Einsatz von Java-Komponenten in verteilten Embedded Systems

Uwe Rastofer[1,2], Ulrich Gall[1,2], Frank Schinkmann[1,2],
Bernd Hindel[2], Jürgen Kleinöder[1]

[1] Universität Erlangen-Nürnberg
Lehrstuhl für Betriebssysteme IMMD-IV
Martensstr. 1, D-91058 Erlangen
{rastofer,gall,schinkmann,kleinoeder}@informatik.uni-erlangen.de
[2] 3SOFT GmbH
Wetterkreuz 19a, D-91058 Erlangen
{rastofer,gall,schinkmann,hindel}@3SOFT.de

Zusammenfassung Die zunehmende Vernetzung von Rechenanlagen dringt immer weiter auch in den Bereich von Embedded Systems vor. Um eine Interaktion zwischen verschiedenen vernetzten Geräten zu ermöglichen, sind allerdings klare Softwareschnittstellen erforderlich. Die vorliegende Arbeit beschreibt, wie Softwarekomponentenarchitekturen sowohl für die Interaktion zwischen Geräten als auch für die Erstellung von Benutzeroberflächen zum Bedienen und Beobachten aus der Ferne verwendet werden können. Die Spezifikation der Interaktionssemantik von Komponenten sowie ihre Anordnung in Benutzeroberflächen erfolgt dabei visuell. Es werden zwei Ansätze für die Verwendung des Komponentenmodells JavaBeans zur Gerätesteuerung vorgestellt. Beim Stellvertreterbasierten Modell werden Geräte lokal durch in Java implementierte Komponenten repräsentiert, die über ein beliebiges Protokoll mit der proprietären Software im Gerät kommunizieren. Im verteilten Modell findet die Interaktion der Komponenten in einer homogenen Java-Umgebung über Java RMI statt.

1 Einführung

Die Anforderungen an Embedded Systems haben sich in letzter Zeit grundlegend verändert. Zu den klassischen Eigenschaften gewinnen zwei neue Aspekte an Bedeutung. Zum einen betrifft dies das Bedienen und Beobachten der Embedded Systems. Hier werden zunehmend komfortable graphische Benutzeroberflächen eingesetzt, bei deren Realisierung der Trend weg von aufwendigen proprietären Lösungen und hin zum Einsatz von Standards wie HTML, HTTP [1] und auch Java [2] geht. Zum anderen arbeiten Embedded Systems immer seltener vollkommen isoliert, sondern tauschen Informationen mit anderen Systemen aus. Ein typisches Beispiel hierfür sind Geräte aus der Unterhaltungselektronik, wo sich Fernsehgerät, Videorecorder und Satellitenreceiver aufeinander abstimmen

und gegenseitig steuern lassen. Die dort verwendeten Techniken sind jedoch proprietär und funktionieren deshalb meist nur zwischen Geräten des gleichen Herstellers.

In beiden oben beschriebenen Fällen muß das Embedded System mit anderen Bedien- oder Steuerungssystemen Daten austauschen – es ist also Teil eines größeren, verteilten Systems. Im Forschungsprojekt "Objektorientierte Verteilte Steuerungssysteme (OVEST)" [1] untersucht, wie sich Softwarekomponenten bei der Entwicklung von verteilten Embedded Systems einsetzen lassen.

Ziele sind hierbei die Senkung des Entwicklungsaufwandes, Gewährleistung von Interoperabilität zwischen Geräten verschiedenen Typs und verschiedener Hersteller, sowie eine einfache Anpassung an anwenderspezifische Umgebungen.

Im folgenden wird zunächst gezeigt, wie sich Embedded Systems als Softwarekomponenten modellieren lassen. Dabei wird besonders auf die Möglichkeiten des Komponentenmodells JavaBeans [3] eingegangen. Anschließend werden zwei verteilte Komponentenarchitekturen vorgestellt, die auf dem JavaBeans-Modell aufbauen.

