

5 years



aicas news

News for Software Developers of Critical Applications

Anniversary Issue 2006

↓ Editorial

Dear Reader,

I am happy to announce that we, the aicas team, celebrate our fifth anniversary. Though work on the JamaicaVM started long before aicas was officially established, we have been supplying the embedded and real-time systems industry with Java technology and consulting for five years.

Much has changed in this time, both in the Market and in our products. In 2001, the acceptance of Java in embedded systems, especially for realtime systems, was limited to early adopters. Now there is much broader interest in Java for embedded applications of all kinds.

Certainly advances in products have contributed to rising interest. The JamaicaVM itself has come a long way. Support for an ever larger set of packages is being added. For example, basic graphics came in 2004. Now the JamaicaVM brings much of J2SE to realtime and embedded applications and we will continue to offer more in the future.

Sincerely,
Dr. James J. Hunt, CEO



aicas Celebrates its Fifth Birthday

Providing realtime Java technology for embedded systems

The aicas team marks five years in business. The JamaicaVM has been available as a commercially supported product for five years as well. Much has happened in this time.

May 2001 saw the founding of aicas GmbH. At the beginning, there

were four partners: Andy Walter, Fridtjof Siebert, Götz M. Fluck, and James J. Hunt; and one product, the JamaicaVM. From the beginning, the principle advantage was real-time garbage collection. This basic technology was the brainchild of Dr. Siebert which earned him the 2003 BMW Scientific Award.

In the fall of 2001, aicas presented for the first time at the SPS/IPC/Drives trade fair in Nürnberg. The following spring, we exhibited at the Embedded Systems in Nürnberg, published the first aicas news, and moved into our office in the Hoepfner castle.

2002 also brought HIDOORS, aicas' first EU project. Lead by Dr. Hunt and running for 33 months, the project supported the rapid, continued development of the JamaicaVM en-

abling the aicas team to implement the Real-Time Specification for Java (RTSJ) and improve the compiler and runtime system. HIDOORS was successfully completed in 2004.

2003 saw rapid expansion: Peter Marggrander joined aicas as CFO that spring; EADS Astrium joined forces with aicas to produce the AERO-VM, a version of the JamaicaVM for satellites; RTSJ support was completed; and the first English version of the aicas news appeared.

2004 was a great leap forward. With the support of basic AWT functionality and third party libraries, the JamaicaVM could support complex graphic applications for the first time and Siemens began using the JamaicaVM in its Simotion product lines. Midyear also saw the start of the HIJA project.

Expansion continued in 2005. The year started with the founding of an independent firm, aicas incorporated, to support the US market. This, along with the development on Swing and debugging, has led to continuing success for 2006.

The JamaicaVM is now used by many industrial automation firms, such as SICK, Saurer, and Multivac and has flown in the EADS Barracuda and will fly with Boeing.

↓ News

The JamaicaVM takes to the air!

The UAV Barracuda flew successfully in May using software written in Java running on the JamaicaVM. The flight demonstrated that Java can be used in aircraft software. Realtime garbage collection along with the RTSJ enables designers to take full advantage of Java features without giving up realtime response.

Boeing selects the JamaicaVM

Barracuda is not the only in flight system to use the JamaicaVM. Boeing has selected the JamaicaVM for use in the 787 Dreamliner. The JamaicaVM helps Boeing meet its development goals on time. A prototype already runs with the JamaicaVM on VxWorks 6. The plane's maiden flight is scheduled for 2007 with first delivery in 2008.

VxWorks 6.3 with RTP support

The JamaicaVM version 3.0 supports the latest VxWorks release from Wind River for both Kernel Processes and Real Time Processes. RTPs provide memory protection between applications and better resource cleanup. For Java applications, the only difference is how RawMemory access is configured. All Java programs run unchanged.

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

```

if (device instanceof MyDevice)
{
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
}
    
```

Annotations in the code: *device != null* (blue), *contains only MySensor* (green), *NullPointerException* (green), *ClassCastException* (red), *NullPointerException* (red).

Figure 1a: Successful instanceof implies that variable device is not null. ...

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 a 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 accesser 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.

```

if (device instanceof MyDevice)
{
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
}
    
```

Annotations in the code: *values(MyDevice.sensor) contains only MySensor* (blue), *NullPointerException* (green), *ClassCastException* (green), *NullPointerException* (red).

Figure 1b: ... Cast always succeeds since MyDevice only contain MySensor. ...

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

```

if (device instanceof MyDevice)
{
    MySensor s = (MySensor) device.sensor;
    int value = s.reading();
}
    
```

Figure 1c: ... Variable *s* is never null since *device* is always correctly initialized. DFA has shown that no exceptions can occur.

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 par-

↓ Events

We would like to welcome you at the following events.

