Report on CSCE 689 Project "Interactive Data Race Checker: IDRC"

Md Abu Obaida
Texas A&M University
Department of Electrical and Computer Engineering
tareqobaida@tamu.edu

Abstract

In this dawning era of concurrent programming, parallel programming is becoming more and more prevalent. Because of the nondeterministic nature of parallel programming, it is notoriously difficult to debug concurrency bugs, moreover attempt to fix one bug may result in deadlock or other concurrency bugs.Though many static and dynamic data race detection tool is proposed in recent years, none of them is interactive in nature. We believe that programmer knows best about his program, there is no automatic tool to provide 100% sound, precise and complete analysis. In this context, a tool for early detection of data races while programmer is coding can be extremely valuable. This paper presents "Interactive Data Race Checker" (IDRC) an interactive tool in the form of Eclipse plugin to provide early warning to Java programmer and give opportunity to fix data races before it become too complicated. IDRC runs static analysis whenever it detects any change in Eclipse Java project, creates customized data race marker and provides information about the data race.

Keywords Data Race, Static Analysis, Eclipse Plugin, Interactive, Java

1. Introduction

Eclipse is the most popular IDE for Java and has extensive support to extend features via plugin development. Though Java has built in support for parallel programming in the form of lock allocation, it's always susceptible to human error to allocate lock properly. Moreover, it's not uncommon for programmers to fall into subtle pitfalls in parallel program. As program grows more complex, so does the difficulty in finding data race bugs. Even using best data race detection tool to fix data race bugs can be quite cumbersome because fixing data race is still manual work and there is no reliable tool for automatic fixing up to now. Moreover it may take long time and sometimes fixing one bug results in another bug. Detecting data race at the development time can make the programmer's life easier. However, interactive data race detection requires static analysis which should be precise enough and fast as well. Several researches have been conducted in past few years to detect data races by static analysis. Very few of them provided a user friendly interface for developers to give early warning while programming. In case of Eclipse, the most popular IDE for Java developers, there is no such interface or plugin for data race detection which is incremental in nature.

IDRC responds to every change made in Java project related to data race. It runs analysis on Abstract Syntax Tree (AST) provided by Eclipse. The AST is a detailed tree representation of the Java source code. Each specific AST node provides specific information about the object it represents. For example MethodDeclaration (for methods), VariableDeclarationFragment (for variable declarations) and SimpleName (for any string which is not a Java keyword). In Eclipse An AST instance serves as the common owner of any number of AST nodes. We are particularly interested to examine AST Nodes related to data race. Each related AST Node of interest is visited to extract information about potential data races. We present novel technique to deal with library calls and different approach alternative to Call Flow Graph construction.

1.1 Contribution

1. An interactive data race detection tool in the form of Eclipse plugin. The plugin can provide significant improvement in user experience incorporating static analysis to detect data race in incremental manner.
2. Concept of using Breadth First Search instead of Call Flow Graph

XXX 'XX, December 12–15, 2014, College Station, Texas, USA..

http://dx.doi.org/10.1145/

3. Simple algorithm to find data race focusing on less false negatives.
4. Simple technique to handle library calls in analysis.
5. Available as Eclipse plugin, easy to use.
6. User's freedom to run IDRC in interactive mode or batch mode.

2. Related Work

A number of tools have been developed to detect data race, both statically and dynamically. TACLE Project [2]developed a plugin for performing type analysis and for constructing the call graph for a Java project inside the JDT environment. J. Lhotak and O. Lhotak [8]developed program analysis visualization tools called Soot-Eclipse plugin. None of the plugin is incremental rather run in batch mode. TACLE provides support for only RTA analysis, on the other hand, Soot based plugin supports points-to analysis.Those plugins are not incremental in the sense that the user has to run the tool after completing his code and there is no way to give indication of potential data races when programmer is editing the code.

Flanagan et al. present Type-Based Race Detection for Java which is based on a formal type system. [11] It supports many common synchronization patterns but not the complete set of patterns. M. Naik et al. [1]present a technique of static analysis for data race detection. Their technique is comprised of several static analyses to successively reduce the pairs of memory accesses potentially involved in a race. L. Halpert et al. present an approach of automatically allocating lock to group of critical sections, connected with interference edge and leaving single critical section (CS) unlocked. They incorporated thread local object analysis and may happen in parallel analysis in their approach which is effective in reducing analysis cost and false positives.

