph as similar to program slicing [19]. To keep the size of a visualized

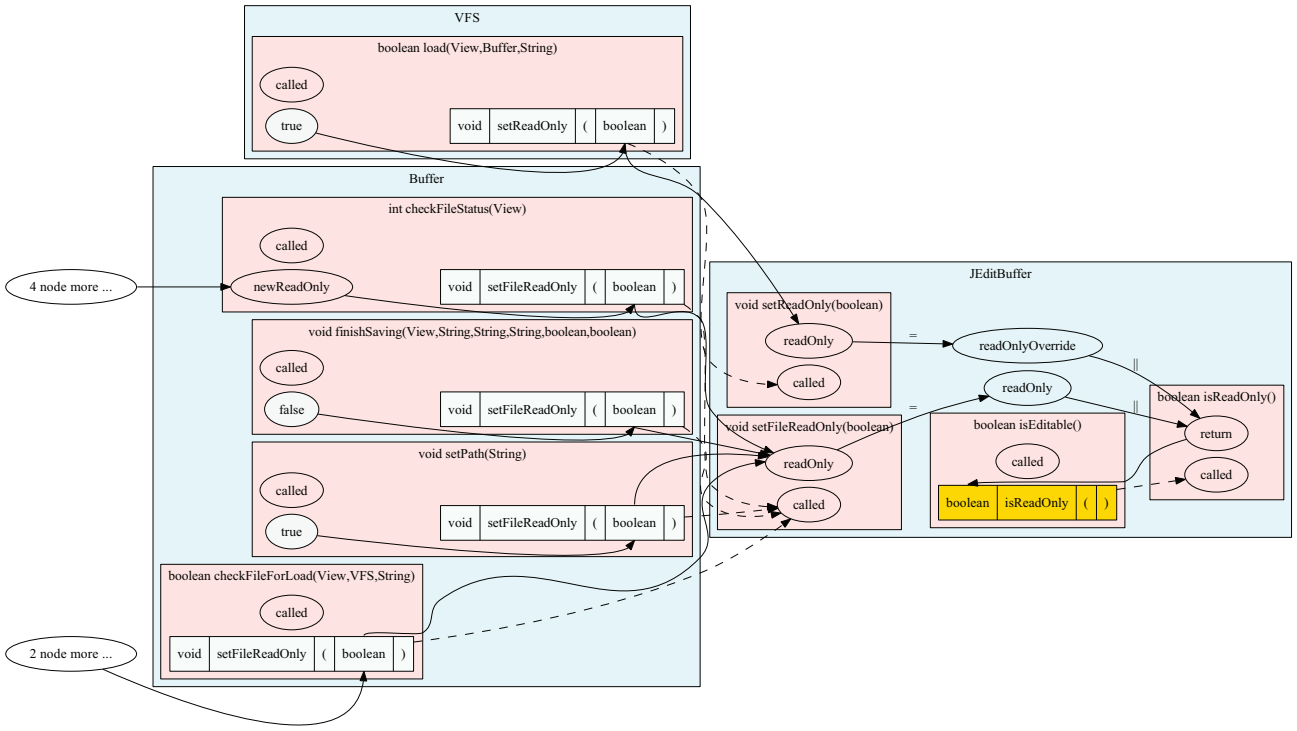
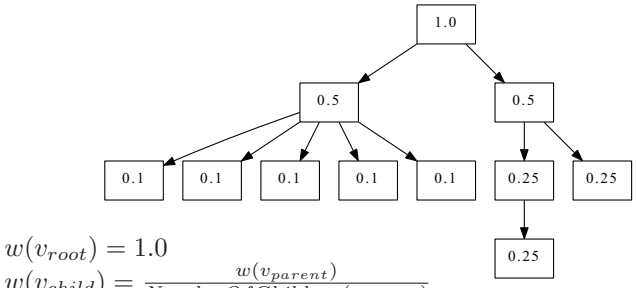


Fig. 6. A screenshot of a Variable Data-flow Graph extracted from JEdit



$$w(v_{root}) = 1.0$$
$$w(v_{child}) = \frac{w(v_{parent})}{NumberOfChildren(v_{parent})}$$

Fig. 5. Fractal values

subgraph manageable, we have employed fractal value [10]. A fractal value $w(v)$ is a weight for a tree node v. The fractal value for the root node v_{root} is defined as $w(v_{root}) = 1.0$. A fractal value of a node is divided to the children as follows.

$$w(v_{child}) = \frac{w(v_{parent})}{NumberOfChildren(v_{parent})}$$

Figure 5 shows an example of this computation. We start a graph traversal with the fractal value 1.0 and terminate the traversal at nodes whose fractal values are less than threshold 0.04 that is experimentally determined. Since the sum of the fractal values of leaf nodes is always 1.0, the threshold 0.04 indicates at most 25 leaf nodes are included for visualization.

