Multithreaded Dependence Graphs for 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

Understanding program dependencies in a computer program is essential for many software engineering activities including program slicing, testing, debugging, reverse engineering, and maintenance. In this paper, we present a dependence-based representation called *multithreaded dependence graph*, which extends previous dependence-based representations, to represent program dependencies in a concurrent Java program. We also discuss some important applications of a multithreaded dependence graph in a maintenance environment for concurrent Java programs.

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, maintenance, complexity measurement of concurrent Java software will become an important issue.

Program dependencies are dependence relationships holding between program statements in a program that are implicitly determined by the control flows and data flows in the program. Intuitively, if the computation of a statement directly or indirectly affects the computation of another statement in a program, there might exist some program dependence between the statements. Program dependence analysis is the process to determine the program's dependencies by analyzing the control flows and data flows in the program.

Many compiler optimizations and program analysis and testing techniques rely on program dependence information, which is topically represented by a *dependence-based representation* (DBR), for example, a *program dependence graph* (PDG) [7, 13]. The PDG, although originally proposed for compiler optimizations, has been used for various software engineering activities including program slicing, debugging, testing, maintenance, and complexity measurements [1, 2, 6, 11, 16, 17]. For example, program slicing, a decomposition technique that extracts program statements related to a particular computation, is greatly benefit from a PDG on which the slicing problem can be reduced to a vertex reachability problem [16] that is much simpler than its original algorithm [23].

DBRs were originally constructed for procedural programs. Recently, as object-oriented software become popular, researchers have applied DBRs to object-oriented programs to represent various object-oriented features such as classes and objects, class inheritance, polymorphism and dynamic binding [4, 5, 12, 14, 15], and concurrency [24, 26]. (for detailed discussions, see related work section).

However, the existing DBRs for object-oriented software can not be applied to concurrent Java programs straightforwardly due to the specific features of Java concurrency model. In order to represent the full range of concurrent Java program, we must extend the existing DBRs for representing concurrent Java programs.

In this paper we present a DBR called *multithreaded dependence graph*, which extends previous DBRs, to represent various types of dependencies in a concurrent Java program. The multithreaded dependence graph 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 between different threads. Finally, we discuss some important applications of a multithreaded dependence graph in a maintenance environment for concurrent Java programs.

The rest of the paper is organized as follows. Section 2 briefly introduces the concurrency model of Java. Section 3 discusses some related work and explains why existing DBRs can not be used to represent concurrent Java programs correctly. Section 4 presents the mul-

tithreaded dependence graph for concurrent Java programs. Section 5 discusses some applications of the graph. Concluding remarks are given in Section 6.

2 Concurrency Model in Java

Java supports concurrent programming with threads through the language and the runtime system. A thread is a single sequential flow of control within a program. It is similar to a sequential program in the sense that each thread also has a start, a sequence of execution, and a stop and at any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program because it can not run independently, rather, it can only run 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 Java, 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.

Any Java program begins its execution from the `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.