Eraser [9]is a dynamic race detection tool based on lockset algorithm which checks if all shared-memory accesses follow a consistent locking discipline. FastTrack [12]is another dynamic analysis tool which introduced the concept of Epoch. Hybrid Dynamic Data Race [10]Detection is a combination of lockset based detection and happens-before based detection.

3. Design

IDRC takes Eclipse Java project as input. The project must have a main method. Java's strong typing dictates that a pair of accesses may be involved in a race only if both access the same instance or static field or both access array elements (and at least one access is a write).[4]Current implementation of the program handles data races on static variables and array accesses. Detection of data race on the access of same instance is left for future version of the work. Figure-1 presents an example of data race that will be used to explain our design and algorithm in rest of the paper.

```
1  public class main {
2  public static void main(String[] args)
       {
3  Thread t1= new Thread(new Thread1());
4  Thread t2= new Thread(new Thread2());
5  t1.start();
6  t2.start();
7  }
8  public class Dtest(){
9  public static int staticVar=0;
10 public static int[] intarray=
       {1,2,3,4,5,6};
11 }
12 public class Thread1 implements
       Runnable{
13 public void run() {
14 Dtest.staticVar=10;
15 Dtest.intarray[0]=100;
16 }
17 }
18 public class Thread2 implements
       Runnable{
19 public void run() {
20 int z=Dtest.staticVar;
21 System.out.println(Dtest.intarray[0]);
22 }
23 }
```

Figure 1. Example Code

The Main class declares a static variable staticVar and a static array intArray. In main method it instantiates two classes(Thread1 and Thread2) which implement Runnable interface. Both threads are started by start() method of Java which eventually executes the run method of respective thread in parallel. In Thread1, inside run method there is a read event of staticVar and another read event of element 0 of intArray. On the other hand, Thread2 has write event on both staticVar and element 0 of intArray. It's clear from the example code that both Thread1 and Thread2 are accessing same memory access of staticVar and element 0 of intArray, there is no particular ordering of execution of those memory accesses and one of the accesses is write access. It implies that both staticVar and element 0 of intArray have data race. For the sake of scalability we consider access to any element of an array is an access to the array as a whole.

We can present the program as a graph consists of nodes and edges. Each node in the graph represents a memory access event. Two nodes are connected via edge if

1. Two nodes are accessing same memory location.
2. One of them is write access
3. There is no synchronization object guarding those two execution events

4. There is no particular order of execution of the events

We are considering first three conditions in current implementation leaving the 4th condition to be considered in future extension of the work. Each thread of the program is presented as super node which holds every memory accesses from the thread. If two nodes from different super node have edge between them, then this a case of potential data race. Figure-2 presents the graph of our sample code. Main thread, Thread1 and Thread2 all are super nodes. Main thread does not contain any nodes because it does not have any memory access in our example code. Thread1 and Thread2 both have two nodes representing memory accesses of staticVar and intArray. The node intArray in Thread1 has edge with the intArray node of Thread2 because they fulfill the conditions of having an edge. Similar thing is happening for StaticVar nodes.

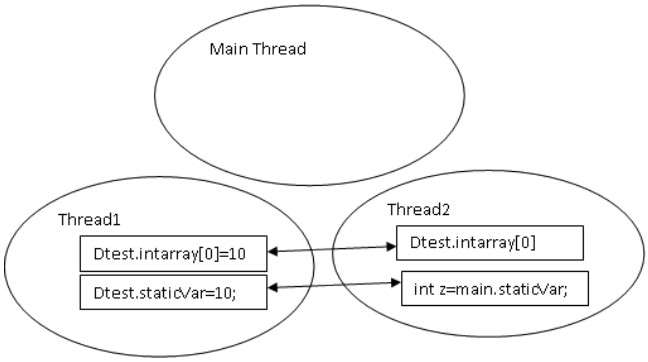

Figure 2. Program representation in graph

3.1 Algorithm

We consider a class as a super node if it implements Runnable interface and instantiated in the program. The class holding main method is considered super node by default. Our algorithm detects super nodes of the program at the starting of analysis. We are not considering the fact that the thread should be started by *Thread.start()* assuming that programmer will start the thread at some point of the program otherwise he is not writing the thread. Once every super node is identified, each and every memory access by the super node is examined. A precise call flow graph is required at this point to find out all the memory accesses by the super node. IDRC uses a different approach instead of call flow graph. It visits all the reachable methods from the super node using Breadth First Search.

