Slicing Concurrent Java Programs

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-02, Japan
zhao@cs.fit.ac.jp

Abstract

Although many slicing algorithms have been proposed for object-oriented programs, no slicing algorithm has been proposed which can be used to handle the problem of slicing concurrent Java programs correctly. In this paper, we propose a slicing algorithm for concurrent Java programs. To slice concurrent Java programs, we present a dependence-based representation called multithreaded dependence graph, which extends previous dependence graphs, to represent concurrent Java programs. We also show how static slices of a concurrent Java program can be computed efficiently based on its multithreaded dependence graph.

1 Introduction

Java is a new object-oriented programming language and has achieved widespread acceptance because it emphasizes portability. Java has multithreading capabilities for concurrent programming. To provide synchronization between asynchronously running threads, the Java language and runtime system uses *monitors*. Because of the nondeterministic behaviors of concurrent Java programs, predicting, understanding, and debugging a concurrent Java program is more difficult than a sequential object-oriented program. As concurrent Java applications are going to be accumulated, the development of techniques and tools to support understanding, debugging, testing, and maintenance of concurrent Java software will become an important issue.

Program slicing, originally introduced by Weiser [19], is a decomposition technique which extracts the elements of a program related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. The task to compute program slices is called *program slicing*. To understand the basic idea of program slicing, consider a simple example in Figure 1 which shows: (a) a program fragment and (b) its slice with respect to the slice criterion (`Total`,14). The slice has only those statements in the program that might affect the value of variable `Total` at line 14. The lines represented by small rectangles are statements that have been sliced away.

Program slicing has been studied primarily in the context of procedural programming languages (for a detailed survey, see [17]). In such languages, slicing is typically performed by using a control flow graph or a dependence graph [6, 11, 8, 15]. Program slicing has many

(a) A program fragment.

```
 1  begin
 2    read(X,Y);
 3    Total := 0.0;
 4    Sum := 0.0;
 5    if X <= 1 then
 6       Sum := Y;
 7       else
 8           begin
 9             read(Z);
10             Total := X * Y;
11          end;
12    end if
13    write(Total, sum);
14  end
```

(b) a slice of (a) on the criterion (Total,14).

```
 1  begin
 2    read(X,Y);
 3    Total := 0.0;
 4    [          ]
 5    if X <= 1 then
 6       [          ]
 7       else
 8           begin
 9             [          ]
10             Total := X * Y;
11          end;
12    end if
13    [                    ]
14  end
```

Figure 1: A program fragment and its slice on criterion (`Total`,14).

applications in software engineering activities such as program understanding [7], debugging [1], testing [10], maintenance [9], and reverse engineering [2].

As object-oriented software becomes popular, recently, researchers have applied program slicing to object-oriented programs to handle various object-oriented features such as classes and objects, class inheritance, polymorphism, dynamic binding [4, 5, 12, 13, 14, 18], and concurrency [20]. However, existing slicing algorithms for object-oriented programs can not be applied to concurrent Java programs straightforwardly to obtain correct slices due to specific features of Java concurrency model. In order to slice concurrent Java programs correctly, we must extend these slicing techniques for adapting concurrent Java programs.

In this paper we present the *multithreaded dependence graph* for concurrent Java programs on which

```
 ce1  class Producer extends Thread {
   2        private CubbyHole cubbyhole;
   3        private int number;
  e4        public Producer(CubbyHole c, int number) {
  s5            cubbyhole = c;
  s6            this.number = number;
   7        }
 te8        public void run() {
  s9          int i=0;
 s10          while (i<10) {
 s11            cubbyhole.put(i);
 s12            System.out.println("Producer #" +
                                this.number + "put:" + i);
 s13            sleep((ubt)(Math.random()*100));
 s14            i=i+1;
  15          }
  16        }
  17  }
ce18  class Consumer extends Thread {
  19        private CubbyHole cubbyhole;
  20        private int number;
 e21        public Consumer(CubbyHole c, int number) {
 s22            cubbyhole = c;
 s23            this.number = number;
  24        }
te25        public void run() {
 s26          int value = 0;
 s27          int i=0;
 s28          while (i<10) {
 s29            value = cubbyhole.get();
 s30            System.out.println("Consumer #" +
                                this.number + "get:" + value)
 s31            sleep((int)(Math.random()*100));
 s32            i=i+1;
  33          }
  34        }
  35  }
ce36  class CubbyHole {
  37        private int seq;
 s38        private boolean available = false;
me39        public synchronized int get() {
 s40           while (available == false) {
 s41                 wait();
  42           }
 s43            available = false;
 s44            notify();
 s45            return seq;
  46        }
me47        public synchronized int put(int value) {
 s48            while (available == true) {
 s49                 wait();
  50           }
 s51           seq = value;
 s52           available = true;
 s53           notify();
  54        }
  55  }
ce56  class ProducerConsumerTest {
me57        public static void main(string[] args) {
 s58           CubbyHole c = new CubbyHole();
 s59           Producer p1 = new Producer(c, 1);
 s60           Consumer c1 = new Consumer(c, 1);
 s61           p1.start();
 s62           c1.start();
  63        }
  64  }
```

