Dynamische Adaption in heterogenen und verteilten eingebetteten Systemen

Der Technischen Fakultät der

Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Meik Felser

Erlangen — 2008

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	06.06.2008
Tag der Promotion:	26.11.2008
Dekan:	Prof. Dr.-Ing. Johannes Huber
Berichterstatter:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr.-Ing. Olaf Spinczyk

Kurzfassung

Eingebettete Systeme erledigen im heutigen Alltag eine Vielzahl von Aufgaben. Die geplante Lebensdauer dieser Systeme ist dabei häufig relativ lang. Daher ist die Wahrscheinlichkeit groß, dass man die Software im Laufe der Zeit verändern möchte. Gründe hierfür kann das Beseitigen von Fehlern sein oder der Wunsch eine Funktionalität hinzuzufügen. In komplexeren Systemen arbeiten häufig mehrere Mikrocontroller in einem Netzwerk kooperierend zusammen. Die beteiligten Mikrocontroller unterscheiden sich dabei nicht nur hinsichtlich ihrer speziellen Sensoren und Peripheriegeräte, sondern auch bezüglich ihrer Leistungsfähigkeit. Die Ressourcen der einzelnen Mikrocontroller sind jedoch in der Regel stark beschränkt, sodass man die Software nachträglich nicht beliebig erweitern kann.

Unter diesen Rahmenbedingungen soll in dieser Arbeit ein System zur dynamischen Adaption der Software auf Mikrocontrollern vorgestellt werden. Dabei sollen größere Steuerknoten, die leistungsstärker sind und über mehr Ressourcen verfügen, Hilfestellung für kleinere Knoten geben. Diesem Ansatz zufolge werden die Mikrocontroller in *verwaltete Knoten* und *Verwalter* eingeteilt. Verwalter bieten dabei Dienste zur Konfiguration und Unterstützung der verwalteten Knoten an.

Eine Voraussetzung zur flexiblen Konfiguration ist, dass die Software in kleinen Einheiten vorliegt. Dazu wird in dieser Arbeit eine Technik vorgestellt, existierende Software in Module aufzuteilen. Der Ansatz arbeitet auf den Objektdateien der Software und ist architekturunabhängig. Darüber hinaus muss die Software nicht speziell vorbereitet werden. Mithilfe der Module kann ein Verwalter die Software für einen verwalteten Knoten wie aus einem Baukasten zusammenbauen.

Die Konfigurationsoperationen umfassen dabei das nachträgliche Installieren, Entfernen und Austauschen von Softwaremodulen. Dabei sind die Operationen zustandserhaltend und können so auch während der Laufzeit durchgeführt werden. Dies ermöglicht das bedarfsgesteuerte Installieren oder das temporäre Entfernen von Modulen (hier als Parken bezeichnet).

Neben Operationen zur Umkonfiguration bieten Verwalter auch die Möglichkeit, Aufgaben im Auftrag der verwalteten Knoten durchzuführen. Hierzu wurde ein schlankes Fernaufrufsystem realisiert, das nur geringe Ressourcenanforderungen an den verwalteten Knoten stellt. Durch Kombination von Umkonfigurationen und Fernaufrufsystem können fast beliebige Funktionen zur Laufzeit von einem verwalteten Knoten auf einen Verwalter verschoben werden.

Um heterogene Hardware zu unterstützen, verfügt das Fernaufrufsystem daher über eine Infrastruktur zur automatischen Anpassung der Datendarstellung zwischen verschiedenen Knoten. Die dafür notwendigen Informationen werden automatisch aus Debuginformationen gewonnen.

Die Realisierbarkeit des Ansatzes wurde durch die prototypische Implementierung eines Verwalters nachgewiesen. Zur Demonstration wird außerdem eine Java-Laufzeitumgebung betrachtet, die eigentlich zu groß ist, um für kleine Knoten geeignet zu sein. Mit dem Verwalter ist es jedoch möglich, die Laufzeitumgebung auf kleinen Knoten zu installieren, indem unbenutzte Funktionen entfernt und Funktionen, welche nur selten verwendet werden am Verwalter ausgeführt werden. Hierdurch können sogar sehr kleine Knoten mit einer vollständigen Java-Umgebung ausgestattet werden.

Abstract

Today, embedded systems are ubiquitous in everyday life. In many cases, these systems are expected to have a long lifespan. For such long-living systems, it is very likely that the software needs to be modified over time. Potential reasons would be the necessity to fix errors or the desire to add new functionality. More complex systems usually consist of several microcontrollers that cooperate over a network to perform the task. In these systems, the involved microcontrollers do not only differ with respect to the attached sensors and peripheral devices, but also in terms of available hardware resources. However, most microcontrollers in such networks are typically very resource constrained. This limits the possibility to subsequently extend their software.

With respect to these conditions, the objective of this thesis is to present a system for the dynamic adaptation of software on microcontrollers. The basic idea is that more powerful nodes, with more available resources and higher performance, should assist smaller nodes. This leads to a classification of microcontrollers as *managed nodes* and *managers*. Managers offer services to configure, assist, and support managed nodes.

A prerequisite for the flexible configuration of nodes is that the software has been partitioned into small units. In this thesis, I present a novel technique to split existing software into modules. The approach is based on the object files of the software; it is architecture independent and does not require any special preparations of the software. The manager can use the resulting modules as a construction kit to assemble the software for the managed nodes.

The manager provides reconfiguration operations to install, remove, and replace software modules. As all operations preserve the state of the software, they can safely be used at run time. This facilitates installation on demand or parking (temporal removal) of modules.

Beside reconfiguration services, managers also provide the service to perform a task on behalf of a managed node. For this purpose, I have implemented a slim remote procedure call system that requires only minimal resources at the managed node. By combining the reconfiguration capability and the remote procedure call system, arbitrary functions can be shifted from a managed node to a manager at run time.

For the support of heterogeneous hardware, the remote procedure call system contains an infrastructure for automatic adjustment of the data representation between different architectures. All necessary information for this service is automatically obtained from the debugging information generated by the compiler.

I evaluate the feasibility of my approach with a prototypical implementation of the manager infrastructure applied to an existing Java Runtime Environment (JRE) that is too big to fit on smaller nodes. With the approach it becomes possible to install the JRE by removing unneeded functionality and placing rarely used functions on the manager. Thereby, even small nodes can be equipped with Java software.

Einige Ansätze und Ideen dieser Arbeit sind auch in den folgenden Publikationen erschienen:

1. SCHRÖDER-PREIKSCHAT, WOLFGANG, RÜDIGER KAPITZA, JÜRGEN KLEINÖDER, MEIK FELSER, KATJA KARMEIER, THOMAS HALVA LABELLA und FALKO DRESSLER: *Robust and Ecient Software Management in Sensor Networks*. In: IEEE Computer Society Press (Herausgeber): *2nd IEEE/ACM International Conference on Communication System Software and Middleware (COMSWARE 2007)*, Januar 2007.

2. FELSER, MEIK, RÜDIGER KAPITZA, JÜRGEN KLEINÖDER und WOLFGANG SCHRÖDER-PREIKSCHAT: *Dynamic Software Update of Resource-Constrained Distributed Embedded Systems*. In: *Embedded System Design: Topics, Techniques and Trends*, Band 231 der Reihe *IFIP International Federation for Information Processing*, Seiten 387–400, Boston, MA, USA, Mai 2007. IFIP, Springer-Verlag

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1
1.1.1 Eingebettete Systeme . . . 1
1.1.2 Netzwerke aus Mikrocontrollern . . . 2
1.1.3 Laufzeitsysteme für Mikrocontroller . . . 3
1.1.4 Anforderungen im Wandel . . . 3
1.1.5 Fernwartungssystem für Mikrocontroller . . . 4
1.1.6 Aktive Hilfestellung durch Verwalter . . . 4
1.1.7 Uniforme Programmier- und Laufzeitumgebung . . . 5
1.2 Zielsetzung der Arbeit . . . 6
1.3 Gliederung der Arbeit . . . 7

2 Grundlagen und Techniken **9**

2.1 Ansätze zur Reduktion des Speicherbedarfs für kleine Knoten . . . 9
2.1.1 Reduktion der Codegröße . . . 9
2.1.2 Codekompression . . . 11
2.1.3 Überlagerungstechnik . . . 11
2.2 Ansätze zur dynamischen Modifikation von Anwendungen . . . 12
2.2.1 Interpretergestützte Ausführung . . . 12
2.2.2 Verändern des Speicherabbildes als Ganzes . . . 13
2.2.3 Austauschen von binären Programmmodulen . . . 13
2.2.4 Vergleich der Verfahren . . . 14
2.2.5 Zustandsanpassung . . . 15
2.3 Ansätze zur Benutzung und Steuerung entfernter Knoten . . . 16
2.3.1 Fernaufruf . . . 16
2.3.2 Verschieben von Anwendungskomponenten . . . 16
2.3.3 Laufzeitsysteme für verteilte Systeme . . . 17
2.3.4 Fernwartungssysteme . . . 17
2.4 Zusammenfassung . . . 18

3 System zur Verwaltung eingebetteter, heterogener, verteilter Knoten **19**

3.1 Verwalter und verwaltete Knoten . . . 19
3.1.1 Permanenter oder temporärer Verwalter . . . 20
3.1.2 Verwalter als Konfigurator . . . 21
3.1.3 Verwalter als Dienstleister . . . 21
3.2 Architektur und Aufgaben eines verwalteten Knotens . . . 23
3.2.1 Software auf einem Mikrocontroller . . . 23
3.2.2 Softwarearchitektur eines verwalteten Knotens . . . 23

3.3 Architektur und Aufgaben eines Verwalters . . . 24
3.3.1 Basisschicht . . . 24
3.3.2 Kontrollschicht . . . 25
3.3.3 Hardwaredatenbank . . . 27
3.4 Zusammenfassung . . . 30

4 Basisschicht eines Verwalters 31
4.1 Aufgaben und Dienste der Basisschicht . . . 31
4.1.1 Installation von Modulen bei Bedarf . . . 32
4.1.2 Parken von Modulen . . . 32
4.1.3 Ersetzen von Modulen . . . 33
4.1.4 Module als entfernten Dienst auslagern . . . 33
4.2 Modularisierung . . . 34
4.2.1 Anforderungen an Module . . . 35
4.2.2 Komponentenmodelle . . . 36
4.2.3 Modularisierung existierender Anwendungen . . . 38
4.2.4 Binärdateien . . . 42
4.2.5 Erstellung der Module . . . 48
4.2.6 Debuginformationen . . . 56
4.2.7 Modularten . . . 60
4.3 Basismechanismen zur Modulverwaltung . . . 61
4.3.1 Installieren von Modulen . . . 61
4.3.2 Entfernen von Modulen . . . 62
4.3.3 Dynamisches Austauschen von Modulen . . . 63
4.4 Verweise auf Module . . . 64
4.4.1 Klassifikation von Verweisen . . . 64
4.4.2 Identifikation von Verweisen . . . 65
4.4.3 Abhängigkeitsgraph . . . 72
4.4.4 Ergebnisse . . . 73
4.5 Aufbau und Architektur . . . 74
4.5.1 Module . . . 74
4.5.2 Verarbeiten von ELF-Dateien und Erzeugen von Modulen . . . 76
4.5.3 Modulverwaltung . . . 79
4.5.4 Konfigurations- und Abbildverwaltung . . . 79
4.5.5 Binder . . . 81
4.5.6 Interaktion mit verwalteten Knoten . . . 81
4.5.7 Fernwartungsmodul (RCM) . . . 85
4.5.8 Schnittstellenbeschreibung und Typinformationen . . . 88
4.6 Operationen zur dynamischen Konfiguration eines Knotens . . . 91
4.6.1 Bedarfsgesteuertes Installieren von Modulen . . . 91
4.6.2 Parken von Modulen . . . 91
4.6.3 Ersetzen und Aktualisieren von Modulen . . . 92
4.6.4 Externe Dienste mittels Fernzugriff . . . 94
4.7 Zusammenfassung . . . 96

5 Kontroll- und Verwaltungsschicht 97
5.1 Konfigurationsfindung . . . 97

5.1.1 Kosten . . . 97
5.1.2 Metriken zur Abschätzung der kostenbeeinflussenden Faktoren . . . 100
5.1.3 Kosten – Nutzen Abwägung . . . 101
5.2 Arten von Strategien . . . 102
5.2.1 Manuelle, interaktive Konfiguration . . . 102
5.2.2 Automatische, anwendungsbezogene Konfiguration . . . 103
5.3 Kontroll- und Verwaltungsschicht für eine Java-Laufzeitumgebung . . . 104
5.3.1 Motivation . . . 104
5.3.2 Java-Laufzeitumgebungen für eingebettete Systeme . . . 104
5.3.3 Dynamisch anwendungsspezifisch angepasste Laufzeitumgebung . . . 106
5.4 Eine JVM als Baukasten . . . 107
5.4.1 Ausführungseinheit . . . 107
5.4.2 Speicherverwaltung . . . 108
5.4.3 Objektverwaltung . . . 109
5.4.4 Threadverwaltung . . . 114
5.4.5 Koordinierungsmechanismen . . . 116
5.4.6 Ausnahmeverarbeitung . . . 117
5.4.7 Codevorbereitung . . . 118
5.4.8 Klassenverwaltung . . . 121
5.4.9 Laufzeittypsystem . . . 122
5.4.10 Klassendaten . . . 123
5.4.11 Ergebnisse . . . 125
5.5 Anwendungsbeispiel KESO . . . 126
5.6 Zusammenfassung . . . 128

6 Zusammenfassung und Ausblick **129**
6.1 Zusammenfassung . . . 129
6.2 Erreichte Ziele . . . 129
6.3 Wissenschaftlicher Beitrag . . . 130
6.4 Weiterführende Arbeiten . . . 131
6.5 Abschließende Bemerkung . . . 132

Literaturverzeichnis **133**

Abbildungsverzeichnis

3.1 Architektur eines verwalteten Knotens 24
3.2 Architektur eines Verwalters 24

4.1 Ablauf beim Erstellen eines Programms 42
4.2 Ablauf beim Binden eines Programms 43
4.3 Daten in einer ELF-Datei und deren Zusammenhang 45
4.4 Schematischer Aufbau einer ELF-Objektdatei 49
4.5 Erstellung von Modulen aus Text- und Datensektion 50
4.6 Teilen der Textsektion in zwei Module 51
4.7 Ein Symbol als Ziel mehrerer Relokationen 53
4.8 Sprungziel direkt als relativer Sprung innerhalb der Sektion codiert 54
4.9 Eigene Sektionen für jede Funktion 56
4.10 Vorgehen bei der Installation von Modulen 62
4.11 Vorgehen beim Austausch eines Moduls 64
4.12 Verändern einer aktiven Funktion 69
4.13 Verändern einer Funktion auf dem Aufrufpfad 71
4.14 Abhängigkeitsgraph eines einfachen Beispiels 73
4.15 Architekturübersicht der Basisschicht 75
4.16 Einzel- oder Multisymbolmodul? 78
4.17 Übergänge zwischen den Modulzuständen 80
4.18 Klassendiagramm des Kommunikationssystems 82
4.19 Ablauf bei Aktivierung des Stellvertreters 84
4.20 Klassendiagramm der Stellvertreter Infrastruktur 85
4.21 Abarbeitung der Befehle im Rahmen einer Unterstützungsanforderung 87
4.22 Klassendiagramm des DWARF-Typsystems 89
4.23 Entfernen von abhängigen Modulen beim Parken 92
4.24 Ablauf einer Ausführung am Verwalter 95

5.1 Benutzeroberfläche einer interaktiven Kontrollschicht 102
5.2 Grundlegender Ablauf einer automatischen Konfiguration 103
5.3 Bausteine einer JVM 107
5.4 Abhängigkeiten der Ausführungseinheit 108
5.5 Abhängigkeiten der Speicherverwaltung 109
5.6 Abhängigkeiten der Objektlayoutverwaltung 110
5.7 Abhängigkeiten der Objekterzeugung 111
5.8 Abhängigkeiten der Speicherbereinigung 112
5.9 Abhängigkeiten der Threadsteuerung 114
5.10 Abhängigkeiten des Schedulers 115

5.11 Abhängigkeiten des Dispatchers 116
5.12 Abhängigkeiten der Koordinierungsmechanismen 116
5.13 Abhängigkeiten der Ausnahmeverarbeitung 117
5.14 Abhängigkeiten des Klassenladers 118
5.15 Abhängigkeiten des Verifiers 119
5.16 Abhängigkeiten des Binders 120
5.17 Abhängigkeiten des Compilers 120
5.18 Abhängigkeiten der Klassenlayoutverwaltung 121
5.19 Abhängigkeiten der Typprüfung 122
5.20 Einfaches Optimierungsbeispiel bei globaler Sicht 122
5.21 Abhängigkeiten des Laufzeittypsystems 123
5.22 Abhängigkeiten der Klassendaten 123

1 Einleitung

1.1 Motivation

1.1.1 Eingebettete Systeme

Betrachtet man den Markt der programmierbaren Systeme, so stellt man fest, dass die Gruppe der eingebetteten Systeme einen nicht zu vernachlässigenden Teil ausmacht. Unter einem *eingebetteten System* versteht man einen programmierbaren Kleinstcomputer, der in ein technisches Gerät integriert ist, um das Gerät zu steuern. Beispiele für eingebettete Systeme finden sich in Geräten der Unterhaltungselektronik, wie MP3-Player, aber auch bei Haushaltsgeräten, wie Waschmaschinen, oder in der Motorsteuerung von Kraftfahrzeugen. Nicht selten wird die Funktion des Gerätes maßgeblich durch den eingebetteten Computer realisiert. Trotzdem nimmt man ihn dabei im Allgemeinen nicht als eigenständige Komponente wahr, sondern empfindet das Ganze als Einheit mit dem umgebenden Gerät.

Als Minicomputer werden heute sogenannte Mikrocontroller eingesetzt. Ein Mikrocontroller vereinigt die Komponenten eines Computers (CPU, Speicher, Bus, ...) mit einer Auswahl an grundlegenden Peripheriegeräten (A/D-Wandler, serielle Schnittstellen, Zeitgeber, ...) auf einem einzigen Chip. Mikrocontroller sind daher klein und kompakt und zeichnen sich außerdem durch ein relativ gutes Preis/Leistungsverhältnis aus.

