Safety Critical Java erleichtert die Zertifizierung sicherheitskritischer Echtzeit-Anwendungen

Thomas Henties

CT SE 2
Siemens AG
Otto-Hahn-Ring 6
81730 München
thomas.henties@siemens.com

Abstract: Der Vortrag informiert über den aktuellen Stand des *Java Community Prozesses* für die beiden *Java Specification Requests*:

- JSR-282 (*Real-Time Specification for Java 1.1*) und
- JSR-302 (*Safty Critical Specification for Java*).

Java war von Anfang an auch für *Embedded-Devices* gedacht, was auch die frühe Spezifikation von Echtzeit-Java (JSR-001) zeigt. Inzwischen werden von einigen Firmen *Java Virtuelle Maschinen* (JVMs) angeboten, die die *Real Time Specification for Java 1.0* (RTSJ) implementieren. Im Laufe des Jahres 2008 wird eine aktualisierte und verbesserte Version der RTSJ verabschiedet werden. Außerdem wird die erste Spezifikation für sicherheitsrelevante Java-Anwendungen (SCSJ) der Öffentlichkeit vorgestellt werden, die auf der neuen *Real-Time Spezifikation* basiert. Dabei wird auf alles verzichtet, was die statischen Analyse-Möglichkeiten zu sehr einschränkt. Oberstes Ziel ist es, die Zertifizierung der Java Applikationen (z.B. nach *DO 178B level A*) zu ermöglichen. Im Folgenden wird anhand der Anforderungen an sicherheitsrelevante Programme gezeigt, wie RTSJ modifiziert wird, um den gewünschten Anforderung gerecht zu werden.

1 Einleitung

Über 90 Prozent aller Mikroprozessoren werden für Echtzeit-Anwendungen eingesetzt, die meist mit ihrer Umgebung interagieren [Cor04]. Solche Systeme müssen nicht nur mit beschränkten Ressourcen realisiert werden, sondern außerdem zuverlässig und vorhersehbar funktionieren.

Zudem werden die funktionalen Anforderungen immer komplexer bei gleichzeitiger Verkürzung der Entwicklungszeiten. Deshalb ist es wichtig, dass die Vorteile von High-Level-Programmiersprachen, deren Methoden und deren Entwicklungswerkzeuge auch für Embedded- und Echtzeit-Anwendungen genutzt werden, um bessere Qualität und höhere Produktivität zu erreichen.

Java hat sich in Desktop- und Serversystemen durchgesetzt, weil es viele unnötige Schwierigkeiten, wie z.B. Typ- und Speicherfehler vermeidet und ein hohes Maß an Sicherheit und Produktivität bietet. Diese Attraktivität hat dazu geführt, dass "abgespeckte" Versionen (*Java Micro Edition*) weite Verbreitung in *Embedded Systemen* wie PDAs, Handys oder Set-Top-Boxen gefunden haben.

Trotz dieser guten Eigenschaften ist Standard-Java für Echtzeit-Anwendungen ungeeignet. Dies liegt hauptsächlich daran, dass Java zwar *Threads* und Synchronisierung unterstützt, das *Scheduling* aber weder genormt noch durch die Applikation beeinflussbar ist. Außerdem verhindert die *Garbage-Collection* und ein zu grob getakteter Timer den Einsatz von Java [Sto07]. Diese Probleme werden durch die *Real-Time Specification for Java* [RTSJ] gelöst. RTSJ hat sich inzwischen unter anderem in Anwendungen der Automatisierungstechnik, der Luft- und Raumfahrt [Bak06], aber auch bei Finanzapplikationen bewährt. Deshalb möchte man diese Technologie jetzt auch für sicherheitskritische Systeme einsetzen. Die Definition der SCSJ [JSR302] und neue Analysewerkzeuge machen die Verwendung von Java in sicherheitskritischen Systemen attraktiv.

2 Der Java Community Process

Der *Java Community Process* (JCP) ermöglicht die Beteiligung von Personen und Organisationen an der Definition von Funktionen und Eigenschaften der Java Plattform. Außer der Spezifikation muss auch eine *Reference Implementation* (RI) und ein *Technology Compatibility Kit* (TCK) verpflichtend erstellt werden. Die *Reference Implementation* (RI) zeigt, dass die Spezifikation auch wirklich implementiert werden kann. Das TCK ist eine *Testsuite*, die alle Anforderungen der Spezifikation vollständig überprüft. Jede Implementierung eines *JSRs* muss alle *TCK-Tests* bestehen, um ihre Spezifikationstreue nachzuweisen. Dadurch wird die Kompatibilität verschiedener Implementierungen gewährleistet.

