Dynamic Slicing of Feature-Oriented Programs

Madhusmita Sahu and Durga Prasad Mohapatra

Abstract We intend to suggest a dynamic slicing algorithm for feature-oriented programs. We have named our algorithm *Execution Trace file Based Dynamic Slicing* (ETBDS) algorithm. The ETBDS algorithm constructs an intermediate program representation known as *Dynamic Feature-Oriented Dependence Graph* (DFDG) based on various dependences exist amongst the program statements. We use an execution trace file to keep the execution history of the program. The dynamic slice is computed by first performing breadth-first or depth-first traversal on the DFDG and then mapping out the resultant nodes to the program statements.

Keywords Feature-oriented programming (FOP) · Dynamic feature-oriented dependence graph (DFDG) · FeatureC++ · Mixin layer

1 Introduction

Feature-Oriented Programming (FOP) is the study of feature modularity in program families and programming models supporting it. The key idea behind FOP is to build software by the composition of features. *Features* are the characteristics of software that distinguish members of a program family. The FOP paradigm is concerned with identifying functionality in the form of features.

The rest of the paper is organized as follows. Section 2 highlights an overview of some previous works. Section 3 presents a brief idea of Feature-Oriented

M. Sahu (✉) · D.P. Mohapatra
Department of Computer Science & Engineering,
National Institute of Technology, Rourkela 769008, Odisha, India
e-mail: 513CS8041@nitrkl.ac.in

D.P. Mohapatra
e-mail: durga@nitrkl.ac.in

A. Nagar et al. (eds.), *Proceedings of 3rd International Conference on Advanced Computing, Networking and Informatics*, Smart Innovation, Systems and Technologies 44, DOI 10.1007/978-81-322-2529-4_40

Programming. Section 4 describes our proposed work on dynamic slicing of feature-oriented programs. Section 5 concludes the paper. We use node and vertex interchangeably in this paper.

2 Overview of Previous Work

Apel et al. [1] presented FeatureC++ with an additional adoptation to Aspect-Oriented Programming (AOP) concepts. Apel et al. [2] presented a novel language for FOP in C++ namely *FeatureC++*. They showed few problems of FOP languages during implementation of program families and proposed ways to solve them by the combination of AOP and FOP features. We find no work discussing the slicing of FOP. We present an approach for dynamic slicing of FOPs using FeatureC++ as a FOP language. We have extended the work of Mohapatra et al. [3] to incorporate feature-oriented features.

3 Feature-Oriented Programming (FOP)

The term Feature-Oriented Programming (FOP) was coined by Christian Prehofer in 1997 [4]. FOP is a vision of programming in which individual features can be defined separately and then can be composed to build a wide variety of particular products. The *step-wise refinement*, where features are incrementally refined by other features, results in a layered stack of features. A suitable technique for the implementation of features is the use of *Mixin Layers*. A Mixin Layer is a static component that encapsulates fragments of several different classes (Mixins) to compose all fragments consistently.

FeatureC++ is an extension to C++ language supporting Feature-Oriented Programming (FOP). Figure 3 shows an example FeatureC++ program. This program checks the primeness of a number. Figure 1 shows different features supported by our prime number checking problem and Fig. 2 shows the corresponding stack of Mixin Layers. Details of FeatureC++ can be found in [1, 2].