Java permits locking of statement by using synchronized block. A memory access is not considered as a potential node if it is guarded by synchronized block or synchronized method. This approach is not complete in the sense that, two memory accesses is thread safe only if they are guarded by same synchronization lock. Automatic detection of this scenario requires a precise points-to analysis which can be expensive in terms of time. We plan to implement the feature in future extension of the work.

Access to same memory location more than once is discarded if the previous access is a write access otherwise previous node is replaced by the new node. It implies that memory access to same variable will represent only one node in super node. This optimization allows us to reduce memory and time overhead. The challenging part in finding nodes is to traverse the super nodes and all other methods that are called inside super node. We implemented a slightly different approach rather than call flow graph which is described in Implementation section. Access type of each memory access is associated with each node. Dealing with Java library calls is another challenge to encounter. Without handling library calls the algorithm is not complete and we may miss potential data races. We present a simple approach to deal with library call which is described in section 3.2. The pseudo code of IDRC algorithm is presented in figure 3. Every node of each super node is added to the collection of the tuple <Thread,variable,access type>. Once the collection is completely constructed, it is examined to find potential data races. A data race marker is created in Eclipse in the location of accessing the variable to visually aid the user to quickly find the data race and take preventive measures.

```
1 Construct Call Flow Graph
2 Identify instantiated classes
3 Filter out instantiated classes to find
      classes that implements Runnable
      interface or holds main method
4 Examine Run method of each thread and
      main method
5 For each thread examine memory accesses
      .
6 Create collection of tuple <Thread,
      variable,access type>
7 Analyse the collection to detect
      potential data races
8 Create marker in Eclipse in respective
      location of the project to indicate
      each data race.
```

Figure 3. Pseudo code

3.2 Handling Library Calls

Dealing with library calls is quite difficult especially for interactive analysis tool. A simple hello world program may involve hundreds of library calls. Most of the time source code of library call is not available to analyze. We propose a summary based library calls handling mechanism which is similar concept to TACLE[2] but significantly different in implementation and usage. TACLE uses summary to construct call flow graph, but to detect data race we don't need to consider call flow graph of library calls. All we need the information about the arguments of library calls and access type of the arguments. For instance the library call

```
System.out.println("from thread1 x="+x)
```

takes one argument and the access type of the argument is read. We propose to build a summary of every library call having information about arguments and access types of arguments. The summary can be used in static analysis whenever any library call is encountered in the process of creating nodes from super nodes. Every argument of the call can be treated as a potential memory access. In this way the time and memory overhead of analyzing library calls can be kept significantly limited and precision of analysis can be improved. In our current implementation of IDRC we are considering every argument of library calls as read access to memory. In future we will construct a complete summary of library calls and incorporate the summary with IDRC.

4. Implementation

The Eclipse IDE allows the developer to extend the IDE functionality via plugins. Being an interactive tool, IDRC takes the advantage of the extension points and API provided by JDT (Java Development Tooling) of Eclipse. JDT adds Java specific behavior to the generic platform resource model and contribute Java specific views, editors, and actions to the workbench. Using JDT API and extension points we can access Java editor in Eclipse and programmatically manipulate Java resources.

IDRC comes in the form of Eclipse plugin. Unlike most other interactive plugins it has the capability of starting and stopping the plugins. The thought behind adding this capability is that, some times we don't need to check data races interactively or even we are not writing any multi threaded program. This functionality becomes handy in these scenarios to avoid unnecessary overheads. IDRC starts it's analysis when user selects a project and click "Start IDRC" menu. Being triggered by the click event it continues to analyze the code whenever any change is detected in the source code. The project information is collected from user selection and stored which is cleared when user clicks "Stop IDRC" menu. Implementation of IDRC is described in the following steps.

4.1 Call Flow Graph

To construct call flow graph IDRC depends on the TACLE [2] implementation of constructing call flow graph. TACLE uses rapid type analysis (RTA) for constructing CFG which has some inherent limitations. It mostly depends on AST provided by Eclipse. It starts its analysis with the main method and traverse all the methods reachable from main method. IDRC requires TACLE implementation of CFG only for determining instantiated classes. In future we plan to replace TACLE CFG with our own implementation to find instantiated classes. Analyzing only the instantiated classes can prune many false positives because if any thread is not instantiated, it will never be executed in the program. It is an engineering decision to consider all classes or only instantiated classes of the program for analysis. It is intuitive that a programmer writes a class to use it at some point of developing life cycle of the code. It may be a wise decision to give early indication of data race even if the class is not instantiated yet. However, in our current implementation of IDRC we are considering only instantiated classes, but it may be avoided which will result in less analysis overhead.