Gleichzeitig sind jedoch auch die Ressourcen, die sie einem Programm zur Verfügung stellen, stark begrenzt. Um den Preis und die Größe gering zu halten, werden Mikrocontroller nicht aus Prozessoren der neusten Generation aufgebaut, sondern enthalten meist günstigere Modelle einer älteren Generation. So sind 8- und 16-bit-Mikrocontroller heute noch deutlich häufiger vorzufinden als 32-bit-Mikrocontroller [Tur02]. Auch die Speicherausstattung ist nicht mit modernen Bürocomputern zu vergleichen. Mikrocontroller haben meist nur wenige Kilobyte Speicher. In der 8-bit-AVR-Mikrocontrollerserie von Atmel gibt es beispielsweise relativ große Varianten [Atm07a] mit 256 KByte Programmspeicher und 8 KByte Datenspeicher (ATmega2560), aber auch sehr kleine Varianten mit nur 1 KByte Programmspeicher und 32 Byte Datenspeicher (ATtiny13) oder sogar Varianten ganz ohne Speicher für Daten (ATtiny11).

1.1.2 Netzwerke aus Mikrocontrollern

Einsatzbereiche von Mikrocontrollern sind vielfältig, im Allgemeinen sind es jedoch Steuer- und Überwachungsaufgaben. Für größere Szenarios, wenn die Leistung eines einzelnen Mikrocontrollers nicht mehr ausreicht, kann ein Netzwerk aus mehreren Mikrocontrollern verwendet werden, um die Aufgabe zu erfüllen. Dabei können die Mikrocontroller koordiniert werden und unterschiedliche Teilaufgaben übernehmen oder selbst organisiert und autonom zur Lösung der Aufgabe beitragen.

Beispiel Maschinensteuerung

Ein Beispiel ist die Steuerung größerer und komplexerer Maschinen. Hier werden mehrere Mikrocontroller eingesetzt, die jeweils eine Teilaufgabe übernehmen. Die besondere Stärke von Mikrocontrollern liegt in der geringen Größe und der großen Robustheit. Daher können sie direkt vor Ort eingesetzt werden, um Sensoren auszulesen und Aktoren zu steuern. Als intelligenter Sensor kann ein Mikrocontroller dabei eine Reihe von Vorverarbeitungsschritten übernehmen, wie zum Beispiel die Digitalisierung oder Filterung der Sensorwerte oder die Verknüpfung von Signalen mehrerer Sensoren. Aber auch einfache Steueraufgaben kann ein Mikrocontroller autonom durchführen, indem die gewonnenen Sensordaten in einer lokalen Entscheidung direkt in die Steuerung von Aktoren umgesetzt werden. Ein Beispiel ist die Abschaltung eines Heizelements bei Überschreitung einer Maximaltemperatur.

Um eine Maschine im Ganzen zu steuern, müssen jedoch die einzelnen Mikrocontroller miteinander kommunizieren, da die einzeln erfassten Daten ausgewertet und zueinander in Beziehung gestellt werden müssen. Wenn dafür eine globale Sicht des gesamten Systems nötig ist, können Mikrocontroller hierarchisch angeordnet sein. Die Koordination übernimmt dann ein übergeordneter Steuerknoten, der über genügend Ressourcen verfügt, um alle Daten für eine Auswertung speichern zu können und um die nächste Aktion zu bestimmen. Eine Hierarchieebene höher kann man sich zusätzlich noch Konfigurationsknoten vorstellen, welche einem Administrator den Zugang zum System ermöglichen. Sie extrahieren Informationen über den Zustand des Systems, um sie zum Beispiel grafisch darzustellen und erlauben dem Administrator das System manuell zu beeinflussen.

Beispiel Sensornetzwerk

Ein anderes Szenario, bei dem mehrere Mikrocontroller benötigt werden, ist ein Sensornetzwerk. Sensornetzwerke bestehen aus kleinen, stark ressourcenbeschränkten Knoten. Jeder Knoten ist durch einen Mikrocontroller realisiert, der mit Sensoren ausgestattet ist. Darüber hinaus ist jeder Knoten mit einer Kommunikationsmöglichkeit versehen, um Informationen mit anderen Knoten auszutauschen und die Sensordaten weiterzuleiten. Der hauptsächliche Einsatzzweck eines solchen Sensornetzwerks ist die kontinuierliche Sammlung und Verarbeitung von Daten von einem beobachteten Objekt. Anwendung finden solche Netzwerke in der Forschung, hier vor allem in der Biologie [CEE+01, CES04, JOW+02], aber auch bei experimenteller Verkehrsüberwachung [RMM+02] oder zur kontinuierlichen Kontrolle der Bodenqualität in der Landwirtschaft [NRC01].

Neben den Sensorknoten enthält das Netzwerk auch noch eine sogenannte Basisstation. Dabei handelt es sich um einen größeren Knoten, der die gesammelten Daten von den einzelnen Sensorknoten entgegennimmt und zur Auswertung weiterleitet oder speichert. Die Sensorknoten agieren weitestgehend selbstständig, mit dem Ziel die gesammelten Daten an eine Basisstation zu übermitteln. Die Knoten sind dabei nicht nur Datenquelle, sondern können die Daten anderer Knoten auch weiterleiten, wenn nicht alle Knoten direkten Kontakt zur Basisstation haben.

1.1.3 Laufzeitsysteme für Mikrocontroller

Da die Ressourcen in einem eingebetteten System immer sehr knapp bemessen sind, ist die effiziente Ressourcennutzung ein besonderes Ziel bei der Softwareentwicklung für solche Systeme. Mehr als bei allgemeinen Computersystemen muss hier der Ressourcenverbrauch von jeder zusätzlichen Funktion bewertet und abgewogen werden. Daher ist die Software für eingebettete Systeme immer Spezialsoftware, die genau für die vorhergesehene Aufgabe ausgelegt ist.

Bei der Entwicklung von Anwendungen für normale PCs steht ein Betriebssystem zur Verfügung, welches die Verwaltung der Ressourcen regelt und den Zugriff darauf abstrahiert. Bei der Softwareentwicklung für eingebettete Systeme ist die Trennung zwischen Betriebssystem und Anwendungsprogramm weniger stark ausgeprägt. Dies liegt hauptsächlich daran, dass durch die Aufweichung der Trennung Ressourcen eingespart werden können, aber auch weil die Betriebssystemschicht häufig im Rahmen der Anwendung mitentwickelt wird. Ein Vorteil dabei ist, dass eine Betriebssystemschicht entsteht, die genau auf die Bedürfnisse der Anwendung zugeschnitten ist und so nur Dienste anbietet, die auch wirklich benötigt werden.

In manchen Bereichen werden auch generische Betriebssysteme eingesetzt. Ein Beispiel hierfür ist OSEK [OSE05], eine Betriebssystemspezifikation für Mikrocontroller im Automobilbereich. Sie definiert eine Vielzahl von Konzepten, Diensten und Abstraktionen wie zum Beispiel ein Aktivitätsträgerkonzept inklusive Scheduler oder ein Konzept zur Koordinierung der einzelnen Aktivitätsträger. Um die Laufzeitumgebung auf die Anwendung anzupassen, werden im Rahmen der Spezifikation bereits verschiedene Konfigurationen definiert. Eine OSEK-Implementierung kann darüber hinaus noch feiner konfigurierbar sein und wird vor dem Einsatz (in einem Konfigurationsprozess) auf die Anwendung zugeschnitten. Durch eine Vielzahl von Auswahlmöglichkeiten wird festgelegt, welche Dienste und Funktionen für die endgültige Software angeboten werden. Hierdurch kommt man ebenfalls zu einer ressourcensparenden angepassten Systemschicht.

Zusammenfassend ist festzuhalten, dass Betriebssysteme oder allgemeiner Laufzeitsysteme für eingebettete Systeme meistens genau auf die Bedürfnisse der Anwendung abgestimmt sind, da zu viel Funktionalität nur zusätzliche Ressourcen benötigt.

1.1.4 Anforderungen im Wandel

Durch eine exakte Anpassung der Systemschicht an die Anwendung können die vorhandenen Ressourcen zwar optimal ausgenutzt werden, allerdings nimmt dadurch die Flexibilität gegenüber dynamischen Veränderungen des Systems oder der zu erfüllenden Aufgabe ab. Die Wahrscheinlichkeit, dass sich die Aufgabe und die gewünschten Funktionen eines Knotens im Laufe des Einsatzes verändern, steigt mit zunehmender Einsatzdauer.

Beispiele hierfür finden sich im Bereich der Sensornetzwerke. Obwohl zu Beginn des Einsatzes eines Sensornetzes zur Datenerfassung festgelegt wird, welche Daten man erfassen möchte, wird in Berichten von realen Einsatzszenarios [SPMC04, JOW+02] immer wieder festgestellt, dass die Anwendung, welche die Messungen durchführt, nach kurzer Zeit verändert werden musste, um besser verwertbare Daten zu erhalten. Dies ist zum einen darauf zurückzuführen, dass es sich um Forschungsarbeiten und Spezialanwendungen handelte, sodass kaum Erfahrungswerte vorliegen, aber auch darauf, dass erst nachdem die ersten Messwerte ausgewertet wurden, festgestellt werden kann, ob durch das gewählte Verfahren überhaupt die gewünschten Daten erfasst werden.

Ein ähnliches Beispiel ist einer Pressemeldung der NASA, über Softwareupdates der Marsrover *Spirit* und *Opportunity*, zu entnehmen [MER06]. Die Marsrover übermitteln Bildmaterial zur Erde. Da die Übertragungskapazität jedoch begrenzt ist, können nicht alle aufgenommenen Bilder übertragen

werden. Im Laufe der Mission wurde daher die Software verändert, sodass eine spezifische Vorfilterung der Bilder stattfindet, um möglichst nur solche Bilder zu übertragen, die relevanten Inhalt haben. Je nach Fragestellung änderte sich hier die Aufgabe während der Mission.

Im Bereich einer Maschinensteuerung werden sich die Aufgaben weniger dynamisch ändern. Aber auch hier kann es notwendig sein, das System neuen Aufgaben anzupassen. Soll beispielsweise ein zusätzlicher oder neuer Sensor anderen Typs hinzugefügt werden, um den Ablaufprozess zu verbessern, so muss dieser von einem Mikrocontroller bedient und die Daten ausgewertet werden.

1.1.5 Fernwartungssystem für Mikrocontroller

Es ist somit ein Mechanismus notwendig, um die Software von Mikrocontrollern nach der Inbetriebnahme zu verändern. Einige Systeme haben einen Updatemechanismus, um neue Softwareversionen einzuspielen und so Fehler in der Software beseitigen zu können. Dazu wird aber meistens ein lokaler Zugriff auf das System vorausgesetzt. Die vorher genannten Beispiele zeigen jedoch, dass ein Updatemechanismus aus der Entfernung von Vorteil ist. Eingebettete Systeme sind, wie der Name schon andeutet, oft an schwer zugänglichen Orten im Einsatz; das macht es jedoch schwierig nachträgliche Veränderungen vorzunehmen. Die Knoten eines Sensornetzwerks können über eine große Fläche verteilt oder an unzugänglichen Orten [CES04] angebracht sein. Es ist somit nicht praktikabel, jeden Knoten aufsuchen zu müssen, um neue Software einspielen zu können.

Im Beispiel der Marsrover wurde ein Fernupdatemechanismus eingesetzt. Die Marsrover sind jedoch mit relativ vielen Ressourcen ausgestattet (25 MHz PowerPC mit 128 MByte RAM) und somit schon im Leistungsbereich von modernen Taschencomputern anzusiedeln und nicht im Bereich kleinster Mikrocontroller.

Das Ziel muss es daher sein, ein Werkzeug zur dynamischen Konfiguration und Wartung von Mikrocontrollern zu erstellen, welches nur sehr geringe Anforderungen an einen Knoten stellt, um ihn nicht zusätzlich zu belasten.

Bei der Konfiguration und Wartung aus der Entfernung sind verschiedene Knoten beteiligt, die nach ihrer Rolle in zwei Gruppen eingeteilt werden: die Knoten, die verändert werden sollen und die Knoten, welche die Veränderung beauftragen und steuern. Im Gegensatz zu dem ähnlich strukturierten Fernupdatesystem DIADEM (siehe Abschnitt 2.3.4) werden im Rahmen dieser Arbeit die Knoten als *verwaltete Knoten* (*managed nodes*) und *Verwalter* (*managing nodes* oder einfach *manager*) bezeichnet, um die Aufgabenverteilung stärker hervorzuheben. Die angebotenen Updatemechanismen müssen so ausgelegt sein, dass sie die Ressourcen des Verwalters nutzen, um die des verwalteten Knotens zu schonen. Die Verwalter sollen somit möglichst viele Aufgaben übernehmen und den Prozess weitestgehend für die verwalteten Knoten vorbereiten, sodass nur unbedingt notwendige Schritte durch die verwalteten Knoten selbst durchgeführt werden müssen. Da in verteilten eingebetteten Systemen oft schon Knoten verschiedener Leistungsklassen vorhanden sind, kann die Aufgabe des Verwalters von einem leistungsstarken Steuerknoten oder bei Sensornetzwerken durch die Basisstation übernommen werden.

1.1.6 Aktive Hilfestellung durch Verwalter

Durch die Kopplung von kleinen, verwalteten Knoten mit größeren Verwalterknoten entsteht eine Struktur, die sich nicht nur für extern angestoßene Updates eignet. Wenn ein verwalteter Knoten davon ausgehen kann, dass ihm ein Verwalter zur Seite steht, so können Updates oder Umkonfigurationen auch dynamisch von einem kleinen Knoten angestoßen werden. Darüber hinaus können ressourcenintensive Funktionen komplett von dem kleinen Knoten auf den Verwalter ausgelagert werden.

Die Einsatzmöglichkeiten eines Mikrocontrollers werden dadurch erweitert. So muss beispielsweise für Anwendungen, die verschiedene Phasen durchlaufen, nicht mehr der gesamte Code auf dem Mikrocontroller Platz finden. Beim Eintritt in eine neue Phase können die Teile der Anwendung, die nicht mehr benötigt werden, durch andere Teile ersetzt werden, welche für die neue Phase benötigt werden. Zum Beispiel kann nach dem Starten des Systems der Code zum Identifizieren und Initialisieren der Peripheriegeräte entfernt und durch Code zur adaptiven Anpassung der Messzyklen ersetzt werden. Sind die optimalen Messzyklen bestimmt, so kann dieser Code wiederum durch Code zur Auswertung, Komprimierung oder Aggregation der Daten ersetzt werden.

Dieses Vorgehen ist, von der Wirkungsweise, mit der Überlagerungstechnik (siehe Abschnitt 2.1.3) vergleichbar. Der Code wird hier allerdings erst durch den Verwalter in das System eingebracht und belegt vorher keinen Speicherplatz auf dem Gerät. Mit der Unterstützung durch den Verwalter kann die Anwendung somit umfangreicher sein als eine Anwendung, die komplett auf dem Mikrocontroller installiert wird. Als Konsequenz daraus kann man in manchen Szenarios einen kleineren und kostengünstigeren Mikrocontroller einsetzen.

Außer der Umkonfiguration kann der Verwalter auch als Dienstleister auftreten und Funktionen für den verwalteten Knoten übernehmen. So kann man die Auswertung der gesammelten Daten zeitweise vom Verwalter durchführen lassen, wenn die dafür benötigten Ressourcen auf dem Mikrocontroller zum Beispiel für eine erneute Adaption der Messzyklen benötigt werden. Ist die Adaption abgeschlossen, so wird die Auswertung wieder vom Mikrocontroller übernommen. Der Verwalter erlaubt es somit zeitweise Ressourcenengpässe zu überbrücken, indem er einem verwalteten Knoten seine Ressourcen zur Verfügung stellt. Die Auslagerung von Diensten kann natürlich auch dauerhaft erfolgen. Dabei ist jedoch zu beachten, dass bei der Dienstnutzung Kommunikationskosten anfallen, die sich auf die Laufzeit und den Energiebedarf des verwalteten Knotens auswirken.

1.1.7 Uniforme Programmier- und Laufzeitumgebung

Ein Werkzeug, welches es ermöglicht mehr Dienste auf einem Mikrocontroller anzubieten als es die Ressourcen eigentlich erlauben würden, ermöglicht es eine Laufzeitumgebung zu verwenden, die generische Dienste anbietet, auch wenn sie von der Anwendung nicht benötigt werden. Konzeptionell sind die Dienste verfügbar, sie werden jedoch nur dann dynamisch installiert, wenn sie auch tatsächlich genutzt werden. Dieses Vorgehen erlaubt die Verwendung von Laufzeitumgebungen, die für den Einsatz auf diesem Mikrocontroller durch ihren Ressourcenbedarf normalerweise ungeeignet sind.

Durch dieses Vorgehen kann man ein heterogenes Mikrocontrollernetzwerk homogenisieren, indem man auf allen Knoten des Netzwerks eine einheitliche uniforme Laufzeitumgebung einsetzt. Zur Homogenisierung der heterogenen Architekturen können Abstraktionsschichten beispielsweise in Form von virtuellen Maschinen eingesetzt werden. Hierbei wird das Programm für eine einheitlich virtuelle Architektur entwickelt. Zur Ausführung muss eine Ausführungsumgebung, die als *virtuelle Maschine* bezeichnet wird, auf der Zielplattform vorhanden sein, die das Programm umsetzt. Alternativ kann das Programm durch einen speziellen Compiler in ein hardwarespezifisches Programm umgewandelt werden. Zusätzlich ist dann jedoch eine meist umfangreiche Bibliothek nötig, welche die nötigen Funktionen der Laufzeitunterstützung bereitstellt.

Die bekanntesten Vertreter von Laufzeitumgebungen basierend auf virtuellen Maschinen sind Java und die .NET-Umgebung. Der Nachteil dieser Systeme liegt jedoch darin, dass das Laufzeitsystem

relativ groß ist, da sie für den Einsatz auf Bürocomputern entwickelt wurden[1]. Die Größe begründet sich hauptsächlich in den umfangreichen Möglichkeiten und Diensten, die eine solche Umgebung bereitstellt.

