

Proving the Absence of RTSJ Related Runtime Errors through Data Flow Analysis *

Dr. Fridtjof Siebert

aicas GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe, Germany
siebert@aicas.com

Abstract

The Real-Time Specification for Java (RTSJ) introduces region based memory management to avoid the need for garbage collection. This region based memory management, however, introduces new possible runtime errors. To ensure that an application developed with the Real-Time Specification for Java executes correctly, it has to be proven that no runtime errors occur.

The use of program-wide pointer analysis for the proof of absence of runtime error conditions such as null pointer uses or illegal casts is still not widespread. Current uses of program-wide pointer analysis focus on applying the results for optimisations in compilers, where a low accuracy of the results leads to missed opportunities for optimisation, which is often tolerable.

This paper presents the application of a program-wide data flow analysis to prove the absence of memory related runtime errors such as those introduced by the RTSJ.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: [Reliability, Validation]; D.3.4 [Processors]: [Compilers]

General Terms Algorithms Languages Reliability Verification

Keywords program analysis, data-flow analysis, Java, Real-Time Specification for Java, context sensitive, pointer analysis, runtime errors

1. INTRODUCTION

The enormous success of Java technology is due to the many advantages, such as higher productivity and safety, the language brings to the developer. Even critical applications, as in automotive or aerospace control, can profit from these advantages [15, 2].

1.1 Scoped Memory in the RTSJ

Language extensions such as the Realtime Specification for Java [9] have made it possible to use Java implementations even though

* This work was partially funded by the European Commission's 6th framework program's HIJA project, IST-511718.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

JTRES'06 October 11-13, 2006, Paris, France
Copyright © 2006 ACM XXXXXXXXXX. . . \$5.00.

a garbage collector may interrupt the execution of normal Java code in an unpredictable way. New specific features such as region based memory management [28] using scoped memory and realtime tasks that cannot access the garbage collected heap make it possible to develop code that has predictable timing behaviour and that can be used for hard realtime tasks.

The use of scoped memory, however, requires that certain runtime error conditions are avoided. These new runtime error conditions are scopes that are nested improperly and assignment that may lead to dangling references.

Scoped memory areas may be nested such that entered memory areas form a tree. Each scope has at most one parent scope in this nesting, but it may have several child scopes. Since the same scoped memory area may be used by different threads simultaneously, each thread must follow the same nesting order. Current implementations of the RTSJ check at runtime that the scopes actually form a proper tree and not a cyclic graph. If a cycle would be created by entering a scoped memory area, a *ScopedCycleException* is thrown at runtime.

Since memory allocated in a scope will be reclaimed after this scope was exited by all threads, it has to be ensured that no dangling references to any objects allocated in a scope will exist when this scope is exited. To ensure this, the RTSJ does not permit to store a reference to an object allocated in scoped memory into a static variable or into another object that was allocated in heap memory, immortal memory or in an outer scope. Current implementations of the RTSJ use runtime checks on all pointer assignments to make sure that no assignments that may lead to dangling references are made. In case an attempt is made to make such an illegal assignment, an *IllegalAssignmentError* is thrown by the virtual machine.

Apart from *ScopedCycleException* and *IllegalAssignmentError*, another runtime error that may occur when using scoped memory is, of course, an *OutOfMemoryError* in case too much memory is allocated in a scope or a scope was dimensioned too small.

1.2 Pointer Analysis

The application of pointer analysis to object-oriented languages such as Java is a relatively new area of research. There are two main uses of the results of pointer analysis: the results can be used to control optimisations of code manipulating tools such as compilers, or the results may be used for the correctness analysis of an application.

1.3 Contributions

In this paper, a new implementation of a context-sensitive and flow-sensitive pointer analysis algorithm for an object-oriented environment is presented.

The pointer analysis results are applied to prove the absence of error conditions. The development of the analysis algorithm was driven by the requirement to reduce the number of false positives in the set of potential errors that is reported. The error conditions include standard Java runtime errors such as null pointer use, illegal casts and illegal array stores, but also memory related runtime errors in applications using the Real-Time Specification for Java [9]. The analysis results are used for the verification of safety-critical Java applications that are developed according to the profiles defined within the HIJA project [16].

1.4 Paper Organisation

The rest of this papers starts with a motivation example that shows a illegal assignment error (chapter 2). Then, chapter 3 gives an introduction into pointer analysis algorithms and presents the Jamaica data flow analysis presented in this paper. Next, chapter 4 presents the HIJA safety critical Java profile, that the analysis is used for, before chapter 5 explains how the pointer analysis can be applied to find runtime errors including RTSJ related errors. Chapter ?? provides coding guidelines that help to improve the accuracy of the analysis. Chapter 8 concludes this paper.

2. MOTIVATING EXAMPLE

A typical use of scoped memories is show in figure 4. In This example, a realtime thread runs it its own scoped memory area *s1*. It starts by reading data from several files. A second memory *s2* area is used to store all the temporary objects required to read those files. However, the data read from the files is allocated in *s2* and copied directly to *s1* causing an *IllegalAssignmentError* at runtime in the call to *Vector.add* made in line 30.

This kind of programming errors are hard to predict since they are caused by an apparently harmless pointer store operation. It typically requires complex manual analysis of the code to determine the possible memory ares of the source and the target of the pointer store operation. In this example, the illegal assignment error occurs inside class *Vector*, even though the code of this class is absolutely correct, the error is in the call at line 30 of the example that passes an argument that cannot be stored into the vector.