2 Embedded Systems als Komponenten

Eine Softwarekomponente wird meist sehr allgemein als "... ein Stück Software. Groß genug um funktional einsetzbar zu sein, aber klein genug, um noch wartbar zu sein..." [4] definiert. Dies sagt jedoch noch nichts darüber aus, wie eine Komponenten verwendet werden kann, weshalb in [5] folgende, konkretere Definition gewählt wurde:

> *Eine Softwarekomponente ist eine funktional abgeschlossene Einheit, die Dienste nach außen anbietet. Sie ist selbstbeschreibend, so daß sie durch entsprechende Programme automatisch analysiert werden kann. Eine Softwarekomponente kann mit anderen Komponenten innerhalb einer bestimmten Infrastruktur, die durch das Komponentenmodell definiert wird, zu komplexeren Anwendungen komponiert werden.*

Komponenten müssen den Spezifikationen des gleichen Komponentenmodells entsprechen, um miteinander interagieren zu können. Darüber hinaus legt das Komponentenmodell fest, in welcher Form Komponenten die Informationen zur Verfügung stellen können, die von Programmen (Builder-Tools) zur Parametrierung und Verbindung der Komponenten benötigt werden. Das wichtigste Komponentenmodell für Java ist derzeit JavaBeans [3].

Die grundlegende Idee ist nun, ein Embedded System als Softwarekomponente so zu modellieren, daß die Funktionalität des Embedded Systems als Dienste der Softwarekomponente anderen Komponenten zur Verfügung gestellt wird. Dabei kann das Embedded System als Ganzes in einer einzigen Komponente realisiert sein oder es können funktionale Teile, aus denen das Embedded System aufgebaut ist, als einzelne Komponenten modelliert werden.

[1] OVEST wird von der Universität Erlangen-Nürnberg und Industriepartnern getragen, sowie von der Bayerischen Forschungsstiftung gefördert.

Die Dienste der Embedded-System-Komponente können in einem bestimmten Komponentenmodell von beliebigen anderen Komponenten genutzt werden. Beim Aufbau von Bedienoberflächen geschieht dies durch die Verbindung der Embedded-System-Komponente mit graphischen Anzeige- und Bedienkomponenten. Die Steuerung einer Embedded-System-Komponente durch eine andere kann in gleicher Weise durch die Verbindung der beiden Komponenten erfolgen.

Zur Realisierung der eben beschriebenen Ideen müßte nur jeder Hersteller zu seinem Embedded System statt einer kompletten Softwarelösung nur eine passende Komponente für ein Standardkomponentenmodell ausliefern. Es ist dann sehr einfach möglich, neue, den jeweiligen Bedürfnissen angepaßte Benutzeroberflächen zu erzeugen und beliebige Embedded Systems miteinander zu koppeln.

Embedded Systems als JavaBeans

Für die weiteren Untersuchungen wurde als konkretes Komponentenmodell JavaBeans [3] ausgewählt. Da JavaBeans auf dem Java-System aufbaut, haben JavaBeans-Komponenten (kurz Beans) eine Reihe von Eigenschaften die ihre Verwendung in heterogenen, verteilten Systemen begünstigen. So sind Beans durch den Java-Bytecode plattformunabhängig. Außerdem können sie während der Laufzeit dynamisch geladen werden und haben sehr einfach Zugriff auf Netzwerkfunktionalität, wie Sockets und JavaRMI.

In der Spezifikation von JavaBeans ist festgelegt, daß jede Bean *Properties* besitzt, die ihre Eigenschaften enthalten. Weiterhin hat sie *Methoden*, die eine Veränderung des internen Zustands auslösen können, und sie kann *Events* erzeugen, um andere Beans von einem Ereignis in Kenntnis zu setzten.