Figure 1 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, i.e., the `Producer`

```
 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 1: A concurrent Java program.

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 Related Work

As an essential representation useful for compiler optimization and program analysis, a DBR has been widely studied in the literatures. In this section, we review some related work on DBRs which directly or indirectly influence our work, and explain why these DBRs can not be applied to concurrent Java programs. Although DBRs have been widely studied in the context of procedural programming languages, it is the first time, to our knowledge, to apply existing DBRs to concurrent Java programs.

3.1 DBRs for Procedural Programs

Ferrante *et al.* [7] presented a DBR called *program dependence graph* to explicitly represent control and data dependencies in a sequential procedural program with single procedure. Horwitz *et al.* [9] extended the program dependence graph to introduce a DBR called *system dependence graph* (SDG) to represent a sequential procedural program with multiple procedures. Cheng [6] presented a DBR called *process dependence net* which is the generalization of the program dependence graph to represent program dependencies in a concurrent procedural program with single procedure. Although these DBRs can be used to represent many features of a procedural program, they lack the ability to represent object-oriented features in concurrent Java programs.

3.2 DBRs for Object-Oriented Programs

Larsen and Harrold [14] extended the SDG which was first proposed to handle interprocedural slicing of sequential procedural programs [9] to the case of sequential object-oriented programs. Their SDGs can be used to represent many object-oriented features such as classes and objects, polymorphism, and dynamic binding. Since the SDGs 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, 18] for sequential procedural programs to compute slices of sequential object-oriented programs. Chan and Yang [4] adopted a similar way to extend the SDGs for sequential procedural programs [9] 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 DBR 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 object-oriented program dependency 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 DBRs can be used to represent many features of sequential object-oriented programs, they lack the ability to represent concurrency issues, and therefore can not be used to represent concurrent Java programs.

Zhao *et al.* [24] presented a DBR called *system dependence net* to represent concurrent object-riented 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 for CC++ programs is a straightforward extension of the SDG proposed by Larsen and Harrold [14] for sequential object-oriented programs, 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-Based Representation for Concurrent Java Programs