Für den Bereich der eingebetteten Systeme wurden spezielle Versionen dieser Laufzeitumgebungen entwickelt, beispielsweise die Java Micro Edition (J2ME) [Whi01] oder das .NET Compact Framework [Nea02] beziehungsweise das .NET Micro Framework [TM07]. Diese Versionen haben zwar einen deutlich geringeren Ressourcenbedarf[2], sind jedoch in ihrer Funktionalität und Flexibilität eingeschränkt und entsprechen somit nicht mehr den ursprünglichen Versionen.

Durch den Einsatz von Verwaltern wird es möglich, den vollständigen Funktionsumfang auch auf kleinen Mikrocontrollern anzubieten. Dadurch kann die Entwicklung von Anwendungen einfacher und effizienter erfolgen, da die Programmierung von kleinen Knoten mit denselben Werkzeugen und unter ähnlichen Bedingungen erfolgen kann wie die Programmierung der großen Knoten. Um sinnvolle Ergebnisse zu erhalten, müssen allerdings trotzdem die Fähigkeiten und Ressourcen der einzelnen Knoten berücksichtigt werden. Denn selbst wenn konzeptionell auf jedem Knoten alle Dienste angeboten werden, so ist zu bedenken, dass die Hilfestellung durch einen Verwalter zusätzliche Kosten verursacht, die sich beispielsweise in einer verlängerten Laufzeit niederschlagen.

1.2 Zielsetzung der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung einer Infrastruktur zur Unterstützung ressourcenschwacher Knoten in einem Netzwerk aus Mikrocontrollern. Ein kleiner Knoten wird dabei durch einen leistungsstärkeren Knoten unterstützt und verwaltet. Eine Aufgabe des Verwalters ist dabei die dynamische Anpassung der Software des verwalteten Knotens an sich ändernde Bedürfnisse. Dadurch könnte die Software auf den kleinen Knoten immer genau auf die jeweilige Situation angepasst werden, ohne den Ressourcenverbrauch pauschal zu erhöhen.

Zusätzlich soll der Verwalterknoten seine Ressourcen dem verwalteten Knoten in Form von Diensten zur Verfügung stellen können. Abstrakt betrachtet können so die Fähigkeiten des kleinen, verwalteten Knotens, über seine eigentliche Leistungsfähigkeit hinaus erweitert werden.

Zur Realisierung dieser Ziele sollen zwei Mechanismen realisiert werden:

- Ein Fernwartungssystem, womit Programmteile in einem kleinen Knoten ausgetauscht und verändert werden können und
- ein Fernaufrufsystem, um die Ressourcen des kleinen Knotens virtuell zu erweitern.

Diese, von der Infrastruktur angebotenen, Unterstützungsmöglichkeiten müssen dabei universell einsetzbar und unabhängig von der verwendeten Software sein. Das heißt, die Mechanismen sollen

[1] Eine einfache “Hello world”-Anwendung in einer JVM, welche durch einen Linux-Prozess ausgeführt wird, belegt zwischen 150 und 200 MB virtuellen Speicher. Auch wenn der Speicher oft nur aus Performancegründen in dieser Höhe angefordert wird, zeigen diese Zahlen dennoch, dass die Desktop-Variante der Java-Umgebung für den Einsatz in eingebetteten Systemen ungeeignet ist.

[2] Während das .NET Compact Framework eine Windows CE Plattform mit mindestens 12 MB RAM benötigt, wird für die kleinste spezifizierte .NET Umgebung, das .NET Micro Framework, als Mindestvoraussetzungen ein 32-bit-ARM-Microcontroller mit mindestens 256 KB RAM und 512 KB Flashspeicher angegeben. Für reale Anwendungen werden allerdings mindestens 300 MB RAM und etwa 1 MB Flashspeicher empfohlen [TM07].

Die J2ME in ihrer kleinsten Konfiguration, CLDC, ist für Geräte mit 16- oder 32-bit-Microcontroller, 192 KB RAM und mindestens 160 KB Flashspeicher spezifiziert. Auch hier wird für reale Anwendungen üblicherweise mehr Speicher benötigt, außerdem ist, im Gegensatz zum .NET Micro Framework, ein Betriebssystem erforderlich.

sich auch auf existierende und nicht speziell vorbereitete Software anwenden lassen. Dabei müssen die Anforderungen an einen kleinen Knoten so gering wie möglich gehalten werden, um ihn nicht zusätzlich zu belasten.

Die Verwendung der Infrastruktur und der entwickelten Mechanismen soll an einem Anwendungsbeispiel gezeigt werden. Hierzu wird eine Java-Laufzeitumgebung für sehr kleine Knoten vorgestellt. Mithilfe der Unterstützung durch einen Verwalterknoten kann sie dynamisch angepasst werden und kann somit den Funktionsumfang einer vollständigen JVM auf dem verwalteten Knoten anbieten.

1.3 Gliederung der Arbeit

In *Kapitel 2* werden einige alternative Möglichkeiten vorgestellt, wie man den Speicher auf eingebetteten Geräten effizient nutzen kann. Außerdem werden verschiedene Techniken vorgestellt, Code zu einem existierenden System hinzuzufügen. Schließlich werden Konzepte vorgestellt, wie mehrere Knoten zusammenarbeiten können, um beispielsweise ihre Ressourcen zu teilen.

In *Kapitel 3* wird das Konzept des Verwalters und der verwalteten Knoten vorgestellt, mit dessen Hilfe die hier besprochenen Aufgaben gelöst werden sollen. Anschließend wird ein Überblick über die Aufgaben der eingesetzten Software gegeben. Dabei wird auch auf die Schichten eingegangen, aus denen sich die Software eines Verwalters zusammensetzt.

Kapitel 4 gibt einen detaillierten Einblick in die Basisdienste des Systems. Es wird ein Verfahren zur Modularisierung der Software vorgestellt. Außerdem wird auf die Problematik eingegangen, zur Laufzeit alle Verweise zwischen den Modulen zu erkennen. Darüber hinaus wird die Realisierung der angebotenen Verwaltungsdienste und -operationen betrachtet und damit die Funktionsweise des Systems näher gebracht.

Das *5. Kapitel* beschäftigt sich mit dem Einsatz und den Kosten der Verwaltungsoperationen. Es werden Konzepte und Möglichkeiten vorgestellt, wie die Verwaltung eines kleinen Knotens vorgenommen werden kann. Als Beispiel wird dabei eine Java-Umgebung untersucht.

In *Kapitel 6* werden die Ergebnisse der Arbeit zusammengefasst und bewertet. Schließlich erfolgt ein Ausblick auf weiterführende Arbeiten.

2 Grundlagen und Techniken

Eine knappe Ressource in eingebetteten Systemen ist der verfügbare Speicherplatz. Darin ist auch eine Motivation für die dynamische Umkonfiguration begründet. Um Speicher einzusparen oder effizienter zu nutzen, gibt es aber auch andere Ansätze, von denen zunächst einige in diesem Kapitel vorgestellt werden sollen. Als nächstes wird die Grundlage der dynamischen Anpassung, das dynamische Verändern von Code zur Laufzeit, betrachtet. Hierzu werden verschiedene Ansätze vorgestellt und miteinander verglichen. Anschließend werden grundlegende Techniken zur Kooperation von Knoten vorgestellt. Die Kooperation ist die Voraussetzung, um entfernte Ressourcen zu nutzen oder entfernte Knoten zu verwalten.

2.1 Ansätze zur Reduktion des Speicherbedarfs für kleine Knoten

Da es ein wesentliches Ziel dieser Arbeit ist, den verfügbaren Speicherplatz auf kleinen Knoten effizienter zu nutzen, sollen auch andere Techniken vorgestellt werden, die den Speicherplatz von Anwendungen reduzieren beziehungsweise den vorhandenen Speicher intelligent ausnutzen. Die meisten Ansätze sind dabei nicht nur als Alternative, sondern auch als Ergänzung, zur dynamischen Umkonfiguration zu sehen.

2.1.1 Reduktion der Codegröße

Es gibt verschiedene Möglichkeiten, den Speicher eines kleinen Knotens optimal auszunutzen. Zum einen sollte man vermeiden, den Speicher mit unnötigem Code zu belegen. Zum anderen gibt es Möglichkeiten, den vorhandenen Code in möglichst wenig Speicher unterzubringen. Im Folgenden werden einige Techniken und Möglichkeiten dazu vorgestellt.

2.1.1.1 Befehlssatz

Beim Hardwaredesign beziehungsweise bei der Auswahl des einzusetzenden Mikrocontrollers kann darauf geachtet werden, dass die Befehle möglichst klein codierbar sind. So bietet die ARM-Architektur einen speziellen Befehlssatz an, den sogenannten *Thumb instruction set* [Sea00], der die Befehle mit

zwei statt vier Byte codiert. Eine ähnliche Möglichkeit bietet der TriCore-Mikrocontroller an. Hier gibt es neben den normalen 32-bit-Befehlen spezielle 16-bit-Versionen der häufig genutzten Befehle, um kleineren Code zu erzeugen [Inf05].

Um die Größe der Befehle zu reduzieren, ist oft die Größe der Operanden eingeschränkt sowie die Operationen auf bestimmte Register begrenzt. Trotz dieser Einschränkungen ist der resultierende Code oft kleiner als bei Verwendung der normalen Befehle. Besonders bei Mikrocontrollern mit sehr wenig Speicher wirkt sich die Einschränkung der Operandengröße weniger stark aus, da auch mit verkürzten Operanden ein größerer Teil des Adressraums ansprechbar ist.

Die Ausführung der 16-bit-Befehle benötigt in der Regel genauso viele Taktzyklen wie die entsprechenden 32-bit-Befehle. Die kompaktere Darstellung des Codes kann sich allerdings positiv auf die Nutzung des Instruktionscaches auswirken. Bei einigen ARM-Prozessoren wird jedoch darauf hingewiesen [ARM06], dass die kompaktere Codierung der Sprungbefehle zu einer leichten Verschlechterung der Sprungvorhersage (*branch prediction*) führen kann.

2.1.1.2 Zwischencode

Unabhängig vom tatsächlich eingesetzten Prozessor, kann man eine Anwendung auch in einem Zwischencode beschreiben, der eine besonders kompakte Darstellung erlaubt. Dem Zwischencode liegt oft eine virtuelle Maschine zugrunde. Zur Ausführung muss der Code interpretiert werden. Es gibt zahlreiche Beispiele, zu den grundlegenden Arbeiten zählt der bei Pascal eingesetzte *P-Code* [Bar81] und die *Experimental Machine* (*EM-1*) [Tan78].

Die kompakte Darstellung wird oft dadurch erreicht, indem man auf ein stackbasiertes Maschinenmodell setzt, wodurch die explizite Angabe der Operanden entfällt. Wird der Code und der Interpreter für einen speziellen Einsatzzweck entworfen, so können außerdem die Befehle genau angepasst werden, wodurch einzelne Befehle relativ mächtig sind.

Der Nachteil dabei ist allerdings, dass die Verwendung eines Interpreters zusätzlichen Speicherplatz benötigt. Außerdem ist die Ausführungsgeschwindigkeit, im Vergleich zu einer direkten Ausführung des nativen Binärcodes, in der Regel langsamer, da das Programm verarbeitet und umgesetzt werden muss.

2.1.1.3 Codeerzeugung

Beim Übersetzen eines Programms aus einer höheren Sprache in Maschinencode können verschiedene Optimierungen eingesetzt werden, um die Größe des erzeugten Codes zu beeinflussen [Muc97]. Eine Möglichkeit ist, Code zu eliminieren, der nicht verwendet wird oder keine Auswirkung auf das Ergebnis des Programms hat. Allerdings lässt sich unerreichbarer Code nicht immer zuverlässig identifizieren, vor allem wenn die Erreichbarkeit von externen Daten oder Ereignissen abhängt. Manche unnötige Anweisungen können jedoch während des Übersetzungsvorganges anhand eines Kontroll- und Datenflussgraphen erkannt und eliminiert werden. Beispielsweise kann die Zuweisung eines Wertes zu einer Variablen entfernt werden, wenn sie später nicht mehr ausgelesen wird.

Neben der Entfernung von unbenötigten Programmteilen hat der Compiler häufig mehrere Möglichkeiten eine Folge von Anweisungen einer höheren Sprache in Maschinensprache umzuwandeln. Dabei unterscheiden sich die Möglichkeiten oft in der Größe des resultierenden Codes und in der geschätzten Ausführungsgeschwindigkeit.

Ein Beispiel hierfür ist die Identifikation von doppeltem Code. Es gibt verschiedene Situationen, in denen Code doppelt erzeugt wird. So ist beispielsweise das Ende einer Funktion oft ähnlich oder gleich.

Ist es gleich, so kann ein Codestück gemeinsam verwendet werden, das am Ende der Funktionen angesprungen wird (*Cross-Jumping*). Bei ähnlichem Code, der semantisch gleich ist und sich zum Beispiel nur in den verwendeten Registern unterscheidet, kann der Compiler versuchen den gemeinsamen Code als separate Funktion zu extrahieren (*prozedurale Abstraktion*) [DEMS00]. Dieser Vorgang stellt in etwa das Gegenteil des *Inlinings* dar. Der Nachteil dabei ist, dass zusätzlicher Aufwand durch den Funktionsaufruf und die Parameterübergabe entsteht; dafür ist der resultierende Code kleiner, was wiederum positive Effekte auf die Nutzung eines Instruktionscaches haben kann.

Die angedeuteten Techniken sind unabhängig von den in dieser Arbeit entwickelten Mechanismen und können daher parallel eingesetzt werden. Die Auswirkungen werden jedoch teilweise sichtbar, da das System, das im Rahmen dieser Arbeit entwickelt wurde, auf der Basis von Funktionen arbeitet und zum Beispiel durch die prozedurale Abstraktion zusätzliche Funktionen erzeugt werden.

2.1.2 Codekompression

Bei der Codekompression wird der binäre Code komprimiert im Speicher abgelegt und erst kurz vor der Verwendung entpackt. Die Dekompression des Codes soll dabei transparent erfolgen. Hierfür gibt es verschiedene Ansätze. Einige Techniken verwenden eine besondere Hardware, um den Code bei Speicherzugriffen automatisch zu dekomprimieren [WC92]. Andere Techniken sind rein in Software realisiert, benötigen dabei allerdings spezielle Werkzeuge um den Code vorzubereiten [KKMS97].

Nachteil dieser Verfahren ist, dass eine Möglichkeit gegeben sein muss, um Adressen im unkomprimierten Code auf komprimierten Code abzubilden. Weiterhin wird, zumindest bei softwarebasierten Verfahren, zusätzlicher Speicherplatz benötigt, um den entkomprimierten Code abzulegen. Schließlich ist noch anzumerken, dass für die Dekomprimierung Rechenzeit und Energie aufgewendet werden muss.

Prinzipiell ist die Kompression von Code parallel zu der in dieser Arbeit vorgeschlagenen Technik einsetzbar. Es muss allerdings eine Abstimmung zwischen den Verwaltungsstrukturen, die zur Dekomprimierung verwendet werden, und den Änderungen, die durch das vorgestellte System vorgenommen werden, stattfinden.

2.1.3 Überlagerungstechnik

Bei der Überlagerungstechnik versucht man Programmteile zu identifizieren, die nicht gleichzeitig genutzt werden, sogenannte *Overlays*. Da diese nicht gleichzeitig im Speicher vorhanden sein müssen, kann man sie abwechselnd an derselben Speicherstelle bereitstellen, sobald man sie benötigt. Ein Speicherbereich kann so durch Überlagerung mehrfach von verschiedenen Codeblöcken verwendet werden. Die Codeblöcke können dabei verschiedene Größen haben und sind nur durch die Größe des Programmspeichers beschränkt. In einem Programm ist es möglich auch mehrere Speicherbereiche vorzusehen, an denen Overlays eingeblendet werden, allerdings kann man ein Overlay normalerweise immer nur an einer festen Adresse einblenden. Die Anzahl der Overlays ist durch den zusätzlich vorhandenen Speicher beschränkt. Bei Systemen mit Harvard-Architektur können Overlays beispielsweise im Datenspeicher abgelegt sein und bei Bedarf in den Programmspeicher geladen werden. Alternativ ist das Nachladen der Overlays von einem Hintergrundspeicher (Bandlaufwerk, Festplatte) denkbar.

Der Vorteil der Überlagerungstechnik ist, dass man Programme ausführen kann, die größer sind als der verfügbare Programmspeicher. Bei der Realisierung kann man zwischen verschiedenen Stufen unterscheiden [Pan68]:

automatisch: Das System bestimmt automatisch die Unterteilung des Programms in Overlays. Außerdem sorgt das System automatisch für die Bereitstellung der benötigten Overlays.

semi-automatisch: Der Entwickler legt die Aufteilung des Programms in Overlays fest. Das Umschalten der jeweils benötigten Overlays wird jedoch automatisch durchgeführt.

manuell: Der Entwickler muss nicht nur die Aufteilung der Software in Overlays manuell durchführen, sondern er muss darüber hinaus durch Anweisungen im Code dafür sorgen, dass das richtige Overlay eingelagert ist, wenn es benötigt wird.

Die automatische Überlagerung kann als Vorläufer der Seitenüberlagerung bei virtuellem Adressraum (*paging*) betrachtet werden. Hierbei stößt eine Hardwareeinheit zur Adressumrechnung das Einlagern von Datenblöcken (Seiten) an, wenn auf Adressen zugegriffen wird, deren Speicherstelle momentan nicht im physikalischen Speicher repräsentiert ist.

Im Vergleich zur automatischen Seitenüberlagerung, zur Bereitstellung eines virtuellen Adressraums, hat die Überlagerungstechnik den Vorteil, dass sie ohne zusätzliche Hardwareunterstützung verwendet werden kann und somit für eingebettete Systeme geeignet ist. Für Echtzeitanwendungen hat die (manuelle) Überlagerung außerdem den Vorteil, dass der Zeitbedarf für die Bereitstellung des benötigten Codes deutlich besser vorhergesagt werden kann als bei einer transparenten Seitenüberlagerung im Rahmen von virtuellem Speicher.