Ein Embedded System hat ebenfalls eine Menge von Eigenschaften und einen internen Zustand, die jeweils durch eine Menge von Werten repräsentiert werden. Dabei kann man zwischen gemessenen und einstellbaren Werten unterscheiden. Bei der Modellierung als Bean werden Meßwerte auf Properties abgebildet, die nur gelesen werden können, während eingestellte Werte sowohl gelesen als auch geschrieben werden können. Zum Lesen und Schreiben einer Property werden zwei Zugriffsmethoden benötigt. Für größere Komponenten ist das einfache Lesen und Schreiben einzelner Properties nicht ausreichend, weshalb man für komplexere Operationen zusätzliche Methoden definieren muß.

Eine Bean erzeugt immer dann ein Event, wenn sich der Zustand des Embedded Systems ändert. Dabei kann ein Event durch sein Auftreten neben dem Zeitpunkt der Änderung auch den neuen Wert übertragen. Die Zustellung eines Events an eine andere Bean erfolgt über Adapter, die bei der Ziel-Bean eine frei wählbare Methode aufrufen. Bevor aber ein Event zugestellt werden kann, muß der Adapter bei der Quell-Bean über eine spezielle Methode registriert werden.

Als einfaches Beispiel ist in Abbildung 1 ein Heizelement als Bean realisiert. Die HeaterBean hat zwei Properties, die eingestellte und die gemessene Temperatur, wobei die Ist-Temperatur nur lesbar ist. Außerdem erzeugt die HeaterBean bei der Veränderung der Soll- oder Ist-Temperatur je ein Event, in dem der neue Wert übermittelt wird. Methoden für komplexe Operationen werden nicht benötigt.

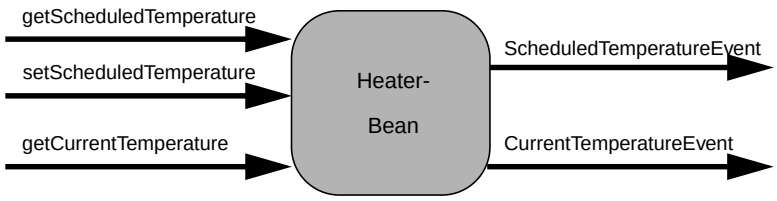

Abb. 1. Ein Heizelement als JavaBean

Für die Verwendung von JavaBeans in verteilten Systemen gibt es jedoch noch ein wichtiges Hindernis. Die Zustellung von Events ist nur an Beans möglich, die in der gleichen Virtuellen Java-Maschine ablaufen. Ein Mechanismus für die verteilte Event-Zustellung an Beans auf anderen Rechner ist nicht vorgesehen. Im folgenden wird deshalb zunächst eine Architektur vorgestellt, die das Problem durch Aufteilung der Komponenten umgeht.

3 Eine Architektur mit Stellvertreterkomponenten

Im JavaBeans-Komponentenmodell können nur Komponenten innerhalb der gleichen Virtuellen Java-Maschine miteinander interagieren. In den eingangs beschriebenen Szenarien wird aber Kommunikation zwischen Komponenten, die über mehrere Rechner verteilt sind, benötigt. Daher wurde in [5] eine Architektur entwickelt, in der die Embedded-System-Komponenten in drei Schichten aufgeteilt werden (Abbildung 2):