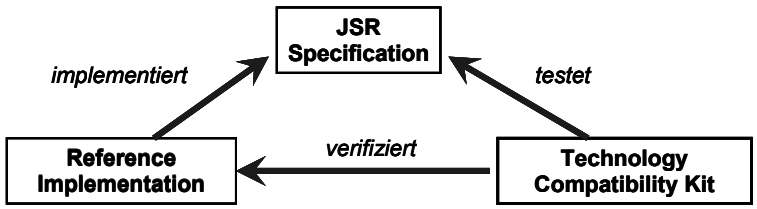

Abbildung 1: Komponenten eines *Java Specification Requests*

Für den JSR-282 (RTSJ 1.1) stehen inzwischen Vorabversionen der Spezifikation und der RI zur Verfügung. Die *Final Specification* wird im Laufe des Jahres 2008 verabschiedet werden. Für den JSR-302 (SCSJ) wird zur Zeit die erste Version der Spezifikation in der *JSR-Expert-Group* diskutiert. Mit der *Final Specification* ist erst nach der Abstimmung über den JSR-282 zu rechnen. Für den JSR-001 (RTSJ 1.0) stehen inzwischen verschiedene *Java VMs* zur Verfügung, unter anderem von Sun, IBM, Aonix und von der deutschen Firma Aicas. Alle genannten Anbieter wollen zukünftig auch JSR-282 und JSR-302 kompatible *VMs* anbieten.

3 Begriffsklärung

In einem Echtzeitsystem ist es erforderlich, dass – neben der Berechnung korrekter Ergebnisse – diese Ergebnisse rechtzeitig, also weder zu früh noch zu spät zur Verfügung gestellt werden. Anhand der Folgen, die eine Nichteinhaltung der Zeitschranken auslösen kann, werden Echtzeitsysteme in drei Kategorien unterteilt: Man spricht von:
soft real time wenn eine unpünktliche Berechnung lediglich zum Qualitätsverlust der Anwendung führt. Beispiele hierfür sind: Videoverschlüsselung und Telefonvermittlung.
hard real time wenn eine unpünktliche Berechnung zu einer Störung, einem Schaden bzw. zu finanziellem Verlust führt. Beispiele hierfür sind: Steuerung von Fahrzeugfunktionen wie z.B. Bremsen, Roboter-Steuerung oder auch Aktienhandels-Programme.
safety critical wenn eine unpünktliche Berechnung dazu führen kann, dass Menschenleben gefährdet sind. Beispiele hierfür findet man in den Bereichen: Luft- und Raumfahrt, Militär, Medizin und Kraftwerksteuerung.

Die *Radio Technical Commission for Aeronautics* hat bereits im Jahr 1992 Richtlinien für die Entwicklung vom Software für die Luftfahrt in den USA definiert. Diese tragen die Bezeichnung **DO-178B** [Do178] und sind in sogenannte *Levels*: A bis E unterteilt; wobei *Level A* die höchsten Anforderungen beinhaltet. Der Erfolg von DO-178B hat sich auch in anderen Branchen und Ländern fortgesetzt und ist die Grundlage zahlreicher Zertifizierungsprozesse.

Es gibt eine ganze Reihe von Werkzeugen, die bestimmte Eigenschaften von Java-Programmen, bzw. *Java class-files*, statisch analysieren. Die SCSJ geht davon aus, dass solche Tools existieren - sie sind jedoch nicht Teil der SCSJ, da hier Wettbewerb und Weiterentwicklung erwünscht ist. Datenflussanalyse, *Model-Checking* und Deduktive Verifikation sind Kandidaten zum Nachweis von *Post-Conditions*, Erreichbarkeit oder *Worst-Case-Execution-Time*. So kann beispielsweise durch eine Analyse leicht überprüft werden, ob die Initialisierungsreihenfolge der verwendeten Klassen zyklusfrei ist.