Der Nachteil ist allerdings, dass das Verfahren nicht transparent für den Softwareentwickler ist und er sich über die Größe der Overlays bewusst werden muss. Auch das Finden einer optimalen Aufteilung in Overlays ist eine nicht triviale Aufgabe.

Im Rahmen dieser Arbeit wird ein System erstellt, das ähnliche Eigenschaften aufweist wie eine automatische Überlagerungstechnik. Die Codeblöcke werden allerdings nicht auf dem Gerät gelagert, sondern bei Bedarf übertragen. Außerdem werden die Codeblöcke durch einen dynamischen Binder angepasst, sodass sie an beliebigen Adressen eingesetzt werden können.

2.2 Ansätze zur dynamischen Modifikation von Anwendungen

Im Folgenden sollen einige Ansätze zur nachträglichen Veränderung von Code zur Laufzeit vorgestellt werden. Die meisten Beispiele sind dabei aus dem Umfeld der Sensornetze entnommen, da die Zielplattform kleine ressourcenbeschränkte Knoten sind. Weitere Systeme, die das dynamische Verändern von Code zur Laufzeit erlauben, sind zum Beispiel bei Hicks [Hic01] zu finden.

2.2.1 Interpretergestützte Ausführung

Eine Möglichkeit, die Anwendung flexibel zu erstellen, ist die Verwendung einer Sprache, die erst zur Laufzeit interpretiert wird. Das Vorgehen entspricht dem in Abschnitt 2.1.1.2 beschriebenen Zwischencode: Eine Anwendung besteht aus einer Reihe von Befehlen, die durch einen Interpreter gelesen, analysiert und umgesetzt werden. Eine Ausprägung dieses Ansatzes sind Skriptsprachen, wie sie beispielsweise bei *SensorWare* [BHS03] eingesetzt werden. Hier werden mobile Agenten, die für den Einsatz auf PDAs gedacht sind, in einer TCL-ähnlichen Skriptsprache programmiert. Eine andere Ausprägung sind virtuelle Maschinen, die häufig durch Interpreter realisiert sind. Ein Beispiel ist *MagnetOS* [LRW+05], welches eine verteilte Java Maschine für PDAs bereitstellt. Für Sensornetze sind die Systeme *VM** [KP05b] und *Maté* [LC02] beispielhaft zu nennen.

Neben der kompakten Darstellung des Codes ist als weiterer Vorteil das relativ einfache Nachladen eines Programms zur Laufzeit zu nennen. Abhängigkeiten zu anderen Programmmodulen oder zum Laufzeitsystem werden durch den Interpreter aufgelöst. Der Code wird dabei dem Interpreter in Form von Daten zur Verfügung gestellt. Speziell bei Harvard-Architekturen sind dabei einige Besonderheiten

zu beachten: Wird der Programmcode im Programmspeicher abgelegt, so müssen die Anweisungen vor der Verarbeitung zunächst in den Datenspeicher kopiert werden. Legt man das Programm hingegen im Datenspeicher ab, so ist zu bedenken, dass dieser bei solchen Systemen oft deutlich knapper als Programmspeicher ausfällt. Allerdings hat man dann den Vorteil, dass die Veränderungen nicht am Programmspeicher vorgenommen werden müssen, dessen Beschreiben, im Gegensatz zum Datenspeicher, oft aufwendiger ist.

Generelle Nachteile dieses Ansatzes wurden bereits genannt: der zusätzliche Speicherverbrauch und die reduzierte Ausführungsgeschwindigkeit. Hinzu kommt noch, dass dieser Ansatz auf die Veränderung der Anwendung beschränkt ist. Grundlegende Systemkomponenten oder der Interpreter selbst können nicht verändert werden.

2.2.2 Verändern des Speicherabbildes als Ganzes

Will man nicht auf eine Skriptsprache oder interpretierten Zwischencode setzen, so kann man den Binärcode auch direkt verändern oder austauschen. Im Bereich von kleinen Knoten gibt es dabei zunächst die Möglichkeit, das Speicherabbild eines Knotens als eine Einheit zu betrachten und durch ein neues zu ersetzen. Dieser Ansatz wird beispielsweise bei *TinyOS* [HSW+00] angeboten. Dort gibt es mit *Xnp* [Cro03, JKB03] und *Deluge* [HC04] zwei ähnliche Systeme, die ein komplettes Speicherabbild übertragen und installieren. Vorteil dabei ist, dass auch grundlegende Teile des Betriebssystems verändert werden können. Nachteil ist, dass auch für kleine Änderungen immer das gesamte Speicherabbild übertragen und neu geschrieben werden muss. Um die Menge der zu übertragenden Daten zu reduzieren, können auch nur die Änderungen übertragen werden, die zuvor aus einem Vergleich des ursprünglichen und des neuen Abbildes extrahiert werden [RL03, JC04]. Meist muss dazu allerdings das ursprüngliche Abbild vorliegen. Da das Speicherabbild auf dem Sensorknoten oft in Flashspeicher geschrieben werden muss, gibt es auch Ansätze, die Änderungen so anzuordnen, dass möglichst wenige Seiten von Änderungen betroffen sind [KP05a].

Keines der genannten Systeme ist in der Lage den Zustand des Knotens über eine Änderung hinweg zu erhalten. Bei allen Systemen ist ein Neustart des Sensorknotens nach der Veränderung vorgesehen.

2.2.3 Austauschen von binären Programmmodulen

Um den Zustand teilweise zu erhalten, bietet es sich an, nicht den gesamten Speicher als Einheit anzusehen, sondern kleinere Programmmodule auszutauschen beziehungsweise zu verändern. Bei der Integration der Programmmodule in das bestehende System gibt es verschiedene Ansätze.

2.2.3.1 Positionsunabhängiger Code und indirekte Adressierung

Positionsunabhängiger Code (*Position Independent Code*, *PIC*) zeichnet sich dadurch aus, dass er keine festen Adressen enthält und somit an beliebige Positionen im Speicher geladen und verwendet werden kann. Sprünge innerhalb des Codes werden durch relative Sprünge realisiert. Die Anbindung an das existierende System kann zum Beispiel über eine Tabelle mit Einsprungadressen realisiert werden. Im Code wird dann anstelle der Adresse nur der Index in die Tabelle verwendet. Allerdings muss die Tabelle an einer festen Position stehen. Auch der Zugriff auf globale Variablen muss über eine Tabelle erfolgen, da ihre relative Position ebenfalls nicht konstant ist.

Positionsunabhängiger Code kommt häufig bei gemeinsam genutzten Bibliotheken zum Einsatz, da sie in mehreren Prozessen an verschiedene Adressen gebunden werden. Ein anderes Beispiel ist

SOS [HKS+05], ein Betriebssystem für Sensorknoten, welches das Nachinstallieren und Austauschen von Programmmodulen zur Laufzeit mithilfe von positionsunabhängigem Code realisiert.

Durch dieses Verfahren kann das Modul im gebundenen Binärcode zum Knoten übertragen und ohne weitere Vorbereitung an beliebiger Adresse eingesetzt werden. Da die Aufrufe zu dem Modul indirekt über eine Tabelle ausgeführt werden, ist auch das Austauschen des Programmmoduls einfach, da lediglich die Adresse in der Tabelle anzupassen ist. Auf der anderen Seite ist die Indirektionsstufe auch ein Nachteil, da das Auflösen zusätzlich Laufzeit benötigt. Ein weiterer Nachteil ist, dass eine Werkzeugkette benötigt wird, die positionsunabhängigen Code erstellen kann. Entsprechende Werkzeuge sind allerdings nicht für alle Plattformen verfügbar. Für die MSP430 Mikrocontroller von Texas Instruments sind beispielsweise keine Compiler verfügbar, um positionsunabhängigen Code zu erzeugen. Für AVR Mikrocontroller unterstützt der GCC diese Art der Codeerstellung. Ein Programmmodul kann hier allerdings maximal 4 KByte groß sein, da das die maximale Entfernung für einen relativen Sprung ist.

2.2.3.2 Vorab-gebundene Programmmodule

Um die Einschränkung von positionsunabhängigem Code zu vermeiden und dennoch die Einbindung neuer Module möglichst einfach zu gestalten, können Module verwendet werden, die bereits vor der Übertragung gebunden wurden. Somit ist kein Binden auf dem Knoten erforderlich. Dafür müssen allerdings Informationen über alle vorhandenen Symbole auf dem externen Rechner vorhanden sein. Außerdem kann ein so vorbereitetes Modul nur für ein System verwendet werden, welches exakt dem Aufbau der gegebenen Symbolinformationen entspricht.

Beispiele aus dem Bereich der Sensornetzwerke sind frühe Versionen von *Contiki* [DGV04]. Teilweise kann auch der Baukasten für virtuelle Maschinen *VM** [KP05b] in diese Kategorie eingeordnet werden, der die Erweiterung des Interpreters mittels extern gebundenen Modulen ermöglicht.

2.2.3.3 Dynamisch gebundene Programmmodule

Um die Module flexibler einsetzen zu können, kann man das Binden auch erst auf dem Knoten durchführen. Ein Beispiel hierfür ist *FlexCup* [MGL+06], ein im Rahmen des *TinyCubus*–Projekts [MLM+05] entwickeltes Updatesystem für Sensorknoten. Auch bei neueren Versionen von *Contiki* [DFEV06] wird auf dynamisches Binden der Module gesetzt.

Der Vorteil des Verfahrens ist, dass die Programmmodule nicht für eine spezielle Speicherbelegung erstellt werden und somit auf mehreren Knoten eingesetzt werden können, auch wenn sich diese leicht unterscheiden. Nachteil ist allerdings, dass nicht nur der Binder auf jedem Knoten vorhanden sein muss, sondern auch die Symbolinformationen. Ebenso müssen neue Symbol- und Relokationsinformationen mit dem Modul zu jedem Knoten übertragen werden.

2.2.4 Vergleich der Verfahren

In Tabelle 2.1 werden die verschiedenen Ansätze anhand einiger Kriterien miteinander verglichen.

Je weniger Infrastruktur am Knoten benötigt wird, desto mehr Speicherplatz steht für die eigentliche Software zur Verfügung. Interpreterbasierte Verfahren und dynamisches Binden benötigen beide umfangreichen Code oder Daten auf dem Knoten. Für das Anwenden von Änderungen ist weniger Code notwendig. Das Einspielen von exakt vorbereiteten Programmmodulen oder Speicherabbildern benötigt hingegen am wenigsten Infrastruktur am Knoten.

Betrachtet man die Menge der zu übertragenden Daten, so ist die Übertragung des gesamten Speicherabbildes das Maximum. Ein guter Wert wird erreicht, wenn nur die veränderten binären

	interpretiert	alles tauschen	nur Änderungen	pos. unabh. (PIC)	vorab-gebunden	dyn. gebunden
geringe Infrastruktur am Knoten	- -	+	-	0	+	- -
geringe Übertragungsmenge	++	- -	+	+	+	-
geringer Aufwand am Knoten	+	+	-	0	+	-
Flexibilität der Modulverwendung	+	0	-	+	-	+
geringe Mehrkosten zur Laufzeit	- -	+	+	0	+	+
feine Granularität	+	- -	0	+	+	+

Tabelle 2.1: Vergleich der Verfahren zum Codeupdate

Daten übertragen werden. Nur bei der Übertragung von speziellem interpretierten Code können unter Umständen noch bessere Werte erreicht werden. Beim dynamischen Binden ist zu beachten, dass neben dem Code zusätzlich Symbol- und Relokationsinformationen am Knoten benötigt werden.

Der Aufwand, um die Änderung auf dem Knoten einzuspielen, ist besonders gering, wenn der Code schon von einem externen Rechner vorbereitet wurde. In diesem Fall müssen weder Tabellen angepasst, noch Code gebunden werden.

Bei der Flexibilität wird betrachtet, wie gut man die Methode für die Veränderung mehrerer Knoten einsetzen kann. Enthält der übertragene Code keinen Bezug zur aktuellen Speicherbelegung, so kann er für verschiedene Knoten verwendet werden, ansonsten kann das Programm nur für genau einen Knoten angewandt werden.

Beim Aufwand zur Laufzeit sticht besonders die Interpretation hervor, da der Code analysiert und umgesetzt werden muss. Alle anderen Verfahren haben kaum oder keinen zusätzlichen Mehraufwand zur Laufzeit.

In der Granularität der Veränderung ist der Hauptunterschied, ob man den gesamten Speicher betrachtet oder einzelne Programmteile verändern kann.

2.2.5 Zustandsanpassung

Die bisher vorgestellten Techniken beschäftigen sich hauptsächlich mit dem Verändern von Code. Der Laufzeitzustand wurde dabei kaum beachtet. Bei den meisten beispielhaft angesprochenen Systemen bleibt der Zustand bei einer Veränderung auch nicht erhalten. Das System wird nach der Veränderung neu gestartet, um ungültige Zeiger zu eliminieren. Lediglich *Contiki* [DGV04] bietet die Möglichkeit an, dass die alte Version eines Programmmoduls seinen Zustand sichert, sodass er von der neuen Version verwendet werden kann. Die Anpassung des Zustands bleibt der Anwendung überlassen.

Will man ungültige Referenzen vermeiden, müssen entweder genügend Informationen zur Verfügung stehen, um alle Referenzen zu kennen oder man führt eine Indirektionsstufe ein. Wie bereits vom positionsunabhängigen Code bekannt, werden dann Aufrufe beispielsweise über eine Tabelle durchgeführt, die einen Bezeichner auf eine Adresse abbildet [BHA+05]. Die Anpassung anderer Werte des Zustands kann im Allgemeinen nicht automatisch erfolgen, da die Bedeutung der einzelnen Elemente unbekannt ist. Unterstützung kann das System nur bei der Erstellung von Sicherungspunkten bieten.

2.3 Ansätze zur Benutzung und Steuerung entfernter Knoten

In diesem Abschnitt werden Techniken und Ansätze vorgestellt, wie entfernte Knoten genutzt werden oder wie verteilte Knoten in Kooperation zusammenarbeiten können.

2.3.1 Fernaufruf

Ein grundlegender Mechanismus, um Dienste auf einem entfernten Rechner zu nutzen, ist der Fernaufruf, auch als *Remote Procedure Call* (*RPC*) oder bei objektorientierter Programmierung als *Remote Methode Invocation* (*RMI*) bezeichnet [TS03, Sun04]. Dabei wird auf dem *Client* eine Funktion aufgerufen, die dann auf dem *Server* ausgeführt wird. Für den Softwareentwickler sieht ein Fernaufruf fast wie ein normaler Funktionsaufruf aus. Dazu ist auf dem Client ein Fernaufrufsystem vorhanden, welches einen Stellvertreter, einen sogenannten *Stub*, für die Funktion bereitstellt. Dieser Stellvertreter wird anstelle der eigentlichen Funktion aufgerufen und bekommt somit die Parameter des Aufrufs. Er veranlasst dann das Erstellen und Absenden der Nachricht an den Server und wartet auf ein Ergebnis. Die Nachricht enthält neben den Parametern mindestens noch eine Identifikation der aufzurufenden Funktion. Diese Nachricht wird von einem Fernaufrufsystem am Server empfangen. Dort wird ein stellvertretender Aufrufer der Funktion bereitgestellt, ein sogenannter *Skeleton*. Dieser bekommt die Parameter aus der Nachricht, ruft die eigentliche Funktion auf und leitet den Rückgabewert an das Fernaufrufsystem weiter, welches ihn zurück zum aufrufenden System schickt. Dort wird der Rückgabewert dem entsprechenden Stellvertreter zugeordnet, der ihn dem ursprünglichen Aufrufer zurückgibt.

Da verschiedene Systeme unterschiedliche Repräsentationen der Daten haben können, muss bei der Realisierung eines Fernaufrufsystems festgelegt werden, wie die Parameter serialisiert und codiert werden (*Marshalling*). Bei der Verwendung eines Fernaufrufs ist zu beachten, dass dieser zwar verwendet werden kann wie ein normaler Funktionsaufruf, jedoch ein anderes Verhalten beinhaltet. Zum einen sind Fernaufrufe um ein Vielfaches langsamer als lokale Aufrufe, zum anderen können durch das zwischengeschaltete Kommunikationssystem Fehler auftreten, die bei einem lokalen Funktionsaufruf nicht möglich sind. Fehlerquellen sind Nachrichtenverlust, -verdopplung oder -veränderung. Im Rahmen eines Fernaufrufsystems gibt es Möglichkeiten einige Fehler zu bearbeiten und damit der Semantik eines lokalen Aufrufs möglichst nahe zu kommen. Die sogenannte *exactly-once* Semantik [Nel81] eines lokalen Funktionsaufrufs ist jedoch nicht immer erreichbar.

2.3.2 Verschieben von Anwendungskomponenten

Im Bereich der mobilen Agenten werden Programme oder Programmteile von einem Rechner auf einen anderen verschoben [Omg00]. Dabei wird zunächst die Anwendung angehalten. Danach werden die Daten, die transferiert werden sollen, identifiziert und serialisiert. Nach der Übertragung werden die Daten auf dem Zielknoten deserialisiert, um anschließend die Ausführung der Anwendung fortzusetzen.

Bei der Migration unterscheidet man zwischen *schwacher* und *starker Migration* [TS03]. Die schwache Migration ist die einfachere Variante. Charakteristisch für sie ist, dass die übertragene Anwendung auf dem Zielknoten immer in ihrem Ausgangszustand gestartet wird. Es ist daher nur die Übertragung des Codes und der initialen Daten nötig. Der aktuelle Laufzeitzustand der Anwendung geht bei der Migration verloren.

Bei der starken Migration hingegen wird zusätzlich der Laufzeitzustand serialisiert und auf dem Zielknoten wieder hergestellt. Hierdurch kann eine starke Migration transparent für die Anwendung

ablaufen. Im Gegensatz zur schwachen Migration ist die Realisierung der starken Migration komplexer, da die Extraktion des Zustands einer Anwendung oft sehr aufwendig ist.