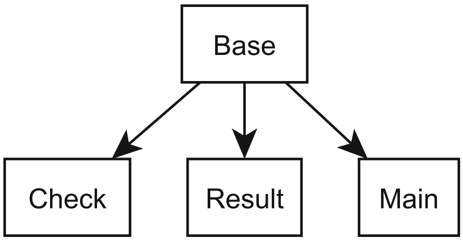


Fig. 1 Features supported by prime number checking problem

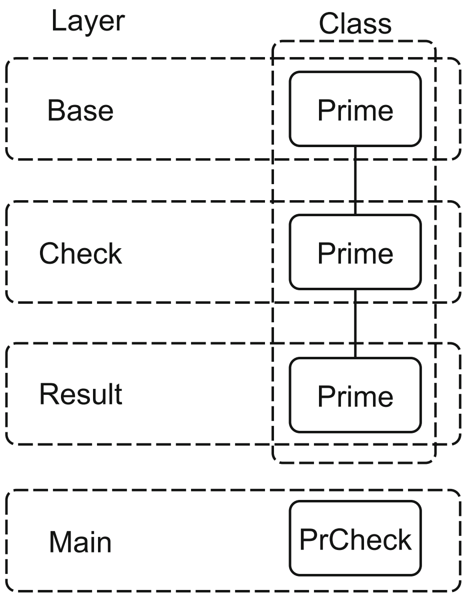


Fig. 2 Stack of mixin layers in prime number checking problem

4 Proposed Work

This section describes our *Execution Trace file Based Dynamic Slicing* (ETBDS) algorithm to compute dynamic slices of FOPs alongwith few definitions.

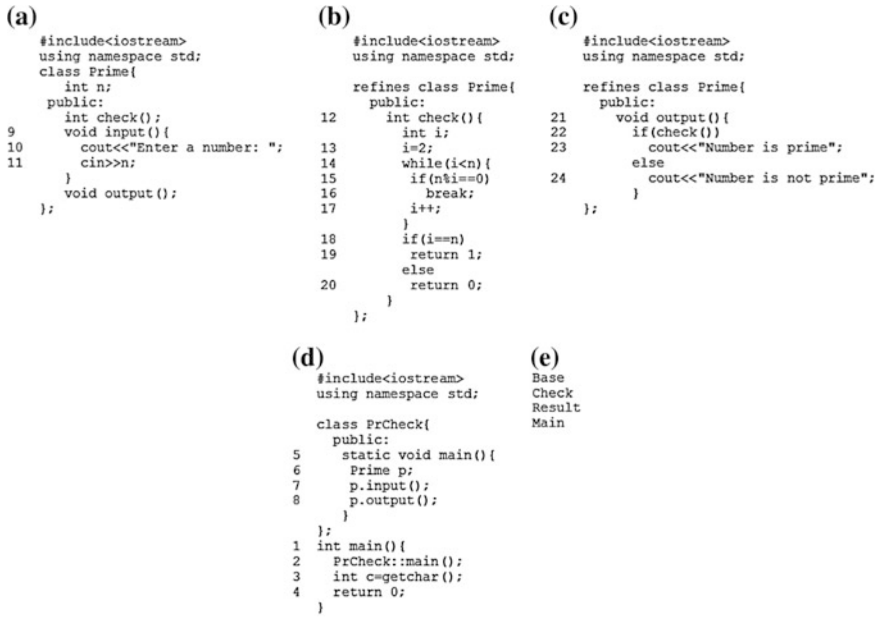
4.1 Definitions

Definition 1: Digraph A *directed graph* or *digraph* is defined as a collection of a finite set, *V*, of elements called *vertices* or *nodes* and the set, *E*, of ordered pairs of elements of *V* called *edges* or *arcs*.

Definition 2: Arc-classified digraph An *arc-classified digraph* is defined as a digraph where more than one type of edges exist between two vertices and the direction of edges between two vertices are not same.

Definition 3: Path A *path* in a digraph is a sequence of edges or arcs connecting a sequence of vertices and the edges are directed in the same direction.

Definition 4: Execution trace The path that an input data to a program actually executes is referred to an *execution trace*. For example, Fig. 4 shows the execution trace of the program given in Fig. 3 for the input data $n = 5$.



(a)
```
    #include<iostream>
    using namespace std;
    class Prime{
       int n;
     public:
       int check();
9      void input(){
10        cout<<"Enter a number: ";
11        cin>>n;
       }
       void output();
    };
```

(b)
```
    #include<iostream>
    using namespace std;

    refines class Prime{
      public:
12      int check(){
          int i;
13        i=2;
14        while(i<n){
15         if(n%i==0)
16           break;
17         i++;
          }
18        if(i==n)
19         return 1;
          else
20         return 0;
        }
    };
```

(c)
```
    #include<iostream>
    using namespace std;

    refines class Prime{
      public:
21      void output(){
22        if(check())
23          cout<<"Number is prime";
          else
24          cout<<"Number is not prime";
        }
    };
```

(d)
```
   #include<iostream>
   using namespace std;

   class PrCheck{
     public:
5     static void main(){
6      Prime p;
7      p.input();
8      p.output();
      }
   };
1  int main(){
2    PrCheck::main();
3    int c=getchar();
4    return 0;
   }
```