4.2 Super Node

Every instantiated thread(Classes which implement Runnable interface) including main thread is considered super node. Set of instantiated classes is taken from CFG. Each element of the set of instantiated classes is checked to find whether it implements Runnable interface or extends Thread class. After the primary scrutiny it is added to the set of super nodes. Using the concept of super nodes enables to filter out single threaded program and classes not related to data races. Data race in single threaded Java program is not viable, since analysis is discarded. Set of super nodes is examined further to find potential nodes that can participate in data race. The candidate methods of super nodes are the run method and the main method. The insight behind choosing only these two methods is that, when a Java program executes, it's main method is executed and when any thread starts, it's run method is executed. Obviously, examining these two methods can lead to the detection of potential data races. Discarding other methods primarily is safe because all other executable methods can be reached from these two methods. It helps to prune false positives as well.

4.3 Node

Eclipse can provide AST for a particular method. AST of each run method and main method is constructed using Eclipse built-in functionalities. Eclipse provides another useful abstract class ASTVisitor to traverse through AST. Each AST of our concern is visited once to extract required information. We are interested in some particular nodes form where we can get our information required for data race analysis.

4.3.1 Assignment Node

Each assignment expression is caught inside visitor class. Left hand side of assignment is considered as write access and right hand side is considered as read access. For example

```
x=y
```

is an assignment expression where program has read access on y and write access on x. Both left and right hand side are separate expressions and sent to expression resolver separately.

4.3.2 VariableDeclarationFragment node

We are particularly interested in the initializer of VariableDeclarationFragment. For instance

```
Int x=y
```

is a VariableDeclarationFragment, where y is the initializer. Intializer is always read operation and sent to expression resolver for further analysis

4.3.3 PostfixExpression and PrefixExpression node

Postfix and prefix expression have both read and write operation. Considering both read and write accesses is unnecessary, expressions send to expression resolver with access type=write.

4.3.4 InfixExpression node

Infix expression is important to examine because it may come as the condition inside if, for loop, while loop etc. It may come as the argument of methods, any expression described above can be infix expression too. Infix expression is handled inside expression resolver. Infix expression is broken into simple expressions and each simple expression is sent to resolve expression recursively until all the simple expressions are analyzed.

4.3.5 Expression Resolver

Expression resolver is an analysis engine at the heart of the implementation. It takes expression as argument and examine each expression to find potential nodes. It traverses backward to find if any expression is inside synchronized block or synchronized method. Any expression like this is rejected for potential data race candidate. Only the expressions having variable binding are considered for further analysis. While checking any variable, IDRC always traverses backward to find the declaration of the variable. Aliasing problem is solved in this way, because all aliases of a variable will have a single declaration. Expression resolver keep track of every analyzed variable, prevent unnecessary analysis of same variable in same super node. For each variable write access is prioritized, for example read access is replaced by write access if write access comes later. Thus it takes only one node for each variable in a super node. Result of visiting each method is sent to further edge analysis.

4.3.6 Breadth First Search

Breadth fist search is used to traverse all the methods called from a run or main method. Calling of other methods come in the form of ClassInstanceCreation and MethodInvocation. Class instance creation implies calling the constructor of the class. BFS continues until each method reachable from super node is visited. BFS is a replacement of TACLE call flow graph because TACLE's CFG is not complete in finding all the called methods from a particular method. The interesting part of using BFS is that, it can be used to make IDRC more efficient while working interactively. IDRC stores only the node information from a method. If a user makes changes inside a method, that particular method will be traversed only and node informations will be updated. IDRC will use the stored informations for other methods thus can save a significant amount of analysis overhead. Currently full implementation of this concept is not available in IDRC. For every change made by user IDRC run analysis if it finds AST is changed.

4.4 Edge Creation, Detect Data Race

IDRC computes all the potential nodes of each super node and stores the information in a collection of tuple <Thread,variable,access type>. By scanning each element of the collection it finds out if there is more than one access on the same variable from different thread and at least one access is write access. It marks such condition as a potential data race and create data race marker in Eclipse project in appropriate position. Scanning through all the components of the collection and comparing against each other is an expensive operation. IDRC stores the checked node and checked super node information and prevents duplicate analysis. Further optimization can be done in this area which is left for future extension of IDRC.