Figure 2: A concurrent Java program.

static slices of the programs can be computed efficiently. The multithreaded dependence graph of a concurrent Java program is composed of a collection of thread dependence graphs each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions between different threads. Once a concurrent Java program is represented by its multithreaded dependence graph, the slices of the program can be computed by solving a vertex reachability problem in the graph.

The rest of the paper is organized as follows. Section 2 briefly introduces the concurrency model of Java. Section 3 discusses some related work. Section 4 presents the multithreaded dependence graph for concurrent Java programs. Section 5 shows how to compute static slices based on the graph. Concluding remarks are given in Section 7.

2 Concurrency Model in Java

Java supports concurrent programming with threads through the language and the runtime system. A thread, which is a single sequential flow of control within a program, is similar to the sequential programs in the sense that each thread also has a beginning, and execution sequence, and an end and at any given time during the runtime of the thread, there is a single point of execution. However, a thread itself in not a program; it can not run on its own, Rather, it runs within a program. Programs that has multiple synchronous threads are called *multithreaded programs* topically. Java provides a *Thread* class library, that defines a set of operations on one thread, like `start()`, `stop()`, `join()`, `suspend()` and `resume()`.

Java uses shared memory to support communication among threads. Objects shared by two or more threads are called *condition variables*, and the access on them must be synchronized. The Java language and runtime system support thread synchronization through the use of *monitors*. In general, a monitor is associated with a specific data item (a condition variable) and functions as a lock on that data. When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data. The code segments within a program that access the same data from within separate, concurrent threads are known as *critical sections*. In the Java language, you may mark critical sections in your program with the *synchronized* keyword. Java provides some methods of `Object` class, like `wait()`, `notify()`, and `notifyall()` to support synchronization among different threads. Using these operations and different mechanism, threads can cooperate to complete a valid method sequence of the shared object.

Figure 2 shows a simple concurrent Java program that implements the *Producer-Consumer* problem. The program creates two threads `Producer` and `Consumer`. The `Producer` generates an integer between 0 and 9 and stores it in a `CubbyHole` object. The `Consumer` consumes all integers from the `CubbyHole` as quickly as they become available. Threads `Producer` and `Consumer` in this example share data through a common `CubbyHole` object. However, to execute the program correctly, the following condition must be satisfied, that is, the `Producer` can not put any new integer into the `CubbyHole` unless the previously put integer has been extracted by the `Consumer`, while the `Consumer` must wait for the `Producer` to put a new integer in the `CubbyHole` is empty.

In order to satisfy the above condition, the activities of the `Producer` and `Consumer` must be synchronized in two ways. First, the two threads must not simultaneously access the `CubbyHole`. A Java thread can handle this through the use of *monitor* to lock an object as described previously. Second, the two threads must do some simple cooperation. That is, the `Producer` must have some way to inform the `Consumer` that the value is ready and the `Consumer` must have some way to inform the `Producer` that the value has been extracted. This can be done by using a collection of methods: `wait()` for helping threads wait for a condition, and `notify()`

and `notifyAll()` for notifying other threads of when that condition changes.

3 Program Slicing for Object-Oriented Programs

In this section, we review some related work on program slicing which directly or indirectly influence our work on slicing concurrent Java programs, and explain why these slicing algorithms can not be applied to concurrent Java programs straightforwardly.

