Static Slicing of Concurrent Object-Oriented Programs

Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima

Department of Computer Science and Communication Engineering

Kyushu University

6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-81, Japan

{zhao,cheng,ushijima}@csce.kyushu-u.ac.jp

Abstract

Program slicing has many applications such as program debugging, testing, maintenance, and complexity measurement. This paper concerns the problem of slicing concurrent object-oriented programs that has not been addressed in the literatures until now. To solve this problem, we propose a new program dependence representation named the system dependence net *(SDN), which extends previous program dependence representations to represent concurrent object-oriented programs. An SDN of a concurrent object-oriented program consists of a collection of procedure dependence nets each representing a main procedure, a free standing procedure, or a method in a class of the program, and some additional arcs to represent direct dependences between a call and the called procedure/method and transitive interprocedural data dependences. We construct the SDN to represent not only object-oriented features but also concurrency issues in a concurrent object-oriented program. Once a concurrent object-oriented program is represented by its SDN, the slices of the program can be computed based on the SDN as a simple vertex reachability problem in the net.*

1   Introduction

Program slicing has many applications including program debugging [1, 11], testing [2], maintenance [8], and complexity measurement [3, 14]. A *program slice* consists of the parts of a program that (potentially) affect the values computed at some program point of interest, referred to as a *slicing criterion*. The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion are called the *program slice with respect to the criterion*. The process of finding program slices is called *program slicing*. Weiser [18] first introduced the concept of a program slice and proposed an algorithm to compute intraprocedural and interprocedural slices of a sequential imperative program. For a target program P he defined a slicing criterion which consists of a program point p and a set of variables V, and a slice S with respect to p and V as a reduced and executable program derived from P by removing zero or more statements, such that S replicates parts of the behavior of P. Ottenstein and Ottenstein [14] defined a slicing criterion for a sequential imperative program consisting of a program point p and a variable v defined or

used at p and proposed to use a *Program Dependence Graph* (PDG) to compute slices. They showed that once a program can be represented by its PDG, the computation of a slice with respect to the slicing criterion can be simplified to a vertex reachability problem in the PDG. Horwitz, Reps, and Binkley [9] also use dependence graphs to compute slices of a sequential imperative program. They proposed an interprocedural program dependence representation, called *System Dependence Graph* to represent a sequential imperative program with multiple procedures and developed a two-pass slicing algorithm to compute slices of the program. Cheng [6] considered the problem of slicing concurrent imperative programs. He proposed a program dependence representation, called the *Process Dependence Net* which is the generalization of the PDG to represent program dependences in a concurrent imperative program with single procedure. In addition to usual control and data dependence arcs as in the PDG, the PDN also contains selection, synchronization, and communication dependence arcs to represent program dependences related to nondeterministic selections, interprocess synchronizations and interprocess communications respectively in the program. Based on the PDN, the problem of slicing concurrent imperative programs can also be simplified to a vertex reachability problem in the net. Tip [17] surveys the existing slicing algorithms for imperative programs.

However, although a number of slicing algorithms have been proposed and studied for imperative programs, little slicing algorithms has been proposed for object-oriented programs. To our knowledge, only two articles deal with the slicing problem of sequential object-oriented programs, and no slicing algorithm for concurrent object-oriented programs exists until now. Larsen and Harrold [13] proposed an algorithm for static slicing of sequential object-oriented programs. They extended the SDG which was first proposed to handle the interprocedural slicing of imperative programs [9] to the case of sequential object-oriented programs. Since the SDGs which they compute for sequential object-oriented programs belong to a class of SDGs defined in [9], they can use the two-pass slicing algorithm introduced in [9, 15] to compute slices of sequential object-oriented programs. Krishnaswamy [12] proposed another approach to slicing sequential object-oriented programs. He uses a program dependence representation called the *object-*

oriented program dependency graph (OPDG) to represent sequential object-oriented programs and compute polymorphic slices of a sequential object-oriented program based on the OPDG. Although the SDG and OPDG can represent many features of sequential object-oriented programs, they provide no mechanism to represent concurrency issues related to nondeterministic selections, interprocess synchronizations and communications in a concurrent object-oriented program.

Slicing concurrent object-oriented programs is considered more difficult than slicing sequential object-oriented programs since in a concurrent object-oriented program, in addition to some object-oriented features such as the classes and their instances, objects, inheritance, polymorphism, dynamic binding and objects, and scoping, there exists also some concurrency issues related to nondeterministic selection, interprocess synchronization and communication. As a result, if we intend to slice concurrent object-oriented programs using dependence graphs, we must develop a program dependence representation to represent either object-oriented features or concurrency issues in a concurrent object-oriented program. However, although there has been much work contributed to represent imperative programs and sequential object-oriented programs using dependence graphs [6, 14, 9, 12, 13], no program dependence representation exists which can be used to represent the full range of a concurrent object-oriented program until now.