4.5 Being Interactive

Eclipse provide an extension point compilationParticipant, which is used to participate in the compilation time. Auto build feature of Eclipse compile the program automatically in the background and user defined compilation participant is invoked in the compile time. Eclipse provide ReconcileContext from which IDRC extracts the changed Java element. IDRC runs the analysis if it finds any content changes or AST changes by examining changed Java element. Thus IDRC will not trigger new analysis until any AST change event happens. Further researches can be carried out to make this process more optimized.

5. Evaluation

Current implementation of IDRC is not capable of handling large Java projects. For analyzing simple Java project it takes 2 second analysis time and uses approximately 1 MB memory on average. IDRC is optimized for not to miss any possible data race rather than producing very few false positives. We believe that dealing with every possible data race is important in development time.

6. Limitation

- Currently IDRC handles data races on static variable and array access. It does not handle data race on same instance of a class.
- It ignores start/join, wait/notify and volatile native codes of Java concurrent library. Ignoring these native codes may results in false positives.
- Current implementation of IDRC is not suitable for large Java programs. We plan to reduce analysis cost to make it more scalable.
- IDRC does not run in background, user editor window may become freeze, means user cannot access the editor at the analysis time.

- It does not consider if same locking object is used to guard several event. If potential data race involving events are not guarded by same locking object, then it's not guaranteed to prevent data race. Precise points-to analysis is required to find if separate objects pointing to same object. Including points-to analysis may result in higher analysis cost which is a trade-off between cost and preciseness.
- It does not handle loops
- Current implementation treats every argument of library method as read access, which is not necessarily true.

7. Future Work

Though current implementation of IDRC has several limitations, it has interesting potential future directions to work with. L. Halpert et al [7]used local object analysis and may happen in parallel analysis in their Component-Based Lock Allocation approach. In order to reduce false positives, similar approaches can be incorporated in IDRC. There are a lot of optimization scopes in making IDRC more scalable and practical, for instance careful investigation of changed java element will save a lot of analysis cost.

We plan to build a complete summary of every Java library method to use in IDRC. That summary can be used instead of default read access assumption of every argument of library calls to make analysis more precise.

8. Conclusion

IDRC is an starting point to develop user friendly, interactive interface to detect potential data race at the development time. It can make the program less error prone and help the programmer to write safe concurrent program. Early detection of data race can be fixed much more easily before it becomes too much complicated. We believe IDRC can remove a huge burden from programmers shoulder, help him to be accustomed with good programming practice. IDRC introduce the concept of using breadth first search instead of whole program call flow graph, which has the potential to make interactive tool to have less analysis overhead. Simple idea of handling library calls helps to achieve more completeness of the analysis with very low additional overhead.

Acknowledgments

I would like to thank Dr. Jeff Huang for his kind support and guidance.

References

[1] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In PLDI'06, pages 308–319, June 2006.

[2] M. Sharp, J. Sawin, and A. Rountev. Building a whole-program type analysis in Eclipse. In Eclipse Technology Exchange Workshop , pages 610, 2005.

[3] Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. Aside: Ide support for web application security. In Proceed- ings of the 27th Annual Computer Security Applications Conference, ACSAC '11, pages 267{276, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076770. URL http://doi.acm.org/10.1145/2076732.2076770.

[4] Mayur Naik , Alex Aiken , John Whaley, Effective static race detection for Java, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, June 11-14, 2006, Ottawa, Ontario, Canada

[5] Eric Bodden , Klaus Havelund, Racer: eective race detection using aspectj, Proceedings of the 2008 international symposium on Software testing and analysis, July 20-24, 2008, Seattle, WA, USA

[6] O. Lhot ak. Spark: A exible points-to analysis framework for Java. Master's thesis, School of Computer Science, McGill University, Montr eal, Qu ebec, Canada, Feb. 2003.

[7] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In PACT, 2007.

[8] J. Lhotak, O. Lhotak. Visualizing Program Analysis with the Soot-Eclipse Plugin,Sable Research Group, McGill University, Montreal, Canada.

[9] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. ACM Transactions on Computer Systems, 15(4):391{411, 1997.

[10] Robert O'Callahan , Jong-Deok Choi, Hybrid dynamic data race detection, ACM SIGPLAN Notices, v.38 n.10, October 2003

[11] Cormac Flanagan , Stephen N. Freund, Type-based race detection for Java, Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, p.219-232, June 18-21, 2000, Vancouver, British Columbia, Canada

[12] Cormac Flanagan , Stephen N. Freund, FastTrack: efficient and precise dynamic race detection, Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, June 15-21, 2009, Dublin, Ireland