Larsen and Harrold [13] proposed a static slicing algorithm for sequential object-oriented programs. They extended the *system dependence graph* (SDG) [11] which was first proposed to handle interprocedural slicing of sequential procedural programs to the case of sequential object-oriented programs. Their SDG can be used to represent many object-oriented features such as classes and objects, polymorphism, and dynamic binding. Since the SDG they compute for sequential object-oriented programs belong to a class of SDGs defined in [11], they can use the two-pass slicing algorithm in [11] to compute slices of sequential object-oriented programs. Chan and Yang [4] adopted a similar way to extend the SDG for sequential procedural programs [11] to sequential object-oriented programs, and use the extended SDG for computing static slices of sequential object-oriented programs. On the other hand, Krishnaswamy [12] proposed another approach to slicing sequential object-oriented programs. He uses a dependence-based representation called the *object-oriented program dependency graph* to represent sequential object-oriented programs and compute polymorphic slices of sequential object-oriented programs based on the graph. Chen *et al.* [5] also extended the program dependence graph to the *object-oriented dependency graph* for modeling sequential object-oriented programs. Although these representations can be used to represent many features of sequential object-oriented programs, they lack the ability to represent concurrency. Therefore, the slicing algorithms based on these representations can not compute static slices of a concurrent Java program correctly.

Slicing object-oriented programs with concurrency issues has also been considered. Zhao *et al.* [20] presented a dependence-based representation called the *system dependence net* to represent concurrent object-oriented programs (especially Compositional C++ (CC++) programs [3]). In CC++, synchronization between different threads is realized by using a single-assignment variable. Threads that share access to a single-assignment variable can use that variable as a synchronization element. Their system dependence net is a straightforward extension of the SDG of Larsen and Harrold [13], and therefore can be used to represent many object-oriented features in a CC++ program. To handle concurrency issues in CC++, they used a approach proposed by Cheng [6] which originally used for representing concurrent procedural programs with single procedures. However, their approach, when applied to concurrent Java programs, has some problems due to the following reason. The concurrency models of CC++ and Java are essentially different. While Java supports monitors and some low-level thread synchronization primitives, CC++ uses a single-assignment variable mechanism to realize thread synchronization. This difference leads to different sets of concurrency constructs in both languages, and therefore requires different techniques to handle.

4 A Dependence Model for Concurrent Java Programs

Generally, a concurrent Java program has a number of threads each having its own control flow and data flow. These information flows are not independent because inter-thread synchronizations among multiple control flows and inter-thread communications among multiple data flows may exist in the program. To represent concurrent Java programs, we present a dependence-based representation called the *multithreaded dependence graph*. The multithreaded dependence graph of a concurrent Java program is composed of a collection of thread dependence graphs each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions between different threads. In this section, we show how to construct the thread dependence graph for a single thread and the multithreaded dependence graph for a complete concurrent Java program.

4.1 Thread Dependence Graphs for Single Threads

The *thread dependence graph* (TDG) is used to represent a single thread in a concurrent Java program. It is similar to the SDG presented by Larsen and Harrold [13] for modeling a sequential object-oriented program. Since execution behavior of a thread in a concurrent Java program is similar to that of a sequential object-oriented program. We can use the technique presented by Larsen and Harrold for constructing the SDG of sequential object-oriented programs to construct the thread dependence graph. The detailed information for building the SDG of a sequential object-oriented program can be found in [13]. In the following we briefly describe our construction method.

The TDG of a thread is an arc-classified digraph that consists of a number of method dependence graphs each representing a method that contributes to the implementation of the thread, and some special kinds of dependence arcs to represent direct dependencies between a call and the called method and transitive interprocedural data dependencies in the thread. Each TDG has a unique vertex called *thread entry vertex* to represent the entry into the thread.

The *method dependence graph* of a method is an arc-classified digraph whose vertices represent statements or control predicates of conditional branch statements in the method, and arcs represent two types of dependencies: *control dependence* and *data dependence*. Control dependence represents control conditions on which the execution of a statement or expression depends in the method. Data dependence represents the data flow between statements in the method. For each method dependence graph, there is a unique vertex called *method entry vertex* to represent the entry into the method. For example, `me39` and `me47` in Figure 3 are method entry vertices for methods `get()` and `put()`.