To address the problem of slicing concurrent object-oriented programs, we propose a new program dependence representation named the *System Dependence Net*, which extends previous dependence representations [6, 9, 13] to represent concurrent object-oriented programs*. The system dependence net of a concurrent object-oriented program can be used to represent not only object-oriented features but also concurrency issues in the program, and allows us to efficiently compute slices of the program by using slicing algorithms introduced in [9, 15]. An SDN of a concurrent object-oriented program consists of a collection of procedure dependence nets each representing a main procedure, a free standing procedure, or a method in a class of the program, and some additional arcs to represent direct dependences between a call and the called procedure/method and transitive interprocedural data dependences. Once a concurrent object-oriented program is represented by its system dependence net, the slicing of the program can be simplified to a vertex reachability problem in the net.

The main contribution of this paper is a new program dependence representation for concurrent object-oriented programs on which slices of concurrent object-oriented programs can be computed efficiently. In addition, since we also consider how to represent a free standing procedure in a concurrent object-oriented program our representation can be extended straightforwardly to represent concurrent imperative programs with multiple procedures. Although here we present our results for CC++ pro-

grams, the notions are general and easy to be applied to those programs written in other statically typed concurrent object-oriented languages.

The rest of the paper is organized as follows. Section 2 briefly introduces the concurrent object-oriented programming language CC++. Section 3 presents the system dependence net for concurrent object-oriented programs[†]. Section 4 describes how to compute a slice of a concurrent object-oriented program based on its system dependence net. Conclusions are given in Section 5.

2 Preliminaries

We assume that readers are familiar with the basic concepts of object-oriented programming languages, and do not give a comprehensive introduction here. Throughout this paper, we will restrict ourselves to CC++ [5], a concurrent object-oriented programming language, which illustrates the basic mechanisms of concurrent object-oriented programming.

CC++ (Compositional C++) is a concurrent object-oriented programming language. CC++ is C++ with six extensions for writing declarative and concurrent programs. Of those extensions, we briefly introduce three ones which will be considered in this paper. The other three extensions that we do not introduce in this paper are constructs for atomic functions and for the unstructured spawning of processes and for the distribution of computations across multiple address spaces. A complete definition of the syntax and semantics of CC++ is given by [4].

Below, we use a sample CC++ program taken from [16] as an example to briefly introduce these extensions. This example implements a single-reader, single-writer stream. Two operations are defined on such a stream: an append and a removal. A removal from an empty stream suspends until the stream is non-empty. Appends to the stream never suspend. This example is shown in Figure 1.

The most basic mechanism for creating parallel threads of control in CC++ is the parallel block. A parallel block looks just like a compound statement in C or C++ with the keyword **par** in front of it. The statements in a **par**-block are executed as parallel processes. The execution of a **par**-block terminates when the execution of all its statements has terminated. See the **main** procedure of the program in Figure 1.

The construct for parallel composition of a variable number of statements is **parfor**. With the exception of the keyword, the syntax of a **parfor** statement is the same as the usual C++ **for** statement:

```
parfor (int i=0; i<N; i++) {
    statement_1;
    statement_2;
    statement_M;
}
```

This is a parallel loop construct in which the iterations of a **parfor** statement are executed as parallel processes. As with the usual C or C++ for loop, the

---

body of each iteration is executed sequentially. Similar to the parallel block introduced above, there is an implicit barrier at the end of a **parfor**. The execution of a **parfor** statement terminates when the execution of all its iterations has terminated.

CC++ defines a new type of variable, a single-assignment variable, denoted by the keyword **sync**. A variable of a **sync** type initially has a special undefined value, and can be assigned a value at most once. Evaluation of an undefined **sync** variable suspends until the variable is assigned a value (Types that are not **sync** are referred to as "mutable" types). Thus, threads of control that share access to a single-assignment variable can use that variable as a synchronization element. See the CC++ class **StreamNode** of the program in Figure 1.

```
   1    #include <iostream.h>
   2    class Stream;
 ce3    class StreamNode  {
   4      private:
   5        int data;
   6        StreamNode *sync next;
  e7        StreamNode (int d)
   8        {
  s9           data = d;
  10        }
  11      friend class Stream;
  12    };
ce13    class Stream  {
  14      private:
  15        StreamNode* head;
  16        StreamNode* tail;
  17      public:
 e18        Stream (void)
  19        {
 s20           head = new StreamNode(0);
 s21           tail = head;
  22        }
 e23        void append (int a)
  24        {
 s25           StreamNode* addition = new
                              StreamNode(a);
 s26           tail->next = addition;
 s27           tail = addition;
  28        }
 e29        int remove (void)
  30        {
 s31           StreamNode* old_head = head;
 s32           head = head->next;
 s33           delete old_head;
 s34           return head->data;
  35        }
  36    }
  37    Stream S;
 e38    void producer (int n1)
  39    {
 s40      for (int i=0; i<n1; i++)  {
 s41        cout << ''[appending '' << i << '']'';
 s42        S.append(i);
  43      }
  44    }
 e45    void consumer (int n2)
  46    {
 s47      for (int i=0; i<n2; i++)
 s48        cout << ''Consumer removes : ''
                        << S.remove() << endl;
  49    }
 e50    int main()
  51    {
 s52      par {
 c53        producer(10);
 c54        consumer(10);
  55      }
 s56      return 0;
  57    }
```

Figure 1:   A sample CC++ program.