4 Änderungen in der RTSJ im Vergleich zur Java Standard Edition

Neben dem bekannten *Heap-Memory* unterstützt RTSJ *Immortal-Memory*, das nie *garbage-collected* wird und das während der gesamten Programmlaufzeit existiert. Außerdem gibt es das sogenannte *Scoped-Memory*, das wie ein *Cactus-Stack* organisiert ist. *Scoped-Memory* muss ebenfalls nicht *garbage-collected* werden, sondern ein *Scope* wird stets komplett freigegeben.

Die von Standard-Java unterstützen Prioritäten müssen nicht verpflichtend beachtet werden, deshalb sind sie für Echtzeit-Anwendungen ungeeignet. RTSJ verwendet 28 zusätzliche, vorrangige Prioritäts-Level. Es werden erweiterbare Scheduling-Protokolle angeboten, um Prioritäts-Inversion zu vermeiden und speziellen Anwendungsbedürfnissen gerecht zu werden. Neben den bekannten *Java-Threads* gibt es *Real-Time-Threads*, die mit einer Real-Time-Priorität behaftet sind, die aber von einer *Garbage-Collection* unterbrochen werden können. Außerdem gibt es *No-Heap-Real-Time-Threads*, die den *Heap* nicht benutzen und deshalb auch keine *Garbage-Collection* auslösen können.

Asynchrone Ereignisse werden nicht an einen dedizierten *Thread* weitergeleitet. Vielmehr wird der von dem Ereignis betroffene *Thread* seine aktuelle Ausführung unterbrechen und einen alternativen, für diesen Fall vorgesehenen Algorithmus ausführen. Diese Vorgehensweise gleicht dem *Exception-Handling*; asynchrone Ereignisse müssen aber nicht an allen Stellen sofort bearbeitet werden.

Da Millisekunden für Echtzeit-Anwendungen nicht genau genug sind, werden in RTSJ-Programmen die Zeiten in Nanosekunden angegeben. Falls der Hardware-Timer keine Nanosekunden unterstützt, dann wird die feinstmögliche Auflösung verwendet. Auf die Unterschiede zwischen RTSJ 1.0 und RTSJ 1.1 [JSR282] wird nicht näher eingegangen.

5 Änderungen in der SCSJ im Vergleich zur RTSJ

SCSJ basiert auf den Ergebnissen des EU-Projekts HIJA [HIJA], wurde aber in vielen Punkten weiterentwickelt und verfeinert, um zu einer dem JCP genügenden Spezifikation und Referenz-Implementierung zu kommen. Das Hauptaugenmerk bei der Definition von JSR-302 ist es, eine Zertifizierung (z.B. nach DO 178B level A) so weit wie möglich zu unterstützen. Deshalb waren einige drastische Einschnitte in den Sprachumfang nötig, allen voran der Verzicht auf Heap-Memory und somit einigen Methoden des *java.lang-Paketes*. Von den Threads werden nur No-Heap-Realtime-Threads verwendet; die Synchronisation kann dadurch rein prioritätsgesteuert erfolgen. SCSJ unterstützt sowohl das *Priority-Inheritance* als auch das *Priority-Ceiling-Emulation-Protocol*, um Prioritäts-Inversion zu vermeiden.

Der von Java propagierte objektorientierte Programmierstil erfordert dynamische Allokation von temporären Objekten; dies wird durch Stack-basiertes *Scoped-Memory* erreicht. Dafür ist keine *Garbage-Collection* erforderlich und Werkzeuge können überprüfen, dass es keine *Dangling-Pointers* in freigegebenen *Scopes* gibt.