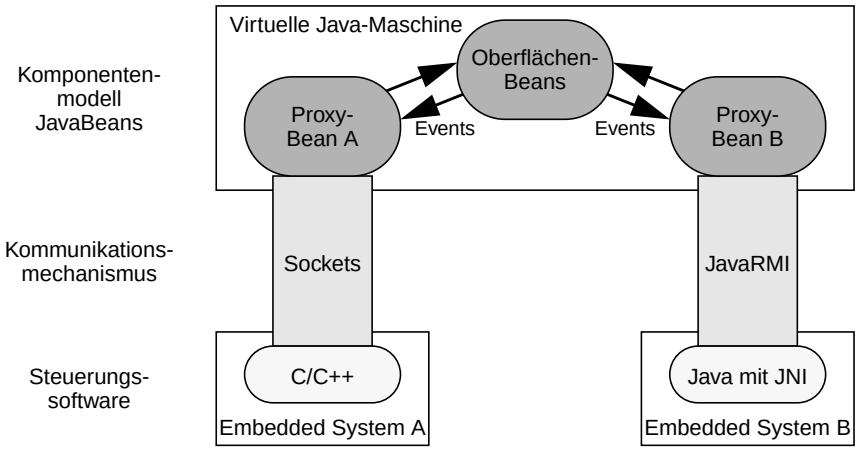

Abb. 2. Eine Architektur mit Stellvertreterkomponenten

1. Die oberste Schicht bildet das Komponentenmodell JavaBeans. Für jedes Embedded System gibt es eine *Stellvertreterkomponente* (*Proxy-Bean*), die zwar Dienste anbietet, sie aber nicht selbst realisiert.
2. Die Proxy-Bean gibt alle Daten an eine *Kommunikationsschicht* weiter, deren Aufgabe es ist, eine Verbindung mit der Steuersoftware auf dem Embedded System herzustellen.
3. Auf dem Embedded System läuft als unterste Schicht die eigentliche *Steuerungssoftware*. Sie reagiert auf die Daten von der Proxy-Bean und schickt Statusinformationen an die Proxy-Bean zurück.

Durch diese Architektur sind auf der Ebene des Komponentenmodells nur die Dienste der Proxy-Beans sichtbar. Die darunterliegenden Schichten werden versteckt. Das hat eine Reihe von Vorteilen:

- Proxy-Beans entsprechen vollständig der JavaBeans-Spezifikation. Sie können daher mit jeder anderen Bean interagieren und in beliebigen Beans-Builder-Tools verwendet werden.
- Es kann eine schon vorhandene Steuerungssoftware wiederverwendet bzw. um einen Kommunikationsmechanismus erweitert werden. Auf dem Embedded System muß keine Virtuelle Java-Maschine vorhanden sein und die Steuerungssoftware muß nicht nach Java portiert werden.
- Es gibt keine Einschränkung bei der Wahl des Kommunikationsmechanismus, da dieser für andere Beans unsichtbar ist. Je nachdem, welche Möglichkeiten auf dem Embedded System vorhanden sind, können beispielsweise Sockets, Java Remote Method Invocation (RMI) [6] oder CORBA [8] verwendet werden.
- Die Virtuelle Java-Maschine, in der alle Beans ihre Events austauschen, kann an einer beliebigen Stelle im verteilten System ablaufen. Bei Benutzeroberflächen für Bedienen und Beobachten aus einem Web-Browser heraus läuft diese virtuelle Maschine im Browser.

Prototypische Implementierung

Zur Evaluierung der beschriebenen Architektur wurde ein Prototyp [5] implementiert, mit dem sich eine Kamera und eine Kaffeemaschine steuern lassen. Als Kommunikationsmechanismus wurde Java RMI [6] gewählt.

In Abbildung 3 ist zu sehen, wie verschiedene Komponenten für graphischen Benutzeroberflächen mit der JavaBeans-fähigen Entwicklungsumgebung *VisualAge* zu einer Bediensoftware zusammengesetzt werden. Die Proxy-Bean für das zu bedienende Gerät (in diesem Fall eine Kamera), im späteren Betrieb normalerweise unsichtbar, ist oben links repräsentiert.

4 Ein verteiltes Komponentenmodell

Während in der gerade beschriebenen Architektur die Verteilung außerhalb des Komponentenmodells realisiert wurde, wird nun eine Erweiterung von JavaBeans