Generally, a concurrent Java program has a number of threads each having its own control flow and data flow. These flows are not independent because inter-thread synchronizations among multiple control flows and inter-thread communications among multiple data

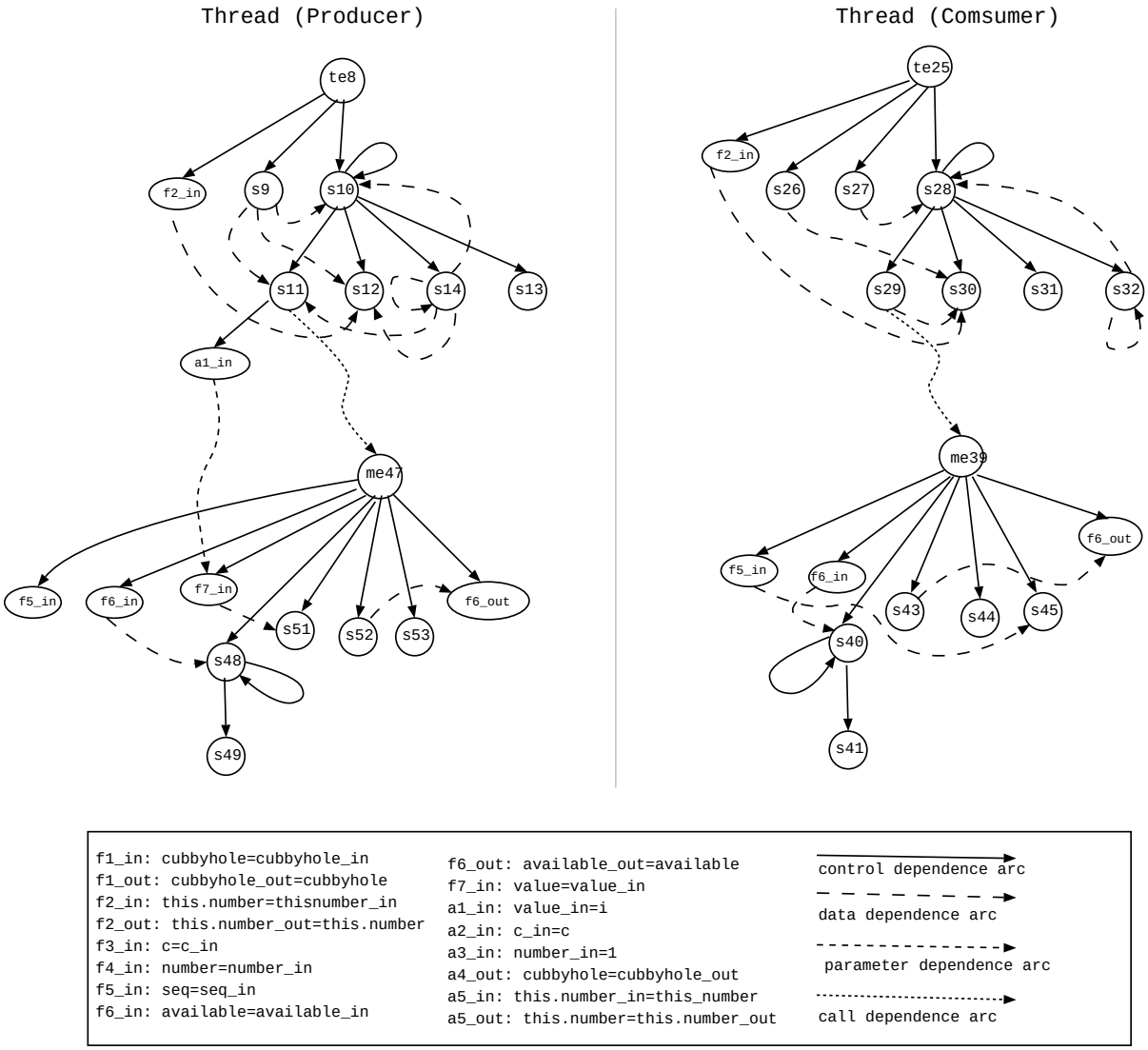

Figure 2: The TDGs for threads `Producer` and `Consumer`.

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 [14] 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 [14]. 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 2 are method entry vertices for methods `get()` and `put()`.

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 2 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 2 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.

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 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 Relations

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 1, methods `put()` and `get()` use Java Object's `notify()` and `wait()` methods to cooperate their activities. This means

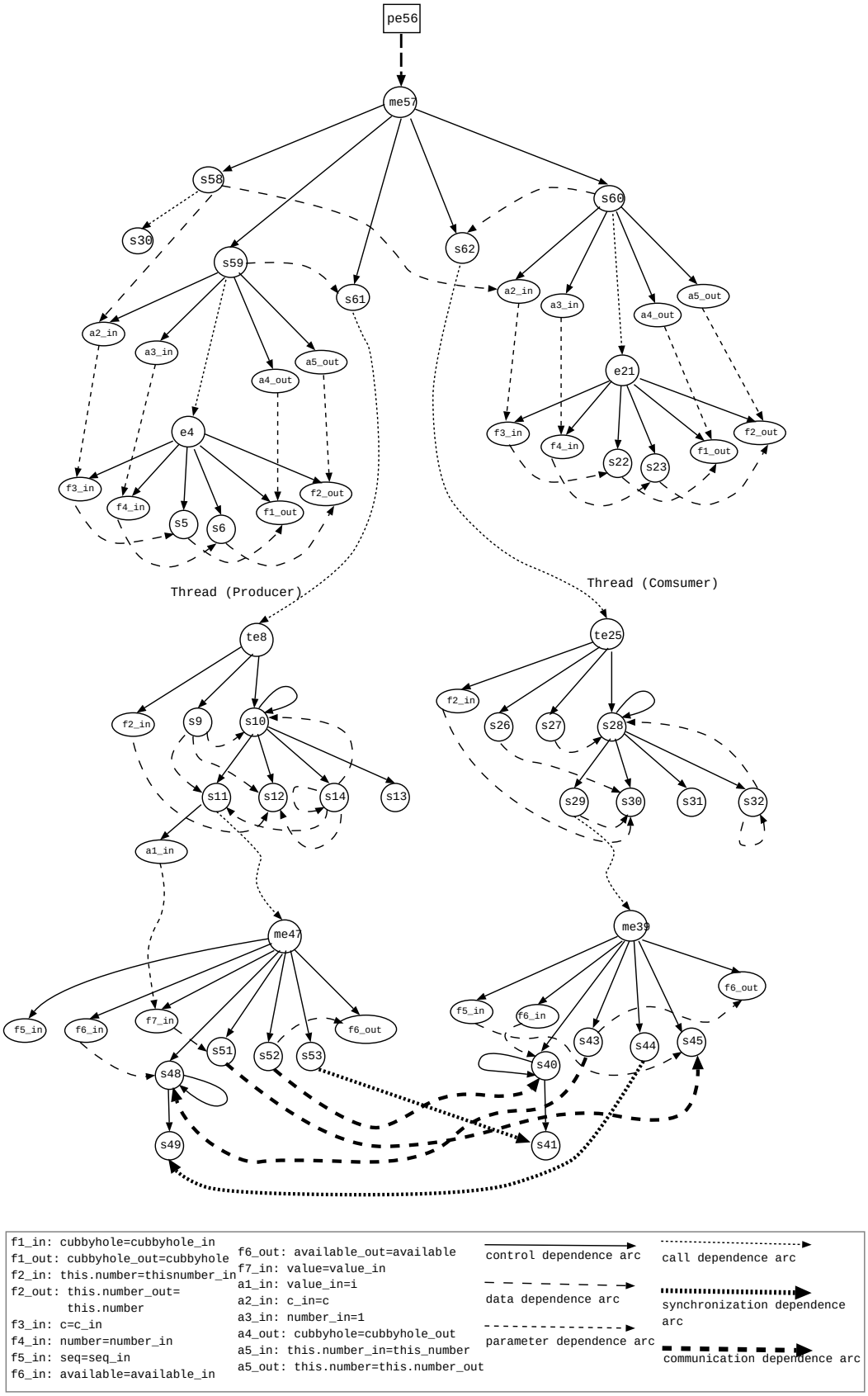

Figure 3: The MDG of a concurrent Java program in Figure 1.

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 3.

(2) Stop-join Relations

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 1, 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 3.

4.2.3 Constructing the MDG

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. There are control dependence arcs to connect the entry vertex to each statement vertex in the method dependence graph of the `main` class. 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, 14]. Their techniques can be directly integrated into our MDG for concurrent Java programs. Moreover, in order to develop a practical MDG for concurrent Java programs, some specific features in Java such as interfaces and packages must be considered. In [27], we presented a technique for constructing a dependence graph for representing interfaces and packages in sequential Java programs. Such a technique can also be integrated directly with our MDG for representing interfaces and packages in a concurrent Java program.

Figure 3 shows the MDG for the program in Figure 1. 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`.

4.3 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 [14] 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

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

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.

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

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.

5 Applications of the MDG

Having MDG as a DBR for concurrent Java programs, we discuss some important applications based on the MDG in a maintenance environment for concurrent Java programs.

5.1 Slicing Concurrent Java Programs