(e)
```
Base
Check
Result
Main
```

Fig. 3 A FeatureC++ program to check the primeness of a number. **a** Base/Prime.h. **b** Check/Prime.h. **c** Result/Prime.h. **d** Main/PrCheck.h. **e** Test-Prime.equation

4.2 The Dynamic Feature-Oriented Dependence Graph (DFDG)

The DFDG is an *arc-classified digraph* consisting of vertices corresponding to the statements and edges showing dynamic dependence relationships exist amongst statements. The following types of dependence edges exist in the DFDG of a feature-oriented program:

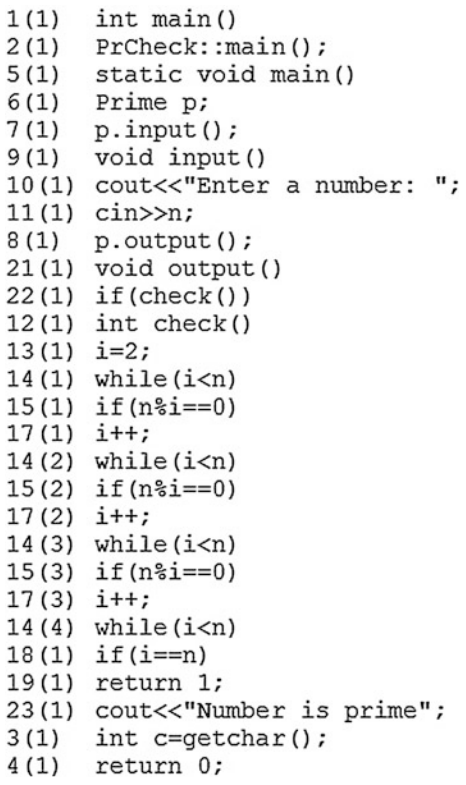
Control dependence edge: *Control dependences* represent the control conditions or predicates on which the execution of a statement or an expression depends. For example, in Fig. 5, the edge between nodes 21 and 22 indicates that node 21 controls the execution of node 22.
Data dependence edge: *Data dependences* represent the flow of data amongst the statements and expressions. For example, in Fig. 5, the edge between nodes 13 and 14 indicates that node 14 uses the value of *i* defined at node 13.
Mixin call edge: *Mixin call* edges denote the entry of a function in a mixin layer in response to a function call in another mixin layer. For example, in Fig. 5, the edge from node 12 to node 22 indicates that node 22 in one mixin layer calls a function check() that is defined in another mixin layer at node 12.
Mixin data dependence edge: *Mixin data dependences* represents the flow of data amongst the statements and expressions in different mixin layers. For example, in

Fig. 5, the edge from node 14 to node 11 indicates that node 14 in one mixin layer uses the value of *n* and *n* is defined in another mixin layer. Similarly, node 22 in one mixin layer uses the value returned by node 19 and node 19 exist in another mixin layer.
Call edge: *Call* edges are used to reflect the entry of a function in response to a function call. For example, in Fig. 5, there is an edge from node 5 to node 2 since node 2 calls a function main defined at node 5 in the same mixin layer.

4.3 Computation of Dynamic Slices

Let *FP* be a feature-oriented program and $G = (V, E)$ be the DFDG of *FP*. We use a slicing criterion with respect to which the dynamic slice of FOP is to be computed. A *dynamic slicing criterion* for *G* has the form $<x, y, e, i>$, where $x \in V$ represents an occurrence of a statement for an execution trace *e* with input *i* and *y* is the variable used at *x*. A *dynamic slice* DS_G of *G* on a given slicing criterion $<x, y, e, i>$ is a subset of vertices of *G* such that for any $x' \in V$, $x' \in DS_G(x, y, e, i)$ if and only if there exists a path from x' to *x* in *G*.