An important feature of fractal value is that the fractal value of a node is the same as its parent if there are no sibling nodes. We continue a graph traversal through a variable if the value of the variable is determined by another single variable, e.g. a parameter incoming from a single caller. We extract a subgraph so that developers can skip such a simple data-flow path on the visualized graph without reading source code. On the other hand, backward traversal stops at methods called by a large number of other methods, e.g. setter/getter methods. Similarly, forward traversal stops at return values and fields used by a large number of methods. In these cases, we add a pseudo node labeled "`more ...`" to the extracted subgraph in order to indicate the graph traversal is terminated at the node. Developers can start another graph traversal from the method by selecting one of the methods.

After a subgraph is extracted, we visualize the subgraph on the screen. Since we focus on interprocedural data-flow paths, the graph view visualizes method call, field access and their parameter vertices connected with summary edges. Other vertices and edges are omitted in the graph view by default.

Our VDFG view is implemented as an Eclipse plug-in. To help developers to investigate source code, the VDFG view interacts with a text editor; a mouse click on a method name or a field name triggers a graph traversal as follows.

- A click on a method declaration shows backward data-flow paths from all parameters of the method and forward data-flow paths from the return value vertex of the method.

- A click on a method call site shows forward data-flow paths from the actual parameters and backward data-flow paths from the return value of the method call. In other words, the graph view shows all data-flow paths related to the selected method call.
- A click on a field declaration or a field reference shows both forward and backward data-flow paths from the vertex corresponding to the field.

Figure 6 shows a subgraph of a VDFG of JEdit visualized when a developer clicked on a method call instruction `isReadOnly()` involved in `isEditable` method.

Our visualization approach is a hierarchical view as similar to DA4Java [7]. A class is represented by a rectangle which contains its methods and fields. Each method involved in the subgraph is also represented as a rectangle including vertices representing the instructions of the method. We have excluded the other methods and fields that are not involved in the subgraph from the graph view, since such methods and fields are irrelevant to the visualized subgraph. We have used Graphviz [20] for graph layout simply because of its availability. Any other graph layout tools are also applicable.

The highlighted (yellow) call-site in Figure 6 has an incoming data-flow edge from `isReadOnly` method. The return value depends on `readOnlyOverride` and `readOnly` fields. The fields are assigned by `setReadOnly` and `setFileReadOnly` methods. `setReadOnly` method is called by `load` method of a file system class, `setFileReadOnly` method is called by 4 methods in a buffer class, respectively. Using the graph, we can infer how the return value of `isReadOnly` method is determined without reading source code.

The graph view provides hyperlinks to Java source code so that developers can quickly confirm the implementation details of a method that are omitted in the graph view. Developers can move to a method declaration, a method call site or a field declaration in a Java editor by selecting a vertex in the graph view.

The graph view also allowed developers to execute an additional forward/backward search on VDFG if necessary. In addition, developers can keep a copy of a visualized graph and restart the search from the preserved copy. Using these features, developers can incrementally explore a large VDFG.

IV. Experiment

A. Case Study: Source Code Reading

To evaluate the effectiveness of our visualization, we had 16 participants work on program understanding tasks. 12 participants are graduate students studying software engineering. They are familiar with Java since they implement tools for their research in Java. 4 participants are developers working in a software company. They develop some package software or enterprise applications written in Java.

We have assigned two tasks for each participant; one task using Eclipse enhanced with the VDFG plug-in and another task using a regular Eclipse 3.4. We have compared the data-flow paths investigated by participants in a limited time slot.