JTRES 2006
Paris
11–13 October 2006

SPS/IPC/Drives 2006
Nuremberg
28–30 November 2006

Embedded World 2007
Nuremberg
13–15 February 2007

ESC Silicon Valley 2007
San Jose
1–5 April 2007

allel 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.

Outlook

The next issue of the aicas news will present how Java is used in industrial automation.

Target Debugging with the JamaicaVM

The latest version of the JamaicaVM supports standard Java debugging facilities

Introduction

As the complexity of software continues to increase, the desire for tools to help develop software increases as well. Dynamic tools such as debugging and monitoring can provide indispensable help for understanding program execution particularly when multiple tasks execute at once. A good debugger can help a developer find errors more quickly without having to recompile code. Monitors and profilers can be used to help the developer understand the dynamic relationships in a system. For these reasons, JamaicaVM implements the Java Virtual Machine Toolkit In-

terface (JVMTI) to provide better support for program development.

An Architecture for Debugging

The Java Platform defines an architecture for debugging: Java Platform Debugger Architecture (JPDA)(1). In order to both reduce the burden on the virtual machine and support remote debugging, JPDA has a client server model. Basic services are provided by a server in the JVM, called an agent, which communicates with an external Java debugger. This architecture has the added advantage of separating low level JVM concerns from the debugger user interface so that all Java debuggers can work with all virtual machines which support the architecture.

Flexibility

JVMTI provides all the facilities necessary for implementing debugging and monitoring tools. For example, one can use JVMTI routines to stop and start Java threads, read and write variables and fields, set and clear breakpoints, and step through code or continue execution. So monitoring and profiling agents can be effective, JVMTI is defined as a light weight native interface in the JVM. This independence makes it possible for vendors to provide special agents adapted to particular tasks and embedded environments. Examples of tools that use such agents include Adriana, Hyades, JBoss, JProfiler, Optimizeit, YAJP, and Yourkit.

↓ **New Release**

JamaicaVM Release 3.0

In addition to basic remote debugging, the latest release of the JamaicaVM brings many improvements and new features both for graphics as well as for realtime applications.

Swing support has been extended considerably. Several large graphics applications are already using the JamaicaVM. The performance has also been improved for quicker response.

The VM has become more flexible. Both thread number and memory size can be dynamically increased. Support for non-contiguous heaps is particularly helpful for surmounting the 32MB memory limits in Windows CE. This flexibility comes from completely reworking the garbage collector to give developers more control over how the collector should run.

Dynamic systems are also easier to manage with the addition of cost monitoring. It is now possible to recognize and intervene when a thread demands too much processor time. This is particularly useful for server applications.

These changes are rounded out with better tools support including a *jamaicac* command. A trial version is available from the aicas website. Try it out today!

Host Debuggers

On the Java debugger side, the Java Debugging Interface (JDI) defines the base functionality that a debugger can use. This interface is implemented by several debuggers including the debugger provided with Sun's JDK called JDB, NetBeans, JBuilder and the Eclipse Java debugger. JDI provides the basic functions such as setting breakpoints, examining variables, stepping, and continuing execution. These commands are encoded using the Java Debug Wire Protocol (JDWP) for sending to the debugging agent in the virtual machine, e.g., over ethernet.

Target Debugging Agents

The debugging agent on the target receives commands from the host, carries them out, and sends the results back to the host. To make it easier to provide diverse debugger implementation and added facilities, JPDA defines an interface between an agent and a Java virtual machine: the Java Virtual Machine Toolkit Interface (JVMTI). The JamaicaVM implements both JVMTI and a debugging agent using ethernet for communicating with a host debugger. The initial version has been tested against Eclipse and JDB with adaptations for other debuggers to follow later.

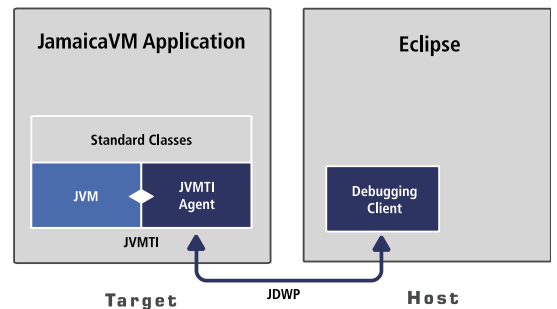


Figure 1: An Architecture for Debugging

In the JamaicaVM

The agent provided with version 3.0 of the JamaicaVM provides basic debugging services enabling developers to use standard debuggers such as the Java debugger in Eclipse to debug an application on a target system. Basic breakpoint, stepping, thread, and object examining functions are available to the developer. More features, including support for compiled code, will be added in later JamaicaVM versions.

(1) <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/index.html>

↓ **Contact**

aicas GmbH
Haid-und-Neu-Straße 18
D-76131 Karlsruhe

aicas incorporated
69 West Rock Avenue
New Haven, CT 06515

email info@aicas.com
web www.aicas.com