

aicas news

Neuigkeiten für Softwareentwickler kritischer Anwendungen

Jubiläumsausgabe 2006

↓ Editorial

Liebe Leser,

dieses Jahr feiern wir vom aicas-Team unser 5-jähriges Jubiläum. Die Arbeit an der JamaicaVM hatte zwar schon lange vor Gründung der aicas GmbH begonnen, seit 5 Jahren jedoch unterstützt aicas die Embedded- und Echtzeit-Branche durch Java-Technologie und Dienstleistungen.

Vieles hat sich in dieser Zeit geändert, sowohl im Markt als auch an unseren Produkten. 2001 war die Akzeptanz von Java auf *Early Adopters* beschränkt. Mittlerweile gibt es ein breites Interesse an Java für Embedded-Applikationen aller Art.

Grund hierfür sind auch verbesserte Tools. Die JamaicaVM hat in dieser Zeit erhebliche Erweiterungen erfahren. Zum Beispiel kam 2004 die Grafikerunterstützung hinzu. Mittlerweile ermöglicht JamaicaVM den Einsatz großer Teile der J2SE in Echtzeit- und Embedded-Anwendungen, und die unterstützten Funktionen werden ständig von uns erweitert.

Ihr

Dr. James J. Hunt, CEO



aicas feiert sein fünfjähriges Jubiläum

Echtzeitfähige Java-Technologie für eingebettete Systeme

Das aicas-Team vollendet das fünfte Jahr der Firmengeschichte und die JamaicaVM ist als kommerzielles Produkt nun ebenfalls seit 5 Jahren verfügbar. Vieles ist in dieser Zeit geschehen.

2001 erblickte die aicas GmbH das Licht der Welt. Zunächst gab es vier Gesellschafter, Andy Walter, Fridtjof Siebert, Götz M. Fluck und James J. Hunt, und ein Produkt, die JamaicaVM. Von Beginn an war der grundlegende Vorteil die Real-Time Garbage Collection. Diese Technologie war eine Entwicklung von Dr. Siebert, für die er später den BMW Scientific Award 2003 erhielt.

Im Herbst 2001 präsentierte aicas erstmals Ihre JamaicaVM auf der SPS/IPC/Drives Messe in Nürnberg. Im Frühling 2002 folgt die Embedded Systems Messe, die ersten aicas news und der Bezug neuer Büros in der Hoepfnerburg.

2002 brachte auch HIDOORS, aicas' erstes EU Projekt. Geleitet von Dr. Hunt unterstützte das 33-monatige Projekt die rapide Entwicklung der JamaicaVM und ermöglichte dem

aicas-Team die Implementierung der Real-Time Specification for Java (RTSJ), sowie die Verbesserung des Compilers und des Laufzeitsystems.

2003 trat Peter Marggrander aicas als CFO bei. EADS Astrium schloss sich mit aicas zusammen um die AERO-VM zu kommerzialisieren, eine Version der JamaicaVM für Satelliten. Die RTSJ-Unterstützung wurde vervollständigt und die ersten englischen aicas news veröffentlicht.

2004 war ein großer Schritt vorwärts. Durch Unterstützung grundlegender AWT Funktionen konnte JamaicaVM zum ersten Mal komplexe grafische Anwendungen ausführen. Siemens begann JamaicaVM in der Simulation Produktlinie zu verwenden. Zur Jahresmitte startete das HIJA Projekt.

Das Wachstum setzte sich 2005 fort. Das Jahr begann mit der Gründung einer unabhängigen Firma, aicas incorporated, um den US-Markt zu bedienen. Die Entwicklung von Swing und Debugging ermöglicht auch weiterhin Erfolg im Jahr 2006.

Die JamaicaVM wird mittlerweile von vielen Firmen in der Industrieautomatisierung wie SICK, Saurer und Multivac genutzt, sie flog mit dem EADS Barracuda und sie wird auch mit Boeing fliegen.

↓ Neuigkeiten

Die JamaicaVM hebt ab!

Das UAV Barracuda flog im Mai erfolgreich mit Java-Software, die auf JamaicaVM lief. Der Flug demonstrierte, dass Java auch in Flugzeug-Software genutzt werden kann. Echtzeit Garbage Collection zusammen mit RTSJ erlaubte den Entwicklern alle Vorteile von Java zu nutzen ohne die Echtzeit-Reaktionsfähigkeit aufzugeben.