We have chosen JEdit, an open source text editor, as a target program since no participants had known its source code. JEdit has a functionality to sound a beep when JEdit cannot execute an action specified by a user. Such a code fragment frequently appears in JEdit. We have randomly selected two source code locations in JEdit 4.3pre11:

- Task A: EditAbbrevDialog.java, Line 153

```
public void actionPerformed(...) {
  ...
  if (editor.getAbbrev() == null  ||
      editor.getAbbrev().length() == 0) {
    getToolkit().beep();  // 153
    return;
  }
  ...
```

- Task B: JEditBuffer.java, Line 2038

```
public void undo(TextArea textArea)
{
 ...
 if(!isEditable())
 {
  textArea.getToolkit().beep(); // 2038
  return;
 }
 ...
```

We have asked the participants to identify conditions which sound a beep and explain how the `if` statements are affected by the external environment such as actions conducted by a user, the status of GUI components and the status of a file system. We have chosen `if` statements since understanding precise `if` conditions is important for bug fix tasks [21]. In Task A, `beep` is called if a text input widget have no text that must be specified by a user to execute the command. In Task B, `beep` is called if file editing is not permitted by a file system. To explain the conditions for Task A and Task B, 8 methods and 13 methods in JEdit must be investigated, respectively. We have asked the participants to write down their answers on a paper.

The time for a task is limited to 30 minutes in order to observe the detailed behavior of a participant, in addition to the limited time for the experiment in the company. The time does not include VDFG construction, since we would like to compare only the effectiveness of the tool. We have observed the activities of each participant with a video camera.

The tasks are sequentially assigned. For each participant, we had a 30 minute session to explain the concept of VDFG and give a trial task to learn our tool. After the introduction, we have assigned a task. 30 minutes later, we have interrupted his or her investigation activity and assigned the other task. Some participants who finished the task within 30 minutes used the remaining time for verifying their answers.

To evaluate the correctness of an answer, we have determined a correct answer for each task in prior to the experiment. The correct answers are defined as data-flow paths. For example, Figure 7 is a very simplified version of the answer of Task B. The graph has three data-flow paths from different methods:

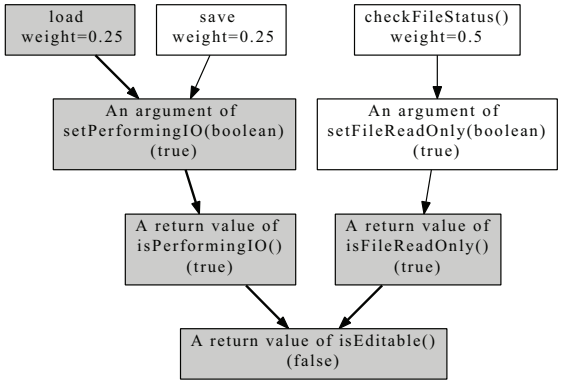


Fig. 7. An example of score computation. The graph is a simplified version of the answer of Task B.

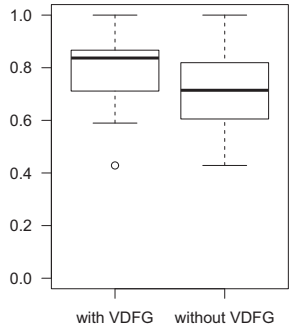


Fig. 8. Comparison of data-flow paths identified by developers

`load`, `save` and `checkFileStatus`. We have computed a score for each answer using the following function:

$$Score(A) \quad = \quad \sum_{v \in V} w(v) \frac{|path(v,m) \cap A|}{|path(v,m)|}$$

In this function, A is a set of data-flow edges included in an answer of a participant. V is a set of source vertices of data-flow paths in the correct answer. m is the target method where `beep` is called. $w(v)$ is a weight value for each vertex determined by fractal values [10] described in Section III-B. If an answer included a gray part of Figure 7, the answer included 3 of 3 edges for `load`, 2 of 3 edges for `save` and 1 of 3 edges for `checkFileStatus`. The resultant score is $0.25 \times \frac{3}{3} + 0.25 \times \frac{2}{3} + 0.5 \times \frac{1}{3} = 0.583$.

Table I shows the assigned tasks and the scores for each participant. We have summarized the resultant scores in Figure 8. The average score of participants working with VDFG is 0.79, while the average score of participants working with a regular Eclipse is 0.71. Wilcoxon signed-rank test shows the difference is statistically significant (p=0.03). The visualization of VDFG enabled participants to more completely discover the data-flow paths implicated in the task.

