Extracting Permission-based Specifications from a Sequential Java Program

Ayesha Sadiq*, Yuan-Fang Li*, Sea Ling* and Ijaz Ahmed†
* Faculty of Information Technology
Monash University, Victoria, Australia.
{ayesha.sadiq, yuanfang.li, chris.ling}@monash.edu
† Department of Computer and Information Science
Dubai Women College, Higher College Technology, United Arab Emirates
iahmed2@hct.ac.ae

***Abstract*—It is expected that muti-core systems will become the dominant computing platform in the next few years. However, the current programming models (such as Java, .Net etc.) do not scale well to exploit the computing power of such multi-core systems. In primitive programming paradigms there exist implicit dependencies between code and proram states, and compilers cannot exploit the potential concurrency present in the program unless the programmer introduces concurrency manually using multi-threading, which is prone to errors such as race conditions and deadlocks. The goal of this research is to help programmers achieve concurrency without mastering the intricacies of this domain. We propose a formal technique and a high-level algorithm to extract implicit dependencies from a sequential Java program in the form of *access permission rights*. The proposed technique performs static analysis of the source code on a modular basis. The inferred permissions can potentially be used by runtime engines such as Java Virtual Machine (JVM) to automatically parallelize sequential programs on multi-core systems and to reason about concurrency.**

***Keywords*-access permissions, static analysis, concurrency, permission inference.**

I. Introduction

In primitive programming models, there exist implicit dependencies between code and shared resources, which means two methods might be dependent on the same mutable (shared) state without the caller knowing about it. Consequently, this information is not revealed to the compiler and programs are not able to exploit potential concurrency present in the system. Writing concurrent applications is a challenging task for programmers because of thread interleaving and heap interference. Therefore, there is a need for qualified tools and techniques to address these issues. Our aim is to exploit implicit concurrency present in a source program.

To help a programmer to reason about concurrency, a number of abstractions have been developed. One such abstraction is called *access permission* [1] which enables a reference to modify (or read) a referenced object in the presence (or absence) of alias(es). There are five different kinds of access permissions for a particular reference `x` i.e. Unique(x), Full(x), Share(x), Pure(x), Immutable(x). Access permission is a novel way to express implicit dependencies in a program, and it has been used to address issues related to safe concurrency [2].

Unlike previous approaches [3][4][5] that try to exploit implicit concurrency but require manual program annotation, our technique will free programmers from specification overhead. It will help programmers to write concurrent applications without mastering the semantics and intricacies of a new concurrent paradigms.

In our previous work [6] we used access permissions to ensure the integrity of permission-based specifications using an already annotated program. In this paper, we propose a formal technique called GAP (Generating Access Permissions) and a high-level algorithm to infer permission-based specifications from a sequential Java program. We develop rules for graph construction and permission inference to extract permission-based dependencies from the source code. The inferred access permissions can potentially be used by runtime machines such as Java Virtual Machine (JVM) to parallelize sequential programs automatically.

II. Methodology

The basic idea is to perform static analysis of source code on a modular basis. First we parse the method statements one by one and extract the required information. A directed graph (defined in Section II-A) is then constructed from this information using graph construction rules given in Section II-B. One graph is generated for each method. Access permissions are then generated

by traversing the graph using permission inference rules given in Section II-C.

A. Graph Notations, Concepts and Conventions

The proposed methodology uses some special nodes (`foo`, `context`) and directed edges (read and write) to represent the extracted information in the form of a graph, such as Figures 5 to 9.

In these graphs, a circle denotes a referenced object (`<var>`) representing a class variable, parameter or local variable. A rectangle is either `foo` (the current method accessing the referenced object or `context` (other method to the referenced object). A dashed arrow is a `readEdge` denoting a read behaviour on the referenced object and a straight arrow is a `writeEdge` denoting the referenced object being modified.

We can safely assume three possible contexts (`Context R` or `Context RW`, `Context N`) at any moment according to the access (read, write and no access) by other references. The kind of access permissions generated in each context depends on the question whether `foo` (*This Reference*) modifies the referenced object (`<var>`).

B. Graph Construction Rules

The graph construction rules can be categorised into *Context*, *Method Call* and *Statement* rules (Figure 1, 2) and 3. The *Method Call* rules describe the ways to add edges according to the post permissions generated for a referenced variable by a called method.

$$\frac{\texttt{<var>}}{}\ \text{(Context-N)}$$

$$\frac{\texttt{<var>}}{\texttt{addreadEdge(<var>, context)}}\ \text{(Context-R)}$$

$$\frac{\texttt{<var>}}{\texttt{addreadEdge(<var>,context)addwriteEdge(context,<var>)}}\ \text{(Context-RW)}$$

Fig. 1. Context Rules.

$$\frac{\texttt{MethodCall(immutable(<var>))}}{\texttt{addreadEdge(<var>, foo) addwriteEdge(<var>, context)}}\ \text{(Imm-M-Call)}$$

$$\frac{\texttt{MethodCall(pure(<var>))}}{\texttt{addreadEdge(<var>, foo) addwriteEdge(context, <var>)}}\ \text{(Pure-M-Call)}$$

$$\frac{\texttt{MethodCall(full(<var>))}}{\texttt{addwriteEdge(foo, <var>) addreadEdge(<var>, context)}}\ \text{(Full-M-Call)}$$

$$\frac{\texttt{MethodCall(share(<var>))}}{\texttt{addwriteEdge(foo,<var>) addwriteEdge(context,<var>)}}\ \text{(Share-M-Call)}$$

$$\frac{\texttt{MethodCall(unique(<var>))}}{\texttt{addwriteEdge(foo, <var>)}}\ \text{(Unique-M-Call)}$$

Fig. 2. Method Call Rules.