Boeing wählt JamaicaVM

Barracuda ist nicht das einzige System, das JamaicaVM im Flug einsetzt. Boeing entschied sich JamaicaVM im neuen 787 Dreamliner zu nutzen. Die JamaicaVM hilft Boeing die Entwicklungsziele rechtzeitig zu erreichen. Ein Prototyp läuft bereits mit JamaicaVM unter VxWorks 6. Der erste Flug ist für 2007 und die Auslieferung für 2008 geplant.

Vxworks 6.3 mit RTP-Support

Die JamaicaVM 3.0 unterstützt sowohl Kernel- als auch Real-Time Processes für die neueste VxWorks Version von Wind River. RTPs bieten Speicherschutz zwischen Anwendungen und bessere Ressourcen-Freigabe. Für Anwendungen ist die Konfiguration des RawMemory-Zugriffs der einzige Unterschied. Alle Java-Programme laufen unverändert.

Zuverlässige Java-Laufzeitfehlererkennung

Verwendung statischer Analysemethoden zur Fehlererkennung und Ressourcenanalyse.

Der enorme Erfolg der Java-Technologie basiert auf den vielen Vorteilen gegenüber anderen populären Programmiersprachen. Starke Typisierung, automatische Speicherbereinigung, Unterstützung der objekt-orientierten Programmierung und eine große Bibliothek von Standardklassen erhöhen die Produktivität und reduzieren die Programmierfehler. Selbst kritische Anwendungen, wie in Steuerungen im Fahrzeug- oder Raumfahrtbereich, profitieren von diesen Vorteilen. Trotzdem können gute Werkzeuge die Produktivität weiter steigern und die Anzahl der Fehler senken. Die wohldefinierte Syntax und Semantik von Java, Reflection und das Bytecode-Format vereinfachen den Bau solcher Werkzeuge. Interessante Ressourcen- und Korrektheitsanalysen von Java-Anwendungen können sowohl auf Quellcode-Ebene als auch auf Bytecode-Ebene durchgeführt werden. Der Fokus der hier beschriebenen Technik liegt dabei auf der Bytecode-Analyse.

