ソフトウェア工学 122－5
（1999. 3. 18）

# Slicing Concurrent Java Programs

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
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 (MDG), which extends previous dependence graphs to represent concurrent Java programs. Finally, we show how static slices of a concurrent Java program can be computed efficiently based on its MDG.*

## 1 Introduction

Java is a new object-oriented (OO) 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 OO 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 slicing, originally introduced by Weiser [23], is a decomposition technique which extracts program elements 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*.

Program slicing has been studied primarily in the context of procedural programming languages [21]. In such languages, slicing is typically performed by using a control flow graph or a dependence graph [7, 11, 9, 18]. Program slicing has many applications in software engineering activities such as program understanding [8], debugging [1], testing [2], maintenance [10], reuse [17], reverse engineering [3], and complexity measurement [18].

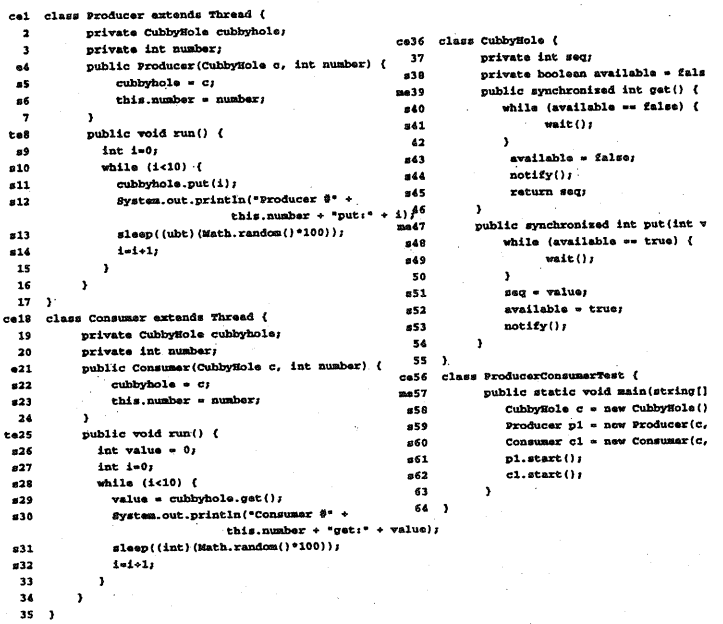
Recently, as OO software becomes popular, researchers have applied program slicing to OO programs to handle various OO features such as classes and objects, class inheritance, polymorphism, dynamic binding [5, 6, 13, 14, 15, 22], and concurrency [24]. However, existing slicing algorithms for OO programs 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 (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 inter-thread synchronization and communication. Once a concurrent Java program is represented by its MDG, the slicing 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 and explains why existing slicing techniques can not handle concurrent Java programs correctly. 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

The Java language support concurrent programming. 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



```
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 = fals
me39       public synchronized int get() {
  s40        while (available == false) {
  s41              wait();
  42         }
  s43         available = false;
  s44         notify();
  s45         return seq;
  s46    }
me47       public synchronized int put(int v
  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[]
  s58        CubbyHole c = new CubbyHole()
  s59        Producer p1 = new Producer(c,
  s60        Consumer c1 = new Consumer(c,
  s61        p1.start();
  s62        c1.start();
  63       }
  64   }
```

Figure 1: A concurrent Java program.

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. Generally, critical sections in Java programs are methods. So you can mark a method as synchronized for synchronizations and communications.

For execution synchronization among different threads, Java provides a few methods of `Object` class, like `wait()`, `notify()`, and `notifyall()`. Using these operations and different mechanism, threads can cooperate to complete a valid method sequence of the shared object.

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 (inclusive), and stores it in a `CubbyHole` object. The `Consumer`, being ravenous, consumes all integers from the `CubbyHole` (the exact same object into which the `Producer` put the integers in the first place) as quickly as they become available.

Threads `Producer` and `Consumer` in this example share data through a common `CubbyHole` object. And you will note that neither the `Producer` nor the `Consumer` makes any effort whatsoever to ensure that the `Consumer` is getting each value produced once and only once. The synchronization between these two threads actually occurs at a lower level, within the `get` and `put` methods of the `CubbyHole` object.

Race condition in the producer-consumer example are prevented by having the storage of a new integer into the `CubbyHole` by the `Producer` be synchronized with the retrieval of an integer from the `CubbyHole` by the `Consumer`. The `Consumer` must consume each integer exactly once.

