Echt Zeit für echte Sicherheit

Sicherheitskritische Echtzeitsysteme mit Java

Andy Walter

Die Komplexität sicherheitskritischer Anwendungen nimmt seit Jahren zu. Die Zahl der Sensoren, die von einem Steuergerät ausgewertet werden müssen, wächst ebenso wie die Modularität von Anwendungen. Diese Anforderungen sind mit den Sprachen C und C++ zunehmend mühsam zu erreichen. Dieser Artikel nennt eine Reihe schwer zu vermeidender Fehlerarten, die Java im Gegensatz zu C/C++ bereits konzeptionell ausschließen kann.

Trotz seiner Sicherheitskonzepte ist Java in zertifizierten Systemen derzeit relativ selten, da sich einige Eigenschaften, welche grundsätzlich die Sicherheit eines Systems erhöhen, beispielsweise die automatische Speicherverwaltung, nicht gut mit den statischen Ansätzen vieler Zertifizierungsstandards vertragen. Teilweise wird im Widerspruch zur Intention gerade dadurch die Komplexität zertifizierter Anwendungen sogar erhöht.

Der Artikel beschreibt verschiedene Ansätze, mit denen Java für den Einsatz in sicherheitsrelevanten Systemen verwendbar gemacht wird.

Die Realtime Specification for Java

Die *Realtime Specification for Java* (RTSJ, [Boll01]) wurde 2001 von der Java-Community als Java Specification Request 1 [JSR1] veröffentlicht und hat sich unterdessen als Standard-Java-Erweiterung für Echtzeit-Java am Markt durchgesetzt.

Normale Java-Anwendungen können von der Speicherverwaltung (Garbage Collection, GC) zu beliebigen Zeitpunkten unterbrochen werden, was für Echtzeit-Anwendungen nicht akzeptabel ist. Abbildung 1 illustriert dieses Verhalten.

RTSJ ermöglicht Echtzeit-Anwendungen auch dann, wenn die verwendete virtuelle Java-Maschine nicht über eine echtzeitfähige Speicherverwaltung verfügt. Dazu wird das Thread-Modell aus Abbildung 2 zugrunde gelegt. Echtzeit-Java-Threads können höhere Priorität haben als die Speicherverwaltung, allerdings dürfen diese Threads nicht auf Objekte zugreifen, die im Einflussbereich der Speicherverwaltung liegen. Stattdessen bietet RTSJ das Modell des *Scoped Memory* an: Ein Speicher-Scope wird explizit betreten, danach landen alle Allokationen in diesem Scope.

Die Scopes sind dabei hierarchisch angelegt, wie Abbildung 3 zeigt. Da Speicher-Scopes in umgekehrter Reihenfolge der Erzeugung wieder verlassen werden müssen, sind Zuweisungen von Objekten an Variablen, die in einem höheren Speicher-Scope residieren, nicht zulässig. Derartige Zuweisungen würden in C/C++ erst deutlich später zu einem schwer zu findenden Dangling-Pointer-Fehler führen, in RTSJ führen sie zu einer erheblich einfacher zu lokalisierenden Exception.

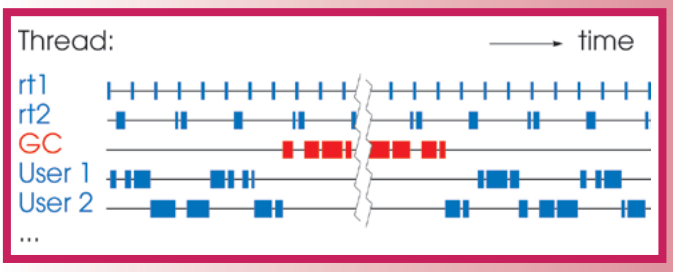

Abb. 2: Thread-Modell in RTSJ: Echtzeit-Threads haben höhere Priorität als die Speicherverwaltung

Abgesehen von Echtzeit-Erweiterungen bringt die RTSJ umfangreiche Funktionalität für den Einsatz von Java in eingebetteten Systemen mit sich. Beispielsweise ermöglicht sie die weitgehend plattformunabhängige Entwicklung selbst systemnaher Programme und Treiber und sorgt für zuverlässigere und leichter portierbare Anwendungen. Für sicherheitskritische Systeme und insbesondere zur Zertifizierung reicht RTSJ alleine jedoch nicht aus.