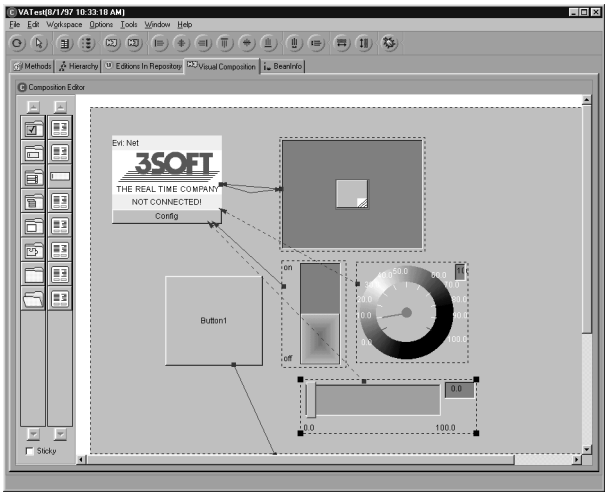

Abb. 3. Aufbau der Bedienoberfläche für eine Kamera

vorgestellt, die Verteilung schon im Komponentenmodell enthält. Die Benutzung dieses verteilten Komponentenmodells ist vor allem dann sinnvoll, wenn sich auf dem Embedded System eine Virtuelle Java-Maschine befindet (und die Steuerungssoftware in Java geschrieben wurde) [7]. Die Kommunikation zwischen den Komponenten kann dann direkt erfolgen, ohne den Umweg über eine zentrale Instanz, in der die Proxy-Beans ablaufen, nehmen zu müssen.

Die wichtigste Entscheidung beim Entwurf des verteilten Komponentenmodells ist die Wahl des Kommunikationsmechanismus. Da in JavaBeans der Datenaustausch über Event-Objekte erfolgt, können objektorientierte Kommunikationsmechanismen sehr einfach verwendet werden. Für JavaBeans kommen daher nur RMI [6] und CORBA [8] in Frage. RMI hat durch die enge Integration in das Java-System einige Vorteile, wie z.B. die Möglichkeit des Verschickens von ganzen Objekten bei einem Methodenaufruf. Dies ist in CORBA nur mit zusätzlichen Mechanismen bzw. zukünftigen Erweiterungen [9] möglich, weshalb für dieses Komponentenmodell RMI gewählt wurde.

Erzeugung RMI-fähiger Beans

Damit man an eine Bean auf einem anderen Rechner Events zustellen kann, muß die Bean Methoden besitzen, die über RMI aufgerufen werden können. Das ist in JavaBeans nicht vorgesehen und kann deshalb nicht bei jeder Bean vorausgesetzt werden. Aus diesem Grund wurde ein Werkzeug (rmiify) entwickelt, das die Methoden einer JavaBean über RMI aufrufbar macht. Das Werkzeug erzeugt automatisch aus einer "normalen" Bean eine RMI-Bean für das verteilte Komponentenmodell.

Die Bean-Factory

In dem verteilten Komponentensystem kann eine Komponente sofort beim Einschalten des zugehörigen Embedded Systems erzeugt und bekannt gemacht werden. Dies ist z.B. für die Komponente sinnvoll, die das Embedded System selbst modelliert.

Andererseits gibt es auch Komponenten, die erst später z.B. von einem Builder-Tool erzeugt werden. Diese Komponenten sollen auf einer frei wählbaren Virtuellen Java-Maschine, die Teil des verteilten Komponentensystems ist, ablaufen. Daher gibt es auf jedem Knoten im Komponentensystem ein Objekt (die Bean-Factory), das eine Bean (genauer jedes beliebige Java-Objekt) lokal erzeugen kann. Ein Builder-Tool kann sich an eine Bean-Factory wenden und die Erzeugung der gewünschten Bean veranlassen. Die Bean-Factory wird außerdem benötigt, um Adapter für die Event-Zustellung zu verteilen.

Als Mechanismus für die Übertragung des Zustandes der Objekte verwendet die Bean-Factory die Object Serialization [10] von Java 1.1.

RMI-Adapter

Nach der Verpackung einer Bean mit dem Werkzeug rmiify ist es möglich, die Methoden der Bean über RMI aufzurufen. Die Bean, die ein Event erzeugt und dadurch den Methodenaufruf anstößt, ist jedoch nicht auf einen RMI-Aufruf vorbereitet. So können bei RMI-Aufrufen zusätzlich RemoteExceptions auftreten, die auch eine RMI-Bean nicht erwartet. Damit die RMI-Bean dies toleriert, müßte der Code der Original-Bean bei der Erzeugung der RMI-Bean modifiziert werden.