C. Access Permission Inference Rules

Access permission inference rules are given in Figure 4. The type of access permission generated depends on the type of edges and presence (or absence) of edges between the referenced object `<var>` and the method accessing it.

D. Inferring Access Permissions: a High-level Algorithm

A high-level algorithm to infer access permissions starts by creating a set $\mathcal{Z}$ which consists of all the class variables (`<var>`) accessed by the current method (`foo`), method parameters (`<param>`) and special nodes called `foo` and `context` such that.

$\mathcal{Z}$ = {<var>$_1$, <var>$_2$, <var>$_3$, ..., <var>$_n$} ∪ {<param>$_1$, <param>$_2$, <param>$_3$, ..., <param>$_n$}∪ {foo, context }.

Step 1: Create graph nodes for all elements of set $\mathcal{Z}$.
Step 2: Choose the required `context` and add edges according to context rules (Figure 1).
Step 3: Assuming that every referenced object of set $\mathcal{Z}$ is to be read by `foo`, add edges according to the *Read-Only* rule (Figure 3).
Step 4: Parse the statements one by one and add edges following *graph construction rules* (Figure 2 and 3).
Step 5: Traverse the constructed graph to generate access permissions using *access permission inference rules*(Figure 4).

III. A Working Example

We apply our methodology to a sample Java program shown in Listing III-1. Two types of access permission are generated for each method, one for `Context R` whereas the other for `Context RW`.

Listing III-1. **A sample Java Program.**

```
1 class Box{
2     static Integer[] coll;
3  public static void createColl() {
4            coll = new Integer[10];}
5  public static void printColl(Integer[] coll) {
6         for(int i=0; i< coll.length;i++)
7            System.out.println("␣"+coll[i]);}
8  public static void IncrColl(Integer[] coll,int x){
9         for (int i=0; i < coll.length; i++)
10                coll[i]=coll[i]+x;}
11 public static void main(String[] args){
12        Box.createColl();
13        Box.printColl(coll);}}
14        Box.IncrColl(coll);
```

Figure 5 elaborates the step-by-step construction of the graph for the method `createColl()`.

$$\frac{\texttt{<Type> <var>}}{}\ \text{(Variable-Declaration)}$$

$$\frac{\texttt{<var>}}{\texttt{addreadEdge(<var>, foo)}}\ \text{(Read-Only)}$$

$$\frac{\texttt{<var> = <value>/<MethodCall(<var1>)>/<Expression>}}{\texttt{addwriteEdge(foo, <var>)}\ \forall a \in Aliasof(<var>)\ \texttt{addaliasEdge(foo, a)}}\ \text{(Value-Flow)}$$

$$\frac{\texttt{<var> = new<Type>}}{\texttt{addwriteEdge(foo, <var>)addreadEdge(<var>, foo)removewriteEdge(context, <var>)removereadEdge(<var>, context))}}\ \text{(New-Object)}$$

Fig. 3. Statement Rules.

$$\frac{\neg\exists\texttt{readEdge(<var>,context)}\wedge\neg\exists\texttt{writeEdge(context,<var>)}\wedge\exists\texttt{readEdge<var>,foo)}\wedge\ \exists\texttt{writeEdge(foo,<var>)}\wedge\ \neg\exists\texttt{aliasEdge(<var>)}}{\texttt{Unique(<var>)}}\ \text{(Unique)}$$

$$\frac{\neg\exists\texttt{writeEdge(context, <var>)}\ \wedge\ \neg\exists\texttt{writeEdge(foo, <var>)}\ \wedge\ \exists\texttt{readEdge(<var>, foo)}\wedge\exists\texttt{readEdge(<var>, context)}}{\texttt{Immutable(<var>)}}\ \text{(Immutable)}$$

$$\frac{\neg\exists\texttt{writeEdge(context, <var>)}\ \wedge\ \exists\texttt{writeEdge(foo, <var>)}\ \wedge\ \exists\texttt{readEdge(<var>, context)}}{\texttt{Full(<var>)}}\ \text{(Full)}$$

$$\frac{\exists\texttt{writeEdge(context, <var>)}\ \wedge\ \exists\texttt{writeEdge(foo, <var>)}\ \wedge\ \exists\texttt{readEdge(<var>, context)}\wedge\ \exists\texttt{readEdge(<var>, foo)}}{\texttt{Share(<var>)}}\ \text{(Share)}$$

$$\frac{\exists\texttt{writeEdge(context, <var>)}\ \wedge\ \neg\exists\texttt{writeEdge(foo, <var>)}\ \wedge\ \exists\texttt{readEdge(<var>, context)}}{\texttt{Pure(<var>)}}\ \text{(Pure)}$$

Fig. 4. Permission Inference Rules.