This example illustrates that the feature of not causing illegal assignment errors at runtime is not an aspect of a single class or method, but an aspect of a whole application. To proof the absence of such errors, all assignments have to be analysed in all possible calling contexts they may be used in.

The pointer analysis presented in this paper solves this problem. One of the results of the pointer analysis is the value sets of all variables in the application. The representation of these values includes the memory area the values were allocated in, such that assignment errors can be found (see section 5.2).

3. POINTER ANALYSIS

3.1 Background

Significant amounts of research have been undertaken in the area of pointer analysis during the last decades [17]. Pointer analysis typically uses static, program-wide data flow analysis, which is an iterative algorithm that determines an upper bound for the set of values each reference variable in an application may hold. In addition to the set of values for each variable, the analysis determines a set of invocations, where an invocation is a method call together with context information at the call. The resulting set of invocations is an upper bound for the set of invocations that may be performed during an actual program run.

The analysis starts with an empty set of variable values and the set of invocations containing only the main routine of the analysed application. In each iteration, the set of possible values

```

1: import javax.realtime.*;
2: import java.io.*;
3: import java.util.Vector;
4: public class Test implements Runnable
5: {
6:     final static LMemory s1 = new LMemory(1000000);
7:     final static LMemory s2 = new LMemory(1000000);
8:
9:     public static void main(String[] args)
10:    {
11:        new RealtimeThread(null,null,null,s1,
12:                            null,new Test()).start();
13:    }
14:
15:    public void run()
16:    {
17:        final Vector results = new Vector();
18:        for(int i=0; i<10; i++)
19:        {
20:            final int n = i;
21:            s2.enter(new Runnable() {
22:                public void run()
23:                {
24:                    try
25:                    {
26:                        RandomAccessFile f =
27:                            new RandomAccessFile("file"+n,"r");
28:                        byte[] a = new byte[(int) f.length()];
29:                        f.readFully(a);
30:                        results.add(a);
31:                        f.close();
32:                    }
33:                    catch (Throwable t)
34:                    {
35:                        t.printStackTrace();
36:                    }
37:                }
38:            });
39:        }
40:    }
41: }

```

Figure 1. Example that causes an *IllegalAssignmentError* in line 30.

each variable may hold is joined with the set of values that are assigned to these variables by any method that is in the invocation set. Also, any new invocation that is performed by a method that is in the set of invocations is also added to this set. The iterative analysis continues as long as these two sets grow, it stops when the smallest fix point has been reached, i.e., when the sets of values and invocations remained constant during a complete iteration over all invocations.

Pointer analysis is called context-sensitive if the context of the caller is part of the representation of invocations and values. Context information usually is the call chain that leads to an invocation. However, context may include other information such as the thread that performs the invocation or environmental information such as the current allocation context when region-based memory management is used [28, 27].

An algorithm that is not context-sensitive is called context-insensitive. Context-insensitive pointer analysis identifies invocations and values by the method that is invoked and the source code position that creates a value, respectively. Context-insensitive analysis significantly reduces the analysis complexity, but it provides results that are not accurate enough for our purposes: values created

in different contexts cannot be distinguished and result in the inability to prove the absence of an error. E.g., the different instances of a container structure that are used by different threads to store different objects cannot be distinguished, a context-insensitive analysis cannot detect that thread local objects stored into a thread local container are not accessible by another thread that uses a different thread local container instance.

In a context-sensitive analysis, the complete call chain is usually used as context. Since this leads to infinite value sets for recursive routines, the call chain has to be reduced to contain no or only a limited number of cycles. However, even with this restriction, the number of possible call chains typically grows exponentially with the application size making context-sensitive analysis difficult to use with real world applications.

Pointer analysis can furthermore be classified as flow-sensitive and flow-insensitive. A flow-sensitive analysis respects the order of statements during the analysis of one routine, while a flow-insensitive analysis ignores this order. A flow-sensitive analysis achieves higher accuracy since information available in the control graph (such as a null pointer check) can be used to reduce the set of values of a variable.

Unlike context-sensitivity, the effect of flow-sensitivity on the performance of the analysis is less critical. Routine-wide, or global data-flow analysis is a technique that is widely applied in optimising compilers at a tolerable cost.

Object-sensitive points-to analysis for Java was presented by Milanova et. al [20]. In object-sensitive analysis, the context information does not consist of the call chain, but the allocation site of the current object (*this* in Java) is used as context information. This approach brings a significant improvement in accuracy compared to a context-insensitive analysis while the analysis complexity grows less than using a context-sensitive analysis based on the complete call chain.

3.2 Jamaica Data Flow Analysis

For the analysis of the correctness of applications using the Real-Time Specification for Java, we have implemented a new pointer analysis algorithm.

The data flow analysis uses the intermediate code representation used by the JamaicaVM [18] static Java compiler. This intermediate code is generated from the Java byte code. Single instructions are more fine-grain than bytecodes, e.g., an array element access is split into four independent instructions that check the array for *null*, obtain the array length, check the index value and finally read the array element. This intermediate representation replaces the Java stack and the local variables used in the bytecode by using the static single assignment form [4, 23] instead. The static single assignment form simplifies the local data flow analysis in a single method since the data flow between intermediate commands is explicit.

During the data flow analysis, two sets are determined: the set of invocations (called methods with their context) and the set reference values for each reference field and reference array's elements. The analysis runs iteratively starting with the *main* function of the application¹. The data flow for all invocations is analysed in each iteration. During the analysis, new values are added to the value sets of fields and array elements and new methods that are found to be called are added to the set of called methods. The algorithm terminates when these sets remained constant during one iteration.