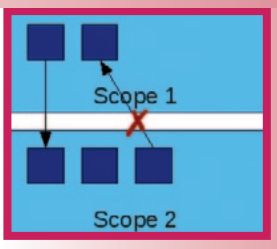

Abb. 3: In RTSJ verhindert Scoped Memory unzulässige Zuweisungen, die beim Verlassen des Scopes zu Dangling Pointers führen könnten, durch das Werfen einer Exception bereits zum Zeitpunkt der Zuweisung

Echtzeitfähige Speicherverwaltung

Eine echtzeitfähige Speicherverwaltung muss in der Lage sein, für jede auszuführende Programmanweisung eine maximale obere Zeitschranke zu garantieren. Außerdem muss sichergestellt werden, dass ein Thread mit höherer Priorität innerhalb einer definierten Zeit einen laufenden Thread unterbrechen kann, selbst wenn das System gerade mit Speicherverwaltung beschäftigt ist. Dies kann erreicht werden, indem die Speicherverwaltung nicht in einem eigenen Thread, sondern ausschließlich zum Zeitpunkt der Allokation eines Objektes ausgeführt wird.

Fridtjof Siebert beschreibt in [Sieb02] und [Sieb07] eine Speicherverwaltung, die für jede Allokation eine obere Zeitschranke garantieren kann und somit deterministisch ist. Abbildung 4 illustriert das Zeitverhalten eines derartigen Systems. Für Threads

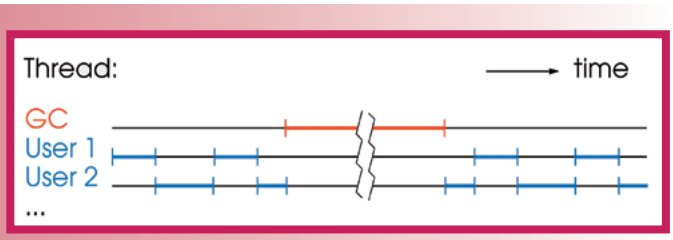

Abb. 1: Normale Java-Threads können von der Speicherverwaltung beliebig unterbrochen werden

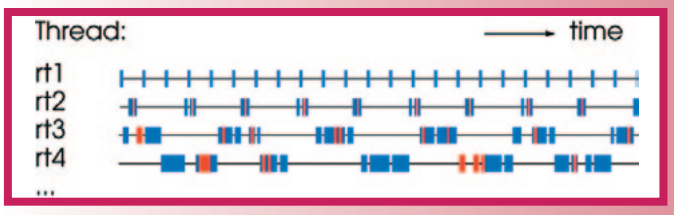

Abb. 4: Eine echtzeitfähige Speicherverwaltung muss deterministisch und inkrementell arbeiten und dabei jederzeit unterbrechbar sein

wie rt1 im Beispiel, die keine Objekte allozieren, lässt sich somit leicht nachweisen, dass keinesfalls eine Beeinträchtigung durch die Speicherverwaltung erfolgt. Wichtig ist jedoch, dass mit diesem Ansatz selbst Threads mit höchster Priorität Objekte erzeugen dürfen, ohne dass die Echtzeitfähigkeit verloren geht.

SCJava - Der Weg zur Zertifizierung

Die Java-Community arbeitet seit 2006 an einer Untermenge der Java-Standard-Klassen, die für den Einsatz in sicherheitskritischen Anwendungen zertifizierbar ist. Der neue Standard, welcher als JSR 302 [JSR302] spezifiziert wird, adressiert dabei den höchsten Sicherheitsstandard der zivilen Luftfahrt, DO-178B, Level A [DO-178B]. Um dies zu erreichen, sind eine Reihe von Einschränkungen erforderlich, u. a. wird auf automatische Speicherverwaltung verzichtet, da nach dem alten Standard der Sicherheitsnachweis zu aufwendig wäre.