Als Zwischenlösung, um die Lücke zwischen den beiden Migrationsarten zu schließen, kann man durch Übergabe von Parametern bei schwacher Migration statt eines festen, einen variablen Einsprungpunkt simulieren. Die Anwendung wählt dabei vor der Migration die Prozedur aus, mit der sie nach der Migration aufgesetzt werden soll, und gibt diese Daten als Parameter der Startfunktion mit. Nach dem Aufsetzen wertet man diesen Parameter aus und springt sofort zur angegebenen Funktion. Dabei geht zwar der Zustand verloren; er kann jedoch vor der Migration dazu genutzt werden, das weitere Vorgehen nach der Migration zu beeinflussen.

Zustandstransfer

Die Übertragung des Zustands spielt nur bei der starken Migration eine Rolle. Ein einfacher Ansatz zur Zustandsübertragung zwischen zwei gleichen Knoten ist, den betroffenen Speicherbereich zu sichern und am Zielknoten an derselben Stelle einzuspielen. In einem heterogenen System ist dieser Ansatz nicht zielführend. Hier müssen die Elemente des Ausführungszustands identifiziert, übertragen und in einer äquivalenten Form auf dem Zielrechner wieder aufgebaut werden. Eine Technik, dies automatisch zu ermöglichen, ist, den Aufrufstack und die lokalen Variablen durch ein Laufzeitsystem zu erfassen und in eine architekturunabhängige Version zu überführen. Für Sprachen wie beispielsweise C++, die zur Laufzeit zu wenige Informationen bieten, um den architekturabhängigen Aufbau des Stacks zu erfassen, kann durch Compilerunterstützung der Code entsprechend verändert werden, um die benötigten Informationen zu sammeln [DR98].

2.3.3 Laufzeitsysteme für verteilte Systeme

Zur gemeinsamen Verwaltung und Nutzung der Ressourcen von mehreren Knoten bieten sich verteilte Betriebssysteme an. Diese bieten den Anwendungen Informationen und Mechanismen an, um die verteilten Ressourcen zu erkennen und zu nutzen. Dazu wird ein Programmiermodell definiert, das den Umgang mit der Verteiltheit vereinheitlichen soll. Im Rahmen dessen kann auch ein Mechanismus angeboten werden, um den Ausführungsort zu bestimmen beziehungsweise zu wechseln.

Eine andere Ausprägung ist ein sogenanntes *Single System Image* (*SSI*). Hierbei werden die vorhandenen Knoten als ein einziges homogenes System dargestellt. Realisiert werden solche Systeme häufig durch gemeinsamen verteilten Speicher (*Distributed Shared Memory*). Die Nutzung von speziellen Ressourcen, die nur an einem Knoten zur Verfügung stehen, ist jedoch nicht immer möglich. Auch der Einfluss auf die tatsächliche Verteilung der Anwendung ist oft eingeschränkt. Ein Beispiel eines solchen Systems ist *MagnetOS* [LRW+05], das eine Java-Umgebung darstellt, die mehrere PDAs zu einer einzigen virtuellen Maschine zusammenfasst.

Neben Laufzeitsystemen für verteilte Systeme gibt es auch noch verteilte Laufzeitumgebungen. Sie betrachten jeden Knoten getrennt und bieten keine spezielle Unterstützung zur Kooperation an. Sie nutzen jedoch andere Knoten, um Dienste der Infrastruktur entfernt bereitzustellen [SGGB99].

2.3.4 Fernwartungssysteme

Zur Steuerung oder Verwaltung entfernter Knoten können Fernwartungssysteme eingesetzt werden. Ein Beispiel hierfür ist das Projekt *DIADEM* (*Distributed Application Independent Maintenance*) [Her87]. Das Ziel des Projekts war die Entwicklung eines Systems zur komfortablen und flexiblen Fernwartung von Rechnern. Unter Fernwartung wurde das Austauschen und Installieren von Anwendungen auf einer

Reihe entfernter Rechner verstanden. Die Wartung umschließt dabei auch das Anpassen eines Dateiformats, falls das für den Betrieb der neuen Programmversion notwendig ist. Die Rekonfiguration der Software zur Laufzeit war jedoch nicht Ziel, ebenso wurde explizit das Verändern des Betriebssystems ausgeschlossen.

Der grundsätzliche Aufbau sieht einen zentralen Rechner vor, der sogenannte *Wärter* (*maintainer*), der Zugriff auf die Softwaredatenbank hat und Wartungsaufträge an die Produktionsrechner gibt. Die Produktionsrechner werden in dem System als *Wartlinge* (*maintainees*) bezeichnet und führen die Wartungsaufträge aus. Um Engpässe bei sehr vielen Wartlingen zu vermeiden, ist ein hierarchischer Aufbau mit Zwischenschichten angedacht. Die Rechner der Zwischenschichten werden *Wärtlinge* (*maintaineers*) bezeichnet und leiten Wartungsaufträge an andere Wärtlinge oder Wartlinge weiter und leiten Ergebnisse zurück an die Auftraggeber.

Die Wartungsarbeiten werden durch Wartungsprogramme beschrieben, welche in einer speziell hierfür entwickelten Sprache Namens *DIALECT* geschrieben sind. Die Wartungsprogramme können entweder komplett zu den Wartlingen übertragen und dort ausgeführt werden (*joborientiert*) oder sie können teilweise am Wärter ausgeführt werden (*dialogorientiert*). Bei der zweiten Möglichkeit werden die Entscheidungen, die in den Wartungsprogrammen definiert sind, am Wärter getroffen und lediglich die Wartungsaktionen auf den Wartlingen initiiert.

Die grundlegende Aufteilung in Knoten, welche die Veränderung organisieren und koordinieren und andere Knoten, die lediglich Aufträge ausführen, ist ähnlich wie die Struktur des in dieser Arbeit vorgestellten Ansatzes.

2.4 Zusammenfassung

In diesem Kapitel wurden zunächst Möglichkeiten beschrieben, die zur effizienten Speichernutzung eingesetzt werden können. Die angesprochenen Techniken können dabei alternativ oder parallel zu dem Ansatz in dieser Arbeit verwendet werden. Anschließend wurden Techniken beschrieben, um Code auf kleinen Knoten zur Laufzeit auszutauschen. Im Rahmen dieser Arbeit wird der Code bereits vorab gebunden. Im Vergleich schneidet diese Möglichkeit gut ab. Zuletzt wurden Ansätze aufgezeigt, wie verschiedene Knoten zusammenarbeiten können. Neben Grundlagen wurde dabei auch ein Fernwartungssystem vorgestellt, das im grundlegenden Aufbau dem in dieser Arbeit vorgestellten System ähnlich ist.

3

System zur Verwaltung eingebetteter, heterogener, verteilter Knoten

In diesem Kapitel wird die Architektur der Software vorgestellt, die zur Verwaltung “kleiner” Knoten in einem Mikrocontrollernetzwerk verwendet wird. Die vorgestellte Infrastruktur ermöglicht es, dass kleine Knoten eines Mikrocontrollernetzwerks flexibel eingesetzt werden können. Gleichzeitig sollen möglichst viele Ressourcen zur Erfüllung ihrer eigentlichen Aufgabe zur Verfügung stehen.

Nach der Einführung des allgemeinen Konzepts werden Einsatzszenarios vorgestellt, um anschließend einen Überblick über die Architektur des Verwaltungssystems zu geben. Die einzelnen Komponenten der Architektur werden in den folgenden Kapiteln detaillierter beschrieben.

3.1 Verwalter und verwaltete Knoten

Ein Netzwerk aus Mikrocontrollern ist meist sehr heterogen. Knoten verschiedenster Architekturen und Leistungsklassen sind miteinander verbunden, um eine Aufgabe gemeinsam zu lösen. Zur Verwaltung des Systems teilen wir die Knoten entsprechend ihrer Leistungsfähigkeit und der verfügbaren Ressourcen in zwei Klassen ein:

- Kleine ressourcenbeschränkte Mikrocontroller werden zu *verwalteten Knoten* (*managed nodes*). Sie sind preiswert und robust und können daher ideal “vor Ort” eingesetzt werden, um beispielsweise Sensordaten zu erfassen und Aktoren zu steuern. Sie sind jedoch zu klein, um sich selbst für neue Aufgaben umzukonfigurieren oder um einer Anwendung alle Dienste einer großen Laufzeitumgebung anzubieten. Um solche Aufgaben zu bewältigen, benötigen sie Unterstützung durch andere Knoten.

- Größere Knoten werden daher als *Verwalter* (*manager*) eingesetzt. Verwalter verfügen über ausreichend Ressourcen, um Dienste für verwaltete Knoten zu übernehmen. Dazu gehören die Konfiguration zur Startzeit und die Umkonfiguration zur Laufzeit. Darüber hinaus können sie auch weitere Unterstützung zur Laufzeit anbieten, indem Dienste von verwalteten Knoten von ihrem Verwalter übernommen werden.

Die Knoten lassen sich nicht immer eindeutig einer der beiden Klassen zuordnen. Unter den kleinen Knoten gibt es immer noch Größenunterschiede, welche die Zugehörigkeit zur einen oder anderen Klasse rechtfertigen. Der Übergang zwischen den Klassen ist daher fließend. Verwalterknoten müssen jedoch genug Ressourcen haben, um die verwalteten Knoten konfigurieren zu können. Durch die Konfiguration wird festgelegt, welche Dienste und Software auf welchem verwalteten Knoten ausgeführt werden und welcher Verwalter welchen kleinen Knoten unterstützt. Es können aber auch kleine Knoten ohne die Fähigkeit, einen anderen Knoten zu verwalten, Dienste für noch kleinere Knoten bereitstellen.

Am unteren Ende der Klasse der Verwalter und gleichzeitig am oberen Ende der verwalteten Knoten kann man Geräte der PDA-Klasse sehen. Dabei handelt es sich um Mikrocontroller mit genügend Ressourcen, um ein komplexes Betriebssystem und mehrere Anwendungen quasi-parallel auszuführen. Ressourcenstarke PDAs können Verwalter für kleinere Knoten sein. PDAs mit weniger Ressourcen können eine Zwischenschicht bilden. Sie können die Rolle von Verwaltern einnehmen, indem sie Dienste für verwaltete Knoten übernehmen. Um dabei Ressourcen zu schonen, muss aber nicht unbedingt die Infrastruktur zur Umkonfiguration vorhanden sein. So sind diese Geräte gleichzeitig verwaltete Knoten, für die ein vollwertiger Verwalter die genaue Konfiguration einrichtet.

3.1.1 Permanenter oder temporärer Verwalter

Je nach Einsatzzweck des Netzwerks können Verwalter verschiedene Aufgaben haben. Man kann dabei zwei grundsätzliche Szenarios unterscheiden:

1. Der Verwalter ist nur zeitweise in das System integriert.

2. Der Verwalter ist fester Bestandteil des Systems und immer vorhanden.

Im ersten Fall kann der Verwalter lediglich zur Konfiguration oder zur Umkonfiguration des Systems eingesetzt werden. Der Verwalter darf keine Aufgaben übernehmen, die während der Laufzeit benötigt werden. Bei der Verteilung der Anwendung und somit bei der Planung der Anwendungsstruktur darf er nicht berücksichtigt werden. Der Verwalter kann lediglich zur erstmaligen Installation und zu späteren Umkonfigurationen des Systems eingesetzt werden. Diese Konstellation ist sinnvoll für ein System, welches relativ statisch ist und kaum dynamischen Veränderungen unterliegt. Veränderungen im System in Form von Neu- oder Umkonfigurationen werden hierbei immer von außen initiiert, da zuvor sichergestellt werden muss, dass ein Verwalter im Netzwerk verfügbar ist. Der Verwalterknoten kann hierbei beispielsweise der Laptop eines Kundendiensttechnikers sein, der zur Überprüfung und Umkonfiguration an das System angeschlossen wird und nach der Umstellung wieder entfernt wird. Hierbei ist zu beachten, dass die Konfiguration des Systems in geeigneter Weise abgespeichert werden muss, um später eine erneute Ankopplung zu ermöglichen.

Ist der Verwalter hingegen fester Bestandteil des Systems, so steht er für ständige Hilfestellung zur Verfügung und kann daher fest in die Konfiguration und Verteilung der Anwendung eingeplant werden. Selten benutzte Funktionalität kann dann von einem kleinen Knoten auf einen Verwalterknoten ausgelagert werden, um die Aufgaben des kleinen Knotens besser zu erfüllen. Den gewonnenen Platz auf dem Mikrocontroller steht damit beispielsweise als Puffer zur Verfügung, um mehr Sensordaten zwischenzuspeichern, bevor sie an einen Verarbeitungsknoten weitergegeben werden müssen. Die Aufträge zur Umkonfiguration können hier auch aus dem System selbst entstehen, beispielsweise als Reaktion auf interne oder externe Ereignisse.

3.1.2 Verwalter als Konfigurator

Ist der Verwalter nicht dauerhaft in das System integriert, so kann er in erster Linie zur Konfiguration des Systems dienen. Die Hauptaufgabe des Verwalters besteht dann in der Verwaltung des Codes, der im Netzwerk ausgeführt werden soll. Der Verwalter kann zur Installation, Entfernung, zum Austausch oder zur Umkonfiguration der Software auf den Knoten genutzt werden. Die Maßnahme wird dabei immer von außen initiiert, zum Beispiel durch einen Administrator. Daher muss der Verwalterknoten auch nicht ständiger Bestandteil des Netzwerks sein. Er kann speziell für die Umkonfiguration in das System eingebracht und, nach dessen Veränderung, wieder entfernt werden.

Um- und Neukonfigurationen können notwendig sein, wenn das System für verschiedene Einsatzzwecke vorgesehen ist. Wird der Einsatzzweck verändert, so ändern sich die Anforderungen und die Aufgaben, die das System erfüllen muss. Daher muss auch die Software auf den Knoten verändert werden. Ein Beispiel hierfür ist eine Maschinensteuerung in einer Produktionsanlage. Während des normalen Betriebs wird kein Verwalter benötigt. Soll jedoch die Produktion auf ein anderes Produkt umgestellt werden, so müssen installierte Steuerungsprogramme entfernt und neue installiert werden. Bei der Deinstallation der alten Anwendung sichert der Verwalter den Zustand, beispielsweise Kalibrierungsdaten, um ihn bei der nächsten Installation der Anwendung wieder einspielen zu können[1]. Bei der Installation der neuen Software kann ein Administrator festlegen, wo die einzelnen Komponenten installiert werden sollen. Der Verwalter unterstützt ihn dabei, indem er mögliche Probleme aufzeigt.

Neben einem veränderten Einsatzzweck kann auch das Einspielen einer neuen Programmversion ein Grund sein, den Verwalter in Anspruch zu nehmen. Die neue Programmversion stellt zusätzliche Funktionalitäten bereit oder beseitigt vorhandene Fehler. Für einen Sensorknoten im Bereich der Datenerfassung möchte man beispielsweise die Filterfunktion zur Vorverarbeitung durch eine andere austauschen, da eine effizientere Version entwickelt wurde. Die neue Version ist jedoch sehr ähnlich zu der vorherigen Version, daher sollen möglichst alle Einstellungen und Daten bei der Installation erhalten bleiben. Der Verwalter analysiert den Zustand und gibt Auskunft darüber, an welchen Stellen die Übernahme des Zustands möglicherweise nicht automatisch funktioniert. Der Administrator kann dann eine Entscheidung treffen und eine halb automatische Übernahme der Einstellungen vornehmen.

Der Verwalter kann bei all diesen Installationsszenarios auch helfen, eine geeignete Verteilung der Anwendung zu finden. Vergrößert sich die Anwendung, beispielsweise durch ein Update, so kann der Fall eintreten, dass sie nicht mehr vollständig auf dem ursprünglichen Knoten installiert werden kann. Anwendungen, die spezielle Eigenschaften des Knotens benötigen, müssen jedoch dort installiert werden. Der Verwalter hilft hierbei, indem er die Anwendung in Module unterteilt, um bestimmte Module, welche nicht ortsgebunden sind, auf andere Knoten auszulagern. Am ursprünglichen Ort wird ein Stellvertreter installiert, der die Aufrufe mittels Fernaufruf weiterleitet. Dies ist besonders für selten benutzte Funktionen sinnvoll.

3.1.3 Verwalter als Dienstleister

Ist der Verwalter dauerhaft in das System integriert, so können die oben beschriebenen Einsatzmöglichkeiten noch flexibler angewandt werden. Da davon ausgegangen wird, dass der Verwalter jederzeit verfügbar ist, kann das System Umkonfigurationen auch selbst initiieren. Zur optimalen Erfüllung der Aufgaben konfiguriert sich das System dann dynamisch und bedarfsgerecht selbst um. Der Verwalter tritt dabei als Dienstleister auf, der sowohl die Konfigurationsaufgaben übernehmen als auch selbst aktiv eingebunden werden kann. Zusätzlich kann ein Verwalter dann zwei grundlegende Dienste zur Verfügung stellen:

[1] Dieser Vorgang wird später als *Parken* bezeichnet.

- Bereitstellen von Code
- Bereitstellen von Funktionalität

Bereitstellen von Code

Das Bereitstellen von Code ermöglicht die Installation von Anwendungskomponenten bei Bedarf. Ein Bedarf kann zum Beispiel durch eine Änderung des Einsatzzweckes entstehen. Im Gegensatz zu einem Verwalter, der nur zeitweise verfügbar ist und daher erst an das System angeschlossen werden muss, kann die Installation hier automatisch ohne Einwirkung durch einen Administrator erfolgen.

Als Beispiel dient ein Kfz-Navigationssystem, welches das CD-Laufwerk des MP3-Autoradios mitverwendet um Kartenmaterial einzulesen. In der Startkonfiguration bietet das Navigationssystem lediglich Richtungshinweise in Form kleiner Pfeile an. Die Kartendarstellung ist nicht installiert, um beispielsweise den Startvorgang zu beschleunigen. Wählt der Anwender nun die Kartendarstellung aus, so würde der Verwalter den entsprechenden Code zu diesem Zeitpunkt installieren. Die Ressourcen, die das installierte Kartendarstellungsmodul benötigt, werden so erst belegt, wenn die Komponente auch tatsächlich verwendet wird.