Algorithm 1 gives our proposed *ETBDS* algorithm.



```
1(1)   int main()
2(1)   PrCheck::main();
5(1)   static void main()
6(1)   Prime p;
7(1)   p.input();
9(1)   void input()
10(1)  cout<<"Enter a number: ";
11(1)  cin>>n;
8(1)   p.output();
21(1)  void output()
22(1)  if(check())
12(1)  int check()
13(1)  i=2;
14(1)  while(i<n)
15(1)  if(n%i==0)
17(1)  i++;
14(2)  while(i<n)
15(2)  if(n%i==0)
17(2)  i++;
14(3)  while(i<n)
15(3)  if(n%i==0)
17(3)  i++;
14(4)  while(i<n)
18(1)  if(i==n)
19(1)  return 1;
23(1)  cout<<"Number is prime";
3(1)   int c=getchar();
4(1)   return 0;
```

Fig. 4 Execution trace of the program given in Fig. 3 for $n = 5$

Working of ETBDS Algorithm

The working of our ETBDS algorithm is exemplified with an example. Consider the example FeatureC++ program given in Fig. 3. The program executes the statements 1, 2, 5, 6, 7, 9, 10, 11, 8, 21, 22, 12, 13, 14, 15, 17, 14, 1, 17, 14, 15, 17, 14, 18, 19, 23, 3, 4 in order for the input data $n = 5$. The execution trace file, shown in Fig. 4, kept these executed statements. Then, applying Steps 6 to 23 of ETBDS algorithm

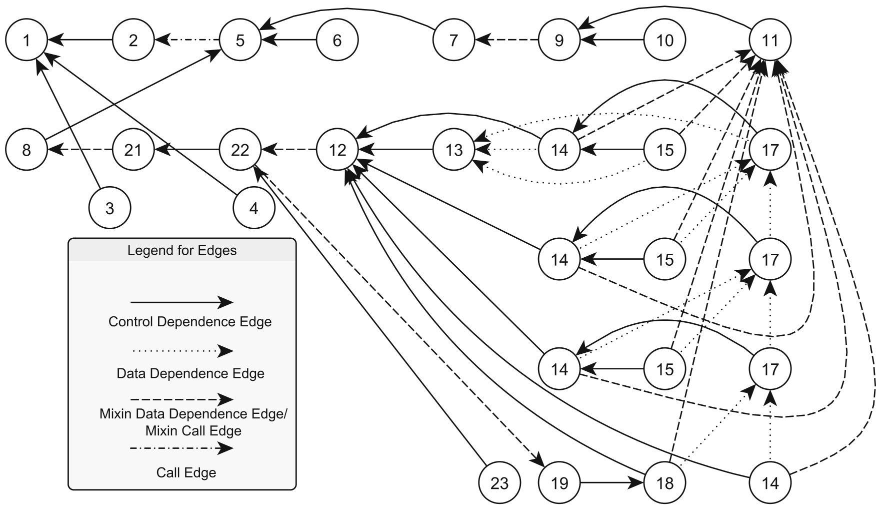


Fig. 5 Dynamic feature-oriented dependence graph (DFDG) for the execution trace given Fig. 4

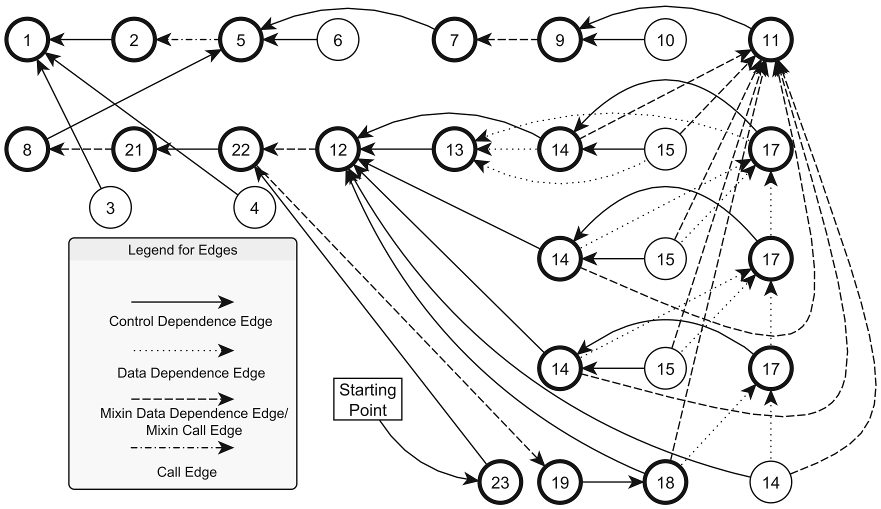


Fig. 6 Dynamic feature-oriented dependence graph (DFDG) for the execution trace given Fig. 4 and node 23 as the starting point

