Reliable Runtime Error Detection in Java Applications

Using Resource Analysis to Investigating Data, Memory, and Thread interactions.

Introduction
The enormous success of Java technology is due to its many advantages over other popular programming languages. Strong typing, automatic memory deallocation through garbage collection, support for object oriented programming, and a large library of standard classes increase programmer productivity and reduce programming errors. Even critical applications, as in automotive or aerospace control, can profit from these advantages. Still, good tools can increase productivity and reduce errors further. Java’s well defined syntax and semantics along with reflection and a common intermediate bytecode format simplify building such tools. Interesting resource and correctness analysis of Java applications can be done both at the source code level and at the byte code level; though the focus here is byte code analysis.

Eliminating Runtime Errors
The aicas team has developed a data flow analysis (DFA) tool to help programmers find and remove runtime errors. By using object context information, DFA produces results that can be used to catch a number of errors that are impervious to local analysis. Its output provides information for pinpointing and correcting data dependent errors.

Null pointer error detection

Dereferencing of null values is a frequent error in Java code. Marking null as a special reference value for tracing during DFA enables the detection of dereferences of null. At any point in the program where a value is dereferenced, when the null value is in the value set of the variable, there is a potential null pointer use. Since all instance and static reference fields in Java are initialized with the null value, the analysis must reliably detect initialization code that overwrites this null value. Given this, the analysis can prove the absence of throws of NullPointerException.

Type cast error detection
Catching type cast errors are done in as similar manner. Java performs a runtime check to ensure that a casted reference is assignable to the target type. If this is not the case, a ClassCastException is thrown. Having exact value sets enables the detection of potential class cast exceptions. At every cast of a variable v to a type T, DFA can check that all values that v may hold are assignable to T. When this is the case, the cast is proven to be always valid.

Array store error detection
Java permits the assignment of reference arrays to array variables of a more general element type. To ensure that storing an element in an array does not store a value of an incompatible element type into an array of a more specific element type, a runtime check is performed on each array store. Having complete value sets makes it possible for each array store to check that all possibly stored values are assignable to all possible target array element types. When this is the case, no array store error may occur at runtime; otherwise the assignment is a potential error.
Threading error detection
Threading is a powerful feature of Java, but it also provides scope for errors that C and C++ programmers seldom think about: unprotected data sharing and deadlocks. To ensure that data accessible from more than one thread is not corrupted, the data must be protected by a lock (synchronize clause or method in Java). Failure to do this can result in difficult to find errors. On the other hand, when more than one resource is locked simultaneously by a thread, then deadlocks can also occur. To avoid deadlocks due to this resource contention, the programmer must ensure that all locks are taken in the same order by all threads and released in the reverse order. DFA finds these errors by examining the thread context of all accessor to each object in the program.

Correct use of memory areas
The Real-Time Specification for Java (RTSJ) adds facilities deallocating memory explicitly; however, improper use of ScopedMemory in particular can lead to runtime errors. DFA detects both scope cycles and improper object assignment errors.

![Figure 1b](image)

Scope Cycle Error detection
Verification of legal scope entry is performed by recording the parent scope whenever a scoped memory area is entered. The parent must match any previously set parent to ensure that it respects the single parent rule. If this is not the case, a possible ScopedCycleException will be reported.

Assignment error detection
Checking assignments in scoped memory is a bit more involved. The value representing an allocated object includes the memory area context of the invocation that allocated it. This information is then used to check all assignments for possible reference errors. When the assigned reference in the store might be allocated in a memory area that is not equal to or an ancestor of the target of the store a possible IllegalAssignmentError is reported by the analysis.

How Data Flow Analysis Works
During execution, aicas’ DFA tool determines two sets of values with contexts: the set of invocations and the set of reference values for each referenced field and referenced array element. The analysis runs iteratively, tracing the call graph starting with main. The data flow for all invocations is analyzed in each iteration. The algorithm terminates when these sets remained constant during one iteration. The analysis distinguishes object initialization and object use to provide better accuracy. Specific properties, such as singleton instances and embedded instances, are also detected to further improve analysis accuracy. When a potential error is found, context information enables the error to be located in the code.

Coding for Effective Analysis
DFA is a purely static analysis that can follow data flow through the application, but cannot detect any semantic relations between variables or objects. This limitation may result in false positives flagged by the analysis, but could be avoided if the semantics of the application were understood. However, the analysis will never produce a false negative; that is it will never fail to report an occurrence of one of the aforementioned runtime errors. Thus when no error is flagged, then DFA has proven that none of these runtime errors can occur. Coding conventions can help make the analysis more accurate by eliminating false positives.
Simple initializer for static fields

Static fields are often used to hold references to a constant object or structure. The initial null value of such a field is overwritten by a non-null value during execution of the static initializer, such that no NullPointerException may occur on any later uses of such a field. Unfortunately, Java permits access to uninitialized static fields, even if they are marked final. For the analysis to detect initial field values correctly, static initializers should

- initialize static fields at declaration,
- not contain static statement sequences,
- not contain any calls to methods that may access static fields being initialized, and
- initialize static fields before other initialization code is run.

A separate constructor that does not access any static fields and is called only after all other static fields are initialized should be used for creating a static instance of the current class.

Separating out constant values

The analysis can be improved further by not initializing simple static fields in a class with a complex static initializer. Moving such fields into a separate class makes it easier for the analysis to show that these fields are actually initialized on their first use.

Initialize instance fields

The initialization of instance fields is a similar issue. Simple, linear assignments of initial values are easy to analyze, whereas complex initialization clauses which reference other fields makes it difficult for the analysis to prove that the other fields are actually initialized. Again, using the keyword final for fields that cannot be assigned after initialization does not help in all cases. For the analysis to detect initial field values correctly, constructors should

- initialize fields at declaration,
- not contain statement sequences intertwined with field declarations,
- not contain any calls to methods that access fields being initialized, and
- initialize instance fields before other initialization code is run.

Use local variables

A very common code sequence for avoiding a NullPointerException is to test for null before calling a method on an object. Unfortunately, this code pattern works only when the reference value is in a local variable. If the value is in a static or an instance field, a concurrent thread could overwrite the value between the null check and the dereferencing of the field. For this reason, DFA cannot assume that the value of a field is non-null within the body of an if statement and it will flag a potential NullPointerException.

The solution for both code safety and DFA is to assign the value to a local variable before operating on it. Local variables can not be modified by other threads. This use of a local variable not only improves the analyzability of the code, but it also makes it more robust. Even if a parallel thread changes the field value, it may not provoke a NullPointerException.

A similar relationship holds for protecting casts with an instanceof test. Again, assigning to a local variable before testing can prevent a ClassCastException. Here too, safer code goes hand in hand with better analyzability.
Do not rely on semantics for casts
Since semantic information is not available to the analysis, it cannot be used to prove the absence of runtime errors whose absence depend on said information. One can argue that over use of casting is bad object-oriented programming style in any case. Avoiding casts by using more classes can result in code that is both better structured and easier to analyze.

Conclusion
The DFA tool can prove the absence of pointer related errors such as null dereferencing and illegal casts, as well as errors related to the use of explicit memory deallocation using the mechanisms available in the RTSJ. Using simple coding guidelines, this result can be improved significantly. DFA will be officially supported with the 3.2 release of the JamaicaVM at the end of this year. A prerelease is available to interested customers upon request. More information can be found on the aicas web site.