One of our purpose for constructing the MDG of a concurrent Java program is to use it for computing static slices of the program. In this section, we define some notions about statically slicing of a concurrent Java program, and show how to compute static slices of a concurrent Java program based on its MDG.

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* $SS(s, v)$ 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.

Since the MDG proposed for a concurrent Java program can be regarded as an extension of the SDGs for sequential object-oriented programs in [14] and procedural programs in [9], we can use the two-pass slicing algorithm proposed in [9, 18] to compute static slices of a concurrent Java program based on the MDG. 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, and then 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. Similar to the backward slicing described above, we can also apply the forward slicing algorithm [9, 18] to the MDG to compute forward slices of concurrent Java programs. Figure 4 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion `(s45, seq)`.

In addition to slicing a complete concurrent Java program, we can also perform slicing on a single Java thread independently based on its TDG. This may be helpful for analyzing a single thread which is not involved in inter-thread synchronization and communication.

A *static slicing criterion* for a thread in 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 thread slice* $SS(s, v)$ of a concurrent Java program 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.

Similarly, we can use the two-pass slicing algorithm proposed in [9] to compute static thread slices of a thread in a concurrent Java program.

5.2 Understanding and Maintenance

When we attempt to understand the behavior of a concurrent Java program, we usually want to know which variables in which statements might affect a variable of

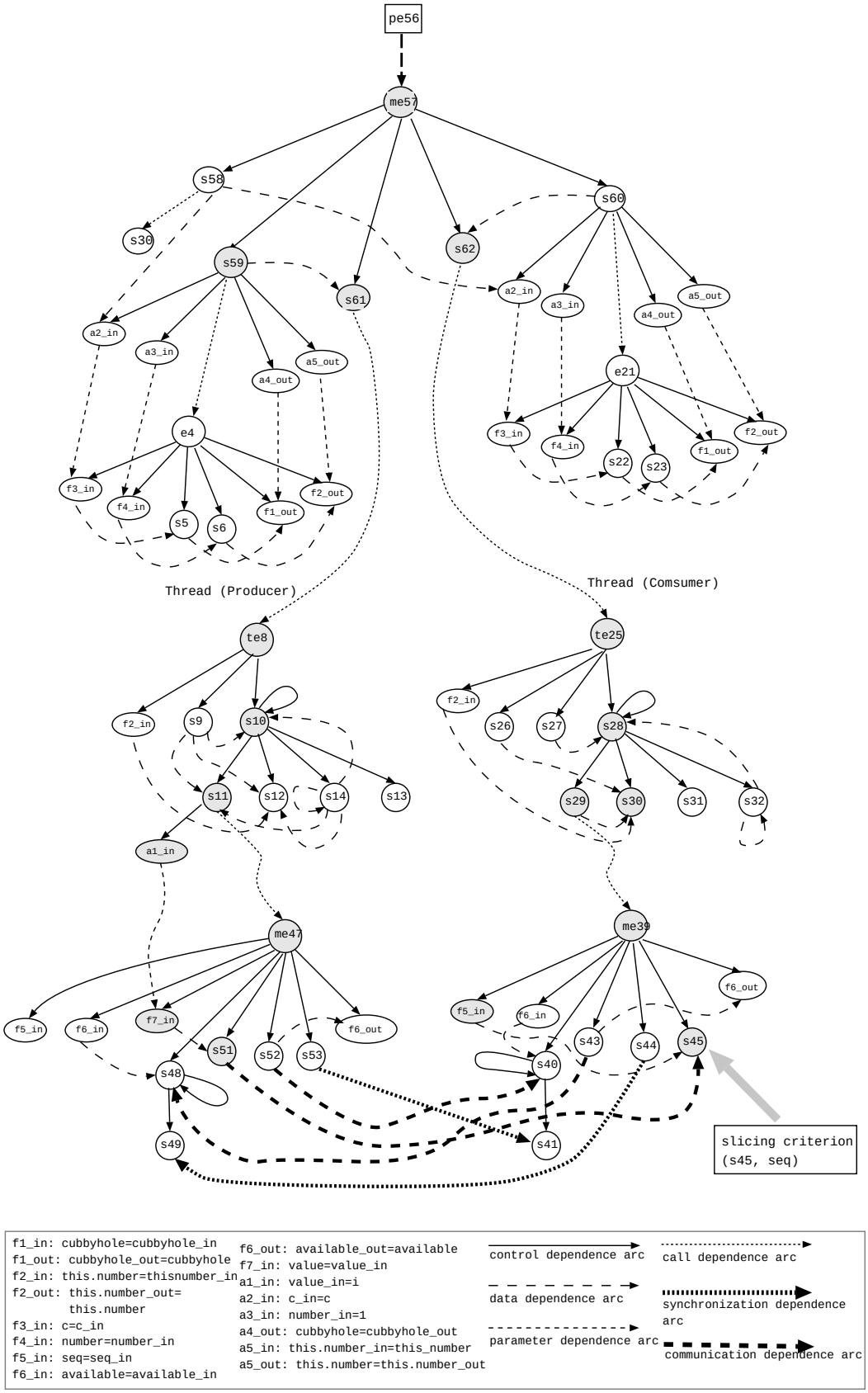

Figure 4: A backward slice with respect to the slicing criterion (`s45, seq`).

interest, and which variables in which statements might be affected by the execution of a variable of interest in the program. As discussed above, the slicing and forward-slicing based on the MDG of a concurrent Java program can satisfy these requirements. On the other hand, one of the problems in software maintenance 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. To maintain a concurrent Java program, it is necessary to know which variables in which statements will be affected by a modified variable, and which variables in which statements will affect a modified variable. The needs can be satisfied by slicing and forward-slicing the program being maintained.

5.3 Complexity Measurement