3   The System Dependence Net for Concurrent Object-Oriented Programs

Object-oriented programs differ from usual imperative programs in many ways. Some of these differences are the classes and their instances, objects, inheritance, polymorphism, dynamic binding and objects, and scoping. These object-oriented features impact the development of object-oriented software, and also impact the development of an object-oriented program representation. Furthermore, in a concurrent object-oriented program, in addition to those object-oriented features, there are some concurrency issues about non-deterministic selections, interprocess synchronizations and interprocess communications. Therefore, to represent a concurrent object-oriented program from a static viewpoint, these important features should be considered carefully.

In this section we propose a new program dependence representation named the system dependence net to represent a concurrent object-oriented program. A system dependence net of a concurrent object-oriented program consists of a collection of procedure dependence nets each representing a main procedure, a free standing procedure, or a method in a class of the program. A system dependence net is an extension of previous program dependence representations [6, 9, 13]. We use CC++ programs as the target programs to show how to represent a concurrent object-oriented program by a system dependence net.

3.1   Procedure Dependence Nets

This subsection describes the procedure dependence net for a main procedure, a free standing procedure, or a method in a class of a concurrent object-oriented program, and shows how to represent the nondeterministic selection, interprocess synchronization and communication in a single procedure/method.

Generally, a procedure/method in a concurrent object-oriented program consists of a number of processes, and therefore, it has multiple control flows and multiple data flows. These flows are not independent because there exist interprocess synchronizations among multiple control flows and interprocess communications among multiple data flows in the program. Moreover, a process in a procedure/method may select a communication partner nondeterministically among a number of processes ready for communication with the process. Therefore, to represent a procedure/method in a concurrent object-oriented, a program dependence representation should be used to represent these features. On the other hand, in order to construct the system dependence net of a concurrent object-oriented program, we should also model the parameter passing between procedures, methods, and a procedure and a method in the representation. Obviously, the procedure dependence graph [9] proposed for representing a single procedure in a sequential imperative program with multiple procedures, although can model the parameter passing between procedures, can not be used to represent a proce-

dure/method in a concurrent object-oriented program since it only considers sequential imperative programs. In this paper, we extend the *process dependence net* [6] to represent a procedure/method in a concurrent object-oriented program. The process dependence net is a generalization of the program dependence graph to represent program dependences in a concurrent imperative program with a single procedure. In addition to usual control and data dependence arcs as in the program dependence graph, the process dependence net also contains selection, synchronization, and communication dependence arcs to represent nondeterministic selections, interprocess synchronizations and interprocess communications in the program, respectively. However, since the process dependence net can not model parameter passing, we add some additional vertices to each process dependence net as the procedure dependence graph [9] to represent the parameter passing. We name the resulting net as the "procedure dependence net" and use it to represent a main procedure, a free standing procedure, or a method in a class of a concurrent object-oriented program.

A procedure dependence net of a procedure/method in a concurrent object-oriented program is an arc-classified digraph such that its vertices represent a statement or a control predicate of a conditional branch statement in the program, and its arcs represent five types of program dependence relationships between statements that were mentioned above. Control dependences represent control conditions on which the execution of a statement or expression depends. Informally, a statement u is directly control-dependent on the control predicate v of a conditional branch statement(e.g., an if statement or while statement) if whether u is executed or not is directly determined by the evaluation result of v. Data dependences reflect the data flow between statements and expressions. Informally a statement u is directly data-dependent on a statement v if the value of a variable computed at v has a direct influence on the value of a variable computed at u. Selection dependences are similar to control dependences but involve nondeterministic selection statements. Informally, a statement u is directly selection-dependent on a nondeterministic selection statement v if whether u is executed or not is directly determined by the selection result of v. Synchronization dependences represent interprocess synchronizations in a single procedure/method. It does not involve data transferring. Informally a statement u is directly synchronization-dependent on another statement v if the start and/or termination of execution of v directly determines whether or not the execution of u starts and/or terminates. Internal-communication dependences represent interprocess communications in a single procedure/method. Informally a statement u in a process is directly internal-communication-dependent on another statement v in another process if the value of a variable computed at v has a direct influence on the value of a variable computed at u by an interprocess communication. On the other hand, to model parameter passing, Like in [9], we creat a formal-in/formal-out vertex at each procedure/method start vertex (i.e., there is a formal-in vertex for each formal parameter of the proce-

dure/method and there is a formal-out vertex for each formal parameter that may be modified by the procedure/method), and an actual-in/actual-out vertex at each call site in the procedure/method (i.e., there is an actual-in vertex for each actual parameter of the procedure/method, and an actual-out vertex for each actual parameter that may be modified by the procedure/method). We also creat a call vertex at each call site of the procedure/method. Moreover, Each actual parameter vertex is control dependent on the call vertex of the procedure/method, and also each formal parameter vertex is control dependent on the procedure/method start vertex. Finally, we treat global variables at the procedure/method entry and call sites in the procedure/method as parameters. Therefore, we should also create actual-in, actual-out, formal-in, formal-out vertices for these variables.