Reference values are identified by the class of the object they represent together with context information. This context informa-

tion is crucial: if it is too detailed, the number of values explodes and the analysis becomes infeasible, while too little context information results in a bad analysis that finds more "false positives", i.e., potential errors that actually cannot occur at run-time.

The context information of the Jamaica data flow analysis consists of the allocation site, the thread that performed the allocation, the current memory area in use, the set of locks held and a form of object context as proposed by Milanova et. al [20].

This analysis also distinguishes object initialisation and object use which significantly improves the analysis accuracy for object oriented applications. Specific properties such as singleton instances and embedded instances are also detected to further improve analysis accuracy.

To reduce the analysis effort, the analysis is split into major and minor iterations. A major iteration analyses all calls in the set of called methods. For the minor iterations, the called methods are placed into separate age groups. If the analysis of a call causes the call or value set to grow, the next iteration is restricted to the methods in the same or younger age groups. This means that repeated analysis of invocations that had now effect on the analysis results are avoided in minor iterations.

Figure 2 shows the output of the analysis on a HelloWorld application. Even though a HelloWorld is very small, the analysis has to analyse all the code that is executed during startup of the virtual machine, which leads to a total of 341 methods that in this case. Each line that is printed by the analysis shows the state after one major iteration. The analysis stabilises after 8 major iterations, while at most 27 minor iterations where needed (in major iteration #3).

During the analysis, 2332 invocations were found for 341 methods, i.e., there are 2332 different calling contexts even though there are only 341 different methods that may be called. 302 different reference values were detected and 847 different sets of these values were created to store the values of variables. The total analysis time was less than 12 seconds.

4. HIJA SAFETY-CRITICAL-JAVA PROFILE

The pointer analysis is intended for the verification of the correctness of safety-critical Java applications using the safety-critical profile defined by the HIJA project. This profile is a severe restriction of the Java libraries, while it has only minor limitations with respect to the use of the Java language. The analysis was driven by the requirements to analyse applications written according to the profile, but the analysis is not restricted to any subset of Java.

4.1 HIJA Project Overview

The HIJA project [16] started in June 2004 to define Java profiles for critical systems and to implement tools for the required verification of Java applications developed for these profiles. There are 12 partners in the HIJA consortium: aicas, Aonix, Bellstream, Fiat Research Centre, FZI Karlsruhe, Thales-Avionics, Telecom Italia, The Open Group, Trialog, the University of Karlsruhe, the University Polytechnic of Madrid, and the University of York,

For the verification of the correctness of applications written using the defined profiles, several static analysis tools are developed within the project. One of these tools is the program-wide pointer analysis presented in this paper. It is used to prove the non-functional correctness of the applications.

Based on the features of the Realtime Specification for Java (RTSJ) [9], a safety-critical profile is defined by HIJA. This profile provides a restricted subset with the aim to permit certification up the DO178B level A [1]. For less critical application domains, business-critical and flexible dynamic systems, less restrictive profiles are also defined as supersets of the safety critical profile. The aspects covered by the profile address the supported thread model,

¹the analysis actually starts with the virtual machine initialisation code that is called before the *main* method. For JamaicaVM, this means that an internal constructor of *Thread* and the static initialiser of class *System* is also initially added to the set of invocations

```

>jamaica -dfa HelloWorld
Jamaica Builder Tool 2.9 Release 5
DFA ITERATION 1.1.1   CALLS:    490 methods: 183 time: 2s mem: 67MB
DFA ITERATION 1.2.15  CALLS:   1215 methods: 251 time: 4s mem: 70MB
DFA ITERATION 1.3.27  CALLS:   2014 methods: 287 time: 7s mem: 73MB
DFA ITERATION 1.4.3   CALLS:   2206 methods: 323 time: 8s mem: 80MB
DFA ITERATION 1.5.9   CALLS:   2332 methods: 341 time: 9s mem: 82MB
DFA ITERATION 1.6.4   CALLS:   2332 methods: 341 time: 10s mem: 86MB
DFA ITERATION 1.7.1   CALLS:   2332 methods: 341 time: 11s mem: 89MB
DFA ITERATION 1.8.1   CALLS:   2332 methods: 341 time: 11s mem: 92MB
DFA DONE: 11583ms TRACED 302 VALUES, 847 VALUE SETS and 2332 INVOCATIONS.

```

Figure 2. Example run of the data flow analysis on a HelloWorld application.

synchronisation mechanism, memory model and annotations that permit tools to perform correctness verification. In parallel, formal verification tools are developed to prove the functional and non-functional (resource use, etc.) correctness of an application.

Safety-Critical systems as defined in DO178B [1] are software systems that are vital in a way such that anomalous behaviour would cause or contribute to a failure of system function resulting in a catastrophic failure condition for an aircraft. To ensure the correctness of a safety-critical software system, stringent verification techniques need to be applied that prove the absence of error conditions.

For Ada, the Ravenscar profile [11] provides a language for safety-critical applications that is powerful enough while remaining analysable. A corresponding profile is required for the use of Java in this area.

To ensure the correctness of a safety-critical software system, stringent verification techniques need to be applied that prove the absence of error conditions.

Dynamic features such as dynamic memory management or even garbage collection or dynamic code loading are not techniques that currently can be verified in a way that would be accepted by certification authorities. For realtime Java to be applicable in such a system, a very stringent subset of the features available in Java, the standard libraries and the RTSJ extensions is required.