Es ist daher Aufgabe des Adapters zwischen den beiden RMI-Beans, den Methodenaufruf über RMI abzuwickeln. Wie in Abbildung 4 dargestellt, wird bei der Bean, die ein Event erzeugt, der RMI-Adapter registriert. Dieser erhält dann über den lokalen Event-Mechanismus einen Methodenaufruf, der über RMI an die Ziel-Bean weitergeleitet wird. Der RMI-Adapter ist außerdem für die Behandlung von Netzwerkfehlern (RemoteExceptions) zuständig.

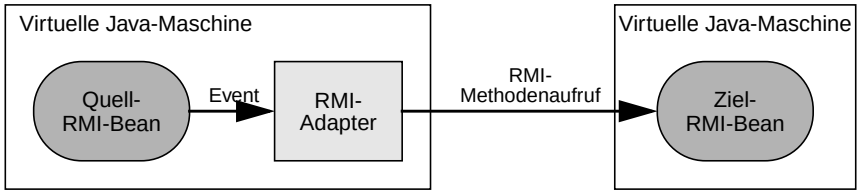

Abb. 4. Event-Zustellung über RMI-Adapter

Da im JavaBeans-Modell die Ziel-Bean bei einer Quell-Bean für ein Event registriert wird, muß im verteilten Modell ein passender RMI-Adapter in der

Virtuellen Java-Maschine, auf der die Quell-RMI-Bean läuft, erzeugt und initialisiert werden. Dies geschieht mit Hilfe der schon beschriebenen Bean-Factory.

Verteilter Namensdienst

Sowohl für die Verwendung in Builder-Tools als auch für das Auffinden von Komponenten untereinander wird ein Namensdienst benötigt. RMI bietet mit der Registry nur einen sehr einfachen Namensdienst an, der für große verteilte Systeme ungeeignet ist. Deshalb wurde ein verteilter, hierarchischer Namensdienst auf Basis von RMI entworfen und implementiert. Der Namensdienst orientiert sich an der Spezifikation des COS Naming Service [11]. Es gibt mehrere Name Server, die jeweils einen lokalen, hierarchischen Namensraum verwalten. Im lokalen Namensraum gibt es Verweise auf Teile des Namensraumes anderer Name Server, wodurch ein verteilter Namensraum entsteht.

Für das verteilte Komponentenmodell wurde ein Namensschema entwickelt, in dem sich Bean-Factories unter dem Namen ihres Host-Rechners anmelden. Namen für Komponenten sind frei wählbar und können beispielsweise nach funktionaler oder geographischer Zusammengehörigkeit gruppiert werden.

Prototyp und Ausblick

Zur Zeit wird ein Prototyp des verteilten Komponentenmodells für Geräte der Unterhaltungselektronik entwickelt. Aufgrund der Erweiterungen, die durch das verteilte Komponentenmodell eingeführt wurden, können für die RMI-Beans jedoch keine JavaBeans-Builder-Tools verwendet werden. Neben den schon beschriebenen Teilen des verteilten Komponentenmodells muß deshalb noch ein Builder-Tool entwickelt werden, das für verschiedene Komponentenmodelle anpaßbar ist.

Ein weiterer Ansatz, das JavaBeans-Modell um Verteilung zu erweitern, ist der Infobus [12]. Dabei tauschen jedoch die Beans ihre Events über einen komplett anderen Mechanismus aus, was einen großen Anpassungsaufwand in existierenden Beans nach sich zieht.

Für heterogene, verteilte Systeme entwickelt die Object Management Group ein verteiltes Komponentenmodell [13], das zu CORBA und JavaBeans kompatibel sein soll. Die Standardisierung ist aber noch nicht abgeschlossen.