We have observed that a visualized VDFG is frequently used by participants. After the participants have selected a method or a variable to obtain a VDFG, they have repeatedly selected a return value vertex or a method parameter vertex in the graph to read source code. Without VDFG, participants have to identify data-flow paths using a call graph view and several search functionalities provided by Eclipse.

We have also observed participants used VDFG to recognize where they have investigated. When a value of a variable depends on two or more methods, participants have to investigate one of data-flow paths and come back to the variable. The participants using VDFG could exhaustively investigate such data-flow paths. On the other hand, participants working without VDFG are hard to identify their previous source code locations. Several participants have lost their previous locations and restarted their investigation from the starting points of the tasks, since they did not explicitly manage the progress of source code reading.

Although 13 of 16 participants performed better using VDFG, three participants P8, P10 and P16 performed better using Eclipse. P16 has the worst score for VDFG because he did not write down several data-flow paths in his answer, even though he has actually investigated the paths. Such an error could be eliminated if the tool automatically recorded the history of investigated files. On the other hand, P8 and P10 answered the correct data-flow using a regular Eclipse. They are familiar with Eclipse because they use Eclipse for daily development tasks. They often clicked on a variable name to highlight the variable and quickly scrolled the editor by a mouse wheel to identify the data-flow paths related to the variable. P8 and P10 have also frequently used keyboard shortcut to obtain method call and field access relationships. Although our VDFG visualized the same information they have referred to, the behavior of two participants are much faster on the regular Eclipse.

Participants have found no problems caused by precision of our data-flow analysis. As described in Section III-A, VDFG may include infeasible edges. Indeed, an infeasible path is caused by the following code fragment for Task A:

```
dialog = new EditAbbrevDialog(..., abbrev, ...);
abbrev = dialog.getAbbrev();
```

Variable `abbrev` in the first line does not depend on the second line. Our VDFG visualized a summary data-flow edge from the return value of `getAbbrev` to a parameter of the constructor call `EditAbbrevDialog`. However, participants have read the source code in a few seconds and simply ignored the infeasible edge.

Our data-flow analysis simply ignores library classes. For example, a method call `JTextField.setText` has no data-flow path to the return value of a method call `JTextField.getText`. However, the participants have never complained the lack of such data-flow paths. Instead, they have investigated data-flow paths of a `JTextField` variable to find the caller methods of `setText` and `getText`.

TABLE II
THE NUMBER OF VARIABLES AND METHODS INCLUDING INFEASIBLE DATA-FLOW PATHS

Software	#class	#method	#m-inf-edge		#m-inf-path		#var	#var-inf		#summary	#summary-inf	
Apache Ant 1.7.0	2342	18928	1720	(9.1%)	332	(1.8%)	52083	3404	(6.5%)	328785	1383	(0.4%)
Apache Batik 1.6	3608	23479	2512	(10.7%)	224	(1.0%)	71088	5803	(8.2%)	516648	4884	(0.9%)
Apache Tomcat 6.0.14	2297	23478	2393	(10.2%)	817	(3.5%)	73176	4057	(5.5%)	484731	4285	(0.9%)
Azureus 3.0.3.4	5378	28943	1814	(6.3%)	665	(2.3%)	90066	3144	(3.5%)	500926	3413	(0.7%)
HSQLDB 1.8.1	381	4876	727	(14.9%)	3	(0.0%)	14465	1380	(9.5%)	98767	88	(0.0%)
JEdit 4.3pre11	1132	7751	704	(9.1%)	304	(3.9%)	24623	1034	(4.1%)	148856	1119	(0.8%)
Spring Framework 2.5.5	4529	28476	1159	(4.1%)	543	(1.9%)	73130	1580	(2.2%)	289827	1803	(0.6%)
Univ Web App	4456	37521	2864	(7.6%)	667	(1.8%)	103261	4794	(4.6%)	533905	2917	(0.5%)
Total	23971	173241	13892	(8.0%)	3351	(1.9%)	501405	25195	(5.0%)	2901355	19892	(0.7%)

TABLE I
THE RESULTANT SCORES. "/VDFG" INDICATES THAT A PARTICIPANT USED VDFG FOR THE TASK.

