Dependence-Based Representation for Concurrent Java Programs and Its Application to Slicing

Jianjun Zhao  
Department of Computer Science and Engineering  
Fukuoka Institute of Technology  
3-30-1 Wajiro-Higashi, Higashi-ku  
Fukuoka 811-0295, Japan  
zhao@cs.fit.ac.jp

Bixin Li  
Department of Computer Science and Engineering  
Southeast University  
Nanjing 210096, China  
bx.li@seu.edu.cn

ABSTRACT

Understanding program dependences 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 concurrent program dependence graph (CPDG), which extends previous dependence-based representations, to represent program dependences in a concurrent Java program. As one application of our dependence analysis technique, we show how static slices of a concurrent Java program can be computed efficiently based on its CPDG.

Keywords  
Concurrent programming, Java, program dependence graph, program slicing

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 nondeterministics behavior of Java threads, understanding a concurrent Java program is usually more difficult than understanding a sequential object-oriented program such as a C++ program. As concurrent Java commercial applications are going to be accumulated, the development of techniques and tools to support debugging, testing, understanding, maintenance, complexity measurement for concurrent Java software will become an important issue.

Program dependences are dependence relationships holding between program elements 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 dependences by analyzing the control flows and data flows in the program.

Many compiler optimizations and program testing and analysis techniques rely on program dependence information, which is topically represented in the form of a program dependence graph (PDG) [6, 11]. The PDG, although originally proposed for compiler optimizations, has been applied to various software engineering activities including program slicing, debugging, testing, maintenance, and complexity measurements [1, 2, 5, 10, 14, 15]. For example, program slicing, a decomposition technique that extracts program statements related to a particular computation, is greatly benefit from a dependence graph representation where slicing problem can be reduced to a vertex reachability problem [14] that is much simpler than its original algorithm [19].

Program dependence analysis originally performed on procedural programs. Recently, as object-oriented software become popular, researchers have applied program dependence analysis to object-oriented programs to handle various object-oriented features such as classes and objects, class inheritance, polymorphism, dynamic binding, and even concurrency issue in object-oriented programs [3, 4, 12, 13, 20, 21].

However, the existing dependence analysis techniques for object-oriented software can not be directly applied to handle some specific issues such as class inheritance and synchronized methods in concurrent Java programs due to specific features of Java concurrency model. In order to represent the full range of concurrent Java program, we must extend the existing program dependence techniques for adapting the analysis of these specific issues in concur-
In this paper we show how to apply program dependence analysis to these specific issues in concurrent Java programs. We present a dependence-based representation called concurrent program dependence graph (CPDG) which extends previous dependence-based representations to represent these dependences in a concurrent Java program. As one application of our dependence analysis technique, we show how to compute static slices of a concurrent Java program based on its CPDG.

The rest of the paper is organized as follows. Section briefly introduces the concurrency model of Java. Section introduces various types of primary program dependences that may exist in a concurrent Java program. Section presents the concurrent program dependence graph for concurrent Java programs. Section shows how to compute static slices based on the net. Concluding remarks are given in Section .

CONCURRENCY MODEL IN JAVA

Java supports multithread programming. Thread is an important concept in a concurrent environment. A thread is a single sequential flow of control within a program. Threads spawn by a process might have different priority and will be scheduled by runtime system. A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program’s environment. Typically, programs that use multiple synchronous threads are called multithreaded programs. Java provides a Thread class library, that defines a set of operations for one thread, like suspend() and resume().

Objects shared by two or more threads are called condition variables, and the access on them must be synchronized. The Java language allows synchronize threads (with keyword synchronized) around a condition variable through the use of monitors. Monitors prevent two threads from simultaneously accessing the same variable. In the Java language, a unique monitor is associated with each instance of every object that has a synchronized method. Whenever control enters a synchronized method, the thread that called the method acquires the monitor for the object whose method has been called. Then other threads can not call a synchronized method on the same object until the monitor is released.