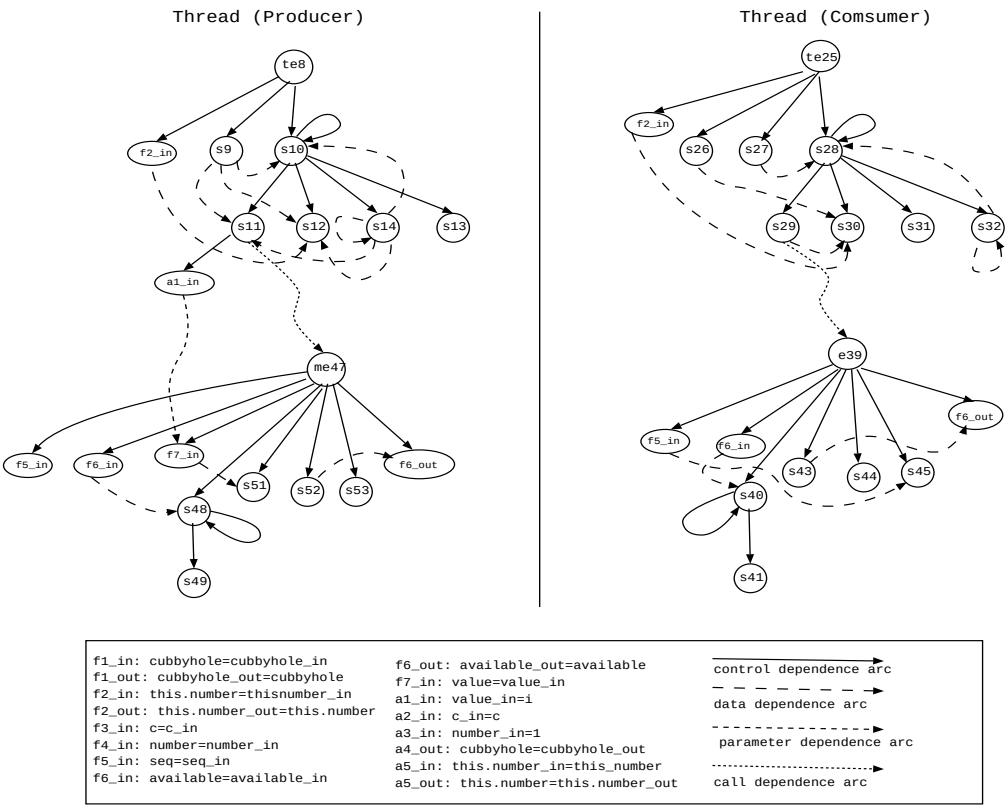

Figure 3: The TDGs for threads `Producer` and `Consumer`.

In order to model parameter passing between methods in a thread, each method dependence graph also includes formal parameter vertices and actual parameter vertices. At each method entry there is a *formal-in vertex* for each formal parameter of the method and a *formal-out vertex* for each formal parameter that may be modified by the method. At each call site in the method, a *call vertex* is created for connecting the called method, and there is an *actual-in vertex* for each actual parameter and an *actual-out vertex* for each actual parameter that may be modified by the called method. Each formal parameter vertex is control-dependent on the method entry vertex, and each actual parameter vertex is control-dependent on the call vertex.

Some special kinds of dependence arcs are created for combining method dependence graphs for all methods in a thread to form the whole TDG of the thread.

- A *call dependence arc* represents call relationships between a call method and the called method, and is created from the call site of a method to the entry vertex of the called method.
- A *parameter-in dependence arc* represents parameter passing between actual parameters and formal input parameter (only if the formal parameter is at all used by the called method).
- A *parameter-out dependence arc* represents parameter passing between formal output parameters and actual parameters (only if the formal parameter is at all defined by the called method). In addition, for methods, parameter-out dependence arcs also represent the data flow of the return value between the method exit and the call site.

Figure 3 shows two TDGs for threads `Producer` and `Consumer`. Each TDG has an entry vertex that corresponds to the first statement in its `run()` method. For example, in Figure 3 the entry vertex of the TDG for thread `Producer` is `te8`, and the entry vertex of the TDG for thread `Consumer` is `te25`.

4.2 Multithreaded Dependence Graphs for Concurrent Java Programs

The *multithreaded dependence graph* (MDG) of a concurrent Java program is an arc-classified digraph which consists of a collection of TDGs each representing a single thread, and some special kinds of dependence arcs to model thread interactions between different threads in the program. There is an entry vertex for the MDG representing the start entry into the program, and a method dependence graph constructed for the `main()` method.

To capture the synchronization between thread synchronization statements and communication between shared objects in different threads, we define some special kinds of dependence arcs in the MDG.