Participant	1st Task	2nd Task	Score with VDFG	Score w/o VDFG
P1 (Student)	A/VDFG	B	0.857	0.781
P2 (Student)	A/VDFG	B	1.000	0.723
P3 (Student)	A/VDFG	B	1.000	0.621
P4 (Industry)	A/VDFG	B	0.857	0.652
P5 (Student)	A	B/VDFG	0.875	0.857
P6 (Student)	A	B/VDFG	0.708	0.429
P7 (Student)	A	B/VDFG	0.621	0.571
P8 (Industry)	A	B/VDFG	0.760	1.000
P9 (Student)	B/VDFG	A	0.733	0.714
P10 (Student)	B/VDFG	A	0.858	1.000
P11 (Student)	B/VDFG	A	0.817	0.714
P12 (Industry)	B/VDFG	A	0.590	0.429
P13 (Student)	B	A/VDFG	0.714	0.590
P14 (Student)	B	A/VDFG	0.857	0.723
P15 (Student)	B	A/VDFG	1.000	0.908
P16 (Industry)	B	A/VDFG	0.429	0.671

B. Infeasible Paths

Infeasible paths in the VDFG of JEdit did not cause a problem for investigation tasks. To estimate the effectiveness of our approach on other software, we have extracted data dependency from various Java applications and compared the result with data-flow edges in VDFG. In VDFG, all assignment statements which write a variable have data-flow edges to all statements that refer to the variable. Consequently, a pair of an assignment and a reference to a variable which has no traditional data dependency is an infeasible data-flow edge in VDFG.

We have extracted intraprocedural data dependency on local variables and formal parameters (excluding `this` and a return value) for each method using Java bytecode analysis. Instance fields and class fields are not included in this analysis since data-flow paths related to these variables depend on alias analysis. In the analysis, we took `try-catch` statements into account. We hypothesized every instruction in a `try` block may throw an exception and jump to a `catch` block.

We have chosen 7 open source software and a web application for a university developed by a company. We analyzed the binary distribution archives of the applications including library classes. The web application developed by a company includes several open source libraries.

```
   int getValue() {
1:   int v = readValue(); // -> line 2, 5
2:   if (v < 0) {
3:     v = 0;             // -> line 5
4:   }
5:   return v;
   }
```

Fig. 9. An example of a method that includes an infeasible edge (from line 3 to line 2) but no infeasible paths

Table II shows the result of our analysis. #class, #method and #var are the number of classes, methods and local variables, respectively. #m-inf-edge indicates the number of methods which have at least one infeasible edge. #m-inf-path indicates the number of methods which have at least one infeasible summary edge. #v-inf indicates the number of variables which cause infeasible edges. #summary indicates the number of summary edges in the VDFG. #summary-inf indicates the number of infeasible summary edges. Each row corresponds to a program. "Total" row excluded redundant classes included in two or more programs. Consequently, the sum of each row is not the same as the bottom row.

The result shows that infeasible edges are caused by only 5% of local variables. This is because a programming habit in Java: developers do not reuse a local variable for another purpose. We have found that about 85% of local variables are assigned only once. 10% of local variables are assigned twice or more, but all reference statements are reachable from all assignment statements.

Surprisingly, infeasible summary edges are included in only 1.9% of methods, about a quarter of methods which include infeasible edges. This indicates that some infeasible data-flow edges did not result in infeasible summary edges. Figure 9 shows an example of such methods. In this code, the `if` statement (line 2) refers to variable `v`; therefore, an infeasible edge is extracted for variable `v` from line 3 to line 2. However, the additional edge does not affect to an existing data-flow path from the return value of `readValue` (line 1) to the return value of `getValue` (line 5).

The result shows that the numbers of infeasible data-flow paths are not so vary for each project. Therefore, in terms of precision of data-flow analysis, our approach is promising to analyze applications other than JEdit.

TABLE III
TIME TO EXTRACT VDFG FROM SOURCE CODE

Software	#LOC	Parse	VDFG	Total
Apache Ant 1.7.0	198,394	65sec	19sec	84sec
Apache Batik 1.6	297,320	155sec	33sec	188sec
Apache Tomcat 6.0.14	322,971	181sec	50sec	231sec
Azureus 3.0.3.4	552,295	353sec	115sec	468sec
HSQLDB 1.8.1	157,388	83sec	12sec	95sec
JEdit 4.3pre11	168,872	108sec	17sec	125sec
Spring Framework 2.5.5	487,177	358sec	120sec	478sec
Univ Web App	29,258	28sec	1sec	29sec

C. Performance

To evaluate the performance of our tool, we have measured the time spent to analyze the software used in Section IV-B. In this analysis, we have excluded library classes. The tool has been executed on a workstation with a 1.80GHz Intel Core2 Duo processor and 2GB main memory running Windows Vista Business and 32-bit Sun Java Virtual Machine 1.6.