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 implement the famous Producer/Consumer problem. In the program, 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.

```
class Producer extends Thread {
    private CubbyHole cubbyhole;
    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        cubbyhole.put(i);
        System.out.println(``Producer #'' +
            this.number + `''put:'' + i);
        sleep((int)(Math.random()*100));
    }
}
```

```
class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        sleep((int)(Math.random()*100));
    }
}
```

```
class CubbyHole {
    private int seq;
    private boolean available = false;
    public synchronized int get() {
        while (available == false) {
            wait();
        }
        seq = value;
        notify();
        return seq;
    }
    public synchronized int put(int value) {
        while (available == true) {
            wait();
        }
        seq = value;
        available = true;
        notify();
        return value;
    }
}
```

```
class ProducerConsumerTest {
    public static void main(string[] args) {
        CubbyHole c = new CubbyHole();
        Producer pl = new Producer(c, 1);
        Consumer cl = new Consumer(c, 1);
        pl.start();
        cl.start();
    }
}
```

Figure 1. A concurrent Java program.
concurrency issue or not. We first introduce some primary program dependences that are not related to concurrency issues. We called them intra-thread program dependences. Intra-thread program dependences are related to a sequential object-oriented program and have been widely studied in the literatures [3, 4, 5, 12, 13, 20]. Then we propose three new types of primary program dependences for concurrent Java programs that are essentially related to the concurrency model of Java, but has not been studied by existing dependence analysis techniques. We called them inter-thread program dependences.

### Intra-thread Program Dependences

#### Control and Data Dependences

There are two types of primary program dependences between statements in a Java thread, i.e., control dependence and data dependence.

*Control dependences* represent control conditions on which the execution of a statement or expression depends in a thread. Informally, a statement $u$ is directly control-dependent on the control predicate $v$ of a conditional branch statement (e.g., an `if`, `switch`, `while`, `for`, or `do-while` statement) if whether $u$ is executed or not is directly determined by the evaluation result of $v$.

*Data dependences* represent the data flow between statements in a thread. Informally, a statement $u$ is directly data-dependent on a statement $v$ if the value of a variable computed at $v$ has a direct influence on the value of a variable computed at $u$.

#### Call and Parameter Dependences

There are three types of primary program dependences between a call and a called method in a Java program.

*Method-call dependences* represent call relationships between a call method and the called method. Informally a method $u$ is method-call dependent on another method $v$ if $v$ invokes $u$.

*Parameter-in dependences* represent parameter passing between actual parameters and formal input parameter (only if the formal parameter is at all used by the called method).

*Parameter-out dependences* represent 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 dependences represent the data flow of the return value between the method exit and the call site.

#### Membership Dependences

There are three types of membership dependences in a Java program.

*Class-membership dependences* capture the membership relationships between a class and its member methods. Informally, a method $u$ is class-membership dependent on class $v$ if $u$ is a member method of $v$.

*Interface-membership dependences* capture the membership relationships between an interface and its member method declarations. Informally, a method declaration $u$ is interface-membership dependent on an interface $v$ if $u$ is a member method declaration of $v$.

*Package-membership dependences* capture the membership relationships between a package and its member classes, interfaces, and sub-packages. Informally, a class/interface/sub-package $u$ is package-membership dependent on a package $v$ if $u$ is a member class/interface/sub-package of $v$.

### Inheritance Dependences

There are two types of inheritance dependences in a Java program to represent Java class extensions and interface extensions. Note that unlike C++, Java does not support *friend* relationships among classes.

*Class-inheritance dependences* capture the inheritance relationships between Java classes. Informally, a class $u$ is class-inheritance dependent on a class $v$ if $u$ is an extending class of $v$.

*Interface-inheritance dependences* capture the inheritance relationships between Java interfaces. Informally, an interface $u$ is interface-inheritance dependent on an interface $v$ if $u$ is an extending interface of $v$.