Beseitigung von Laufzeitfehlern

Das aicas-Team hat ein Datenflussanalyse-Werkzeug (DFA) entwickelt, um Programmierern zu helfen Laufzeitfehler zu finden und zu entfernen. Durch Nutzung von Objekt-Kontextinformationen erzielt DFA Resultate, mit denen Fehler gefunden werden, die für lokale Analysen nicht erkennbar sind. Das Ergebnis ermöglicht das Finden und Korrigieren von möglichen Laufzeitfehlern.

```

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

Annotations in the code:
 - Blue circle around `device instanceof MyDevice` with label `contains only MySensor`
 - Red circle around `(MySensor)` with label `ClassCastException`
 - Green circle around `device.sensor` with label `NullPointerException`
 - Red circle around `s.reading()` with label `NullPointerException`

Bild 1a: Erfolgreiche instanceof-Prüfung zeigt, dass device nicht null sein kann...

NullPointerException

Dereferenzierung von `null`-Werten ist ein häufiger Fehler in Java-Code. `null` als speziellen Referenzwert zur Verfolgung während der DFA zu markieren ermöglicht die Entdeckung von Verwendungen von `null`. Wo ein Wert dereferenziert wird, wird möglicherweise eine `NullPointerException` ausgelöst, wenn der `null`-Wert in der Wertemenge liegt. Da Instanzen und statische

Referenzfelder in Java mit `null` initialisiert werden, muss die Analyse zuverlässig Initialisierungs-Code entdecken, der diesen `null`-Wert überschreibt. Dann kann die Analyse die Abwesenheit von `NullPointerExceptions` beweisen.

Type-Cast Fehlererkennung

Fehler bei der Typumwandlung werden in ähnlicher Weise gefunden. Java führt Laufzeitprüfungen durch um sicherzustellen, dass eine umgewandelte Referenz dem Zieltypen zuweisbar ist. Gilt dies nicht, wird eine `ClassCastException` geworfen. Exakte Wertemengen ermöglichen die Entdeckung potentieller `ClassCastExceptions`. Bei jeder Umwandlung einer Variable `v` in einen Typ `T` kann die DFA prüfen, ob alle Werte, die `v` annehmen kann, `T` zuweisbar sind. Ist dies der Fall, dann ist die Zulässigkeit des Casts bewiesen.

Array-Store Fehlererkennung

Java erlaubt die Zuweisung von Referenz-Arrays an Array-Variablen eines allgemeineren Elementtyps. Um sicherzustellen, dass die Speicherung eines Elements in einem Array nicht den Wert eines inkompatiblen Typs in einem Array speichert, wird eine Laufzeitanalyse bei jeder Array-Speicherung durchgeführt. Vollständige Wertemengen zu kennen ermöglicht es für jede Array-Speicherung zu prüfen, ob alle möglichen gespeicherten Werte allen Ziel-Array-Typen zuweisbar sind. Ist dies der Fall, dann kann keine `ArrayStoreException` auftreten, anderenfalls wäre die Zuweisung ein potentieller Fehler.

Threading-Fehlererkennung

Threading ist ein mächtiges Feature von Java, aber es ermöglicht auch schwere Fehler über die Programmierer nur selten nachdenken: Ungeschütztes Data-Sharing und Deadlocks. Um von mehreren Threads benutzte Daten zu schützen, werden in Java Monitore in Form von `synchronized`-Blocks oder -Methoden verwendet. Dies nicht zu tun kann zu schwer auffindbaren Fehlern führen. Andererseits können Deadlocks entstehen, wenn mehrere Ressourcen gleichzeitig von einem Thread gesperrt werden. Um hierbei Deadlocks zu vermeiden, muss sichergestellt werden, dass alle Monitore von allen Threads in der gleichen Reihenfolge betreten

werden. DFA findet solche Deadlocks durch die Analyse des Thread-Kontexts bei jeder Synchronisation.

Verwendung von Scoped Memory

Die Real-Time Specification for Java (RTSJ) ermöglicht mittels `ScopedMemory` Speicher explizit freizugeben. Jedoch kann inkorrekte Verwendung von `ScopedMemory` zu Laufzeitfehlern führen. DFA erkennt `MemoryArea`-Zyklen und unzulässige Zuweisungen zwischen Objekten in verschiedenen `ScopedMemory`s.

```

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

Annotations in the code:
 - Blue circle around `device instanceof MyDevice` with label `contains only MySensor`
 - Red circle around `(MySensor)` with label `ClassCastException`
 - Green circle around `device.sensor` with label `NullPointerException`
 - Red circle around `s.reading()` with label `NullPointerException`

Bild 1b: ... und der Typ-Cast ist immer erfolgreich, wenn MyDevice.sensor nur MySensor

Scope Cycle Fehlererkennung

Für die Überprüfung, ob das Betreten eines Scopes zulässig ist, wird für alle `enter-Aufrufe` der jeweilige Parent-Scope aufgezeichnet. Findet die Analyse hierbei für denselben Scope unterschiedliche Parent-Scopes, ist die Einhaltung der "Single Parent"-Regel nicht gesichert, es kann also zu `ScopedCycleExceptions` kommen, die von der Analyse angezeigt werden.

Erkennung von Zuweisungsfehlern

Zuweisungen von Objekten in `ScopedMemory` zu prüfen ist aufwändiger. Die Analyse merkt sich das `MemoryArea` jedes möglichen allozierten Objekts, in der es alloziert wurde. Diese Information wird dann genutzt um alle Zuweisungen auf mögliche Referenzfehler zu prüfen. Wenn die zugewiesene Referenz in einem Speicherbereich alloziert wurde, der nicht identisch ist oder ein Parent-Scope des Zielobjektes, wird ein möglicher `IllegalAssignmentError` bei der Analyse gemeldet.