In Table III, column "Parse" indicates the time to construct abstract syntax trees, variable tables, a class hierarchy tree and a method call graph using MASU Framework [22]. Column "VDFG" indicates the time to construct VDFG using the information obtained from the parser.

Our visualization tool required only a few minutes to construct the whole VDFG for a Java program; for each mouse click on a method name, it takes a few seconds to traverse and visualize a subgraph of VDFG. The performance is efficient enough for daily tasks.

V. CONCLUSION

We have proposed a visualization of interprocedural data-flow paths to support source code reading tasks. Through the experiment, we have shown our tool could help developers more completely investigate data-flow paths. Although our analysis generates infeasible data-flow paths, the number of such edges is limited. The participants of the experiment have found no problems on infeasible data-flow paths.

Our data-flow analysis is control-flow-insensitive, context-insensitive and object-insensitive. However, developers did not consider the limitation as a problem since developers know that various features of Eclipse are also control-flow-insensitive, context-insensitive and object-insensitive.

In the future work, we would like to evaluate how precision of data-flow analysis techniques affect source code reading tasks. We are also planning to apply our control-flow-insensitive data-flow analysis to approximate slice-based software metrics [23]. A combination of our work and Jász [15] work is also interesting since such a combination takes both control-flow and data-flow into account but may be faster than traditional program slicing.

ACKNOWLEDGMENT

We thank Mr. Takeshi Murayama and developers of Hitachi Government & Public Corporation System Engineering, Ltd. for supporting our experiment.

This work was supported by KAKENHI (No.23680001).

REFERENCES

[1] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 185–194.

[2] R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," in *Proceedings of GUIDE 48*, 1983.

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[4] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. L. Jr., "Code bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 455–464.

[5] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *Proceedings of the 8th International Workshop on Program Comprehension*, 2000, pp. 241–247.

[6] M. Desmond, M.-A. Storey, and C. Exton, "Fluid source code views," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, 2006, pp. 260–263.

[7] M. Pinzger, K. Gräfenhain, P. Knab, and H. C. Gall, "A tool for visual understanding of source code dependencies," in *Proceedings of the 16th IEEE International Conference on Program Comprehension*, 2008, pp. 254–259.

[8] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, 1990.

[9] J. Krinke, "Visualization of program dependence and slices," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 168–177.

[10] H. Koike, "Fractal views: a fractal-based method for controlling information display," *ACM Transactions on Information Systems*, vol. 13, no. March, pp. 305–323, 1995.

[11] R. DeLine, G. Venolia, and K. Rowan, "Software development with code maps," *Communications of the ACM*, vol. 53, pp. 48–54, 2010.

[12] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 352–357, 1984.

[13] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, "Experimental evaluation of program slicing for fault localization," *Empirical Software Engineering*, vol. 7, no. 1, pp. 49–76, 2002.

[14] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu, "Matching dependence-related queries in the system dependence graph," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 457–466.

[15] J. Jász, Árpád Beszédes, T. Gyimóthy, and V. Rajilich, "Static execute after/before as a replacement of traditional software dependencies," in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2008, pp. 137–146.

[16] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering*, 2009, pp. 383–392.

[17] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, 1995, pp. 77–101.

[18] H. Shah, C. Görg, and M. J. Harrold, "Visualization of exception handling constructs to support program understanding," in *Proceedings of the 4th ACM Symposium on Software Visualization*, 2008, pp. 19–28.

[19] D. Binkley, N. Gold, and M. Harman, "An empirical study of static program slice size," *ACM Transactions on Software Engineering Methodology*, vol. 16, no. 2, p. 8, 2007.

[20] Graphviz Project. [Online]. Available: http://www.graphviz.org/

[21] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[22] MASU Framework. [Online]. Available: http://masu.sourceforge.net/

[23] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 1, pp. 2:1–2:27, 2007.