4.2.1 Synchronization Dependencies

We use synchronization dependence to capture dependence relationships between different threads due to inter-thread synchronization.

- Informally, a statement u in one thread is

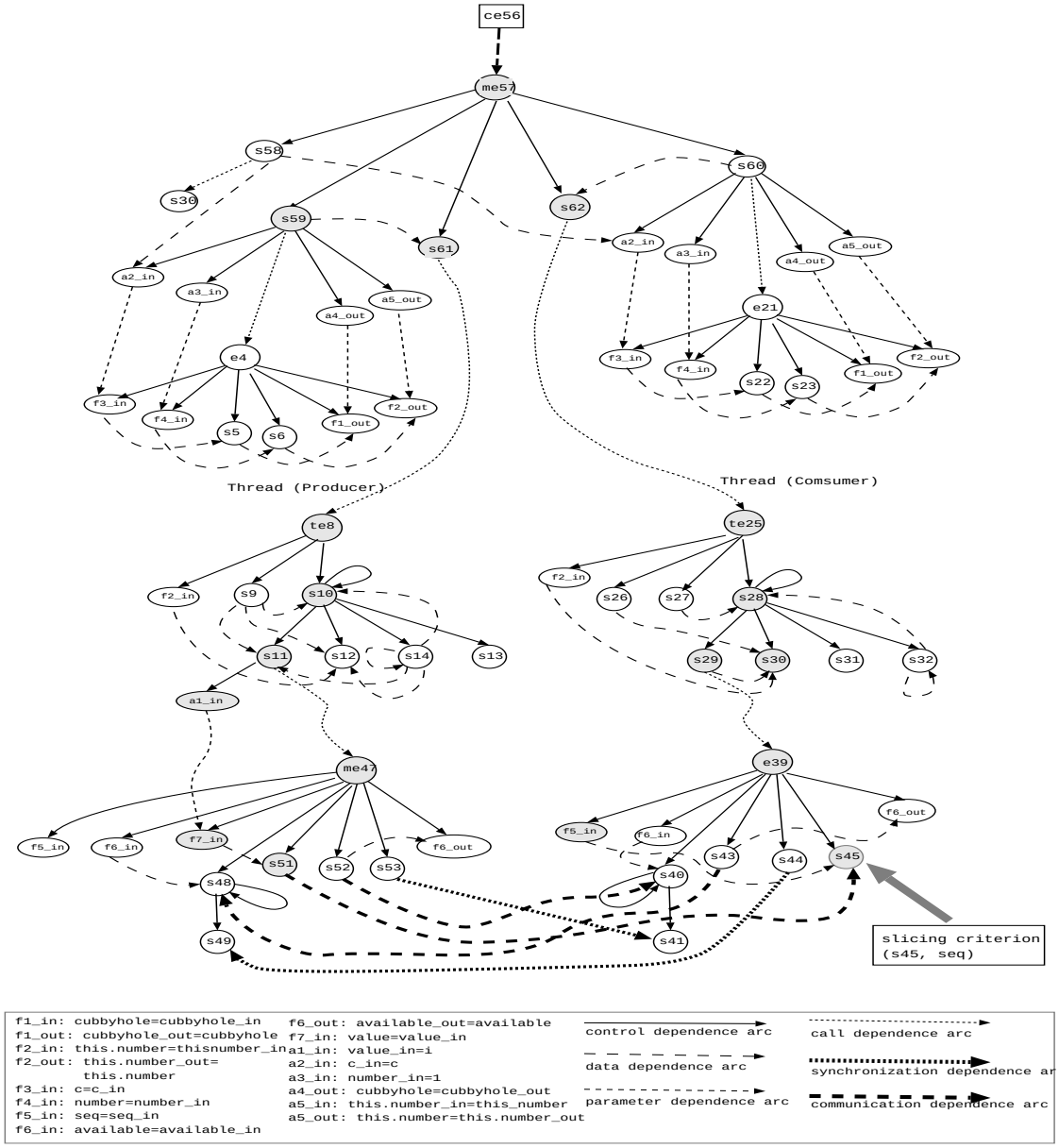

Figure 4: The MDG of a concurrent Java program in Figure 2.

synchronization-dependent on a statement v in another thread if the start or termination of the execution of u directly determinates the start or termination of the execution of v through an inter-thread synchronization.

In Java synchronization dependencies among different threads may be caused in several ways. We show how to create synchronization dependence arc for each of them.