Wie die DFA funktioniert

Das aicas DFA-Werkzeug bestimmt zwei Wertemengen: Die Menge von Aufrufen und die Menge von Referenzwerten für jedes Feld und Array-Element. Die Analyse verläuft iterativ und verfolgt den Aufrufgraph beginnend mit `main`. Der Datenfluss aller Aufrufe wird in jeder Iteration

analysiert. Der Algorithmus endet, sobald die Mengen während einer Iteration konstant bleiben. Die Analyse unterscheidet zwischen Objekt-Initialisierung und Objektnutzung. Sonderfälle wie Singletons und eingebettete Instanzen werden ebenfalls entdeckt um die Genauigkeit weiter zu steigern. Wird ein Fehler gefunden, ermöglichen Kontextinformationen den Fehler im Code zu lokalisieren.

Programmieren für Analyse

DFA ist eine rein statische Analyse, die dem Fluss der Daten durch die Anwendung folgen kann, aber keine semantischen Beziehungen zwischen Variablen oder Objekten entdeckt. Diese Einschränkung kann dazu führen, dass sogenannte *false positives* berichtet werden, also mögliche Fehler, die nach semantischer Analyse gar nicht auftreten können. Allerdings wird die Analyse niemals ein *false negative* Ergebnis liefern, ein vorhandener Fehler wird mit Sicherheit gefunden. Das bedeutet, wenn keine Fehler gefunden wurden hat die DFA bewiesen, dass keiner dieser Laufzeitfehler auftreten kann. Programmierkonventionen können dabei helfen die Analyse genauer zu machen und damit *false positives* zu vermeiden.

```

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

Bild 1c: ... schließlich kann Variable *s* nie null sein da *device.sensor* immer korrekt initialisiert wird. DFA zeigt das keine Exception

Initialisierung statischer Felder

Statische Felder werden oft genutzt um Referenzen auf konstante Objekte oder Strukturen zu speichern. Der initiale *null*-Wert wird während der Ausführung der Klasseninitialisierung überschrieben, so dass keine *NullPointerException* bei einer späteren Nutzung des Feldes auftreten kann. Leider erlaubt Java den Zugriff auf nicht-initialisierte statische Felder selbst wenn sie *final* sind. Damit die Analyse initiale Feldwerte korrekt erkennen kann, sollte der Static-Initialisierer einer Klasse

- statische Felder bei der Deklaration initialisieren
- keine statischen Anweisungen enthalten
- keine Methoden aufrufen, die auf statische Felder zugreifen können, die gerade initialisiert werden

- statische Felder initialisieren bevor anderer Initialisierungscode ausgeführt wird

Um eine statische Instanz der aktuellen Klasse zu erstellen sollte ein separater Konstruktor benutzt werden, der nicht auf statische Felder zugreift, und erst aufgerufen wird, nachdem alle anderen statischen Felder initialisiert wurden.

Instanzenfelder initialisieren

Die Initialisierung von Instanzenfeldern ist ähnlich. Lineare Zuweisungen der Werte an Instanzenfelder sind einfach zu analysieren, während komplexer Initialisierungscode, der andere Instanzenfelder referenziert, es der Analyse erschwert zu zeigen, dass diese Felder initialisiert sind.

Die Nutzung von *final* für Felder, die nach ihrer Initialisierung nicht geändert werden können, hilft nicht immer. Damit die Analyse die zugewiesenen Feldwerte korrekt erkennt, sollte der Konstruktor

- Felder direkt bei der Deklaration initialisieren
- keine Anweisungen enthalten, die mit Feld-Deklarationen verflochten sind
- keine Aufrufe zu Methoden enthalten, die auf Felder zugreifen, welche gerade initialisiert werden
- Instanzenfelder initialisieren bevor anderer Initialisierungscode ausgeführt wird

Lokale Variablen nutzen