Note that the usual procedure dependence graph [9]can be regarded as a special case of the procedure dependence net which does not contain the selection, synchronization, and communication dependence arcs.

Example. Figure 2 shows CC++ procedures `main`, `producer`, and `consumer` in Figure 1 and their procedure dependence nets.

3.2 Class Dependence Nets

In this subsection we use a class dependence net to represent each single class, and describe how to construct the class dependence net for interacting classes and derived classes. We also show how to represent polymorphism.

Single Classes

In order to efficiently construct the system dependence net of a concurrent object-oriented program and perform analysis, we propose a program dependence represent named the *class dependence net* (CDN), which extends the *class dependence graph* (CDG)[13] for a single class in sequential object-oriented programs to represent a single class of a concurrent object-oriented program. A CDN captures not only the control and data dependence relationships concerning intraprocess control flows and data flows as in the CDG but also selection, synchronization and communication dependence relationships concerning nondeterministic selections, interprocess synchronizations and communications in a class of a concurrent object-oriented program. Moreover, the class dependence net represents the parameter passing between methods in the class. A class dependence net is an arc-classified digraph which consists of a collection of procedure dependence nets described previously and some additional arcs and vertices. Each procedure dependence net represents a method in the class. In the class dependence net, there is a class start vertex to represent the entry of the class and a method start vertex for each method to represent the entry of the method in the class. The class start vertex is connected to each method start vertex in the class by the class membership arc. If there is a call in a method to call another method in the class, we connect the procedure dependence nets of the two methods at call sites. In
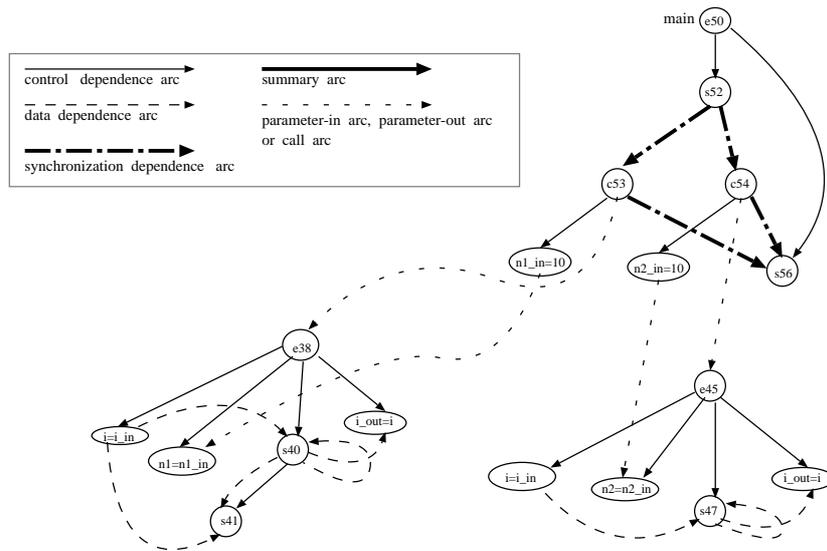
Figure 2: The **main**, **producer**, and **consumer** procedures in Figure 1 and their procedure dependence nets.

this case, a *call* arc is added between a call vertex of a method and the start vertex of the procedure dependence net of the called method, and *parameter-in* and *parameter-out* arcs are added to connect actual-in and formal-in vertices, and formal-out and actual-out vertices. Note that parameter-in and parameter-out arcs represent the parameter passing between methods in the class. In [9], interprocedural slices are computed by solving a graph reachability problem on an SDG. To obtain precise slices, the computation of a slice must preserve the calling context of called procedures, and ensure that only paths corresponding to legal call/return sequences are considered. To facilitate the computation of interprocedural slicing that considers the call context, an SDG represents the flow dependences across call sites. A *transitive flow of dependence* occurs between an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. The transitive flow of dependence can be caused by data dependences, control dependences, or both. A *summary* arc models the transitive flow of dependence across a procedure call. Similar to [9], we also use summary arcs to represent this kind of transitive flow of dependences in the class dependence net. Moreover, for a class of a concurrent object-oriented program, interprocess synchronizations and communications may exist not only in a single method but also between different methods in the class. The class dependence graph [13] can not represent such concurrent issues since it only considers to represent a single class in a sequential object-oriented program. To represent interprocess communications between different methods in a class of a concurrent object-oriented program, we introduce a new kind of program dependence arcs named the *external-*

communication dependence arcs into the class dependence net. External-communication dependences represent interprocess communications between different methods in a class. Informally a statement u in a process of a method is directly external-communication-dependent on another statement v in another process of another method if the value of a variable computed at v has a direct influence on the value of a variable computed at u by an interprocess communication between this two methods. the external-communication dependence differs from the internal-communication dependence which reflects the interprocess communication in a single method of a class.