Since the early versions of the RTSJ are available, work on defining subsets for the use in safety-critical systems has been performed. Of important influence to the HIJA project is the profile defined by Puschner and Wellings [22] and the Assessment of Java for High Integrity Systems by Kwon et al [19].

4.2 The Safety-Critical Profile

The HIJA safety-critical Java profile poses severe restrictions on the Java programs developed according to the profile. There is no provision for garbage collection in the the profile. The execution is split into two distinct phases: an initialisation phase and a mission phase. During the mission phase dynamic loading, thread creation and object allocation in non-local memory areas is not permitted. Instead, local memory areas for each task are used for allocation of temporary objects in the mission phase. Annotations are used to document which methods or classes are safe to be used in the mission phase.

4.2.1 Memory Management

No garbage collection is used in the safety-critical profile. In RTSJ terms, this means that no allocation in heap memory will be performed, since the heap is under the control of the garbage collector. Instead, all memory allocation will use either immortal memory, which is never reclaimed, or scoped memory, which provides a safe region-based memory management.

Since memory allocated in immortal memory will never be reclaimed, repeated dynamic allocation in this memory area will eventually cause the system to run out of available memory. Therefore, allocation in immortal memory is restricted to the initialisation phase. All objects that need to be shared by different schedulable objects need to be allocated statically in immortal memory in the initialisation phase. Once the mission phase has started, the amount of immortal memory that is used will remain constant, there is no possibility to obtain an out of memory error due to allocation in immortal memory during the mission phase.

However, the allocation of temporary objects during the mission phase cannot be prohibited without sacrificing the object-oriented programming style that is encouraged by Java. To provide a safe means of allocation of temporary objects by period threads and event handlers during the mission phase, each of these schedulable objects has its own scoped memory region. No other scoped memory regions may be used, i.e., nesting of scoped memory regions is not permitted. The assignment rules for scoped memory regions in the RTSJ then automatically ensure that these temporary objects may not be shared between different schedulable objects.

The scopes will be entered automatically on each release of a schedulable object, and exited after the execution of the schedulable object returns for this release. This ensures that all temporary objects allocated during one release will be reclaimed before the next release. Since the scopes are local to each schedulable object, the allocation of temporary objects in one task is not affected by the allocation in any other task. Any allocation-related errors in one schedulable object, such as uncontrolled allocation or memory leaks, may not affect any other schedulable object.

With this memory model, what remains to be checked by static analysis is that no assignment errors may occur at runtime. An assignment error occurs whenever a reference that is allocated in scoped memory is assigned to a static variable or to an object that is allocated in a memory area that may have a longer life-span, such as heap, immortal or a surrounding scoped memory. Since heap memory is not used in the profile and nesting of scoped memory is not possible, all the static analysis has to show is that there are no assignments of temporary objects into static variables or objects in immortal memory.

To avoid running out of memory within scoped memory and the runtime stack that is local to one schedulable object, it is furthermore required to perform a worst-case heap and stack use analysis for each schedulable object. This analysis can be performed locally to one schedulable object, it is simplified significantly compared to the full RTSJ.

```

1: Object[] a = new Object[10];
2: String[] b = new String[10];
3:
4: a[0] = "A String";      // ok
5: a[1] = new Integer(3); // ok
6: a = b;                 // ok
7: a[2] = "Another String" // ok
8: a[3] = new Integer(3); // error!

```

Figure 3. Illustration of array store runtime error.

5. APPLICATION OF POINTER ANALYSIS RESULTS

There are four aspects the pointer analysis results are used for: The absence of runtime exceptions such as null pointer uses and illegal casts, the correctness of synchronisation, the correctness of the region-based memory management available in the RTSJ and the determination of worst-case memory allocation and worst-case stack use.

5.1 Runtime Error Detection

5.1.1 Absence of null pointer uses

Making *null* a special value for references that is traced by the data flow analysis permits also to proof the absence of null pointer uses. The detection of a potential presence of a null pointer use is straightforward: At any point in the program that dereferences a pointer, if the *null* value is part of the value set of the dereferenced variable, there is a potential null pointer use.

Since all instance and static reference fields in Java are initialised with the *null* value, for a useful null pointer use analysis it is essential that the presence of initialisation code that overwrites this *null* value is detected reliably.

5.1.2 Absence of type cast errors

A type cast in Java performs a runtime check that ensures that the casted reference is assignable to the target type. If this is not the case, a *ClassCastException* is thrown.

The availability of the exact value sets permits the detection of potential class cast exception. At every cast of a variable v to a type T , it can be checked that all values that v may hold are assignable to T . If this is not the case, the cast cannot be proven not to cause an exception.

5.1.3 Absence of array store errors

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 every array store. Figure 3 illustrates a code sequence that causes such a runtime check to fail.

With the availability of complete value sets of all variables, it is now possible for each array store to check if all possibly stored values are assignable to all possible target array element types. If this is the case, no array store error may occur at runtime, else the assignment is a potential error.

5.2 Correctness of region-based memory management

To be able to verify the correctness of assignments of objects allocated in scoped memory and the absence of scope cycles, the context information for invocations and types is extended by the current allocation context. This allocation context is identified by

the corresponding memory area instance. On a call to *enter* of a memory area, the context is set to that memory area for the invocation of *run* method that executes in this area.