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 prevent this from happening by locking an object. When an object is locked by one thread and another thread tries to call a synchronized method on the same object, the second thread will block until the object is unlocked. Second, the two threads must do some simple coordination. That is, the `Producer` must have some way to indicate to the `Consumer` that the value is ready and the `Consumer` must have some way to indicate that the value has been retrieved. The `Thread` class provides a collection of methods: `wait()`, `notify()`, and `notifyAll()` to help threads wait for a condition and notify other threads of when that condition changes.

## 3 Program Slicing for OO Programs

As an essential analysis technique for decomposing programs, program slicing has been widely studied in the literatures. 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. Although program slicing has been widely studied in the context of procedural programming languages (for a detailed survey, see [21]), program slicing of OO software is just starting, and to the best of our knowledge, the work presented in this paper is the first time to apply and extend previous slicing techniques to concurrent Java programs.

Larsen and Harrold [14] proposed a static slicing algorithm for sequential OO programs. They extended the *System Dependence Graph (SDG)* which was first proposed to handle interprocedural slicing of sequential procedural programs [11] to the case of sequential OO programs. Their SDGs can be use to represent many OO features such as classes and objects, polymorphism, and dynamic binding. Since the SDGs they compute for sequential OO programs belong to a class of SDGs defined in [11], they can use the two-pass slicing algorithm introduced in [11] for sequential procedural programs to compute slices of sequential OO programs. Chan and Yang [5] adopted a similar way to extend the SDGs for sequential procedural programs [11] to sequential OO programs, and use the extended SDG for computing static slices of sequential OO programs. On the other hand, Krishnaswamy [13] proposed another approach to slicing sequential OO programs. He used a program dependence representation called the *Object-Oriented Program Dependency Graph (OPDG)* to represent sequential OO programs and compute polymorphic slices of sequential OO programs based on the OPDG. Chen *et al.* [6] also extended the program dependence graph to the *Object-Oriented Dependency Graph (ODG)* for modeling sequential OO programs. Although these representations can be used to represent many features of sequential OO programs, they lack the ability to represent concurrency. Therefore, the slicing algorithms based on these representations can not compute correct static slices for concurrent Java programs.

In the meantime, slicing OO program with concurrency has been also considered. Zhao *et al.* [24] presented a dependence-based representation called the *System Dependence Net (SDN)* to represent concurrent OO programs (especially Compositional C++ (CC++) programs [4]). 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 SDN for CC++ programs is a straightforward extension of the SDG proposed by Larsen and Harrold [14] for sequential OO programs, and therefore can be used to represent many OO features in a concurrent OO program. However, the method they used to handle concurrency has some problems when applied to concurrent Java Programs.

First, the concurrency models of CC++ and Java is essentially different. While Java supports monitors and some low-level thread synchronization primitives, CC++ uses a single-assignment variable to the thread synchronization. This difference leads to different sets of concurrency constructs in both languages, and therefore requires different techniques.

Second, the construction of the SDN in [24] simply copies the construction method of the SDGs for sequential OO programs, and did not consider the multithread features in a concurrent OO program. This, among other things, may lead to incorrect static slice when applied to concurrent Java programs.

## 4 A Program Dependence Model for Concurrent Java Programs

Generally, a concurrent Java program consists of a number of threads each having its own control flow and data flow. These flows are not independent because there exist inter-thread synchronizations among multiple control flows and inter-thread communications among multiple data flows in the program. To represent concurrency issues in Java programs, we present a dependence-based representation called the *Multithreaded Dependence Graph* (MDG). The MDG of a concurrent Java program is composed of a collection of TDGs each representing a single thread in the program, and some special kinds of dependence arcs to represent thread interactions between different threads. The construction of the MDG for a complete concurrent Java program has two phases: we first construct the TDG for each single thread, and then combine the TDGs for all threads in the program at synchronization and communication points by adding some special kinds of dependence arcs between these points.

### 4.1 Thread Dependence Graphs for Single Java Threads

The *Thread Dependence Graph* (TDG) is used to represent a single Java thread in a concurrent Java program. it is similar to the SDG presented by Larsen and Harrold [14] for modeling a sequen-

tial OO program. Since execution behavior of a thread in a concurrent Java program is similar to that of an sequential OO program. We can use the similar technique used by Larsen and Harrold for constructing the SDGs of sequential OO programs to construct the TDG. The detailed information for building the SDG for a sequential OO program can be found in [14]. In the following we give a brief description of 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, that is, *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 in order 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 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 inter-thread synchronization and inter-thread communication 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. A *wait* vertex is a vertex that denotes a `wait()` method call in a thread. A *notify* vertex is a vertex that denotes a `notify()` or `notifyall()` method call in a thread.