Moreover, since the instance variables of a class are accessible to all methods in the class, we regard them as global variables to methods in the class, and create formal-in and formal-out vertices for all instance variables that are referenced in the method. The exception to this representation for instance variables is that our construction omits formal-in vertices for instance variables in the class constructor and formal-out vertices for instance variables in the class destructor. In addition to the things mentioned above, the CDN for a CC++ class should also represent the effects of *return* statements. A *return* statement causes a method to return a value to its caller. In a CDN, a vertex for each *return* statement is connected to its corresponding call vertex by a parameter-out arc. Moreover, if a actual-in parameter may affect the returned value, we add a summary arc between the actual-in vertex and the call vertex. This kind of summary arcs facilitate interprocedural slicing.

Example. Figure 3 shows the CDN for a CC++ class **Stream** in Figure 1. In the figure, a rectangle represents the class start vertex and is labeled by the statement label related to the class entry. Circles represent statements in the class, including method

start, and are labeled with the corresponding statement number in the class. Ellipses in dashed line represent actual parameter vertices and Ellipses in solid line represent formal parameter vertices. For example, $ce13$ is the class start vertex, and $e18$, $e23$ and $e29$ are the start vertices of methods. Bold dashed arcs represent class membership arcs that connect the class start vertex to each start vertex of the methods. Therefore, $(ce13, e18)$, $(ce13, e23)$, and $(ce13, e29)$ are class membership arcs. Each start vertex of the methods is the root of a subnet which is itself a procedure dependence net corresponding to the method. Hence each subnet may contain control, data, selection, synchronization, and communication dependence arcs, parameter-in and parameter-out arcs, and summary arcs. Moreover, the CDN can be also represented the interprocess communication between methods in the class. For example, there is an external-communication dependence arc from **s26** to **s32** since there exists an interprocess communication between these two statements which are in different methods of class **Stream** via variable **next** which has the **sync** type. Finally, constructor method **Stream** has no formal-in vertices for the two instance variables, since these variables cannot be referenced before they are allocated by the class constructor.

Interacting Classes

In a concurrent object-oriented program, a class may instantiate another class through a declaration or by using an operator such as **new**. When class $c1$ instantiates class $c2$, there is an implicit call to $c2$'s constructor. To represent this implicit constructor call, similar to [13], we adds a call vertex in $c1$ at the location of the instantiation. We also adds actual-in and actual-out vertices at the call vertex to match the formal-in and formal-out vertices in $c2$'s constructor. When there is a call site in method m_1 in c_1 to method m_2 in the public interface of c_2, we connect the call vertex in c_1 to the start vertex of m_2 to form a call vertex, and also connect actual-in and formal-in vertices to form parameter-in arcs and actual-out and formal-out vertices to form parameter-out arcs. Therefore we can get a new CDN which represents a partial program by connecting these two CDNs. For example, the program in Figure 1 contains a statement $s25$ in the method of the class **Stream** which instantiates object of type **StreamNode**. Figure 4 shows the representation of such interaction classes. The construction includes adding actual-in and actual-out vertices at call vertex for $s25$ to match the formal-in and formal-out vertices associated with $e7$ which is the method start vertex of **StreamNode**, and connecting the call vertex for $s25$ to the method start vertex for $e7$ to form a call arc, actual-in and formal-in vertices to form parameter-in arcs and actual-out and formal-out vertices to form parameter-out arcs.

Derived Classes

A *derived class* in an object-oriented program is the refinement of a class into a more specialized class. Derived classes can be defined based on existing classes via *inheritance*. Inheritance is a relation between classes that allows definition and implementation of a class to be built on another. Therefore, inheritance permits a derived class to inherit attributes from its parent classes, and extend, restrict, redefine or replace them in some way. Just as inheritance facilitates code reuse, an efficient graph representation for a derived class should facilitate the reuse of analysis information.

Polymorphism

Another important feature of object-oriented languages is *polymorphism*. In an object-oriented program, a polymorphic reference can, over time, refer to instances of more than one class. Therefore, the static representation should represent this dynamic feature of object-oriented paradigm.

In CC++, polymorphic method calls occur when an indirect reference to a class is made through a pointer dereference, and the type of the referenced object is unknown at compile time. In such cases, compilers add code to resolve the type of the pointer dereference dynamically. A CDN must provide a mechanism for such dynamic operations to be represented during static analysis. In this paper, a CDN represent such polymorphic method calls by using a way that all possible destinations of a method call are included in the representation, unless the type can be determined statically. We use a *polymorphic choice* vertex similar to [13] to represent the possible destinations of the polymorphic call in the CDN. A polymorphic choice vertex represents the selection of a particular call given a set of possible destinations.