Note that those dependences such as control, data, method-call, parameter-in, parameter-out, class-membership, class-inheritance dependences have been studied for sequential object-oriented programs in the literatures [3, 12, 13, 20]. However, the interface-membership, package-membership, and interface-inheritance dependences have not been studied yet, and are new types of primary program dependences that can be used to represent some specific features in a (concurrent) Java programs.

### Inter-thread Program Dependences

#### Synchronized-membership Dependences

One primary program dependence related to inter-thread synchronization is synchronized-membership dependence. *Synchronized-membership dependences* capture the membership relationships between a class and its synchronized member methods. Informally, a method $u$ is synchronized-membership dependent on a class $v$ if $u$ is a synchronized member method of $v$.

The differences between synchronized-membership dependence and class-membership dependence is that the former captures the relationship between a class and its synchronized methods and the later captures the relationship between a class and its methods which are not synchronized. According to the semantics of the Java programming language, they have different meanings.

#### Synchronization and Communication Dependences

we use two types of program dependences called *synchronization dependences* and *communication depend-
dences, which first introduced in [5], to capture the synchronizations and communications between two threads in a concurrent Java program.

Synchronization dependences capture the Inter-thread synchronization between two synchronized threads. Informally, a statement \( u \) of one thread is synchronization dependent on a statement \( v \) of another thread if the start and/or termination of execution of \( u \) directly determines whether or not the execution of \( v \) starts and/or terminates by an inter-thread synchronization.

Communication dependences represent inter-thread communications between two synchronized threads. Informally a statement \( u \) in one thread is directly communication-dependent on another 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 \( v \) by an inter-thread communication.

A DEPENDENCE GRAPH FOR CONCURRENT JAVA PROGRAMS

It has been shown that a dependence graph representation such as the program dependence graph (PDG) [6, 8] for procedural programs, has many application in software engineering activities since it provides a powerful framework for control flow and data flow analysis. This motivates us to present a similar representation to explicitly represent dependences in a concurrent Java program. In this section, we show how to construct the concurrent program dependence graph for explicitly representing various types of primary program dependences in concurrent Java programs. The CPDG of a concurrent Java program consists of a collection of dependence graphs which can be used to represent Java methods, classes and their extensions and interactions, interfaces and their extensions, packages, and complete programs respectively. It partially takes advantage of constructing techniques of previous dependence-based representations [3, 8, 12, 13, 20]. It is not trivial to construct such a representation since we have to consider many cases not only related to a conventional Java program, but also related to a multi-threaded Java program.

Dependence Graphs for Methods

We use the method dependence graph (MDG for short) to represent each method in a concurrent Java program. The MDG of a method is an arc-classified digraph whose vertices are connected by several types of dependence arcs. The vertices of the MDG represent statements or control predicates of conditional branch statements in the method. There is an unique vertex called method start vertex to represent the entry of the method. In order to model parameter passing between methods, an MDG also includes formal parameter vertices and actual parameter vertices. At the 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 there is an actual-in vertex for each actual parameter at call site and an actual-out vertex for each actual parameter that may be modified by the called method. In addition, at each call site of the method, a call vertex is created for connecting the called method.

The arcs of the MDG represent two types of dependence relationships in a method, i.e., control dependences, and data dependences. In addition, each formal parameter vertex is control dependent on the method start vertex, and each actual parameter vertex is control dependent on the call statement.