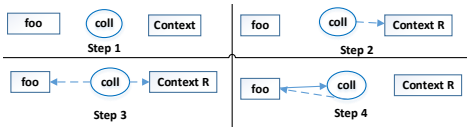

Fig. 5. Graph construction for the method `createColl()` in `Context R`.

The graph of method `createColl()` would be the same in `Context R` and `Context RW` (Figure 6) as there does not exist any context (represented as `Context N`) for the referenced variable `coll` at the time of object creation.

Fig. 6. Graph for the method `createColl()` in `Context R` and `Context RW`.

Following Step 5, we traverse the graph to generate the possible access permission. Let us start from Pure access permission. The one condition for Pure access permission says that there should be an edge from the `context` node to the `coll` node; as there is no edge from `context` to `coll` node, the access permission cannot be Pure. Now we check for Unique access permission, the one condition of the Unique rule states that there should not be an edge between `coll` and `context`; as there is no edge from `coll` to `context` node, so the algorithm generates a Unique access permission for collection `coll`.

Figure 7 shows the graph generated for the method `printColl(coll)` in `Context R` and `Context RW`. The algorithm ignores the state changes for local variables such as line `for (int i=0; i< coll.length;i++)` that only changes the loop state.

Fig. 7. Graph for the method `printColl(coll)` in `Context R` and `Context RW`.

In `Context R`, algorithm generates an Immutable permission for the referenced variable `coll` and a Pure(`coll`) permission in `Context RW`. Figure 8

represents a graph for the method `IncrColl()` in `Context R` and `Context RW`. The algorithm generates Full(`coll`) permissions in `Context R` and Share(`coll`) permissions in `Context RW`.

Fig. 8. Graph for the method `IncrColl(coll)` in `Context R` and `Context RW`.

Figure 9 shows the graph generated for the `main()` method using *Method Call* rules. The `main` method does not require any permission to start its execution and it is not dependent on any context.

Fig. 9. Graph for the `main()` method having nested method calls.

The algorithm generates Unique(`coll`) permissions. We can parallelize the execution of `main()` method if we can track the access permissions on the referenced objects.

IV. Related Work

Ferrara et al. [3] presented a technique to infer permission based specification from a Chalice program [7]. This technique uses symbolic values (annotations) to represent access permissions in the system and performs heap analysis of the code to infer dependency information. We perform static analysis of the source code and our technique does not pose a second level annotation (symbolic values) overhead on the programmer. Æminium [4] is a concurrency by default programming paradigm that aims to develop massively concurrent applications. In Æminium , programmers manually add permission based specifications in the program to control the dependency information between operations. Our technique automatically extracts permission based specification from a sequential program. It will reduce the specification overhead that Æminium approach poses to the programmer to synchronizing the shared data. Haskell [5], a functional programming language uses I/O monad to parallelize the execution of methods to avoid race-conditions but only one permission is used for the whole system and will create a bottleneck for highly concurrent applications. Unlike Haskell, our technique can specify the exact state and permitted operations associated with a certain object. A tool named Daikon [8] performs dynamic analysis of a concurrent program to infer likely specifications based on invariants. This technique uses access permissions to verify the correctness of a already concurrent program. Unlike Daikon, our technique automatically infers permission based specifications from a sequential Java program.

V. Discussion and Future Work

In this paper we propose a method of automatically inferring, from sequential Java programs, access permissions, which can be eventually used to parallelise such sequential programs. Our technique consists of three main tasks: parsing, graph construction and graph traversal. The graph construction and traversal rules have been formally defined to avoid ambiguity. Graph traversal is simple and computationally efficient as it does not involve any cycles or expensive steps like backtracking. For future work, we plan to (a) verify the correctness of our inferred permission; (b) extend our analysis to incorporate alias control information; and (c) implement a technique to automatically parallelize a sequential program based on the access permissions.

References

[1] J. Boyland, *Checking Interference with Fractional Permissions*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 55–72.

[2] N. Cataño, I. Ahmed, R. I. Siminiceanu, and J. Aldrich, "A case study on the lightweight verification of a multi-threaded task server," *Sci. Comput. Program.*, vol. 80, pp. 169–187, 2014.

[3] P. Ferrara and P. Müller, "Automatic inference of access permissions," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012, pp. 202–218.

[4] S. Stork, K. Naden, J. Sunshine, M. Mohr, A. Fonseca, P. Marques, and J. Aldrich, "Æminium : A permission-based concurrent-by-default programming language approach," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 1, pp. 1–42, 2014.

[5] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[6] R. I. Siminiceanu, I. Ahmed, and N. Cataño, "Automated verification of specifications with typestates and access permissions," *ECEASST*, vol. 53, 2012.

[7] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with chalice," in *Foundations of Security Analysis and Design V*. Springer, 2009, pp. 195–222.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, Dec. 2007.