Durch den Wegfall der Speicherverwaltung verzichtet *Safety Critical Java* (SCJava) jedoch auf einen großen Sicherheitsvorteil von Java gegenüber Ada, C und C++. Zur Speicherverwaltung kommt hier das aus RTSJ bekannte Scoped Memory zum Einsatz. Dies ist ein deutlicher Fortschritt gegenüber dem bisher insbesondere in C++-Anwendungen verbreiteten Weg, Speicher-Pools zu verwenden. Durch die zunehmende Komplexität von Software wird die manuelle Speicherverwaltung jedoch fehleranfälliger. Die Komplexität der Anwendungen könnte durch den Einsatz einer automatischen Speicherverwaltung deutlich reduziert werden, sofern nachgewiesen werden kann, dass sie deterministisch und echtzeitfähig ist.

Trotz dieser Einschränkung ist SCJava ein guter erster Schritt in Richtung Zertifizierbarkeit von Java-Anwendungen für sicherheitskritische Anwendungen und ein deutlicher Schritt nach vorne gegenüber Ada oder gar unsicheren Sprachen wie C oder C++.

DO-178C: OOT hebt ab

In sicherheitsrelevanten Anwendungen war es bislang üblich, jede Art von Dynamik zu verbieten, objektorientierte Technologie (OOT) sehr stark einzuschränken und automatische Speicherverwaltung komplett zu verbieten. In der Luftfahrtindustrie hat man das Potenzial von OOT unterdessen erkannt und erlaubt mit dem Nachfolgestandard DO-178C [SG5] ausdrücklich die Verwendung einer automatischen Speicherverwaltung, sofern nachgewiesen werden kann, dass das System dadurch nicht unsicher oder undeterministisch wird. Im Sinne der Softwarequalität bleibt zu hoffen, dass das Vorbild aus der Luftfahrt auch in anderen Industriezweigen Schule machen wird. Wir betrachten daher im Folgenden den für OOT relevanten Teil der DO-178C detaillierter, mit besonderem Augenmerk auf die Bedeutung für Java-Anwendungen.

Sicherheitskritische Software in zivilen Flugzeugen muss bisher gemäß DO-178B (USA) bzw. ED-12B (EU) zertifiziert werden. Der Standard aus dem Jahr 1992 berücksichtigt noch keine OOT, was die Zertifizierung von Java-Anwendungen erschwert: Einerseits erleichtert OOT die Entwicklung robuster Anwendungen und beseitigt gefährliche Fehlerquellen, andererseits bringt es eine Reihe neuer Fehlermöglichkeiten, die bei der Zertifizierung adressiert werden müssen. Die Arbeitsgruppe SG-5 der SC-205/WG-71-Versammlung arbeitet derzeit an Richtlinien zur Zertifizierung von OOT, die in den Nachfolgestandard DO-178C einfließen. Für jede OOT-Eigenschaft muss nachgewiesen werden, dass keine dadurch bedingten Fehler in der Anwendung enthalten sind. Die gute Nachricht für Java-Entwickler ist, dass ein Großteil der OOT-Schwachpunkte in Java bereits aufgrund der Sprachspezifikation oder der automatischen Speicherverwaltung ausgeschlossen werden kann.

Für sicherheitskritische OOT-Anwendungen ist zu zeigen, dass folgende Schwachpunkte nicht auftreten:

Vererben und Überschreiben

Vererbung ist eines der Hauptmerkmale von OOT. Die Kapselung von Code und Daten erleichtert die Wiederverwendung von Code innerhalb einer Anwendung oder sogar anwendungsübergreifend. Wichtig ist, dass auch in Unterklassen die Spezifikation von Elternklassen eingehalten wird. Barbara Liskovs Substitutionsprinzip definiert die erforderliche Typkompatibilität formal unter Verwendung von Vor- und Nachbedingungen sowie Invarianten: In Unterklassen dürfen Vorbedingungen nicht verstärkt und Nachbedingungen und Invarianten nicht abgeschwächt werden gegenüber der Oberklasse. Die Verifikation einer Klasse muss daher auch mit allen Unterklassen möglich sein. Umgekehrt muss jede Klasse anhand ihrer eigenen Spezifikation und der ihrer Oberklassen verifiziert werden können. Zur formalen Verifikation bieten sich Java-Annotationen an, um Vor- und Nachbedingungen direkt mit dem Anwendungscode festzuhalten.