Eine verbreitete Weise um *NullPointerExceptions* zu verhindern, ist auf *null* zu testen, bevor auf ein Objekt zugegriffen wird. Leider funktioniert dies nur, wenn der Referenzwert in einer lokalen Variable gespeichert ist. Ist der Wert in einem Feld gespeichert, könnte ein gleichzeitig laufender Thread den Wert zwischen *null*-Check und Dereferenzierung des Feldes überschreiben. Deshalb kann die DFA nicht annehmen, dass der Wert eines Feldes innerhalb der *if*-Anweisung nicht *null* ist, und sie wird eine potentielle *NullPointerException* melden.

Die Lösung für sichereren Code und bessere Analyse ist die Referenz einer lokalen Variable zuzuweisen, bevor dann mit dem Wert in der Variablen gearbeitet wird. Lokale Variablen können nicht durch andere Threads verändert werden. Diese Nutzung lokaler Variablen verbessert nicht nur die Analysierbarkeit des Codes, sondern macht ihn auch robuster. Selbst ein parallel-laufender Thread, der den Wert des Feldes ändert, provoziert keine *NullPointerException*.

↓ **Veranstaltungen**

Wir möchten Sie auf folgenden Veranstaltungen willkommen heißen:

JTRES 2006, Paris
11–13 October 2006

SPS/IPC/Drives 2006, Nürnberg
28–30 November 2006

Embedded World 2007, Nürnberg
13–15 February 2007

ESC Silicon Valley 2007, San Jose, USA
1–5 April 2007

Eine ähnliche Situation besteht bei Typ-Casts und *instanceof*-Tests. Wiederum kann die Zuweisung zu einer lokalen Variable vor dem Test eine *ClassCastException* verhindern. Auch hier geht sicherer Code Hand in Hand mit einer besseren Analysierbarkeit.

Keine semantischen Typ-Casts

Da semantische Informationen der Analyse nicht zur Verfügung stehen, können sie nicht genutzt werden um die Abwesenheit eines Laufzeitfehlers zu beweisen, der von dieser Information abhängt. Die übermäßige Nutzung von Typ-Casts ist schlechter objektorientierter Programmierstil. Typ-Casts durch das Einführen zusätzlicher Klassen zu verhindern kann zu einem besser strukturierten und besser analysierbaren Code führen.

Fazit

Das DFA-Werkzeug kann die Abwesenheit von Zeigerfehlern, wie *null*-Dereferenzierung und unerlaubte Typ-Casts, sowie von Fehlern, die mit der Nutzung der Mechanismen für explizite Speicherfreigabe in der RTSJ zusammenhängen, beweisen. Die Ergebnisse können dabei durch die Einhaltung einfacher Programmierrichtlinien signifikant verbessert werden. DFA wird offiziell im JamaicaVM Release 3.2 unterstützt, die Ende dieses Jahres erscheinen wird. Ein Pre-release steht interessierten Kunden vorab auf Anfrage zur Verfügung. Weitere Informationen dazu finden Sie auf der aicas Website.

Ausblick

Die nächste Ausgabe der aicas news zeigt wie Java in der Industrieautomatisierung eingesetzt wird.

Zielsystem-Debugging mit der JamaicaVM

Die neueste Version von JamaicaVM unterstützt standard Java Debugging Funktionen

Mit der steigenden Komplexität von Software wächst auch das Verlangen nach Werkzeugen, die bei der Softwareentwicklung helfen. Debugging- und Monitoring-Tools erleichtern das Verstehen des Programmablaufs, insbesondere, wenn mehrere Tasks gleichzeitig laufen. Ein Debugger ermöglicht dem Entwickler das schrittweise Ausführen seiner Anwendung und Betrachten des Systemzustandes. Monitore und Profiler erleichtern das Verständnis der Programmdynamik. Daher unterstützt die neueste Version der JamaicaVM das Java Virtual Machine Toolkit Interface (JVMTI).

↓ **Neue Release**

JamaicaVM Release 3.0

Neben einfachem Remote-Debugging bringt die JamaicaVM 3.0 viele Verbesserungen und Features für grafische und Echtzeit-Anwendungen.

Die Swing-Unterstützung wurde beträchtlich erweitert und die Geschwindigkeit verbessert. In etlichen großen grafischen Anwendungen ist JamaicaVM bereits im Einsatz.

Die VM ist flexibler geworden durch eine komplette Überarbeitung des Garbage-Collectors, um Entwicklern mehr Kontrolle über seine Arbeitsweise zu geben. Die Anzahl der Threads und die Größe des Speichers können dynamisch erweitert werden. Unterstützung nicht-zusammenhängender Heaps ist besonders hilfreich um das 32MB Speicherlimit bei Windows CE zu überwinden.