3.3 The System Dependence Net

In this subsection, we discuss how to construct the system dependence net for a concurrent object-oriented program. A system dependence net of a concurrent object-oriented program is an arc-classified digraph which consists of a collection of procedure dependence nets each representing a main procedure, a free standing procedure, or a method in a class of the program, and some additional arcs to represent direct program dependences between a call and the called procedure/method and transitive interprocedural data dependences.

We construct the system dependence net for a concurrent object-oriented program by connecting calls in the partial system dependence net to methods in the CDN for each class. It contains connecting call vertices to the start vertices of methods to form call arcs, actual-in vertices to formal-in vertices to form parameter-in arcs, and formal-out vertices to actual-out vertices to form parameter-out arcs. We add the summary arcs for methods in a previously analyzed class between the actual-in and actual-out vertices at call sites.

Moreover, to create system dependence nets, we connect procedure dependence nets for the main procedure and other free standing procedures in the program at call sites. A call arc is added between a procedure call vertex and the start vertex of the procedure dependence net of the called procedure. Actual-in and formal-in vertices are connected by parameter-in arcs, and formal-out and actual-out vertices are connected by parameter-out arcs.

Example. Figure 4 shows the complete SDN of the CC++ program in Figure 1. The construction of this
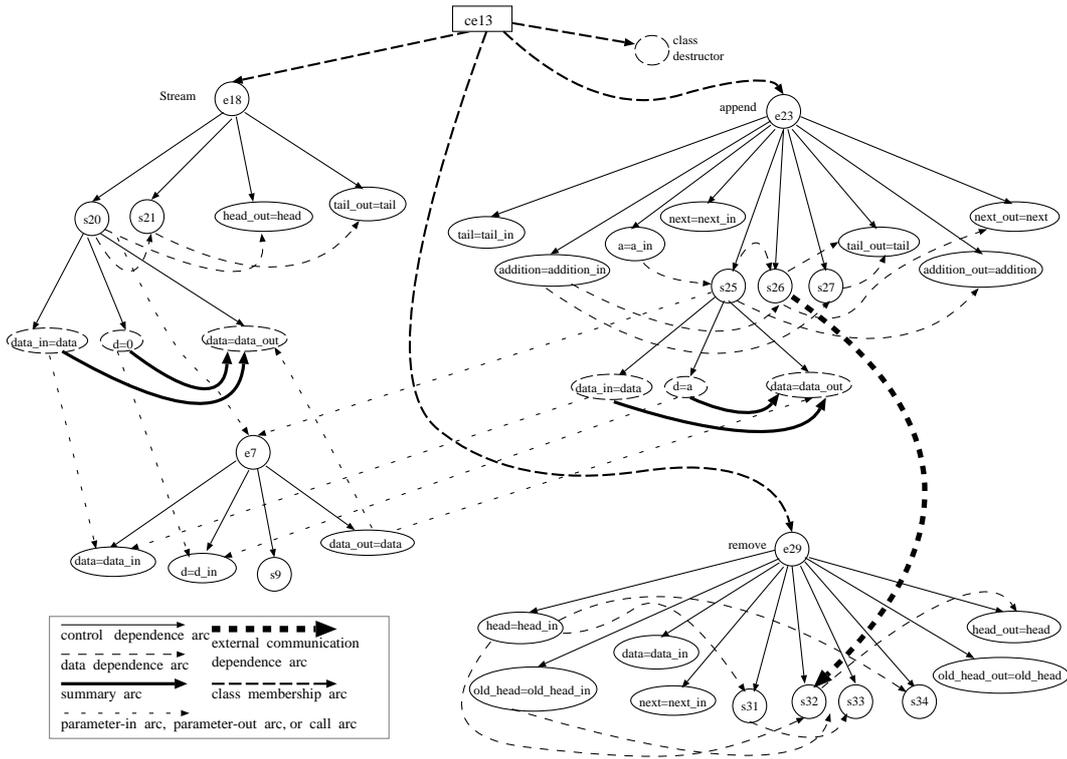
Figure 3:   A CC++ class `Stream` in Figure 1 and its class dependence net.

net includes the construction of the PDN of the `main` procedure and each free standing procedure including `producer` and `consumer`, the construction of the CDN of each class including `Stream`, and the connection of each net using call, parameter-in and parameter-out arcs.

4   Slicing Concurrent Object-Oriented Programs

In this section we show how to compute slices of a concurrent object-oriented program based on its SDN. Once a concurrent object-oriented program is represented by its SDN, the slices for the program can be computed as the ways similar to [9, 13] for imperative programs and sequential object-oriented programs.

4.1   Computation of a Static Slice

We first introduce some notions about statically slicing a concurrent object-oriented program.

A *static slicing criterion* for a concurrent object-oriented program is a tuple (s, v), where s is a statement in the program and v is a variable used at s, or a method call called at s. A *static slice* $SS(s, v)$ of a concurrent object-oriented program on a given static slicing criterion (s, v) consists of all statements in the program that possibly affect the value of the variable v at s or the value returned by the method call v at s.

Statically slicing a concurrent object-oriented program on a given static slicing criterion is to find the static slice of the program with respect to the criterion.