Statischer Methoden-Dispatch bringt ein weiteres Problem mit sich: Welche Implementierung einer überschriebenen Methode letztendlich aufgerufen wird, wird durch den *deklarierten* Typ des Objektes bestimmt - und nicht durch den *echten*. Dies erschwert die Wiederverwendbarkeit von Code, da die Implementierung von Methoden einer Oberklasse unter Umständen angepasst werden muss, um mit Objekten von abgeleiteten Klassen zu funktionieren. Das Verhalten bei C++ ist besonders verwirrend: Sowohl statischer als auch dynamischer Dispatch sind möglich und können sogar beide für verschiedene Methoden derselben Klasse verwendet werden. In Java ist der Methoden-Dispatch immer dynamisch. Der tatsächliche Typ eines Objektes ist also ausschlaggebend dafür, welche Methode letztendlich aufgerufen wird.

Auch Mehrfachvererbung birgt Gefahren: Oftmals ist unklar, welche Implementierung einer Methode aufgerufen wird. Java vermeidet dies, indem Mehrfachvererbung nur für Schnittstellen, aber nicht für Klassen möglich ist. Damit kommt trotz Mehrfachvererbung stets nur eine Implementierung für den Aufruf infrage. Dennoch müssen Entwickler und Gutachter Sorge tragen, dass verschiedene Spezifikationen einer Methode einander nicht widersprechen. In C++ dagegen müsste dieser Abgleich aufwendig auf Implementierungsebene erfolgen.

Parametrischer Polymorphismus

Parametrischer Polymorphismus, der in Java in Form von Generics bzw. in C++ in Form von Templates vorhanden ist, stellt eine Typ-konsistente Möglichkeit zum Wiederverwenden von Code dar, wenn die Verwendung von Unterklassen nicht möglich oder unpraktisch ist. Beispielsweise typsichere Listen, deren einzelne Methoden unabhängig vom jeweiligen Typ sind, die aber jeweils nur Elemente desselben Typs beinhalten sollen. Zur Zertifizierung muss jede Instanz eines parametrierten Typs verifiziert werden. Generics in Java sind typsicher. So weit wie möglich finden die zugehörigen Überprüfungen bereits während der Compilierung statt.

Typkonvertierung

Typkonvertierung kann - nicht nur bei OOT-Anwendungen - erforderlich sein. Das stark typisierte Java erkennt die meisten Typfehler bereits zur Übersetzungszeit. Spätestens zur Laufzeit bewirkt ein Typfehler eine Exception - an der Stelle würde

C/C++ mit korrupten Daten weiter arbeiten. Zur Zertifizierung ist jedoch ein statischer Nachweis erforderlich. Statische Analyse, beispielsweise auf Basis einer Datenflussanalyse, kann die korrekte Typisierung einer Anwendung überprüfen. Dieser Ansatz ist grundsätzlich auch für C/C++-Anwendungen möglich. Die Aussagekraft einer Datenflussanalyse über ein Java-Programm ist jedoch deutlich stärker als bei C/C++ [HuToSie08].

Methodenüberladung

Methodenüberladung kann die Lesbarkeit eines Programmes verbessern. In Verbindung mit impliziter Typkonversion kann es jedoch zu Mehrdeutigkeit kommen. Daher sollten Programmierrichtlinien klarstellen, unter welchen Umständen Überladung erlaubt ist und die Verwendung von impliziter Typkonvertierung möglichst ganz verbieten.

Exceptions

Die meisten OOT-Sprachen unterstützen das Werfen von Exceptions anstatt der Rückkehr aus einer Methode mit einem normalen Rückgabewert. Dies ist grundsätzlich sicher und komfortabel, kann aber zu Problemen führen, wenn eine Exception erst sehr tief im Aufrufstack (oder gar nicht) behandelt wird.

In Java gibt es Checked und Unchecked Exceptions. Bereits zur Übersetzungszeit wird sichergestellt, dass geworfene Checked Exceptions die jeweilige Methode nicht unbehandelt und undeklariert verlassen können. Unchecked Exceptions dagegen treten bei Laufzeitfehlern wie Division durch Null oder Bereichsüberprüfungen auf und müssen nicht explizit behandelt werden. Sie werden von der virtuellen Java-Maschine bei schwerwiegenden Fehlern geworfen, die ohne Ausnahmebehandlung zum Absturz führen würden. Mithilfe von Datenflussanalyse lässt sich mathematisch nachweisen, dass alle Unchecked Exceptions behandelt wurden.