5 Zusammenfassung

Java und das Internet erobern zunehmend auch den Bereich der Embedded Systems [14] [15]. Um Interaktion zwischen vernetzten Embedded Systemen zu ermöglichen, sind klar definierte Schnittstellen und Protokolle erforderlich. Zur Zeit dominieren im Bereich der Embedded Systems allerdings noch proprietäre, ad-hoc definierte Protokolle.

Im OVEST-Projekt wird untersucht, wie Komponentenmodelle (z.B. JavaBeans) verwendet werden können, um Kommunikation zwischen Geräten verschiedenen Typs und verschiedener Hersteller zu ermöglichen. Dieser Ansatz

hat einige Vorteile: Zur Implementierung der verteilten Anwendungen können handelsübliche JavaBean-Entwicklungsumgebungen verwendet werden. Dadurch können auch komfortable graphische Benutzeroberflächen visuell gestaltet werden. Dabei kann auf umfangreiche Softwarekomponenten aus dem Internet zurückgegriffen werden.

Es wurden zwei unterschiedliche Realisierungsmöglichkeiten für diesen Ansatz vorgestellt. Beim Stellvertreter-basierten Ansatz werden Geräte durch Proxy-Komponenten repräsentiert. Diese Proxy-Komponenten enthalten die notwendige Software, um das (im allgemeinen) proprietäre Protokoll mit dem Gerät abzuwickeln und hinter einer JavaBeans-konformen Schnittstelle zu verbergen. Dadurch können unmodifizierte JavaBean-Entwicklungsumgebungen, wie sie bereits erhältlich sind, verwendet werden.

Beim verteilten Interaktionsmodell befinden sich die JavaBeans in den einzelnen Geräten und die Interaktion wird über Java RMI realisiert. Dadurch sind keine Proxy-Komponenten erforderlich. Dieses Modell wird zur Zeit an einem praktischen Anwendungsfall mit Geräten aus der Unterhaltungselektronik untersucht.

Literatur

1. Peter Kraus. *Die Homepage eines Embedded Systems – neue Wege für Service Software.* Embedded Intelligence '97, Sindelfingen, 1997.
2. Ulrich Gall, Bernd Hindel. *Bedienen und Beobachten von Embedded Systemen mit Java.* In: Design & Elektronik 4/97, Magnamedia Verlag, 1997.
3. Sun Microsystems. *JavaBeans Specification.* Version 1.01, 1997.
4. Jalal Fehhi. *Web developer's guide to JavaBeans.* Coriolis Group, 1997.
5. Michael Holzheu. *Entwicklung einer internetfähigen Remote Steuerung für Embedded Systeme mit Hilfe von RMI und JavaBeans.* Diplomarbeit DA-I4-97-20, Universität Erlangen-Nürnberg, 1997.
6. Sun Microsystems. *Java Remote Method Invocation Specification.* 1997.
7. Jürgen Kleinöder. *Java – die Embedded Sprache von morgen?* Embedded Intelligence '97, Sindelfingen, 1997.
8. Object Management Group. *The Common Object Request Broker: Architecture and Specification.* Revision 2.2, OMG Document formal/98-02-01, 1998.
9. Object Management Group. *Objects-by-value Request For Proposal.* OMG Document orbos/96-06-14, 1996.
10. Sun Microsystems. *Java Object Serialization Specification.* 1997.
11. Object Management Group. *CORBAservices: Common Object Services Specification.* OMG Document formal/97-12-02, 1997.
12. Lotus Development Corp. *Infobus 1.1 Specification.* 1998.
13. Object Management Group. *CORBA Component Model Request For Proposal.* OMG Document orbos/97-06-12, 1997.
14. Sun Microsystems. *EmbeddedJava Specification.* Public Review Draft 1.0, 1998.
15. Sun Microsystems. *Sun Announces Key Consumer Technology To Bring The Power Of The Java^{TM} Platform To Auto, TV And Phone Markets.* Press Release 980324-04, San Francisco, 1998.