Die Bereitstellung von Code lässt sich noch sinnvoller einsetzen, wenn eine Komponente ausgetauscht wird. Bei einem dauerhaft vorhandenen Verwalter kann der Austausch einer Komponente je nach Betriebsmodus auch fest eingeplant und daher zur Laufzeit von den Mikrocontrollern angefordert werden. So könnte eine Komponente durch eine andere ersetzt werden, weil die vorhandenen Ressourcen nicht ausreichen, um beide Komponenten gleichzeitig zu betreiben. Im Beispiel des Navigationssystems können die Komponenten zur 2D–Darstellung der Karte durch den Code zur 3D–Darstellung ausgetauscht werden, sobald der Benutzer umschaltet. Da dann nicht beide Komponenten gleichzeitig auf dem Mikrocontroller vorhanden sein müssen, können kleinere und günstigere Controller eingesetzt werden.

Auch das automatische Deinstallieren von Komponenten kann vorteilhaft sein, wenn die freigegebenen Ressourcen sinnvoll durch andere Komponenten genutzt werden können. Da die Navigationssoftware das CD-Laufwerk benötigt, um Kartenmaterial zu lesen, kann die MP3-Abspielsoftware während des Navigationsbetriebs deinstalliert werden. Der freie Speicherplatz kann zur Ablage zusätzlicher Symbole der Kartendarstellung genutzt werden.

Bereitstellen von Funktionalität

Wie bereits beschrieben wurde, kann eine Anwendung, die als Ganzes zu groß für den Mikrocontroller ist, verteilt installiert werden. Einige Funktionalitäten können dann von anderen Knoten oder vom Verwalter bereitgestellt werden. Mithilfe eines permanent verfügbaren Verwalters ist man in der Lage die Verteilung zusätzlich zur Laufzeit den Bedürfnissen anzupassen. Während der Startphase einer Anwendung werden zum Beispiel relativ häufig neue Objekte und Aktivitätsträger erzeugt. Später wird diese Funktionalität kaum mehr in Anspruch genommen. Nach der Startphase kann die Komponente somit auf den Verwalter oder einen anderen Knoten ausgelagert werden. Dies hat den Vorteil, dass die dadurch frei werdenden Ressourcen durch Komponenten genutzt werden können, die häufiger benötigt werden. Die ausgelagerte Funktionalität steht jedoch weiterhin zur Verfügung, da sie von einem anderen Knoten erbracht wird.

Durch die dynamische und automatische Konfiguration und Veränderung der Anwendung können die Ressourcen eines Mikrocontrollers deutlich flexibler genutzt werden. Durch dynamisches und transparentes Entfernen und Hinzufügen von Code können so Anwendungen auf Knoten ausgeführt

werden, welche ohne die Unterstützung durch einen Verwalter gar nicht auf dem entsprechenden Knoten realisiert werden können, da die vorhandenen Ressourcen nicht ausreichen.

3.2 Architektur und Aufgaben eines verwalteten Knotens

Ein verwalteter Knoten muss eine Softwarekomponente besitzen, um die Verwaltung von einem anderen Knoten aus zu ermöglichen. Bevor wir die Aufgaben dieser Komponente genauer betrachten, soll jedoch zunächst der allgemeine Aufbau der Software auf einem Mikrocontroller beleuchtet werden. Auf dieser Grundlage kann die hier benötigte Komponente besser eingeordnet werden. Außerdem wird dadurch deutlicher, welcher Teil der Software hauptsächliches Ziel der Verwaltungsoperationen ist.

3.2.1 Software auf einem Mikrocontroller

Die Software auf einem Mikrocontroller kann, ähnlich wie die Software auf einem Personalcomputer, konzeptionell in zwei Schichten aufgeteilt werden: die Anwendungs- und die Systemschicht. Die Anwendungsschicht enthält die vom Benutzer gewünschte Programmlogik und setzt auf der Systemschicht auf. Die Systemschicht, welche bei Desktop-Computern meist das Betriebssystem darstellt, kann bei Mikrocontrollern sehr unterschiedlich ausfallen.

Im einfachsten Fall ist die Systemschicht für die Initialisierung der Hardware zuständig und bietet eine Schnittstelle an, um die vorhandenen Peripheriegeräte anzusteuern. Ist der verwaltete Knoten größer, sodass auf dem Knoten mehr als eine Anwendung gleichzeitig ausgeführt werden kann, so wird eine komplexere Systemschicht benötigt, deren Aufgabe auch die Verwaltung der vorhandenen Ressourcen einschließt. In diesem Fall kann man schon von einem Betriebssystem reden, welches sich um die Zuteilung der globalen Ressourcen kümmert und, soweit möglich, Schutzmechanismen anbietet, um die Anwendungen voneinander zu isolieren. Soll das nötige Wissen über die Details der Hardware bei der Anwendungsentwicklung weiter verringert werden, so müssen Systemschichten weitergehende Dienste und Abstraktionen anbieten. Beispiel hierfür ist die automatische Verwaltung der Ressourcen innerhalb einer Anwendung oder die Abstraktion der Hardware und des Systems durch generische Schnittstellen. In dieser Arbeit fassen wir alle Arten von Systemschichten, die eine gewisse Abstraktion für die Anwendungsentwicklung bereitstellen, unter dem Oberbegriff *Laufzeitumgebung* zusammen.

3.2.2 Softwarearchitektur eines verwalteten Knotens

Um Unterstützung von einem Verwalter zu bekommen, muss ein verwalteter Knoten in der Lage sein, mit dem Verwalter zu kommunizieren. Hierzu wird eine Kommunikationsverbindung benötigt und eine Software zur Steuerung der Kommunikation, die auf dem Knoten installiert sein muss. Zu den generellen Aufgaben dieser Komponente zählt das Weiterleiten von Anfragen der Anwendung an den Verwalter und das Umsetzen der Kommandos des Verwalters sowie die Koordination der Operationen mit dem laufenden System.

Die Anwendung wird sich an den Verwalter wenden, wenn beispielsweise eine Funktionalität benötigt wird, die nicht lokal ausgeführt werden kann. Zur Koordination mit dem laufenden System überträgt der Knoten Informationen über den aktuellen Systemzustand an den Verwalter, sodass dieser seine Sicht auf das System aktualisieren kann, bevor er eine Aktion, wie zum Beispiel das Installieren eines Moduls, ausführt. Führt der Verwalter eine Aktion durch, so sendet er Kommandos an den Knoten, der sie umsetzt. Um den Knoten möglichst wenig zu belasten, sind die Kommandos sehr einfach gehalten und beschränken sich meist auf das Auslesen oder Beschreiben einer Speicherstelle. Abbildung 3.1 zeigt die schematische Darstellung der Software auf einem verwalteten Knoten.

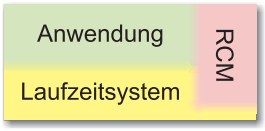


Abbildung 3.1: Architektur eines verwalteten Knotens

Zentraler Bestandteil auf dem verwalteten Knoten ist eine Softwarekomponente, die als Fernwartungsmodul (*Remote Configuration Module*, *RCM*) bezeichnet wird. Sie führt die Kommunikation mit dem Verwalter durch. Das RCM kann Code über die Kommunikationsschnittstelle empfangen und ihn auf dem Knoten installieren. Neben Code kann das RCM auch Daten in das Gerät einspielen oder auslesen. Details zum genaueren Ablauf werden in Kapitel 4, Abschnitt 4.5.7 vorgestellt.

Das RCM stellt zum Teil eine Anwendung dar, da es das Kommunikationsprotokoll mit dem Verwalter abwickelt und dazu die Kommunikationsschnittstelle der Laufzeitumgebung verwendet. Es ist jedoch sehr eng mit der Laufzeitumgebung verbunden, da Veränderungen am installierten Code immer mit dem laufenden System koordiniert werden müssen, und kann daher auch als Teil der Laufzeitumgebung angesehen werden.

3.3 Architektur und Aufgaben eines Verwalters

Im Allgemeinen hat der Verwalter die Aufgabe, einen kleinen Mikrocontroller beziehungsweise die Software auf einem solchen Knoten zu verwalten. Das Augenmerk ist dabei besonders auf die Anpassung der Laufzeitumgebung gerichtet. Daher muss der Verwalter Informationen über die Laufzeitumgebung haben. Zusätzlich benötigt er die Möglichkeit, auf einen entfernten Knoten einzuwirken und Mechanismen, um die Software zu verändern. Abbildung 3.2 zeigt eine grobe Übersicht der Architektur. Die Aufgaben sind in drei Komponenten eingeteilt:

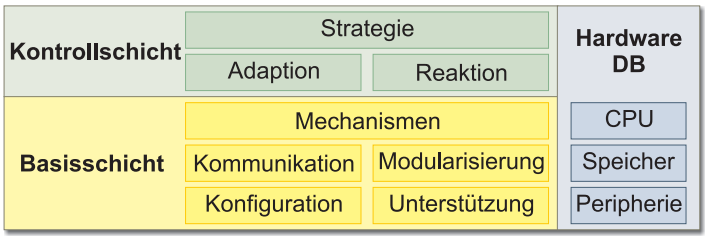


Abbildung 3.2: Architektur eines Verwalters

Die *Basisschicht* stellt die allgemeine Infrastruktur bereit, um Software auf einem entfernten Knoten zu verwalten. Darauf aufbauend steuert und unterstützt eine *systemspezifische Kontrollschicht* die Verwaltung durch detaillierte Informationen über die eingesetzte Laufzeitumgebung. Daneben gibt es eine *Hardwaredatenbank*, welche Informationen über die verwalteten Knoten vorhält, aktualisiert und bereitstellt.

3.3.1 Basisschicht

Die Basisschicht stellt grundlegende Mechanismen zur Konfiguration der Software auf einem verwalteten Knoten bereit. Diese Basismechanismen sind unabhängig von der verwalteten Software und können

daher universell eingesetzt werden, sowohl zur Veränderung der Anwendung als auch zur Anpassung der Laufzeitumgebung. Dem detaillierten Aufbau dieser Schicht widmet sich Kapitel 4. Die Aufgaben dieser Schicht können grob wie folgt charakterisiert werden:

Kommunikation mit einem verwalteten Knoten. Um Konfigurationsaufgaben durchzuführen, muss eine Kommunikationsmöglichkeit mit dem verwalteten Knoten bestehen. Die Basisschicht stellt hierzu eine Schnittstelle zur Verfügung die sowohl intern genutzt wird als auch extern für die anderen Schichten zur Verfügung steht. Außerdem stellt die Basisschicht ein Gerüst bereit, um Anforderungen von einem verwalteten Knoten an die entsprechende Stelle der Kontrollschicht weiterzuleiten.

Modularisierung der Software. Zur flexiblen Verwaltung der Software wird sie in kleine Einheiten, sogenannte *Module*, aufgeteilt. Die Modularisierung entscheidet hierbei, in welcher Granularität der Code verwaltet wird. Das kann auf Dateibasis erfolgen oder feingranularer auf Funktionsebene. Das Ergebnis der Aufteilung sind Module mit definierten Schnittstellen und Abhängigkeiten.

Basismechanismen zur Konfiguration der Software. Auf Basis der erstellten Module werden Mechanismen angeboten, um Module auf einem Knoten zu installieren, sie gegen andere auszutauschen oder zu entfernen.

Konsistenzsicherung auf dem verwalteten Knoten. Wird die Software auf einem verwalteten Knoten verändert, so muss dabei sichergestellt werden, dass wieder ein konsistenter Zustand entsteht. Das heißt, es muss beispielsweise dafür gesorgt werden, dass für Module, welche auf einem Mikrocontroller installiert werden, immer alle Abhängigkeiten erfüllt sind. Wäre eine Abhängigkeit nicht erfüllt, so könnte das Modul auf dem Mikrocontroller nicht ausgeführt werden, da es undefinierte Auswirkungen nach sich ziehen würde.

Infrastruktur für dynamische Unterstützung. Mithilfe der Basismechanismen zur Konfiguration können Unterstützungen realisiert werden wie beispielsweise das bedarfsgesteuerte Installieren oder das entfernte Ausführen von Funktionen. Diese Unterstützung stellt die Basisschicht in Form eines Gerüsts zur Verfügung, um den Einsatz durch die Kontrollschicht zu vereinfachen und zu unterstützen.

3.3.2 Kontrollschicht

Während die Basisschicht hauptsächlich Mechanismen bereitstellt, um das System zu verändern, ist die Hauptaufgabe der Kontrollschicht, diese Mechanismen einzusetzen, um das System zu verwalten, anzupassen und zu kontrollieren. Die genaue Vorgehensweise in der Kontrollschicht ist dabei stark vom Einsatzzweck abhängig. Dennoch lassen sich Teilaufgaben herauskristallisieren:

Optimale Konfiguration der Laufzeitumgebung für eine Anwendung. Als Ausgangsposition oder nach einer Veränderung muss eine neue optimale Konfiguration der Laufzeitumgebung gefunden werden.

Eine Laufzeitumgebung besteht üblicherweise aus verschiedenen Diensten, wobei manche Dienste in optimierten Varianten für Spezialfälle vorliegen können. Ziel ist es, die Dienste und Varianten auszuwählen, die von der Anwendung benötigt werden und sie entsprechend der verfügbaren Ressourcen auf dem oder den Knoten zu verteilen. Für Dienstvarianten muss dabei vorgegeben sein, unter welchen Bedingungen und für welche Fälle sie eingesetzt werden können.

Adaption der Laufzeitumgebung bei Veränderung der Anforderungen. Veränderte Anforderungen sollen eine Umkonfiguration der Laufzeitumgebung nach sich ziehen, um den neuen Zielen gerecht zu werden.
Um festzustellen, ob eine Adaption der Laufzeitumgebung erforderlich ist, müssen die Voraussetzungen und Bedingungen, die zur Auswahl der ursprünglichen Konfiguration geführt haben, bekannt sein. Treffen sie nicht mehr zu, so muss die Konfiguration angepasst werden. Der Zustand der Laufzeitumgebung soll dabei von der vorherigen Konfiguration in die neue Konfiguration transferiert werden. Je nach Umfang der Adaption kann es sich dabei um eine einfache Aufgabe handeln oder um einen komplexen Transformationsvorgang.

Erfassung von Veränderungen. Veränderungen der Anforderungen müssen erkannt werden, um eine angemessene Reaktion zu veranlassen.
Anforderungen können durch äußere Einflüsse oder durch den Ablauf der Anwendung verändert werden. Ein äußerer Einfluss stellt zum Beispiel das manuelle Eingreifen eines Administrators dar. Hierfür muss die Kontrollschicht geeignete Eingabemöglichkeiten bereitstellen. Änderungen der Anforderungen durch den Programmablauf lassen sich an der Ausführung von bestimmten Programmteilen festmachen. Dort können Benachrichtigungsmechanismen in die Anwendung integriert werden, welche die Kontrollschicht anstoßen, eine Adaption durchzuführen.

Zur Erfüllung dieser Aufgaben sind in der Kontrollschicht Strategien hinterlegt, welche entscheiden, wie die Mechanismen der Basisschicht anzuwenden sind. Um eine sinnvolle Konfiguration zu erstellen, muss die Strategie nach dem Verhalten der Software ausgerichtet sein. Daher enthält die Kontrollschicht semantisches Wissen über die Software bzw. die Laufzeitumgebung und ist somit anwendungsabhängig.

Als Grundlage der Entscheidungsfindung kann die Infrastruktur eine abstrakte Sicht des Systems zur Verfügung stellen. Dazu müssen die verfügbaren Knoten und deren aktuelle Konfiguration sowie der gesamte Code, der im Netzwerk ausgeführt wird, bekannt sein. Informationen über die verfügbaren Knoten und deren Eigenschaften werden durch die Hardwaredatenbank bereitgestellt. Die aktuelle Konfiguration der Knoten wird durch die Basisschicht erfasst und abgestimmt. Aufgabe der Kontrollschicht ist es, die Sicht durch die Verwaltung des Codes zu vervollständigen und es somit zu ermöglichen, eine Strategie zu implementieren.

Die Realisierung der Strategie kann auf unterschiedliche Arten erfolgen:

- Durch die interaktive Mitwirkung eines Administrators.
  Hierbei wird eine grafische Benutzeroberfläche angeboten, die das System grafisch darstellt. Dadurch kann ein Überblick vermittelt werden, welche Module oder Komponenten auf welchem Knoten installiert sind. Ausgehend von dieser aktuellen Sicht bietet die Oberfläche Möglichkeiten an, das System interaktiv zu beeinflussen und zu verändern.

- Automatische Adaption auf Basis einer hinterlegten Strategie.
  Neben einer interaktiven Konfiguration kann auch eine Strategie implementiert werden, die automatisch auf veränderte Anforderungen und Einflüsse reagiert. Beispielsweise können automatisch verschiedene Varianten von Diensten gegeneinander ausgetauscht werden, um immer die optimale Variante für die jeweilige Situation zu nutzen. Für eine einfache Realisierung wird dem System eine Rangfolge der Varianten vorgegeben. Anhand der Voraussetzungen für den Einsatz einer Variante und den Rahmenbedingungen, unter denen sie einsetzbar ist, wählt das System dann die Beste in der Rangliste aus.
  Soll auch das automatische Auslagern oder Verschieben von Modulen berücksichtigt werden,

so wird die Strategie deutlich komplexer. Hier können beispielsweise Regeln zur Verteilung der Komponenten vorgegeben sein, die von der Strategie umgesetzt werden, um die möglichen Konfigurationen auf einige sinnvolle einzuschränken.

- Anwendungsgesteuerte automatische Adaption.
  Schließlich kann die Strategie auch Teil der Anwendung selbst sein. Die Anwendung enthält dazu ein besonderes Modul, welches eine Schnittstelle anbietet, um Informationen über die aktuelle Konfiguration des Systems und die Auslastung der einzelnen Knoten abzufragen. Die Anwendung kann auf Basis dieser Informationen eine Umkonfiguration des Systems beauftragen.

Durch die starke Anwendungsabhängigkeit der Kontrollschicht ist es nicht möglich, eine feste Architektur anzugeben. In Kapitel 5 werden die Möglichkeiten einer Kontrollschicht für eine Java-Laufzeitumgebung vorgestellt. Dabei wird auch die Vorgehensweise einer möglichen Strategie deutlich.