Dynamische Speicherverwaltung

Für komplexe Aufgaben wird häufig temporärer Speicher benötigt. Der komfortabelste und sicherste Weg zur Verwaltung solcher Daten ist eine automatische Speicherbereinigung. Die Arbeitsgruppe SG-5 hat alle Fehlermöglichkeiten, die im Zusammenhang mit dynamischer Speicherverwendung auftreten können, identifiziert. Der einzige dieser Fehler, der durch den Einsatz eines echtzeitfähigen Garbage Collectors nicht bereits konzeptionell ausgeschlossen werden kann, ist die Verwendung von zu viel Speicher durch die Anwendung.

- *Mehrdeutige Referenzen:* Objekte überlappen im selben Speicherbereich. Dies kann verhindert werden durch Verwendung eines einzigen Allokators (pro Speicherbereich), sofern die Sprache - wie Java - keine Zeigerarithmetik erlaubt. In C oder C++ wäre es erforderlich, sicherzustellen, dass keine Zeigerarithmetik verwendet wird, was relativ schwer ist, da selbst gewöhnliche Array-Zugriffe auf Zeigerarithmetik angewiesen sind. Mehrdeutige Referenzen entstehen auch, wenn Objekte nach der Zerstörung weiterverwendet werden.
- *Speicherfragmentierung:* Die Speicherverwaltung muss sicher stellen, dass freier Speicher trotz Fragmentierung auch für große Objekte verwendet werden kann. Die meisten Java-Garbage-Kollektoren können dies garantieren. C- und C++-Anwendungen sind insbesondere in eingebetteten Systemen an der Stelle gefährdet.
- *Speicherlöcher:* Nicht mehr benötigte Objekte müssen schnell genug wieder freigegeben werden. Einige Java-Garbage-Kollektoren können dies sicherstellen.
- *Zu wenig Speicher:* Die Anwendung muss sicherstellen, dass der verfügbare Speicher groß genug ist, damit alle gleichzeitig erreichbaren Objekte hineinpassen.
- *Deallokation lebendiger Objekte:* Hierdurch entstehen Dangling Pointers - ein gängiges Problem in C- und C++-Anwendungen. Alle Java-Garbage-Kollektoren lösen dies.
- *Verschobene Objekte:* Einige Speicherverwaltungen verschieben Objekte im Speicher, um Fragmentierung zu beseitigen. In dem Fall muss sichergestellt werden, dass alle Zugriffe auf die neue Speicheradresse erfolgen.
- *Undeterministische Allokation oder Deallokation:* Dynamische Speicherverwaltungen dürfen die Anwendung nicht unerwartet unterbrechen. Die meisten Garbage-Kollektoren, welche die anderen genannten Schwachpunkte verhindern können, scheitern am Determinismus, beispielsweise der unter C++ verbreitete Boehm Garbage Collector. Der Garbage Collector der virtuellen Java-Maschine JamaicaVM garantiert Determinismus [Sieb02,Sieb07] und verhindert alle anderen Schwachpunkte in diesem Kapitel mit Ausnahme von „Zu wenig Speicher“.

Um auszuschließen, dass die genannten Schwachpunkte objektorientierter Systeme ein Problem in einer Anwendung verursachen, existieren diverse Möglichkeiten, wie Abbildung 5 zeigt. C-Anwendungen für eingebettete Systeme, die nicht zertifiziert werden müssen, arbeiten häufig mit manueller Allokation und überlassen die Probleme dem Anwendungsentwickler. Insbesondere die Fragmentierung ist damit sehr schwer auszuschließen.

Objekt-Pools sind nur wenig besser: Anstatt ein temporäres Objekt nach der Verwendung zu zerstören, geht es in einen Objekt-Pool zurück und kann wiederverwendet werden. Dies spart insbesondere bei aufwendig zu erzeugenden Objekten Zeit und beseitigt die typische „malloc-Fragmentierung“. Gleichzeitig führt es eine neue Art von Fragmentierung ein: Freie Objekte können in anderen Pools noch existieren, aber der Pool für Objekte des gerade benötigten Typs ist leer. Im Gegensatz zur manuellen Heap-Allokation ist dieses Fragmentierungsproblem durch den Anwendungsentwickler wenigstens lösbar.