(1) Wait-notify Relationships

A synchronization can be realized by using `wait()` and `notify()/notifyall()` method calls in different threads. For such a case, a synchronization dependence arc is created from a vertex u to a vertex v if u denoted a `notify()` or `notifyall()` call in thread `t1` and v denotes a `wait()` call in thread `t2` for some thread object `o`, where threads `t1` and `t2` are different. A special case is that there are more than one threads waiting for the notification from a thread `t`. For such a case, we create synchronization dependence arcs from the vertex denoted `notify()` call of `t` to each vertex denoted `wait()` call of the other threads respectively.

For example, in the program of Figure 2, methods `put()` and `get()` use Java Object's `notify()` and `wait()` methods to cooperate their activities. This means that there exists synchronization dependencies between `wait()` method call in `Producer` and `notify()` method call in `Comsumer`, and between `notify()` method call in `Producer` and `wait()` method call in `Consumer`. So we can create synchronization dependence arcs between `s53` and `s41`, and between `s44` and `s49` as showed in Figure 4.

(2) Stop-join Relationships

Another case that may cause inter-thread synchronization is the stop-join relationship, that is, a thread calling the `join()` method of another thread may proceed only after this target thread terminates. For such a case, a synchronization dependence arc is created from a vertex u to a vertex v if u denotes the last statement in thread `t1` and v denotes a `join()` call in thread `t2`, where threads `t1` and `t2` are different.

4.2.2 Communication Dependencies

We use communication dependence to capture dependence relationships between different threads due to inter-thread communication.

- Informally a statement u in one thread is directly communication-dependent on a statement v in another thread if the value of a variable computed at u has direct influence on the value of a variable computed at v through an inter-thread communication.

Java uses shared memory to support communication among threads. Communications may occur when two parallel executed threads exchange their data via shared variables. In such a case, a communication dependence arc is created from a vertex u to a vertex v if u denotes a statement s_1 in thread `t1` and v denotes a statement s_2 in thread `t2` for some thread object `o`, where s_1 and s_2 shares a common variable and `t1` and `t2` are different. A special case is that there is more than one thread waiting for the notification from some thread `t`, and there is an attribute `a` shared by these threads as a communication element. In such a case, we create communication dependence arcs from each statement containing variable `a` of the threads to the statement containing variable `a` in thread `t` respectively.

For example, in the program of Figure 2, methods `put()` and `get()` use Java Object's `notify()` and `wait()` methods to cooperate their activities. In this way, each `seq` placed in the `CubbyHole` by the `Producer` is extracted once and only once by the `Consumer`. By analyzing the source code we know that there exist inter-thread communication between statement `s51` in thread `Producer` and statement `s45` in `Comsumer` which share variable `seq`. Similarly, inter-thread communications may also occur between statements `s52` and `s40` and between `s43` and `s48` due to shared variable `available`. As a result, communication dependence arcs can be created from `s52` to `s40`, `s51` to `s45`, and `s43` to `s48` as showed in Figure 4.

4.2.3 Constructing the MDG

In Java, any program begins execution with `main()` method. The thread of execution of the `main` method is the only thread that is running when the program is started. Execution of all other threads is started by calling their `start()` methods, which begins execution with their corresponding `run()` methods. The construction of the MDG for a complete concurrent Java program can be done by combining the TDGs for all threads in the program at synchronization and communication points by adding synchronization and communication dependence arcs between these points. For this purpose, we create an entry vertex for the MDG that represents the entry into the program, and construct a method dependence graph for the `main()` method. Moreover, a *start arc* is created from each `start()` method call in the `main()` method to the corresponding thread entry vertex. Finally, synchronization and communication dependence arcs are created between statements related to thread interaction in different threads. Note that in this paper, since we focus on concurrency issues in Java, many sequential object-oriented features that may also exist in a concurrent Java program are not discussed. However, how to represent these features in sequential object-oriented programs using dependence graphs has already been discussed by some researchers [4, 5, 12, 13]. Their techniques can be directly integrated into our MDG for concurrent Java programs. Figure 4 shows the MDG for the program in Figure 2. It consists of two TDGs for threads `Producer` and `Consumer`, and some additional synchronization and communication dependence arc to model synchronization and communication between `Producer` and `Consumer`.

5 Slicing Concurrent Java Programs

In this section, we show how to compute static slices of a concurrent Java program. We focus on two types of slicing problems: slicing a single thread based on the TDG and slicing the whole program based on the MDG.