3.3.3 Hardwaredatenbank

Die Hardwaredatenbank hat die Aufgabe, den anderen Schichten Informationen über die vorhandenen Knoten zur Verfügung zu stellen. Die Informationen betreffen in erster Linie die Knoten, die aktuell verwaltet werden. Darüber hinaus soll die Hardwaredatenbank aber auch Informationen über Knoten enthalten, die dem System neu hinzugefügt werden und potenziell verwendet werden können. Hierdurch kann die Kontrollschicht eine Sicht der Umgebung aufbauen und die Adaptionsstrategie auf eine globalere Sicht stützen. Die Basisschicht benötigt die Eigenschaften der Knoten, um die entsprechenden Werkzeuge auszuwählen oder die Mechanismen zu parametrisieren. Außerdem ist die Hardwaredatenbank dafür verantwortlich, die aktuelle Konfiguration zu einem Knoten zu verwalten, was besonders bei temporären Verwaltern eine Rolle spielt. Im Folgenden werden die Aufgaben der Hardwaredatenbank genauer beschrieben.

Bereitstellung der Knoteneigenschaften Für jeden Knoten müssen die wichtigsten technischen Eigenschaften zur Verfügung stehen, sodass die Adaption darauf abgestimmt werden kann. Benötigte Informationen sind:

- Architekturtyp und Geschwindigkeit der CPU
  Der Typ ist für die Basisschicht entscheidend, um festzustellen, ob Code für die entsprechende Architektur verfügbar ist. Außerdem müssen die Werkzeuge (z. B. dynamischer Binder) entsprechend dem Typ ausgewählt werden. Die Geschwindigkeit der CPU ist für die Kontrollschicht relevant, da diese nicht-funktionale Eigenschaft die Adaptionsstrategie oder die Verteilung beeinflussen kann.
- Speicherausstattung
  Über den verfügbaren Speicher muss bekannt sein, von welchem Typ (z. B. RAM, Flashspeicher, ...) er ist und wie er genutzt werden kann (für Code, Daten oder beides). Darüber hinaus ist die Menge des Speichers und der Adressbereich für die Basismechanismen und die Strategie relevant.
- Integrierte und extern angeschlossene Peripherie
  Mikrocontroller verfügen über integrierte Peripheriegeräte. Um diese der Anwendung zur Verfügung zu stellen, muss die Kontrollschicht entsprechenden Code installieren.
  Ebenso verhält es sich mit externer Peripherie, wie zum Beispiel Sensoren oder Aktoren. Sie sind mit einem integrierten Peripheriegerät, beispielsweise einem A-/D-Wandler, verbunden und benötigen ebenfalls zusätzlichen Code zur Ansteuerung.

Darüber hinaus haben Peripheriegeräte Einfluss auf die Verteilung, da sie zum Teil einzigartige Ressourcen darstellen, die es nötig machen, dass eine Anwendungskomponente auf einem bestimmten Knoten ausgeführt werden muss.

Bereitstellung der aktuellen Konfiguration Neben den technischen Eigenschaften eines Knotens benötigt die Basisschicht auch detaillierte Informationen über die Software, die auf dem Knoten installiert ist. Diese Informationen bezeichnen wir als aktuelle Konfiguration. Sie wird benötigt, um den initialen Zustand des Verwalters aufzubauen, sobald ein Verwalter an einen verwalteten Knoten gekoppelt wird.
Um diese Unterstützung für das An- und Abkoppeln von temporären Verwaltern zu realisieren, muss beim Abkoppeln eines verwalteten Knotens vom Verwalter die derzeit aktuelle Konfiguration von der Basisschicht in die Hardwaredatenbank geschrieben werden.

Verwaltung neu hinzugefügter Knoten Um eine aktuelle Sicht des Systems wiederzugeben, muss die Hardwaredatenbank auch Informationen über Knoten bereitstellen, die dem Netzwerk dynamisch hinzugefügt werden. Dazu muss sie aktualisiert werden, sobald ein neuer Knoten bereitsteht.
Da neue Knoten auch neue Ressourcen darstellen, ist es auch ein Ziel, sie möglichst schnell in das existierende System zu integrieren. Zu diesem Zweck ist die Benachrichtigung der Kontrollschicht vorgesehen, welche daraufhin eine Umkonfiguration anstoßen kann und den neuen Knoten bei zukünftigen Entscheidungen berücksichtigt.

3.3.3.1 Mögliche Realisierung der Hardwaredatenbank

Eine genaue Implementierung der Hardwaredatenbank soll im Rahmen dieser Arbeit nicht dargelegt werden. Hier sollen aber einige Konzepte zur Realisierung angesprochen werden.

Generell sind zwei grundsätzlich verschiedene Herangehensweisen möglich:

- Eine lokale Datenbank bei jedem Verwalter
- Ein gemeinsam genutzter Datenbankdienst in einem Netzwerk für alle Verwalter

Der Vorteil einer lokalen Datenbank liegt darin, dass sie relativ einfach zu realisieren ist und Zugriffe ohne Mehraufwand durchgeführt werden können. Eine lokale Hardwaredatenbank ist daher bei Verwendung eines einzigen Verwalters die beste Wahl. Arbeiten mehrere Verwalter in einem System zusammen, müssen die lokalen Datenbanken unter Umständen miteinander abgeglichen werden, insbesondere wenn ein verwalteter Knoten von einem Verwalter zu einem anderen Verwalter wechseln soll. Dieser Abgleich der Konfiguration würde bei einem gemeinsam genutzten Datenbankdienst implizit erfolgen, indem der ursprüngliche Verwalter die aktuelle Konfiguration in der Datenbank ablegt und der neue Verwalter diese bei Ankopplung des Gerätes dort ausliest.

Bei mehreren kooperierenden Verwaltern bietet sich daher ein gemeinsam genutzter Dienst an, allerdings ergeben sich die Vor- und Nachteile erst aus der konkreten Realisierung. Man kann grob zwei Ansätze unterscheiden:

- Eine zentrale Datenbank
- Eine verteilte Datenbank

Für eine zentrale Datenbank benötigt man einen zentralen Knoten, der diesen Dienst anbietet. Ein solcher Ansatz hat die üblichen Vor- und Nachteile eines zentralen Dienstes. Vorteil ist, dass die Daten wirklich nur einmal vorgehalten werden. Außerdem ist der Ort des Dienstes fest, sodass der Dienst leicht zu erreichen ist und sein Ort statisch konfiguriert werden kann. Allerdings hat der Ausfall des Dienstes Einfluss auf alle Verwalter. Zudem kann der zentrale Dienst zum Engpass werden, wenn zu viele Verwalter gleichzeitig Daten abfragen oder eintragen wollen.

Bei einem verteilten Datenbankdienst kann jeder beteiligte Verwalter einen Teil der Daten abspeichern. Dabei gibt es noch unterschiedliche Ansätze, wie die Zuordnung der Daten zu den einzelnen Knoten vorgenommen und wie eine globale Suche auf dem Datenbestand realisiert wird.

Eine Möglichkeit ist der Einsatz von Peer-to-Peer-Techniken [LCP+05]. Dabei gibt es den Ansatz, ein strukturiertes Netzwerk zu bilden oder auf einem unstrukturierten Netzwerk zu arbeiten. Auf einem strukturierten Netzwerk wird eine globale Abbildung definiert, um zu einem Suchschlüssel den Knoten zu finden, der das gewünschte Datum enthält. Nachteil dabei ist, dass der Aufenthaltsort der Daten nicht selektiv festgelegt werden kann und somit die Daten nicht auf den Knoten abgelegt werden können, von denen sie am häufigsten benötigt werden. Bei einem unstrukturierten Netzwerk werden Suchanfragen an alle bekannten Knoten gesendet, welche ihrerseits die Anfrage weiterleiten, falls sie das entsprechende Datum nicht besitzen. Da der Ort eines Datums hier frei bestimmbar ist, eignet sich dieser Ansatz besser für die betrachteten Szenarios. Skalierungsprobleme, die dieser Ansatz mit sich bringt, wirken sich erst in großen globalen Netzwerken aus, sind bei kleinen lokalen Netzwerken jedoch zu vernachlässigen.

Eine Alternative zur Organisation als Peer-to-Peer Dienst stellt ein zentraler Koordinator dar, der den Aufenthaltsort der Daten kennt, verwaltet und einen Suchdienst anbietet. Die Daten selbst liegen dann je nach Strategie beispielsweise bei dem Knoten, der sie zuletzt gebraucht oder verändert hat. Dieser Ansatz weist allerdings wieder ähnliche Nachteile, wie ein komplett zentraler Dienst auf.

Abschließend lässt sich feststellen, dass die Auswahl der verwendeten Technik vom Einsatzzweck abhängt. Für stark dynamische Systeme ist ein Ansatz mit Peer-to-Peer Technik sinnvoll. In den meisten Fällen wird jedoch eine Hardwaredatenbank mit zentraler Komponente ausreichend sein.

3.3.3.2 Erfassung der Eigenschaften

Das Hinzufügen von neuen Knoten und das Erfassen der Eigenschaften und Daten eines Knotens kann manuell oder automatisch erfolgen. Die manuelle Erfassung eignet sich für Netze, bei denen selten Knoten hinzugefügt oder ausgetauscht werden. Hier kann der Administrator die Daten manuell für jeden neuen Knoten in die Datenbank eintragen. Für dynamische Netze ist ein automatischer Ansatz sinnvoller.

Um die Daten automatisch zu erfassen, können selbstbeschreibende Knoten eingesetzt werden. Ansätze zur Selbstbeschreibung von Sensoren existieren zum Beispiel mit *IEEE 1451* [IEE98]. Dabei wird eine genaue Beschreibung der Eigenschaften der vorhandenen Sensoren am Knoten gespeichert und kann von dort ausgelesen werden. Eine umfangreichere Selbstbeschreibung wird im Rahmen des Kommunikationsprotokolls *TTP/A* [PE03] angesprochen. Hier stehen zusätzlich Angaben zum Mikrocontroller selbst zur Verfügung. Allerdings werden diese Daten nicht unbedingt auf dem Gerät gespeichert, sondern es findet ein Abgleich mit einer Datenbank statt, in die mit der eindeutigen Identifikation des Knotens indiziert wird. Ein ähnlicher Ansatz kann auch mit der hier vorgestellten Hardwaredatenbank realisiert werden.

3.3.3.3 Erfassen von neuen Knoten

Das automatische Erfassen von neuen Knoten hängt stark vom eingesetzten Kommunikationsmedium ab. Systeme, die über ein Netzwerk gekoppelt werden, erlauben eine hohe Dynamik. Außerdem kann das Kommunikationsmedium von allen beteiligten Knoten gemeinsam verwendet werden. Hier können Ansätze auf der Basis von Broad- oder Multicast-Nachrichten eingesetzt werden. Solche Suchdienste werden zum Beispiel in Systemen wie *Jini* [Wal99], *Universal Plug and Play* [UPn03] oder *Bonjour* [SCK06, CK06b, CK06a] eingesetzt. Diese Systeme sind zwar für sehr kleine Knoten unter Umständen nicht direkt nutzbar, da zum Teil umfangreiche Zusatzdienste definiert werden, der generelle Ansatz lässt sich jedoch auch mit wenigen Ressourcen umsetzen: Der verwaltete Knoten meldet seine Verfügbarkeit mittels Broad- oder Multicast. Der Datenbankdienst reagiert auf die entsprechenden Nachrichten, fragt gegebenenfalls Informationen von dem neuen Knoten ab und fügt sie der Hardwaredatenbank hinzu.

Punkt-zu-Punkt Verbindungen wie beispielsweise RS232 werden meist weniger dynamisch genutzt. Hier kann auch nur der direkt beteiligte Verwalter einen neuen Knoten entdecken und einen Eintrag in der Hardwaredatenbank vornehmen. Dazu muss die Punkt-zu-Punkt Schnittstelle jedoch überwacht werden. Bei seltenen Veränderungen ist daher das manuelle Eintragen der Daten durch einen Administrator eine sinnvolle Alternative und stellt in der Regel keine Einschränkung dar.

Neben diesen Verbindungsarten können auch Bussysteme eingesetzt werden, um verwaltete Knoten an Verwalter anzubinden. Dabei ist die Erkennung neu angeschlossener Geräte oft durch die Hardware in Verbindung mit der Treibersoftware möglich.

Zusammenfassend lässt sich feststellen, dass es eine Reihe von Möglichkeiten gibt, wie die Hardwaredatenbank realisiert werden kann. Die Auswahl hängt dabei von vielen Faktoren ab, wie beispielsweise die Anzahl der Verwalter und verwalteten Knoten, die dynamische Fluktuation der Knoten und der Art des eingesetzten Kommunikationsmediums.

3.4 Zusammenfassung

In diesem Kapitel wurde das System zur Verwaltung eingebetteter verteilter heterogener Knoten vorgestellt. Zunächst wurde das Konzept der Verwalter und verwalteten Knoten eingeführt. Dabei werden ressourcenstarke Knoten des Systems als Verwalter für kleine, ressourcenschwache Knoten verwendet. Durch die Verwalter kann die Software auf einem kleinen Knoten manuell und automatisch den aktuellen Anforderungen angepasst werden.

Anschließend wurde die Software auf einem Mikrocontroller betrachtet und die Aufgaben angesprochen, die ein verwalteter Knoten erfüllen muss. Zuletzt wurde die Architektur der Software auf der Seite eines Verwalters vorgestellt. Hier wurden drei Bestandteile identifiziert, deren Aufgaben kurz dargelegt wurden.

4
Basisschicht eines Verwalters

In diesem Kapitel wird die Basisschicht zur Verwaltung der Software auf kleinen Knoten vorgestellt. Diese bietet vor allem allgemeine Dienste und Mechanismen an und stellt eine Infrastruktur zur Konfiguration der Software auf einem entfernten Knoten dar. Darauf aufbauend kann ein Verwalter für ein konkretes System erstellt werden. Dazu sind zum Teil detaillierte Informationen über das System notwendig, die durch eine *systemspezifische Kontrollschicht* bereitgestellt beziehungsweise realisiert werden. Mit der Kontrollschicht beschäftigt sich das nächste Kapitel. Hier wird zunächst die Analyse und Modularisierung der Software beschrieben. Anschließend werden grundlegende Mechanismen vorgestellt, die auf der Basis der modularisierten Software ausgeführt werden können. Zuletzt werden Details einiger Vorgehensweisen am Aufbau der Basisschicht verdeutlicht.

4.1 Aufgaben und Dienste der Basisschicht

Eine der Hauptaufgaben eines Verwalterknotens ist die Organisation und Verwaltung der Software auf einem verwalteten Knoten. Wie bereits in Abschnitt 3.2.1 beschrieben, umfasst die Software auf einem Knoten dabei die Anwendung und die Systemschicht. Zur Verwaltung werden diese Bestandteile nicht als ganze Blöcke behandelt, sondern in feingranulare Strukturen unterteilt. Diese Verwaltungseinheiten werden im Folgenden als *Module* bezeichnen. Die Form dieser Module wird in Abschnitt 4.2 genauer beschrieben.

Bei der Verwaltung von Modulen unterscheidet die Basisschicht nicht zwischen Laufzeitsystem und Anwendung: Alle Bestandteile werden gleichermaßen in Module unterteilt. Die Verwaltungs- und Konfigurationsoperationen der Basisschicht müssen daher systemunabhängig funktionieren und auf alle Module anwendbar sein. Die Anwendung der Operationen auf diese Module muss allerdings durch eine entsprechende Strategie gesteuert werden. Diese Strategie ist wiederum systemspezifisch und kann eine Unterscheidung zwischen Anwendungsmodulen und Systemmodulen treffen. Die Strategie ist in der Kontrollschicht definiert und kann automatische Konfigurationsänderungen vornehmen oder teilweise manuelle Interaktion mit einem Administrator erfordern.

Die Basisschicht stellt somit hauptsächlich *generische Operationen zur Verwaltung und Konfiguration* eines Knotens zur Verfügung. Diese Operationen sollen im Folgenden nochmals motiviert und

dargestellt werden. Auf Details zur Realisierung beziehungsweise der Implementierung wird später ab Abschnitt 4.3 eingegangen.

4.1.1 Installation von Modulen bei Bedarf

Bei der Analyse der Software erzeugt die Basisschicht einen Abhängigkeitsgraphen (siehe Abschnitt 4.4.3) und versucht daran festzustellen, welche Module benötigt werden. Für einige Module hängt es jedoch von Ereignissen zur Laufzeit ab, ob sie tatsächlich benötigt werden oder nicht. Da der Verwalter die Analyse vor der Laufzeit durchführt, kennt er diese Ereignisse nicht und wird solche Module folglich als notwendig einstufen.

Um diesen Fall zu optimieren, soll es möglich sein, dass Module erst bei Bedarf nachinstalliert werden. Anstelle des Moduls wird ein kleiner Stellvertreter in die Anwendung integriert, der den Verwalter benachrichtigt, sobald das Modul benötigt wird. Zu diesem späteren Zeitpunkt kann das Modul dann installiert werden. Es muss dabei so integriert werden, dass anschließend nicht mehr der Stellvertreter, sondern die eigentliche Funktionalität des Moduls verwendet wird. Außerdem darf der Zustand der Software nicht verändert werden. Das heißt, dass die Installation des Moduls transparent für die Software auf dem Mikrocontroller verläuft.

Vor der Installation sind einige Rahmenbedingungen zu überprüfen. So müssen die Abhängigkeiten des neuen Moduls erfüllt sein. Außerdem muss geprüft werden, ob ausreichend Speicherplatz auf dem Zielgerät zur Verfügung steht. Gegebenenfalls müssen weitere Module installiert werden, um neu entstehende Abhängigkeiten zu erfüllen, oder unbenutzte Module entfernt werden, um ausreichend Speicherplatz zu schaffen.

Diese Operation wird an mehreren Stellen durch die Kontrollschicht beeinflusst beziehungsweise gesteuert. Am Anfang steht die Information, dass ein Modul nicht in allen Situationen benötigt wird und somit die Entscheidung, welche Module erst bei Bedarf installiert werden sollen. Diese Entscheidung wird genauso von der Kontrollschicht getroffen wie die Aussage, welche Module deinstalliert werden sollen, falls der verfügbare Speicherplatz nicht zur Installation ausreicht.