Durch Cost-Monitoring sind auch dynamische Systeme einfacher zu handhaben. Es ist nun möglich zu erkennen und einzugreifen, wenn ein Thread zuviel Prozessorzeit fordert. Dies ist vor allem bei Server-Anwendungen sinnvoll.

Die Änderungen werden durch einen besseren Support von Tools inklusive eines jamaicac-Befehls abgerundet. Eine Demoversion steht zum Testen auf unserer Website bereit.

Eine Architektur für Debugging

Die Java-Plattform definiert eine Architektur für Debugging, die Java Platform Debugger Architecture (JPDA)(1). Ein Client-Server Modell erlaubt dabei Remote Debugging über TCP/IP auf dem Zielsystem. Grundlegende Dienste werden von einem Server in der JVM, dem sogenannten Agenten, bereitgestellt. Dieser kommuniziert mit einem externen Java-Debugger. Diese Architektur hat den Vorteil, low-level JVM-Anforderungen vom Debugger-Benutzerinterface zu trennen, so dass alle Java-Debugger mit allen virtuellen Maschinen zusammenarbeiten, die JVMTI unterstützen.

Flexibilität

JVMTI spezifiziert eine Schnittstelle für Debugging- und Monitoring- Werkzeuge. Beispielsweise können darüber Java-Threads gestartet und gestoppt werden, Variablen und Felder gelesen und geschrieben, Breakpoints gesetzt, sowie Code schrittweise ausgeführt werden. Damit

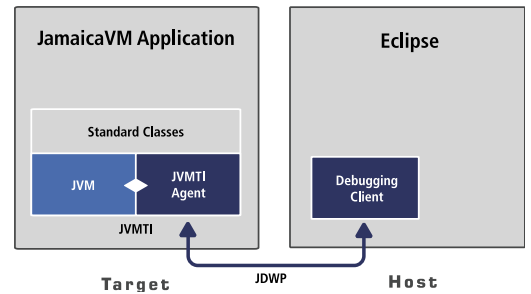


Figure 1: Eine Architektur für Debugging

Host-Debugger

Auf Seiten des Java-Debuggers definiert das Java Debugging Interface (JDI) die grundlegende Funktionalität des Debuggers. Die Schnittstelle wird durch verschiedene Debugger implementiert, darunter auch der in Suns JDK enthaltene Debugger, JDB, NetBeans, JBuilder und Eclipse. JDI spezifiziert grundlegende Funktionen wie das Setzen von Breakpoints, Untersuchen von Variablen, Single Stepping und Fortsetzung des Programms. Diese Befehle werden durch das Java Debug Wire Protocol (JDWP) beispielsweise über Ethernet dem Debugging-Agenten in der virtuellen Maschine übermittelt.

Target Debugging-Agenten

Der Debugging-Agent auf dem Target erhält die Befehle des Hosts, führt sie aus, und übermittelt die Ergebnisse zurück an den Host. Um es einfacher zu machen diverse Debugger-Implementationen zu unterstützen, definiert JPDA ein Interface zwischen dem Agenten und der Java Virtual Machine: das Java Virtual Machine Toolkit Interface (JVMTI). Die JamaicaVM enthält sowohl JVMTI als auch einen Debugging-Agenten, der Ethernet zur Kommunikation mit einem Host-Debugger nutzt. Die erste Version wurde gegen Eclipse und JDB getestet, wobei Tests mit weiteren Debuggern folgen werden.

Monitoring- und Profiling-Agenten effektiv arbeiten können, ist JVMTI als ein schlankes native Interface in der JVM definiert. Eine Reihe von kommerziellen und freien Agenten für spezifische Anbieter sind verfügbar. Unter anderem basieren Adriana, Hyades, JBoss, JProfiler, Optimizeit, YAJP und Yourkit auf solchen Agenten.

In der JamaicaVM

JamaicaVM 3.0 enthält einen Agenten, der grundlegende Debugging-Dienste unterstützt. Damit ist die Verwendung von Standard-Debuggern wie zum Beispiel Eclipse möglich, um Anwendungen auf dem Zielsystem zu debuggen. Dabei stehen dem Entwickler einfache Breakpoint-, Stepping-, Thread- und Objekt-Untersuchungsfunktionen zur Verfügung. Weitere Features, wie Unterstützung für kompilierten Code, werden in späteren Versionen der JamaicaVM folgen.

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

↓ **Kontakt**

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

tel. +49 721 663 968-0
 fax +49 721 663 968-99
 email info@aicas.com
 web www.aicas.com