5.1 Slicing a Single Thread

In addition to slicing a complete concurrent Java program, sometimes we need a slice on a single thread of the program for analyzing a single thread independently which the thread interactions can be ignored. Such a slice can be computed based on its TDG. We define some slicing notions of a single thread as follows.

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

As we mentioned previously, the TDG for a thread is similar to the SDG for a sequential object-oriented program [13]. Therefore, we can use the two-pass slicing algorithm in [11] to compute static slices of the thread based on its TDG. In the first phase, the algorithm traverses backward along all arcs except parameter-out arcs, and set marks to those vertices reached in the TDG. In the second phase, the algorithm traverses backward from all vertices having marks during the first phase along all arcs except call and parameter-in arcs, and sets marks to reached vertices in the TDG. The slice is the union of the vertices in the TDG that have marked during the first and second phases.

Similarly, we can also apply the forward slicing algorithm [11] to the TDG to compute forward slices of a thread.

5.2 Slicing a Complete Program

Slicing a complete concurrent Java program may also use the two-pass slicing algorithm proposed in [11], which the MDG can be used as a base to compute static

slices of the program. This is because that the MDG for a concurrent Java program can also be regarded as an extension of the SDG for a sequential object-oriented program [13]. Whereas slicing a single thread does not involve thread interactions, slicing a complete concurrent Java program may definitely involve some thread interactions due to synchronization and communication dependencies between different threads. In the following we first define some notions for static slicing of a concurrent Java program, then give our slicing algorithm that is based on [11].

A *static slicing criterion* for a concurrent Java 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* of a concurrent Java 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.

The two-pass slicing algorithm based on MDG can be described as follows. In the first step, the algorithm traverses backward along all arcs except parameter-out arcs, and set marks to those vertices reached in the MDG. In the second step, the algorithm traverses backward from all vertices having marks during the first step along all arcs except call and parameter-in arcs, and sets marks to reached vertices in the MDG. The slice is the union of the vertices of the MDG have marks during the first and second steps. Figure 4 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion `(s45, seq)`.

Similarly, we can also apply the forward slicing algorithm [11] to the MDG to compute forward slices of concurrent Java programs.

In addition to computing static slices, the MDG is also useful for computing dynamic slices of a concurrent Java program. For example, we can use a slicing algorithm, similar to [1, 6], to compute dynamic slices of a concurrent Java program based on the corresponding static slices and execution history information of the program.

Program slicing is useful in program understanding. For example, when we attempt to understand the behavior of a concurrent Java program, we usually want to know which statements might affect a statement of interest, and which statements might be affected by the execution of a statement of interest in the program. These requirements can be satisfied by slicing the program using an MDG-based slicing and forward-slicing algorithms introduced in this paper.

6 Cost of Constructing the MDG

The size of the MDG is critical for applying it to the practical development environment for concurrent Java programs. In this section we try to predicate the size of the MDG based on the work of Larsen and Harrold [13] who give an estimate of the size of the SDG for a sequential object-oriented program. Since each TDG in an MDG is similar to the SDG of a sequential object-oriented program, we can apply their approximation here to estimate the size of the TDG for a single thread in a concurrent Java program. The whole cost of the MDG for the program can be got by combining the sizes of all TDGs in the program.

Table 1 lists the variables that contribute to the size of a TDG. We give a bound on the number of parameters for any method (ParamVertices(m)), and use this bound to compute upper bound on the size of a method (Size(m)). Based on the Size(m) and the number of methods Methods in a single thread, we can compute the upper bound Size(TDG) on the number of vertices in a TDG including all classes that contribute to the size of the thread.

ParamVertices(m) = Params+ObjectVar+ClassVar.
Size(m) = O(Vertices*CallSites*(1+TreeDepth*
(2*ParamVertices))+2*ParamVertices).
Size(TDG) = O(Size(m) * Methods).

Based on the above result of a single thread, we can compute the upper bound on the number of vertices Size(MDG) in an MDG for a complete concurrent Java program including all threads.

Size(MDG) = $\sum_{i=1}^{k}$ Size(TDG_i).

Note that Size(TDG) and Size(MDG) give only a rough upper bound on the number of vertices in a TDG and an MDG. In practice we believe that a TDG and an MDG may be considerably more space efficient.