5.2.1 Absence of cycles between scopes

Verification of the absence of scope cycles is performed by recording an ordering relation whenever a scoped memory area is entered in a context that uses another scoped memory area as a surrounding allocation context. Whenever a new relation is added to this order, it is checked that this new relation will still respect the single parent rule defined by the RTSJ. If this is not the case, a possible *ScopedCycleException* will be reported.

5.2.2 Verifying assignments

The value that represents an allocated object includes the memory area context of the invocation that allocates the object. This information can then be used to check for all reference stores if an *IllegalAssignmentError* might occur during runtime. If the assigned reference in the store might be allocated in a memory area that is not equal to or a parent of the target of the store, then a possible *IllegalAssignmentError* will be reported by the analysis.

When applying the analysis to the example from figure 4, the illegal assignment is detected reliably. Figure ?? shows the output of the data flow analysis tool when run on this example. The first line of the output is a summary of the problem: There is a potential illegal assignment to an array at line 530 in class *Vector* (at bytecode number 48). The following two lines show the assignment that causes the problem: the target of the assignment is the array allocated in class *Vector* at line 148, while the assigned value is the reference to the byte array allocate in line 28 of *Test.java* which is stored in the scoped memory *LTMemory* created in line 7 of *Test.java*, which is the scoped memory *s2*.

Lines 4 through 10 give the context information for the failing assignment instruction. The context is the method *Vector.addElement* with the allocation context being the *LTMemory* created in *Test.java* in line 7, which is *s2*. Line 5 shows the current thread, which is the *RealtimeThread* created in *Test.java* in line 11. Line 6 and 7 show the potential values passed to this routine.

Finally, to guide the user to the source of the problem, lines 9 through 17 show an example of call chain that leads to the problem. Here, we can see that the call to *Vector.add* in line 30 of *Test.java* is on the call chain.

Since the call chain is not used as context information by the analysis, since this would lead to an explosion in the number of invocations, it is possible that several different call chains lead to the same invocation. This is why for each invocation only one example call chain is shown.

5.3 Worst-case memory usage

The accurate invocation graph that is a result of the presented analysis can be used for automatic analysis of worst-case memory demand of the threads that are part of one application. For this, a traversal of the invocation graph and summing of the memory demand of each invocation results in the total memory demand of each method.

For this analysis, however, additional constraint information on maximum recursion depths, maximum loop counts, and maximum sizes of allocated arrays is required. This information is not available from the pointer analysis. Instead, we use additional tools that permit annotations in the source code or in separate files to provide these constraints. The correctness of these annotations needs to be verified. Within the HIJA project [16], we are investigating the use of formal verification based on the KeY verifier [3] for this.

```

1: POTENTIALLY ILLEGAL ASSIGNMENT: to array at java/util/Vector.java:530[48]
2:   java/lang/Object[] [ONEINSTANCE]:5846:5811:1830 (Vector.java:148[18])
3: <= byte[]:5994:5814:1830 (Test.java:28[34])
   {IN: javax/runtime/LTMemory[SINGLETON]:1357 (Test.java:7[13])}
4: IN METHOD method java/util/Vector.addElement(Ljava/lang/Object;)V
   [MemoryArea: javax/runtime/LTMemory[SINGLETON]:1357 (Test.java:7[13])]
5: in thread: javax/runtime/RealtimeThread:1829 (Test.java:11[0])
6: (arg[0] : java/util/Vector:5811:1830 (Test.java:17[0]),
7:  arg[1] : byte[]:5994:5814:1830 (Test.java:28[34])
   {IN: javax/runtime/LTMemory[SINGLETON]:1357 (Test.java:7[13])})
8:
9:   1: method java/util/Vector.addElement(Ljava/lang/Object;)V (Vector.java:530)
10:  2: method java/util/Vector.add(Ljava/lang/Object;)Z (Vector.java:680)
11:  3: method Test$1.run()V (Test.java:30)
12:  4: method javax/runtime/MemoryArea.enter(Ljava/lang/Runnable;Z)V (MemoryArea.java:1126)
13:  5: method javax/runtime/MemoryArea.enter(Ljava/lang/Runnable;)V (MemoryArea.java:1092)
14:  6: method javax/runtime/ScopedMemory.enter(Ljava/lang/Runnable;)V (ScopedMemory.java:274)
15:  7: method Test.run()V (Test.java:21)
16:  8: method javax/runtime/RealtimeThread$Logic.run()V (RealtimeThread.java:192)
17:  9: method java/lang/Thread.go()V (Thread.java:1153)

```

Figure 4. Detailed output of the data flow analysis when run on the example from figure 4.

5.3.1 Worst-case heap memory use

For the worst case heap use, the amount of allocation needs to be summed up recursively starting from the main method of each thread. Amounts of allocations performed within loops need to be multiplied by the maximum loop count and allocations in recursive methods can be determined by multiplying the maximum recursion depth with the allocation performed within one recursive cycle.

5.3.2 Worst-case stack use

The maximum stack use can be determined by an algorithm similar to the worst-case heap use, only that stack use does not need to be summed up for different calls that originate in the same method, but it is sufficient to use the maximum stack use of all called methods. Also, the stack use is independent of loop counts, but it strongly depends on maximum recursion depths such that the stack use in a recursive method is the product of the stack use in one recursive cycle and the maximum recursion depth.

5.3.3 First experience

First experience with worst-case memory use determination show that the number of additional constraints that are required is limited such that this information can be provided manually even for moderately complex applications. The analysis itself is fast enough even for larger applications.