- *Synchronization dependence arc* captures the dependence relationships 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 and/or termination of the execution of $u$ directly determinates the starts and/or termination of the execution of $v$ through an inter-thread synchronization.

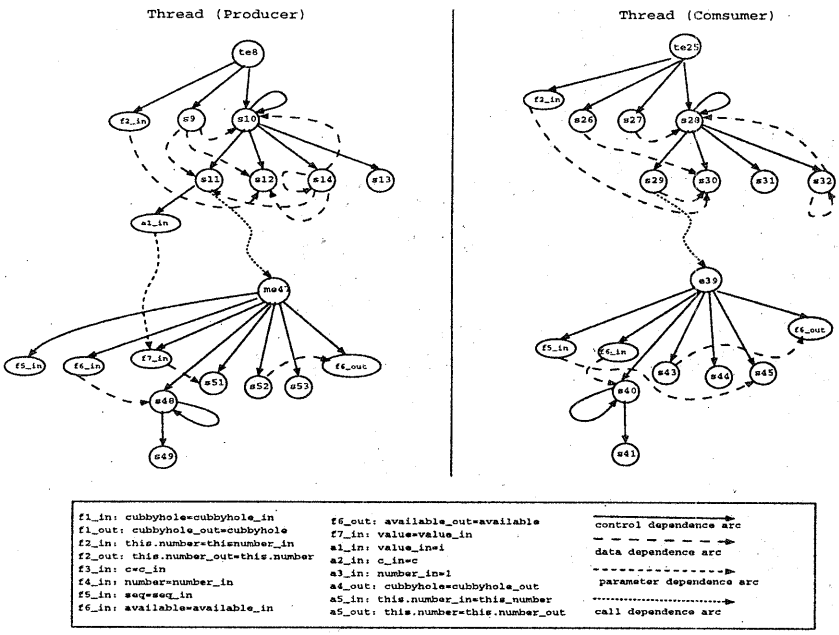



Figure 2: The TDGs for threads `Producer` and `Consumer`.

A synchronization dependence arc is created from a vertex $u$ to a vertex $v$ if $u$ is a *notify* or *notifyall* vertex in thread `t1` and $v$ is a *wait* vertex in thread `t2` for some thread object `o`, where threads `t1` and `t2` are different. Note that in the case of that there is more than one thread waiting for the notification from some thread `t`, we create synchronization dependence arcs from the *notify* vertex of `t` to each *wait* vertex 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. In this way, each `seq` placed in the `CubbyHole` by thread `Producer` is retrieved once and only once by thread `Consumer`. Therefore, there exist synchronization actions between `weit()` method call in thread `Producer` and `notify()` method call in `Comsumer` which are sharing one object `CubbyHole`. This implies that synchronization dependencies may exist between these synchronization points, that is, between `notify()` method call in thread `Producer` and `wait()` method call in thread `Consumer`, and between `notify()` method call in thread `Consumer` and `wait()` method call in thread `Producer`. As a result, synchronization dependence arcs can be created from `s53` to `s41`, and `s44` to `s49` as showed in Figure 3.

- *Communication dependence arc* represents dependence relationship 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.

Note that in the case of 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, we create communication dependence arcs from each attribute variable `a` in each statement of the threads to the statement containing the 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 retrieved 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 shares variable `seq`. This implies that communication dependence may exist between statements `s51` and `s45`. Similarly, communication dependencies may also exist between statements `s51` and `s45`, `s43` and `s48`. As