7 Concluding Remarks

In this paper we presented the *multithreaded dependence graph* (MDG) on which static slices of concurrent Java programs can be computed efficiently. The MDG of a concurrent Java program consists of a collection of thread dependence graphs each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions. Once a concurrent Java program is represented by its MDG, the slices of the program can be computed by solving a vertex reachability problem in the graph. Although here we presented the approach in term of Java, we believe that many aspects of our approach are more widely applicable and could be applied to slicing of programs with a monitor-like synchronization primitive, i.e., Ada95's protected types. Moreover, in order to develop a practical slicing algorithm for concurrent Java programs, some specific features in Java such as interfaces and packages must be considered. In [22], we presented a technique for constructing a dependence graph to represent interfaces and packages in a sequential Java program. Such a technique can be integrated directly into the MDG for representing interfaces and packages in concurrent Java programs.

The slicing technique introduced in this paper can only handle the problem of statement slicing. For large-scale software systems developed in Java, statement slicing may not be efficient because the system usually contains numerous components. For such a case, a new slicing technique called *architectural slicing* [21] can be used to perform slicing at the architectural level of the system. In contrast to statement slicing, architectural slicing can provide knowledge about the high-level structure of a software system [21]. We are considering to integrate the architectural slicing into our statement slicing framework to support slicing of large-scale software systems developed in Java not only at the statement level but also at the architectural level. We believe that this approach can be helpful in understanding

Table 1 Parameters which contribute to the size of a TDG.

Vertices	Large number of statements in a single method
Arcs	Large number of arcs in a single method
Params	Largest number of formal parameters for any method
ClassVar	Largest number of class variables in a class
ObjectVar	Largest number of instance variables in a class
CallSites	Largest number of call sites in any method
TreeDepth	Depth of inheritance tree determining number of possible indirect call destinations
Method	Number of methods

large-scale software systems developed in Java.

Now we are implementing a slicing tool using JavaCC [16], a Java parser generator developed by Sun Microsystes, to computing static slices of a concurrent Java program based on its MDG.

Acknowledgements

The author would like to thank the anonymous referees for their valuable suggestions and comments on earlier drafts of the paper.

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] J. Beck and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceeding of the 15th International Conference on Software Engineering*, pp.509-518, Baltimore, Maryland, IEEE Computer Society Press, 1993.

[3] 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.

[4] J.T. Chan and W. Yang, "A Program Slicing System for Object-Oriented Programs," *Proceedings of the 1996 International Computer Symposium*, Taiwan, December 19-21, 1996.

[5] J. L. Chen, F. J. Wang, and Y. L. Chen, "Slicing Object-Oriented Programs," *Proceedings of the APSEC'97*, pp.395-404, Hongkong, China, December 1997.

[6] J. Cheng, "Slicing Concurrent Programs – A Graph-Theoretical Approach," in P. A. Fritzson (Ed.), "Automated and Algorithmic Debugging, AADEBUG '93," Lecture Notes in Computer Science, Vol.749, pp.223-240, Springer-Verlag, 1993.

[7] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding Function Behaviors through Program Slicing," *Proceedings of the Fourth Workshop on Program Comprehension*, Berlin, Germany, March 1996.

[8] J.Ferrante, K.J.Ottenstein, and 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.

[9] 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.

[10] R. Gupta, M. J. Harrold, and M. L. Soffa, "Program Slicing-Based Regression Testing Techniques," *Journal of Software Testing, Verification, and Reliability*, Vol.6, No.2, pp.83-112, June 1996.

[11] 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.

[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] R. C. H. Law and R. B. Maguire, "Debugging of Object-Oriented Software," *Proceeding of the 8th International Conference on Software Engineering and Knowledge Engineering*, pp.77-84, June 1996.

[15] 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.

[16] Sun Microsystems, *http://www.suntest.com/JavaCC*.

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

[18] F. Tip, J. D. Choi, J. Field, and G. Ramalingam "Slicing Class Hierarchies in C++," *Proceedings of the 11th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.179-197, October, 1996.

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

[20] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pp.312-320, August 1996.

[21] J. Zhao, " Applying Slicing Technique to Software Architectures," *Proc. Fourth IEEE International Conference on Engineering of Complex Computer Systems*, pp.87-98, August 1998.

[22] J. Zhao, "Applying Program Dependence Analysis to Java Software," *Proc. Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pp.162-169, Tainan, TAIWAN, December 1998.