Software metrics have many applications in software engineering activities including program understanding, debugging, testing, analysis, and maintenance, and project management. One could imagine that once some complexity metrics could be proposed for concurrent Java programs, they should be helpful in the development of concurrent Java programs. Because the MDG of a concurrent Java program represents data flows properties in the program, based on the MDG, we can define some metrics for measuring the complexity of concurrent Java programs from different viewpoints. For instance, the metric defined by the sum of all program dependencies in a concurrent Java program can be used to measure the total complexity of the program, the metric defined by the number of all synchronization and communication dependencies in a concurrent Java program can be used to measure the complexity of concurrency in the program, and the proportion of those program dependencies concerning concurrency (i.e, synchronization and communication dependencies) to all program dependencies in a concurrent Java program can be used to measure the degree of concurrency in the program.

6 Concluding Remarks

In this paper we presented the *multithreaded dependence graph* (MDG) which extends previous DBRs to represent various types of dependencies in a concurrent Java program. 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 inter-thread synchronization and communication. We also discussed some important applications of the MDG in a maintenance environment for concurrent Java programs. 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.

The MDG introduced in this paper can only represent dependencies in a concurrent Java program at the statement level. For large-scale software systems developed in Java, such dependencies may not be efficient because the system usually contains numerous components and we are more interested in high-level dependence relations such as dependencies between components. For such a case, a new dependence analysis technique called *architectural dependence analysis* [25, 19] can be used to determine dependencies between components at the system's architectural level. In contrast to traditional dependence analysis, architectural dependence analysis is designed to operate on an architectural specification of the system, rather than the source code of a conventional program. Architectural dependence analysis provides knowledge of dependencies for the high-level architecture of a software system, rather than the low-level implementation details of a conventional program. We are now considering to integrate the architectural dependence analysis into our traditional dependence analysis framework to support dependence analysis of large-scale software systems developed in Java not only at the statement level but also at the architectural level. We believe that such a approach can be helpful in developing and understanding large-scale software systems developed in Java.

Now we are developing a dependence analysis tool using JavaCC [21], a Java parser generator developed by Sun Microsystems, to automatically construct the multithreaded dependence graph for concurrent Java programs.