Note that object-oriented program developers prevent users of a class from directly manipulating instance variables within an object (i.e., the state of object) by providing methods as an interface for setting and observing the state of the object. Therefore, object-oriented programs substitute many variable references with method calls that simply return a value. In order to let us compute slice on the values returned by a method, we use the above slicing criterion which takes method calls into account.

Since the SDN proposed in this paper for concurrent object-oriented programs is an extension of the SDGs in [9, 13], we can use the two-pass slicing algorithm [9, 13] to compute slices of a concurrent object-oriented program based on the system dependence net.

The two-pass slicing algorithm proposed by Horwitz *et al.* can be briefly described as follows based on the SDG. The details for the algorithm can be found in [9, 15].

In the first phase, the algorithm traverses backward along all arcs except parameter-out arcs, and marks those vertices reached in the SDG. In the second phase, the algorithm traverses backward from all vertices marked during the first phase along all arcs except call and parameter-in arcs, and marks reached
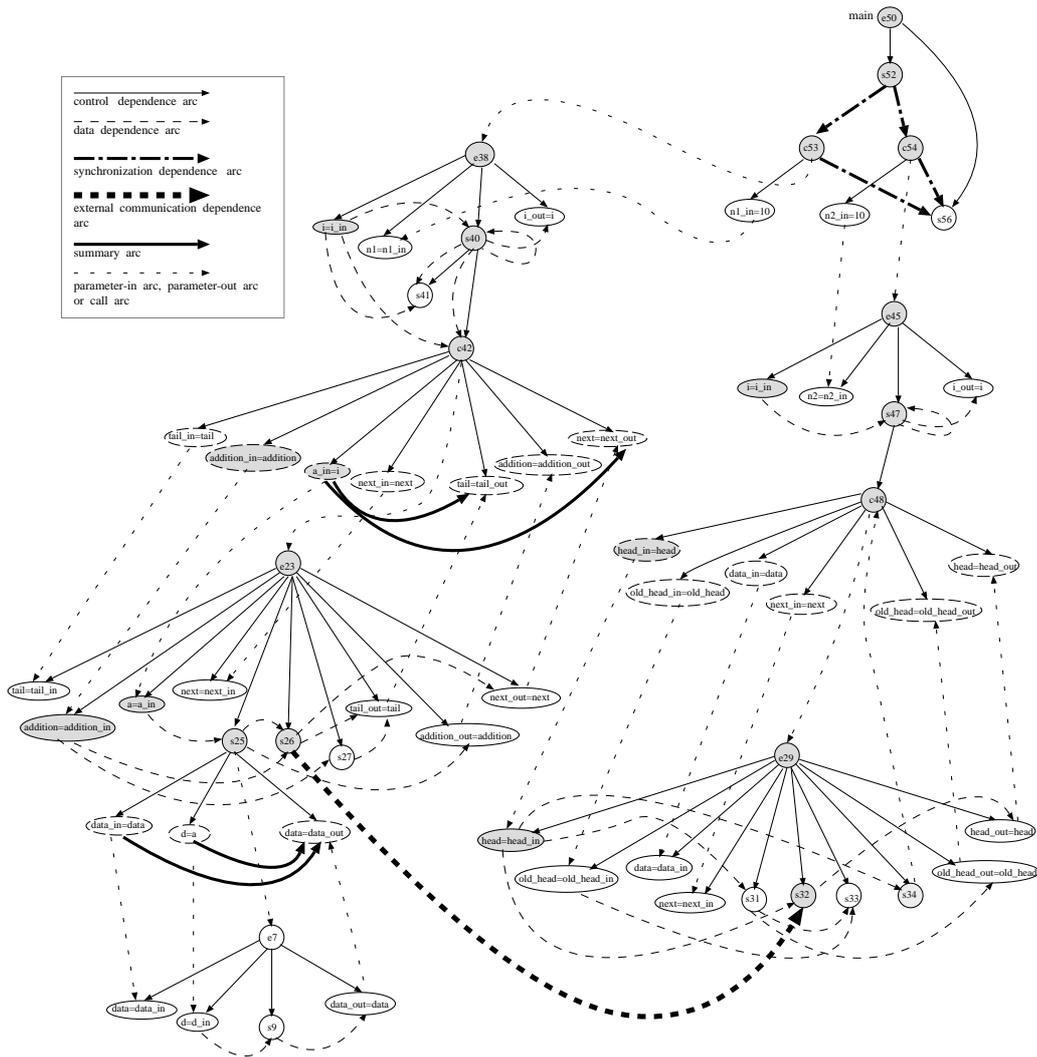
Figure 4: The SDN of the program in Figure 1 and a slice with shaded vertices.

vertices in the SDG. The slice is the union of the vertices of the SDG marked during the first and second phases.