5.1 Ausführungsmodell: Mission

Safety-Critical-Applikationen werden als sogenannte *Safelets* (analog zu *Applets* und *Midlets*) ausgeführt. Ein *Safelet* besteht im wesentlichen aus drei Phasen:
Initialisierung: Alle Klassen müssen à priori verfügbar sein und eine zyklusfreie Initialisierungsreihenfolge muss explizit angegeben werden. *Threads* werden vor der eigentlichen Mission-Phase reserviert und initialisiert. Während der gesamten Initialisierungsphase wird kein *Heap-Memory* verwendet. Die verwendeten Datenstrukturen liegen im sogenannten *Mission-Memory*, dies ist eine Unterklasse von *Scoped-Memory*.
Mission-Phase: Ausführung der Run-Routine, die typischerweise den zyklische Aufruf einer oder mehrerer Methoden beinhaltet. Es können prioritätsbehaftete asynchrone *Event-Handler* verwendet werden. Für dynamische Speicheranforderungen steht nur *Scoped-Memory* zur Verfügung. Nach der Beendigung der *Mission-Phase* folgt die ***Cleanup-Phase***, die durch verschiedene Mechanismen erreicht werden kann. Die Mission kann neu gestartet, oder eine weitere Mission zur Ausführen vorgesehen werden. Das *Mission-Memory* wird dafür neu initialisiert aber nicht physisch freigegeben.

5.2 Konformitätsebenen

Um den widerstrebenden Interessen nach einem möglichst komfortablen Sprachumfang einerseits und dem Wunsch nach einfacher Analysierbarkeit andererseits gerecht zu werden, definiert SCSJ drei Konformitätsebenen. Der Hauptunterschied besteht darin, inwieweit geschachtelte Missionen möglich sind, aber auch der Sprachumfang wird auf jeder Ebene erweitert. Code von niedrigen Konformitätsebenen kann auch in komplexeren Missionen auf höherer Konformitätsebene verwendet werden. Die genauen Details der Abgrenzung sind zur Zeit noch in der Diskussion.

6 Zusammenfassung

Der Sprachentwurf von Java zielt darauf ab, möglichst viele Fehler bereits zur Übersetzungszeit zu finden; auch die Ausnahmebehandlung ist integraler Bestandteil der Sprache, ebenso wie *Threads* und deren Synchronisation. Java unterstützt Objektorientierung, Kapselung und *Information-Hiding*, zudem bietet es eine große Menge vordefinierter *Packages* in einer weltweit standardisierten Klassen-Hierarchie an. Durch sichere Objektreferenzen kann es keinen *General-Protection-Fault* oder *Segmentation-Fault* geben; auch Feldgrenzen werden überwacht; wenn möglich, schon zur Übersetzungszeit.

Alle Probleme, die den Einsatz von Java für Echtzeit- und sicherheitskritische Systeme verhindert haben, werden in der SCSJ ausgeräumt: Die Anwendung hat volle Kontrolle über das *Scheduling* und kann auf spezielle Speicherbereiche zugreifen; es werden hochauflösende Timer unterstützt und *Worst-Case-Execution-Times* sind berechenbar. Das Potential zum Nachweis von Korrektheits-Eigenschaften durch Werkzeuge ist durch eine präzise Definition der Semantik wesentlich höher als bei C oder C++. Sobald die *Virtuellen Maschinen* die JSR-282 (RTSJ) und JSR-302 (SCSJ) unterstützen von den Zertifizierungsbehörden anerkannt sind, bringt der Einsatz von Java eine hohe Produktivität, da - neben den bekannten Vorteilen - die geforderte Qualität leichter zu erreichen und nachzuweisen ist.

Literaturverzeichnis

[Bak06] Baker, J. et. al.: A Real-time Java Virtual Machine for Avionics - An Experience Report. In 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), IEEE Computer Society, 2006, S. 384-396

[Cor04] Corsaro, A.: Techniques and patterns for safe and efficient real-time middleware. PhD Washington University. http://portal.acm.org/citation.cfm?id=1087703&dl=ACM&coll=

[Do178] RTCA: DO 178B, Software Considerations in Airborne Systems and Equipment Certification. http://en.wikipedia.org/wiki/DO-178B, 1992

[HIJA] aicas GmbH: HIJA Safety Critical Java Proposal. http://www.hija.info, 2005

[JSR282] JSR 282: RTSJ version 1.1. http://www.jcp.org/en/jsr/detail?id=282, 2008

[JSR302] JSR 302: Safety Critical Java Technology. http://www.jcp.org/en/jsr/detail?id=302, 2008

[RTSJ] Bollella G. et. al.: The Real-Time Specification for Java. Addison-Wesley, 2000

[Sto07] Mark Stoodley et. al.: Real-Time Java, Part 1: Using the Java language for real-time systems. http://www-128.ibm.com/developerworks/java/library/j-rtj1/, 2007