Figure 2. A complete concurrent Java program and its CPDG.
Dependence Graphs for Classes

we use the class dependence graph (CDG for short) to represent a single Java class. The CDG of a Java class is an arc-classified digraph which consists of a collection of method dependence graphs each representing a single method in the class, and some additional vertices and arcs to model parameter passing between different methods in a class. There is an unique class start vertex for the class to represent the entry of the class, and the class start vertex is connected to the method start vertex of each method in the class by class-membership dependence arcs. If a method invokes another method in the class, the method dependence graphs of two methods are connected at call site. In such a case, a call dependence arc is added between a call vertex of a method and the method start vertex of the method dependence graph of the called method, and parameter dependence arcs are added to connect actual-in and formal-in vertices, and formal-out and actual-out vertices to model parameter passing between the methods in the class. Unlike C++, Java does not support global variables. However, the instance variables of a Java class are accessible to all methods in the class, and therefore we can regard them as “global variables” to every method in the class, and create formal-in and formal-out vertices for all instance variables that are referenced in the methods. Similar to [8], we use summary dependence arcs to represent the transitive flow of dependences in our class dependence graph.

Dependence Graphs for Classes Related to Concurrency

We also use a class dependence graph to represent a class which contains synchronized methods. However, we have to extend the graph to represent interthread synchronizations and communications between threads that are related to the class. There is an unique class start vertex for the class to represent the entry of the class, and the class start vertex is connected to the synchronized-method start vertex of each method dependence graph in the class by synchronized-membership dependence arcs. Interthread synchronizations and communications between synchronized threads can be represented by synchronization and communication dependence arcs in the CDG.

Dependence Graphs for Complete Concurrent Java Programs

We use the concurrent program dependence graph (CPDG) to represent a complete concurrent Java program. The CPDG of a concurrent Java program is an arc-classified digraph which consists of a collection of dependence graphs each representing a single method in the class, and some additional vertices and arcs to model parameter passing between different methods in a class, and interthread synchronizations and communications between different threads.

To construct the CPDG for a concurrent Java program, we first construct the class dependence graph for the main class, then connect the class dependence graph of the main class to other methods in other classes at call sites. A method-call dependence arc is added between a method call vertex and the start vertex of the method dependence graph of the called method. Actual and formal parameter vertices are connected by parameter dependence arcs. We add summary arcs for methods in a previously analyzed class between the actual-in and actual-out vertices at call sites.

Figure 2 shows the complete CPDG of the program in Figure 1. The construction of this net includes the construction of the MDG of the main class and each free standing procedure including producer and consumer, the construction of the CDN of each class including Stream, and the connection of each net using call, parameter-in and parameter-out arcs.

APPLICATION TO SLICING

The most direct application of CPDG is to slice concurrent Java programs since the explicit representation of various program dependences in concurrent Java programs makes the CPDG ideal for computing slices of a concurrent Java program.

Program slicing, originally introduced by Weiser [19], is a decomposition technique which extracts from program statements 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. Program slicing has many applications in software engineering activities such as program understanding, debugging, testing, maintenance, reverse engineering, and complexity measurement. For more information, see Tip’s survey on program slicing techniques [18].

In the following, we introduce some notions about statically slicing of a concurrent Java program.

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 CPDG proposed for a concurrent Java program can be regarded as an extension of the SDGs for sequential object-oriented programs [12] and procedural programs [8], we can use the two-pass slicing algorithm proposed in [8, 12] to compute static slices of a concurrent Java program based on the CPDG. In the first step, the algorithm traverses backward along all arcs except parameter-out arcs, and set marks to those vertices reached in the CPDG, 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 CPDG. The slice is the union of the vertices of the CPDG have marks during the first and second steps. Similar to the backward slicing described above, we can also apply the forward slicing algorithm [8] to the CPDG to compute forward slices of concurrent Java programs. Figure 2 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion \((s37, \text{seq})\).

In addition to slicing a complete concurrent Java program, we can also perform slicing on Java classes independently based on the class dependence graphs.

**CONCLUDING REMARKS**

In this paper we presented a dependence-based representation named concurrent program dependence graph (CPDG) which extends previous dependence-based representations to represent these dependences in a concurrent Java program. Finally, as one application of our dependence analysis technique, we showed how to compute static slices of a concurrent Java program based on its CPDG. In addition to program slicing, we believe that the CPDG can also be used as an underlying representation to develop other software engineering tools for concurrent Java programs.

**References**