Example. We give an example to show how to compute a slice for a concurrent object-oriented program based on its SDN. Figure 4 shows the SDN of the program in Figure 1 and a slice. The slice is represented in shaded vertices and computed with respect with the slicing criterion (`s32, head`). On the first pass, the algorithm marks all shaded vertices except the formal-out vertex `addition_out=addition`. On the second pass, the algorithm marks the formal-out vertex `addition_out=addition`.

In software maintenance, one of the problems as we know is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. Therefore, to main-

tain a concurrent object-oriented program, it is necessary to know which statements in which processes will be affected by a modified statement. The needs can be satisfied by forward-slicing the program being maintained. Similar to the backward slicing described above, we can also apply some forward slicing algorithms [9] to our SDNs to compute forward slices. Note that we can reuse all analysis information because compared with the backward slices computed no additional information is required to compute forward slices.

Moreover, the SDN of a concurrent object-oriented program is useful in computing not only static slices of the program as we described previously but also dynamic slices of the program. For example, similar to the descriptions in [1, 6], we can compute a dynamic slice of a concurrent object-oriented program based on the corresponding static slice of the program and

the program's execution history information that can be collected by an execution monitor. The detailed discussion of this problem is beyond of the scope of this paper.

4.2 The Cost of Constructing the System Dependence Net

Since the SDN that we compute for concurrent object-oriented programs is an extension of the PDN [6] and the SDG defined in [9] for sequential imperative programs and the SDG defined in [13] for sequential object-oriented programs, we can directly apply the results of cost analysis for the PDN and SDGs to our approach. The details for the cost analysis of SDGs for sequential imperative programs with multiple procedures and sequential object-oriented programs can be found in [9, 13, 15], and the details for the cost analysis of PDNs for concurrent imperative program with single procedure can be found in [10].

5 Conclusions

We have proposed a new program dependence representation named the *System Dependence Net*, which extended previous program dependence representations to represent concurrent object-oriented programs. The system dependence net of a concurrent object-oriented program can be used to represent not only object-oriented features but also concurrency issues in the program, and allows us to compute slices of the program efficiently. A system dependence net of a concurrent object-oriented program consists of a collection of procedure dependence nets each representing a main procedure, a free standing procedure, or a method in a class of the program, and some additional arcs to represent direct program dependences between a call and the called procedure/method and transitive interprocedural data dependences. Although here we presented the approach in term of CC++, other versions of this approach for other concurrent object-oriented programming languages are easily adaptable because they share their basic execution mechanisms with CC++.

Since program slicing of imperative programs plays important roles in program debugging, testing, maintenance, and understanding, we believe that the approach to slicing concurrent object-oriented programs proposed in this paper are also useful in development of concurrent object-oriented programs. Now we are developing a slicing tool based on the approach proposed in this paper for slicing concurrent object-oriented programs and also intend to develop a debugger to localize bugs by combining program slicing technique. The target language for development is the CC++ which is one of the most promising concurrent object-oriented programming language.

References

[1] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.

[2] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.

[3] J. M. Bieman, L. M. Ott, "Measuring Functional Cohesion," *IEEE Transaction on Software Engineering*, Vol.20, No.8, pp.644-657, 1994.

[4] P. Carlin, M. Chandy and C. Kesselman, *"The Compositional C++ Language Definition,"* Technical Report CS-TR-93-02, Department of Computer Science, California Institute of Technology, 1993.

[5] K. M. Chandy, C. Kesselman, *"CC++: A declarative Concurrent Object-Oriented Programming Notation,"* in G. Agha, P. Wegner, and A. Yonezawa (ed.) *Research Directions in Concurrent Object-Oriented Programming*, pp.281-313, MIT Press, 1993.

[6] J. Cheng, "Slicing Concurrent Programs – A Graph-Theoretical Approach," *Proceedings of First International Workshop on Automated Algorithmic Debugging, Lecture Notes in Computer Science*, Vol.749, pp.223-240, Springer-Verlag, May, 1993.

[7] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.

[8] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.

[9] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.

[10] Y. Kasahara, J. Cheng and K. Ushijima, "A Task Dependence Net Generator for Concurrent Ada Programs," *Proceeding of the IPSJ & KISS Jpont International Conference on Software Engineering*, pp.315-322, Japan, November 1993.

[11] M. Kamkar, N. Shahmehri, P. Fritzson, "Bug Localization by Algorithmic Debugging and Program Slicing," *Proceedings of International Workshop on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science*, Vol.456, pp.60-74, Springer-Verlag, 1990.

[12] A. Krishnaswamy, *"Program Slicing: An Application of Object-oriented Program Dependency Graphs,"* Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[13] L. D. Larsen and M. J. Harrold, *"Slicing Object-Oriented Software,"* *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.

[14] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

[15] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding Up slicing," *Proceeding of Second ACM Conference on Foundations of Software Engineering*, pp.11-20, December 1994.

[16] P. A. G. Sivilotti and P. A. Carlin, *"A Tutorial for CC++,"* Technical Report CS-TR-94-02, Department of Computer Science, California Institute of Technology, 1994.

[17] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, Vol.3, No.3, pp.121-189, September, 1995.

[18] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.