Bei der Stack-Allokation werden lokale Objekte auf dem Aufruf-Stack gespeichert und beim Verlassen der Methode wieder gelöscht. Dies verringert die Fragmentierungsgefahr, erschwert es gleichzeitig aber, Objekte über mehrere Threads zu verwenden. Die Verwendung von Speicher-Scopes kann hier helfen, diese können aber wiederum zu Fragmentierung führen. Zuordnungsregeln für Scoped Memory können Dangling Pointers ausschließen - eine Gefahr, die bei Stack-Allokation in C und C++ weiterhin besteht.

Automatische Speicherbereinigung ist am komfortabelsten und ist, wenn sie die in DO-178C genannten Schwachpunkte ausschließen kann, die sicherste Art, mit dynamischem Speicher umzugehen. Der Garbage Collector sollte sorgfältig ausgewählt werden: Einige finden freien Speicher nicht in allen Fällen schnell genug oder der Speicher kann durch Fragmentierung ausgehen. Andere sind nicht deterministisch und kön-

Technik	Mehrdeutige Referenzen	Speicherfragmentierung	Speicherlöcher	Zu wenig Speicher	Deallokation lebendiger Objekte	verschobene Objekte	Undeterministische Allokation / Deallokation
Manuelle Allokation	✗	?	✗	✗	✗	N/A	4
Objekt Pool	✗	?	✗	✗	✗	N/A	4
Stack Allokation	✗	4	4	✗	✗	N/A	4
Scope Allokation	4	4	4	✗	✗	N/A	4
Automatische Speicher Bereinigung	4	4	4	✗	4	4	4

4 = automatisch verwaltet, ? = durch die Anwendung
N/A = nicht anwendbar, ? = schwer sicher zu stellen

Abb. 5: Speicherverwaltungstechniken und damit verbundene Problemstellungen

nen keine obere Zeitschranke für Allokationen und Deallokationen garantieren. Die echtzeitfähige Speicherverwaltung der JamaicaVM kann dies garantieren, wie oben gezeigt wurde.

Virtualisierung
Virtualisierung erleichtert die Portierbarkeit und Wiederverwendbarkeit einer Anwendung und reduziert in der Regel die Komplexität. Java-Anwendungen laufen generell in einer virtuellen Umgebung, der Java Virtual Machine. Die größte Schwachstelle in dem Zusammenhang ist, dass der interpretierte Code nicht ausreichend validiert wurde, weil er als Daten behandelt wird. DO-178C lässt Virtualisierung grundsätzlich zu, verlangt aber, dass jede Schicht getrennt verifiziert wird: Zur Verifikation des Interpreters wird die Anwendung als Daten betrachtet. Zusätzlich ist eine Zertifizierung des interpretierten Codes erforderlich, indem der Interpreter als Ausführungsplattform betrachtet wird.

Verifikation von Anwendungen

Zahlreiche Fehlerquellen, die unter C/C++ zu schwer lokalisierenden Laufzeitfehlern und undeterministischem Verhalten führen, rufen unter Java im besten Fall bereits einen Compilerfehler und im schlimmsten Fall eine Laufzeitausnahme hervor. Um auch Laufzeitfehler ausschließen zu können, eignet sich Java aufgrund der fehlenden Zeigerarithmetik besonders für statische Analysen mit formalen Methoden: Durch Datenflussanalyse lässt es sich ausschließen, dass bestimmte Fehler in der Anwendung vorkommen bzw. es lassen sich die Stellen eindeutig eingrenzen, an denen die jeweiligen Fehler auftreten könnten. Dabei wird für jedes Vorkommen einer Variablen die zugehörige Wertemenge bestimmt. Kann somit beispielsweise gezeigt werden, dass ein Variablenvorkommen an einer bestimmten Stelle im Programm unter keinen Umständen den Wert Null enthalten kann, so kann durch die Variable geteilt werden, ohne dass an der Stelle eine DivisionByZeroException auftritt.

Die Aussagekraft von Datenflussanalyse ist bei C- oder C++-Anwendungen deutlich schwächer: Sie könnte sich nicht darauf verlassen, die Wertemengen aller Variablenvorkommen sicher zu kennen, da beispielsweise fehlerhafte Zeigerarithmetik an anderer Stelle des Programmes den Variablenwert verändert haben könnte. Abbildung 6 zeigt, welche Fehlerklassen zu welchem Zeitpunkt gefunden werden können.