6. CODING GUIDELINES

The data flow analysis is a purely static analysis that may follow data flow through the application, but that cannot detect any semantic relations between variables or objects. These limitations may result in false positives, i.e., potential errors that are flagged by the analysis, but that may actually never occur during runtime when the semantics of the application is understood. However, the analysis will never provide any false negatives, i.e., if the analysis does not report a certain runtime error at a specific location of the application, then it has been proven that this runtime error cannot occur.

A number of coding guidelines can help to produce application that are easier to analyse. This chapter tries to provide such guidelines to improve the analysis results.

6.1 Use simple Initialiser to set static fields early

Static fields are often used to hold references to a constant object or structure. Often, the initial null value of such a field is overwritten by a non-null value during execution of the static initialiser, such that no *NullPointerException* may occur on any later uses of such a field. Unfortunately, Java permits to access uninitialised static fields, even if they are marked final. The following example illustrates this:

```

1: public class test1 // bad static field init.
2: {
3:   static final Object a = new test1(false);
4:   static final String msg = new String("hello");
5:
6:   test1(boolean printMsg)
7:   {
8:     if (printMsg)
9:     {
10:      System.out.println(msg.toString());
11:    }
12:  }
13: }

```

Here, the DFA cannot detect that the access to *msg* in method *static_test* will not be executed when this constructor is called from the static initialiser, so a false positive potential *NullPointerException* will be flagged in line 10.

Basically, for the analysis to detect initial field values correctly, static initialisers should

- initialise static fields directly at their declaration
- not contain static statement sequences
- not contain any calls to methods that may access the static fields that are being initialised
- initialise static fields before any more complex initialisation code is executed

A better example for how to achieve the same effect as in the above code is given here:

```

1: public class test2 // better static field init
2: {
3:   static final String msg = new String("hello");
4:   static final Object a = new test2();

```

```

5:
6: private test2()
7: {
8: }
9:
10: test2(boolean printMsg)
11: {
12:     if (printMsg)
13:     {
14:         System.out.println(msg.toString());
15:     }
16: }
17: }

```

In this second example, a different constructor that does not access any static fields is used, and this constructor is called only after all other static fields were initialised.

6.2 Use separate classes for constant values

To go one step further, the analysis improves if simple static fields are not initialised in a class with a complex static initialiser at all. If these fields are instead moved into a very simple separate class, it is easier for the analysis to show that these fields are actually initialised on their first use. The following code shows how this could be done in the example from test1 above:

```

1: public class test3 // good static field init
2: {
3:     static final Object a = new test3(false);
4:
5:     static class Messages
6:     {
7:         static final String msg = new String("hello");
8:     }
9:
10: test3(boolean printMsg)
11: {
12:     if (printMsg)
13:     {
14:         System.out.println(Messages.msg.toString());
15:     }
16: }
17: }

```

6.3 Initialise instance fields early

The initialisation of instance fields is an issue that is very similar to that of static fields: Simple, linear assignments of initial values are simple to analyse, while a complex initialisation cause that contains paths that may access uninitialised fields makes it difficult for the analysis to prove that a field is actually initialised. Again, even using the keyword `final` for fields that cannot be assigned after initialisation does not help in all cases since even `final` instance fields can be accessed by tricky before they are initialised. The following code illustrates how a control flow that accesses such a field leads to a false positive *NullPointerException*:

```

1: public class test4 // bad instance field init
2: {
3:     final Object a = f(false);
4:     final String msg = new String("hello");
5:
6:     Object f(boolean printMsg)
7:     {
8:         if (printMsg)
9:         {
10:            System.out.println(msg.toString());
11:        }
12:        return this;
13:    }
14: }

```

Basically, for the analysis to detect initial field values correctly, constructors should

- initialise fields directly at their declaration
- not contain statement sequences intertwined with field declarations
- not contain any calls to methods that may access the fields that are being initialised
- initialise instance fields before any more complex initialisation code is executed

A better example for how to achieve the same effect as in the above code is given here:

```

1: public class test5 // good instance field init
2: {
3:     final String msg = new String("hello");
4:     final Object a = f(f);
5:
6:     Object f()
7:     {
8:         return this;
9:     }
10:
11: Object f(boolean printMsg)
12: {
13:     if (printMsg)
14:     {
15:         System.out.println(msg.toString());
16:     }
17:     return this;
18: }
19: }

```

6.4 Use local variables for null checks

A very common code sequence that usually avoids a *NullPointerException* for an instance method call is illustrated here:

```

1: if (ref != null)
2: {
3:     ref.f();
4: }

```

The same pattern is used to avoid *NullPointerException* on accesses to instance fields:

```

1: if (ref != null)
2: {
3:     ref.field1 = ref.field2;
4: }

```

Unfortunately, this code pattern works only if the reference value *ref* is in a local variable. If *ref* is a static or an instance field, a concurrent thread may overwrite the value between the null check and the dereferencing of the field. For this reason, the DFA analysis also cannot assume that the value of the field is non-null within the body of the if-statement in this case and it will flag a potential *NullPointerException*.

The solution to obtain code that is both safe of *NullPointerExceptions* and that can be analysed by the DFA not to cause such an exception can use a local variable. Local variables in Java may never be modified by other threads. The modified code for an instance method call is

```

1: Type localref = ref;
2: if (localref != null)
3: {
4:     localref.f();
5: }

```

The corresponding code for an instance field access is

```

1: Type localref = ref;
2: if (localref != null)
3: {
4:     localref.field1 = localref.field2;
5: }

```

This use of a local variable does not only improve the analysability of the code, but it also makes it more robust: Even if a parallel thread may change the field value, it may not provoke a *NullPointerException* here.