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] 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, 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," *Proc. 1996 International Computer Symposium*, December 1996.

[5] J. L. Chen, F. J. Wang, and Y. L. Chen, "Slicing Object-Oriented Programs," *Proc. 1997 Asia Pa-*

cific Software Engineering Conference, pp.395-404, December 1997.

[6] J. Cheng, "Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems," *Proc. IEEE-CS 17th Annual COMPSAC*, pp.231-240, 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] M. Kamkar, N. Shahmehri, P. Fritzson, "Bug Localization by Algorithmic Debugging and Program Slicing," *Proc. International Workshop on Programming Language Implementation and Logic Programming*, *Lecture Notes in Computer Science*, Vol.456, pp.60-74, Springer-Verlag, 1990.

[11] B. Korel, "Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol.24, pp.103-108, 1987.

[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] D. Kuck, R.Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp.207-208, 1981.

[14] L. D. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," *Proc. 18th International Conference on Software Engineering*, March 1996.

[15] B. A. Malloy and J. D. McGregor, A. Krishnaswamy, and M. Medikonda, "An Extensiblesible Program Representation for Object-Oriented Software," *ACM Sigplan Notices*, Vol.29, No.12, pp.38-47, 1994.

[16] 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.

[17] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.

[18] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding Up Slicing," *Proc. 2nd ACM Conference on Foundations of Software Engineering*, pp.11-20, December 1994.

[19] J.A. Stafford and A.L. Wolf, "Architectural-level Dependence Analysis in Support of Software Maintenance," *Proc. 3rd International Software Architecture Workshop*, pp.129-132, November 1998.

[20] V. Sarkar, "A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs," *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, Vol.16, No.9, pp.965-979, 1992.

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

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

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

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

[25] J. Zhao, "Using Dependence Analysis to Support Software Architecture Understanding," in M. Li (Ed.), *New Technologies on Computer Software*, pp.135-142, International Academic Publishers, September 1997.

[26] J. Zhao, J. Cheng, and K. Ushijima, "A Dependence-Based Representation for Concurrent Object-Oriented Software Maintenance," *Proc. 2nd Euromicro Conference on Software Maintenance and Reengineering*, pp.60-66, March 1998.

[27] J. Zhao, "Applying Program Dependence Analysis to Java Software," *Proc. 1998 International Computer Conference*, Tiannan, Taiwan, December 1998.