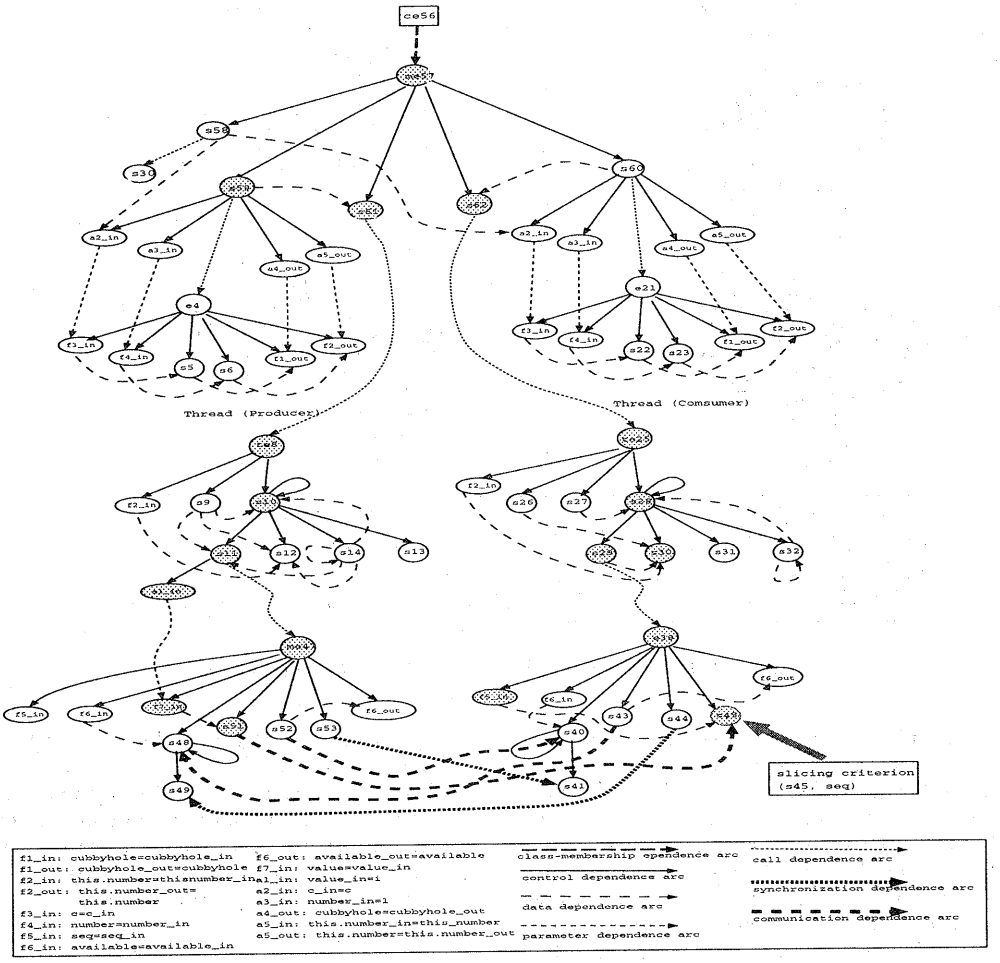



Figure 3: The MDG of a concurrent Java program in Figure 1.

a result, communication dependence arcs can be created from **s52** to **s40**, **s51** to **s45**, and **s43** to **s48** as showed in Figure 3.

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

## 5 Slicing Concurrent Java Programs

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 concurrent Java programs based on the 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 OO programs in [14] and procedural programs in [11], we can use the two-pass slicing algorithm proposed in [11, 14] 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 [11] to the MDG to compute forward slices of concurrent Java programs. Figure 3 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion `(s37, 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 [11, 14] to compute static thread slices of a thread in a concurrent Java program.

## 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 done by Larsen and Harrold [14] who gave an estimate of the size of the system dependence graphs (SDGs) of sequential OO programs. Since each TDG of the MDG is similar to an SDG of a sequential OO program, we can apply their approximation here to estimate the size of the TDGs of threads, and then combine the results to form the whole cost of the MDG of a concurrent Java 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(MDG)* provides only a rough upper bound on the number of vertices in an MDG. In practice 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 inter-thread synchronization and communication. Once a concurrent Java program is represented by its MDG, the slicing 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 [25], we presented a technique for constructing a dependence graph for representing interfaces and packages in sequential Java programs. Such a technique can be used directly for representing interfaces and packages in concurrent Java programs.

Now we are developing a slicing tool using JavaCC [20], a Java parser generator developed by Sun Microsystes, to automatically construct MDGs of concurrent Java programs. We are also implementing an MDG-based slicer for concurrent Java programs.

## 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*,

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 |


pp.384-396, Charleston, South California, ACM Press, 1993.

[3] 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.

[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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[15] 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.

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

[17] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated Support for Legacy Code Understanding," *Communications of ACM*, Vol.37, No.5, pp.50-57, May 1994.

[18] 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.

[19] 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.

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

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

[22] 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.

[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," *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pp.312-320, August 1996, IEEE Computer Society Press.

[25] 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.