6.5 Use local variables for instanceof checks

A similar code pattern as for null checks is often employed for type casts:

```

1: if (ref instanceof NewType)
2: {
3:     .. ((NewType) ref) ..;
4: }

```

Unfortunately, also this code pattern works only if the reference value *ref* is a local variable. If *ref* is a static or an instance field, a concurrent thread may overwrite the value between the instanceof check and the type cast. For this reason, the DFA analysis also cannot assume that the value of the field is of the desired type within the body of the if-statement in this case and it will flag a potential *ClassCastException*.

The solution to obtain code that is both safe of *ClassCastExceptions* and that can be analysed by the DFA not to cause such an exception can use a local variable. Local variables in Java may never be modified by other threads. The modified code for an instance method call is

```

1: Type localref = ref;
2: if (localref instanceof NewType)
3: {
4:     .. ((NewType) localref) ..;
5: }

```

This use of a local variable does not only improve the analysability of the code, but it also makes it more robust: Even if a parallel thread may change the field value, it may not provoke a *ClassCastException* here.

6.6 Avoid casts that depend on semantic information

Since semantic information is not available to the analysis, it cannot automatically prove the absence of runtime errors that require this semantic information. The following code sequence illustrates this problem:

```

1: public class semantic_info
2: {
3:     public void do()
4:     {
5:         Object transaction;
6:         Object data;
7:
8:         if (pay)
9:         {
10:            transaction = new Payment();
11:            data = new Amount();
12:        }
13:        else if (close)
14:        {
15:            transaction = new CloseAccount();
16:            data = new Reason(message);
17:        }
18:
19:        handle(transaction,data);
20:    }

```

```

21:
22: void handle(Object transaction, Object data)
23: {
24:     if (transaction instanceof Payment)
25:     {
26:         ((Payment)transaction).pay((Amount)data);
27:     }
28:     else if (transaction instanceof CloseAccount)
29:     {
30:         ((CloseAccount)transaction).close((Reason)data);
31:     }
32: }
33: }

```

One might argue that this kind of code is bad object-oriented programming style. A solution that is better style and easier to analyse would avoid the casts by using more classes:

```

1: interface Transaction
2: {
3:     abstract void handle();
4: }
5: class PayTrans extends Payment
6: implements Transaction
7: {
8:     Amount amount;
9:     public void handle()
10:    {
11:        pay(amount);
12:    }
13: }
14: class CloseTrans extends CloseAccount
15: implements Transaction
16: {
17:     Reason reason;
18:     public void handle()
19:    {
20:        close(reason);
21:    }
22: }
23:
24: public class semantic_info2
25: {
26:     public static void main(String[] args)
27:     {
28:         Transaction trans;
29:         if (pay)
30:         {
31:             trans = new PayTrans();
32:             trans.amount = new Amount();
33:         }
34:         else if (close)
35:         {
36:             trans = new CloseTrans();
37:             trans.reason = new Reason(message);
38:         }
39:
40:         trans.handle();
41:     }
42: }

```

This case illustrates how the DFA analysis can actually force a better object-oriented design, since this better design permits the automatic proof that runtime errors are absent.

7. RELATED WORK

Research on pointer analysis algorithms has been performed since over 25 years [17, 25]. Among the list of open issues that were identified by Michael Hind [17] are scalability, precision of results, addressing client's needs and the support for Java and object-oriented languages.

Efficient points-to analysis that scales well to real-world applications where originally restricted to context-insensitive and flow-insensitive analysis. Steensgard provided the first context-insensitive and scalable implementation [26]. More recently, context-sensitive points-to analysis that scales well to large C applications has been presented by Fähndrich et al [13, 12], however, these approaches are either flow-insensitive or unification based. Unification based algorithms regard assignments as bidirectional such that the source and target variable of an assignment has the same value set, which leads to significantly reduced precision compared to the inclusion based approach that is presented here. Furthermore, these earlier approaches differ from the presented approach since they are not field-independent, while the approach presented here maintains separate value sets for each field.

Recent work on the use of Binary Decision Diagrams (BDDs) has shown that the application of BDDs can enable the context-sensitive analysis that would otherwise be intractable due to an exponential growth in the size of the invocation set [7, 29, 30]. Whaley and Lam have achieved a fast alias analysis by cloning methods for every context of interest for the analysis such that context-insensitive analysis over the cloned methods can be used instead of a context-sensitive one. They used BDDs for the representation of invocations and showed that this can result in an efficient analysis [29]. Their approach has permitted the analysis of real world Java applications.

Zhu and Calman [30] use BDDs for a symbolic representation of the invocation graph and a symbolic transfer function to represent the effect of a routine. They show that their approach can reduce the exponential analysis effort for context-sensitive analysis to be almost as fast as a corresponding context-insensitive analysis.

The use of BDDs for the representation of invocation graphs and transfer functions is a technology that is orthogonal to the ideas presented in this paper. It can be expected that similar mappings to binary functions can be found for the suggested approach such that the use of BDDs may reduce the analysis effort. Of course, the use of BDDs does not avoid the potentially exponential space and time requirements since the efficiency of BDDs strongly depends on finding a good ordering of binary variables.