	C	C++	Java	Java + Datenfluss Analyse
Typfehler	✗	✗	o	✔
Deadlocks	✗	✗	✗	✔
Dangling Pointers	✗	✗	✔	✔
Fragmentierter Speicher	✗	✗	✔	✔
Array Zugriff	✗	✗	o	✔
Race Conditions	✗	✗	✗	✔
Uninitialisierte Variablen	✗	✗	✔	✔
Nullpointer Deref.	✗	✗	o	✔
Division durch Null	✗	✗	o	✔
Ausnahmen Behandlung	✗	o	o	✔

✗ Gefahr, o Laufzeit Fehler, ✔ Während der Entwicklung gefunden

Abb. 6: Mithilfe von Datenflussanalyse können zahlreiche Fehlerklassen zur Entwicklungszeit ausgeschlossen werden

Zusammenfassung

OOT im Allgemeinen und Java im Besonderen verbessern die Effizienz von Entwicklern, indem sie die Komplexität großer Anwendungen reduzieren. Automatische Speicherverwaltung, ein starkes Typsystem und das Verbot von Zeigerarithmetik und Mehrfachvererbung verhindern zuverlässig schwer zu findende Fehler, die häufig in C- und C++-Anwendungen vorkommen. RTSJ ist eine gute, sichere und verbreitete Erweiterung, um Echtzeit-Anwendungen in Java zu schreiben. Es ist ebenfalls ein gutes, aber alleine noch nicht ausreichendes Fundament für zertifizierte Java-Anwendungen, beispielsweise nach DO-178B. In Kombination mit einer echtzeitfähigen Speicherverwaltung sind Zertifizierungen bis hin zu Level C möglich. Der derzeit entwickelte Nachfolgestandard DO-178C definiert klare Regeln, die bei der Verwendung von OOT und automatischer Speicherbereinigung gelten. Dies ist ein Meilenstein in der Entwicklung künftiger sicherheitskritischer Anwendungen, die wesentlich schwierigere Aufgaben zu erfüllen haben als die derzeitig im Einsatz befindlichen und daher bessere Werkzeuge erfordern.

Literatur und Links

[Boll01] G. Bollela, Real-Time Specification for Java, Addison-Wesley, 2001
[DO-178B] Software considerations in airborne systems and equipment certification, Advisory Circular DO-178B, Radio Technical Commission for Aeronautics, 1992, Errata Issued 3-26-99
[HuToSie08] J. J. Hunt, I. Tonin, F. Siebert, Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time Java programs, in: Volume 343 of ACM International Conference Proceeding Series, 2008
[JSR1] Java Specification Request 1, Real-time Specification for Java, final release 3: RTSJ version 1.02, `http://jcp.org/en/jsr/detail?id=1`
[JSR282] Java Specification Request 282, RTSJ version 1.1, `http://jcp.org/en/jsr/detail?id=282`
[JSR302] Java Specification Request 302, Safety Critical Java Technology, `http://jcp.org/en/jsr/detail?id=302`
[SG5] J. Chelini, P. Heller und SG-5, Technical Supplement Template (Draft OOT supplement to support DO-178C, Revision C of 25th of June 2009), Technical Report RTCA/DO-OOT, SC-205/WG-71 Plenary, 2009
[Sieb02] F. Siebert, Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages, aicas Books, 2002
[Sieb07] F. Siebert, Realtime Garbage Collection in the JamaicaVM 3.0, in: 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007), 2007, ACM Press
[Walt09] A. Walter, Java in Safety Critical Systems, in: Embedded World Conference, 2009

Andy Walter war nach seinem Informatikstudium in Saarbrücken als wissenschaftlicher Mitarbeiter am Forschungszentrum Informatik in Karlsruhe tätig. 2001 hat er gemeinsam mit Kollegen die aicas GmbH gegründet, den Hersteller der echtzeitfähigen Java Virtual Machine JamaicaVM. Heute ist er als Chief Operations Officer für die Geschäftstätigkeit von aicas in Europa und Asien verantwortlich.
E-Mail: anwalt@aicas.com