and using the trace file shown in Fig. 4, the *Dynamic Feature-Oriented Dependence Graph* (DFDG) is constructed. The DFDG of the example program given in Fig. 3 is shown in Fig. 5 with input data $n = 5$. Suppose, we want to find the dynamic slice of the output statement at node 23. Then, starting from node 23 and applying the Step 25 of the ETBDS algorithm, the breadth first traversal yields the vertices 23, 22, 21, 19, 8, 5, 2, 1, 18, 12, 11, 17, 9, 7, 14, 17, 14, 17, 14, 13 and the depth first traversal yields the vertices 23, 22, 21, 19, 18, 12, 11, 17, 14, 17, 14, 17, 14, 13, 9, 7, 5, 2, 1, 8. Both the traversals yields the same result i.e., the same set of vertices in the slice. These vertices are shown bold in Fig. 6. The statements corresponding to these vertices are found using the Steps 26 to 27 of the ETBDS algorithm. This produces the required dynamic slice consisting of statements numbered 1, 2, 5, 7, 8, 9, 11, 12, 13, 14, 17, 18, 19, 21, 22, 23.

Algorithm 1 ETBDS Algorithm

```
1: Execute the program for a given input.
2: Keep each executed statement in a trace file in the order of execution.
3: if the program contains loops then
4:     Keep each executed statement inside the loop in a trace file after each time it
       has been executed.
5: end if
6: Make a vertex in the DFDG for each statement in the trace file.
7: for each occurrence of a statement in the trace file do
8:     Make a separate vertex in the DFDG.
9: end for
10: if if node $y$ controls the execution of node $x$ then
11:     Insert control dependence edge from $x$ to $y$.
12: end if
13: if if node $x$ uses a variable defined at node $y$ then
14:     Insert data dependence edge from $x$ to $y$.
15: end if
16: if if node $y$ calls a function defined at node $x$ in the same mixin layer then
17:     Insert call edge from $x$ to $y$.
18: end if
19: if if node $x$ in one mixin layer uses a variable or value defined at or returned from
    node $y$ in another mixin layer then
20:     Insert mixin data dependence edge from $x$ to $y$.
21: end if
22: if if node $y$ in one mixin layer calls a function defined at node $x$ in another mixin
    layer then
23:     Insert mixin call edge from $x$ to $y$.
24: end if
25: Do the breadth-first or depth-first traversal throughout the DFDG taking any
    vertex $x$ as the starting point of traversal where $x$ corresponds to the statement of
    interest.
26: Define a mapping function $g : DS_G(x, y, e, i) \rightarrow$FP.
27: Map out the yielded slice obtained in Step 25 throughout the DFDG to the source
    program $FP$ using $g$.
```

5 Conclusion

We presented an algorithm for dynamic slicing of feature-oriented programs. First, we executed the program for a given input and stored the execution history in an execution trace file. Then, we constructed an intermediate representation called *Dynamic Feature-Oriented Dependence Graph* (DFDG) based on various dependences. The DFDG was traversed in breadth-first and depth-first manner and the resultant nodes were mapped to the program statements to compute the dynamic slice.

References

1. Apel, S., Leich, T., Rosenmuller, M., Saake, G.: FeatureC++: on the symbiosis of feature-oriented and aspect-oriented programming. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'05), pp. 125–140. Springer, Berlin (2005)
2. Apel, S., Leich, T., Rosenmuller, M., Saake, G.: FeatureC++: feature-oriented and aspect-oriented programming in C++. Technical report (2005)
3. Mohapatra, D.P., Sahu, M., Mall, R., Kumar, R.: Dynamic slicing of aspect-oriented programs. Informatica **32**(3), 261–274 (2008)
4. Prehofer, C.: Feature-oriented programming: a fresh look at objects. In: Proceedings of 11th ECOOP, Lecture Notes in Computer Science, pp. 419–443. Springer, Berlin, Heidelberg (1997)