Rountev et al [24] have extended Andersen's [5] context-insensitive analysis for C for the use with Java. Their analysis is context- and flow-insensitive. It uses *constraint* and *field annotations* to accurately handle virtual method calls and trace the flow of values through different fields. Furthermore, analysis is restricted to potentially called methods. The implementation presented in this paper uses similar techniques to accurately trace virtual calls and fields. An object-sensitive points-to analysis for Java was presented by Milanova et. al [20].

The use of static analysis for the verification of safety-critical applications or the use of the Real-Time Specification for Java is proposed by some recent publications. Blanchet et al [8] present a static analyzer for the formal verification of safety properties in large safety-critical C applications. They use an abstract interpretation approach that is not limited to heap analysis but that covers the arithmetic domain as well. Their analyzer detects potential "fatal errors" such as out-of-bounds array accesses that would have an undefined result. The approach covers error conditions that cannot be handled by the approach presented here since arithmetic values are not analyzed here. However, their approach is not adapted to the analysis of object-oriented code with complex heap structures, so a combination of these approaches may result in a more complete analysis of safety features of an application.

The correctness of region based memory management in the RTSJ can be ensured by the use of an extended static type system based on ownership types as proposed by Boyapati et al [10]. Unlike the approach presented here, additional annotations to the

source code are required. Salagnac et al [14] propose the use of escape analysis for region-based memory management to automatically determine life-spans of objects and therefore be able to use RTSJ memory areas for the allocation of these objects.

A different approach to avoid runtime errors in region based memory management are Scoped Types presented by Andreae et al. [6]. In this approach, types are related to scopes by annotations such that it can be statically that no assignments between instances in different scopes are made by checking that the types of these instances are associated to the same scopes.

8. CONCLUSION

The presented pointer analysis permits the proof of absence of pointer related errors such as null pointer use and illegal casts, but also errors related to the use of region based memory management using the mechanisms available in the Real-Time Specification for Java. The analysis serves as a basis for worst-case memory use analysis enabling static analysis to avoid resource related runtime errors. This is of particular interest for safety-critical code that needs to be certified.

It has been shown that this approach scales well to medium size real-world applications and can prove the correctness of a high percentage of statements that are potential sources of runtime errors. It can be expected that, using simple coding guidelines, this result can still be improved significantly.

Future work needs to focus on the development of coding guidelines that ensure a more accurate analysis and that avoid certain obvious problems. Also, enhancements in the analysis accuracy may be achieved by recording additional information such as the set of threads that modify a field.

An extension of the analysis to determine the value sets of non-reference variables such as booleans may further help to improve the accuracy of the analysis. Analysing non-references as well will, e.g., help to exclude code that is de-activated by a boolean variable that is always *false*.

References

- [1] Rtc/do-178b software considerations in airborne systems and equipment certification, 1992.
- [2] Aero-vm, the hard realtime virtual machine for onboard space systems. www.aero-vm.com, 2003.
- [3] W. Ahrend, T. Baar, B. Becker, R. Bubel, M. Giese, R. Hähnle, W. Menel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The key tool. Technical Report 2003-05, Department of Computer Science, Chalmers University of Technology and Göteborg University (2003), 2002.
- [4] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.
- [5] L. Andersen. *Program Analysis and Specialization for the C Programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.
- [6] Chris Andreae, James Noble, Yvonne Coady, Celina Gibbs, Jan Vitek, and Tian Zhao. Stars: Scoped types and aspects for real-time systems. In *Proceedings of 20th European Conference on Object-Oriented Programming, ECOOP 2006*, 2006.
- [7] Marc Berndt, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM Press.
- [8] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on*

- Programming language design and implementation*, pages 196–207, New York, NY, USA, 2003. ACM Press.
- [9] Greg Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.
- [10] Chandrasekhar Boyapati, Alexandru Salcianu, Jr. William Beebe, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 324–337, New York, NY, USA, 2003. ACM Press.
- [11] A. Burns, B. Dobbins, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. *Proceedings of Ada-Europe 98*, 1411:263–275.
- [12] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *SAS '00: Proceedings of the 7th International Symposium on Static Analysis*, pages 175–198, London, UK, 2000. Springer-Verlag.
- [13] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 253–263, New York, NY, USA, 2000. ACM Press.
- [14] D. Garbervetsky G. Salagnac, S. Yovine. Fast escape analysis for region-base memory management. In *Proceedings of the 1st Int. Workshop on Abstract Interpretation for Object-Oriented Languages (AIOOL'05)*, jan 2005.
- [15] Hidoors, high integrity object-oriented realtime systems. www.hidoors.org, 2002-2004.
- [16] Hija, high-integrity java. www.hija.info, 2004-2006.
- [17] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, June 2001.
- [18] Jamaica virtual machine. www.aicas.com/jamaica, 1999-2006.
- [19] J. Kwon, A. Wellings, and S. King. Assessment of the java programming language for use in high integrity systems. Technical Report YCS 341 (2002), University of York, 2002.
- [20] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.
- [21] *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.
- [22] P. Puscher and A. J. Wellings. A profile for high integrity real-time java programs. *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.
- [23] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, New York, NY, USA, 1988. ACM Press.
- [24] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55, New York, NY, USA, 2001. ACM Press.
- [25] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis.*, chapter 7, pages 189–234. Prentice-Hall, 1981.
- [26] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.
- [27] Mads Tofte. A brief introduction to Regions. In Richard Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 186–195, Vancouver, October 1998. ACM Press.
- [28] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997. An earlier version of this was presented at [21].
- [29] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.
- [30] Jianwen Zhu and Silvan Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 145–157, New York, NY, USA, 2004. ACM Press.