4.1.2 Parken von Modulen

Als Gegenstück zum nachträglichen Installieren von Modulen möchte man sie auch wieder entfernen, sobald sie nicht mehr benötigt werden. Module können jedoch nicht ohne Weiteres entfernt werden, wenn sie von anderen installierten Modulen referenziert werden. Dennoch kann man sich ein Gegenstück zum bedarfsgesteuerten Installieren vorstellen. Diese Operation wird hier als *Parken* bezeichnet da im Gegensatz zu dem, was mit dem Begriff “Deinstallation” assoziiert wird, der Zustand des Moduls bis zu einer späteren Installation erhalten bleibt.

Beim Parken wird der Code der Funktionen, zusammen mit den Daten, die ausschließlich durch dieses Modul referenziert werden, vom Knoten entfernt. Dabei sichert der Verwalter den aktuellen Zustand der Daten, um das Modul zu einem späteren Zeitpunkt wieder transparent integrieren zu können. Anstelle des Moduls installiert er anschließend einen kleinen Stellvertreter, welcher die Basisschicht informiert, sobald das Modul wieder benötigt wird. Die spätere Reintegration des Moduls realisiert das bedarfsgesteuerte Installieren, bei dem auch der zuvor gesicherte Zustand wieder restauriert wird.

Das Parken von Modulen kann somit dazu genutzt werden, Module zeitweise vom verwalteten Gerät zu entfernen und die davon belegten Ressourcen für andere Aufgaben freizumachen. Besonders sinnvoll kann das Parken für Funktionalitäten eingesetzt werden, die nur phasenweise oder selten benutzt werden. Eine Anwendung, die beispielsweise je nach Tageszeit unterschiedliche Messwerte aufzeichnet, kann die unbenutzten Messfunktionen parken, um so mehr Platz für Messwerte zu haben.

In einem Betriebssystem können Dienste, welche nur relativ selten verwendet werden, wie zum Beispiel das Starten von neuen Anwendungen oder das Bereitstellen von neuen Aktivitätsträgern, geparkt werden, wenn sie in absehbarer Zeit nicht genutzt werden.

4.1.3 Ersetzen von Modulen

Beim Ersetzen von Modulen soll ein Modul durch ein anderes Modul ausgetauscht werden. Mit diesem Mechanismus ist es beispielsweise möglich, ein fehlerhaftes Modul durch eine neue, korrigierte Version zu ersetzen. Das neue Modul muss dazu schnittstellenkompatibel mit dem alten Modul sein. Üblicherweise ist auch die Funktionalität der Module gleich. Man kann diesen Mechanismus aber auch dazu nutzen, zwischen verschiedenen Modulen umzuschalten, welche dieselbe Aufgabe erledigen, jedoch auf unterschiedliche Art und Weisen und unter unterschiedlichen Rahmenbedingungen. In Abschnitt 5.4 werden solche Szenarios am Beispiel einer Java-Umgebung vorgestellt.

Das Austauschen von Modulen basiert auf denselben Mechanismen wie das Parken und Installieren. Das aktuelle Modul wird vom verwalteten Gerät entfernt, anschließend wird die neue Version installiert. Da sich die Arbeitsweise des Moduls verändert haben kann, muss neben dem Austauschen des Programmcodes möglicherweise auch eine Anpassung des Zustands erfolgen. Zu diesem Zweck wird mit dem Entfernen des alten Moduls auch der dazugehörende Zustand gesichert. Vor dem Installieren des neuen Moduls muss die Möglichkeit einer Zustandskonvertierung bestehen. Diese Konvertierung ist modulspezifisch und hängt unter anderem von der Version des alten Moduls als auch von der des neuen Moduls ab. Der Auftraggeber des Austausches, üblicherweise die Kontrollschicht, muss daher eine Konvertierungsfunktion zur Verfügung stellen. Der angepasste Zustand wird dann mit dem neuen Modul in den Knoten integriert.

4.1.4 Module als entfernten Dienst auslagern

Die bisher vorgestellten Verwaltungsoperationen dienten hauptsächlich dazu, die Softwarekonfiguration eines verwalteten Knotens zu erstellen und zu verändern. Der Verwalter kann aber auch aktiv in die Ausführung der Aufgaben mit einbezogen werden, indem Teile der Anwendung von ihm ausgeführt werden. Zu diesem Zweck können Module vom verwalteten Knoten entfernt werden, um sie stattdessen auf dem Verwalterknoten zu installieren. So ist es möglich, die Ressourcen des Verwalterknotens für den verwalteten Knoten zu nutzen. Auf einem verwalteten Knoten können dadurch Anwendungen ausgeführt werden, für welche die tatsächlich vorhandenen Ressourcen des Knotens nicht ausreichen würden.

Dies ist beispielsweise notwendig, falls die Software durch Weiterentwicklung mit der Zeit anwächst und nicht mehr vollständig von dem verwalteten Knoten ausgeführt werden kann. Anstatt Funktionen komplett zu entfernen, können sie durch Auslagern dennoch genutzt werden.

Ein anderes Szenario stellt die Nutzung einer Laufzeitumgebung dar, mit deren Hilfe kleine Knoten auf einem höheren Abstraktionsniveau dargestellt werden. Durch den Einsatz einer etablierten Laufzeitumgebung könnte man eine große Vereinheitlichung verschiedener Knoten erreichen. Diese Umgebungen sind jedoch ursprünglich für leistungsstärkere Knoten entwickelt worden (siehe Abschnitt 5.3.2). Das Auslagern von selten benötigten Funktionen ist eine Möglichkeit, eine solche Laufzeitumgebung auf jedem kleinen Knoten zur Verfügung zu stellen. Somit steht die komplette Laufzeitunterstützung der Umgebung zur Verfügung und kleine Knoten können theoretisch genauso programmiert werden wie große Knoten.

Das Verwenden eines Moduls als entfernter Dienst wird durch einen Fernaufruf realisiert. Anstelle des Moduls wird ein Stellvertreter in die Anwendung integriert. Hierzu wird der Mechanismus des

Austauschens verwendet, der in ähnlicher Form auch im Rahmen des Parkens eingesetzt wird, um ein Modul durch einen Stellvertreter zu ersetzen. Wird das Modul dann verwendet, so leitet der Stellvertreter den Aufruf an den Verwalter weiter. Dabei müssen die benötigten Daten zum Zielknoten transportiert werden. Da der Verwalter üblicherweise eine andere Hardwarearchitektur hat als die verwalteten Knoten, müssen die Daten entsprechend umgewandelt werden. Die Umwandlung, *Marshalling* genannt, wird dabei empfangsseitig durchgeführt. Das heißt, dass der Verwalter die Daten im Format des verwalteten Knotens empfängt und sie dann in sein eigenes Format umwandelt. Dadurch soll die Belastung des verwalteten Knotens möglichst gering gehalten werden.

Für die Durchführung des Marshallings werden die genaue Struktur und die genauen Typen der Daten und Parameter benötigt. Diese Information kann von außen, zum Beispiel in Form einer erweiterten Schnittstellenbeschreibung, zugeführt werden. Im Rahmen dieser Arbeit wird jedoch auch ein Mechanismus vorgestellt, die Informationen automatisch aus zusätzlichen compilergenerierten Informationen zu extrahieren (siehe Abschnitt 4.5.8).

4.2 Modularisierung

Um Software zu partitionieren oder zu verteilen, muss man sie in Abschnitte zerschneiden, sogenannte Verteilungseinheiten, die als Basis der Operation dienen. Nachfolgend soll die Aufteilung in solche Bausteine beschrieben werden. Dabei dient die Zergliederung nicht nur der Verteilung, sondern bildet auch die Ausgangsbasis, um die Struktur der Anwendung zu erfassen und zu analysieren. Die resultierenden kleinen Einheiten stellen die *Module* dar, die bereits in vorherigen Abschnitten angesprochen wurden.

Erstellt man eine Anwendung von Beginn an unter dem Gesichtspunkt der Modularisierung, so kann bereits in der Entwicklungsphase die Einteilung in Module vorgesehen werden. Eine Möglichkeit besteht darin, die Software mithilfe eines Komponentenmodells zu entwickeln. Dadurch wird man automatisch eine entsprechende Strukturierung in Komponenten vornehmen, welche dann unsere Module darstellen können. Komponentenmodelle werden in Abschnitt 4.2.2 kurz beschrieben.

Um ein modulares Design zu erstellen und die Modularität im Laufe der Entwicklung sicherzustellen, muss bei der Entwicklung zusätzlicher Aufwand in Form von Planungs- und Entwicklungszeit investiert werden. Diese Mehrkosten werden jedoch häufig nicht in Kauf genommen, da der Vorteil einer späteren Aufteilung in einzelne Einheiten bei der Entwicklung nicht hinreichend begründet werden kann.

Daher sollen für die in dieser Arbeit vorgestellte Basisinfrastruktur keine besonderen Modularisierungsanforderungen an die Software gestellt werden. Es soll stattdessen beliebige existierende Software verwendet werden können, auch wenn sie nicht vorwiegend mit dem Gedanken der Modularität entwickelt wurde.

Die erste Aufgabe des Systems ist somit die Aufteilung der Software in Module. Die Zergliederung soll dabei an funktionalen Grenzen der Software orientiert sein. Um automatisch eine sinnvolle Aufteilung vorzunehmen, ist inhaltliches Wissen über die Software nötig. Die Basissoftware eines Verwalters ist jedoch allgemein verwendbar und unabhängig von der eingesetzten Software. Semantische Informationen über die Funktion der Software werden erst durch die höhere Kontrollschicht dem System zugeführt. Der wichtigste Ansatzpunkt ist daher die Frage, an welchen Stellen man die Software in Module aufteilen kann.

Hier bieten die grundlegenden Programmstrukturen einen möglichen Ansatzpunkt, auch ohne explizites inhaltliches Wissen eine nachvollziehbare Aufteilung vorzunehmen. Eine grundlegende Strukturierung der meisten Programme ist durch ihre Organisation in einzelne Funktionen und Daten beziehungsweise, bei objektorientierter Programmierung, in Klassen und Objekte gegeben. Die Ba-

sissoftware nutzt genau diese implizit gegebene Struktur und nimmt eine Aufteilung der Software an diesen Grenzen vor.

Dieses Vorgehen wird in diesem Abschnitt genauer beschrieben. Zunächst werden die Anforderungen definiert. Anschließend werden verschiedene Möglichkeiten beschrieben, um zu Modulen zu gelangen und auf ihre Vor- bzw. Nachteile hingewiesen.

4.2.1 Anforderungen an Module

In Abschnitt 4.1 wurden die Konfigurationsoperationen dargelegt, die von der Basissoftware zur Modulverwaltung angeboten werden sollen. Um diese Dienste anzubieten, werden folgende Anforderungen an die Module beziehungsweise an die Aufteilung in Module gestellt:

- Die Abhängigkeiten eines Moduls müssen vollständig ermittelbar sein.
  Durch eine Aufteilung entstehen Abhängigkeiten zwischen den resultierenden Modulen. Ein automatisches System muss in der Lage sein, diese Abhängigkeiten zu erkennen. Dadurch können die Voraussetzungen für den Einsatz eines Moduls ermittelt werden. Außerdem kann festgestellt werden, wie stark ein Modul mit anderen Modulen vernetzt ist. Dies kann als Basis einer Bewertung dienen, wie viele Ressourcen das Parken oder die Auslagerung eines Moduls kostet.
  Ein Modul muss daher *selbstbeschreibend* sein. Das heißt, alle Abhängigkeiten müssen entweder explizit beschrieben sein oder sich aus dem Modul erkennen lassen.

- Ein Modul muss eine definierte Schnittstelle haben.
  Wird ein Modul auf einen anderen Knoten ausgelagert und dort ausgeführt, so müssen alle Parameter und Rückgabewerte zwischen den beiden Systemen transportiert werden. Um das zu ermöglichen, werden Informationen über die verwendeten Typen benötigt. Für ein automatisches System müssen die Typinformationen sehr detailliert sein. Größeninformationen reichen zwar für den Transport der Daten zwischen zwei Rechnern aus, allerdings nicht zur Anpassung der Daten zwischen verschiedenen Architekturen. Da die beteiligten Systeme jedoch von unterschiedlichen Architekturen sein können, muss der genaue Aufbau und die Repräsentation im Speicher zur Verfügung stehen.

- Module müssen zur automatischen Verarbeitung geeignet sein.
  Es wird gefordert, dass die Struktur der Module von einem Werkzeug weiterverarbeitet werden kann. Voraussetzung dazu ist, dass alle Informationen in maschinenlesbaren Formaten vorliegen. Ein Werkzeug soll die Module erkennen und beispielsweise nach der Vorgabe eines Verteilungsplans eine Anwendung aus den Modulen erstellen. Dabei müssen die Module automatisch verknüpfbar sein. Eine weitere Bedingung ist, dass automatisch verifiziert werden kann, ob die Abhängigkeiten der einzelnen Module erfüllt sind, um bereits im Vorfeld die Zulässigkeit einer Konfiguration zu prüfen.

- Module müssen für verschiedene Architekturen verfügbar sein.
  Da Module auf Knoten mit verschiedenen Architekturen eingesetzt werden sollen, muss eine Möglichkeit bestehen, sie für verschiedene Architektur zu erstellen. Dabei ist es wünschenswert, dass ein Modul in einem architekturunabhängigen Format definiert und automatisch in eine architekturspezifische Version umgewandelt wird.

- Die Aufteilung in Module muss nachvollziehbar sein.
  Wenn nicht alle Entscheidungen automatisch getroffen werden können, so soll ein Experte die

Möglichkeit haben, diese Entscheidungen zu treffen oder eine entsprechende Strategie in der Kontrollschicht zu verwirklichen. Als Experte eignet sich der Architekt oder der Entwickler der Software. Um auf ihn als Experten zurückgreifen zu können, muss er die automatische Zergliederung der Software nachvollziehen können. Nur wenn der Experte weiß, was die einzelnen Module darstellen, ist er in der Lage, mit ihnen zu arbeiten und Entscheidungen zu treffen.

- Die Aufteilung in Module soll unabhängig vom Entwicklungsprozess stattfinden.
  Um auch existierende Anwendungen in Module aufzuteilen, muss der Aufteilungsprozess weitestgehend unabhängig vom Entwicklungsprozess durchgeführt werden können. Eine Rückkopplung zum Entwicklungsprozess darf jedoch stattfinden und ist sogar gewünscht, da dadurch die Aufteilung verbessert werden kann. Die Unabhängigkeit vom Entwicklungsprozess fördert jedoch die vielseitige Anwendbarkeit des Ansatzes in unterschiedlichen Bereichen.

Das generelle Ziel der Aufteilung in Module ist es, die Software zu analysieren, um der Kontrollschicht eine strukturierte Sicht auf die Software geben zu können. Durch die Kontrollschicht kann dann externes Wissen über die Struktur des Programms hinzugefügt werden. Diese Informationen können verwendet werden, um die Software zu partitionieren, aber auch um Informationen zu extrahieren, die Einfluss auf die Verteilungsstrategie haben.

4.2.2 Komponentenmodelle

Wie bereits erwähnt, können Komponentenmodelle ein Hilfsmittel sein, eine Anwendung strukturiert zu erstellen und definierte Module zu erhalten, die in diesem Kontext als Komponenten bezeichnet werden [Szy98]. Es gibt verschiedene Motivationen, ein Komponentenmodell zu verwenden. Einer der wichtigsten Punkte dabei ist die Wiederverwendbarkeit der einzelnen Komponenten. Hinter diesem Ziel steckt der Ansatz, dass man neue Anwendungen aus vorgefertigten Komponenten zusammensetzen möchte.

Um das zu erreichen, müssen die Komponenten klar abgegrenzt und unabhängig voneinander sein. Die Verknüpfungspunkte der Komponenten müssen durch wohl definierte Schnittstellen und Abhängigkeiten beschrieben werden. Die Umgebung, in der eine Komponente verwendet werden kann, ist somit genau spezifiziert. Um von den Komponenten zu einer Anwendung zu gelangen, kann man dann die einzelnen Komponenten an ihren Schnittstellen miteinander verbinden und sie zu einer Anwendung kombinieren wie mit einer Art Baukasten.

Ein Komponentenmodell sorgt für die Kompatibilität der Komponenten, indem definiert wird, wie die Schnittstellen und Abhängigkeiten einer Komponente zu beschreiben sind. Anhand der Beschreibung kann dann festgestellt werden, ob Schnittstellen kompatibel sind und ob zwei Komponenten miteinander verbunden werden können. Die Beschreibung ist dabei in einem maschinenlesbaren Format abgelegt, sodass die Konsistenz von Verbindungen automatisch geprüft werden kann. Das Aufbauen einer Anwendung aus Komponenten wird üblicherweise durch ein Werkzeug unterstützt, welches die Verknüpfung der Komponenten durchführt und sicherstellt, dass die Abhängigkeiten aller Komponenten erfüllt sind.

Anstatt die Komponenten direkt miteinander zu verbinden, ist auch der Ansatz verbreitet, die Verbindung über ein Komponentenframework herzustellen. Die einzelnen Komponenten werden hier nur mit diesem Framework verknüpft, welches intern die Kommunikation zwischen den Komponenten realisiert. Ein solches Framework enthält außerdem noch Basisdienste, welche von den Modulen genutzt werden können, um Zugriff auf Ressourcen außerhalb des Komponentenframeworks zu erlangen, zum Beispiel für Dateioperationen.

Durch die strenge Kapselung der Komponenten ist außerdem sichergestellt, dass der Datenaustausch zwischen zwei Komponenten ausschließlich über wohl definierte Schnittstellen stattfindet. Daher ist es mithilfe eines Komponentenframeworks auch leicht möglich, verteilte Komponentensysteme zu erstellen, indem die Kommunikation zwischen den Komponenten transparent durch das Framework auf andere Rechner geleitet wird.

Darüber hinaus kann ein Komponentenframework das dynamische Hinzufügen und Austauschen von Komponenten zur Laufzeit unterstützen. Dabei bietet die Infrastruktur oft die Möglichkeit, zu einem Modul "*startup*" und "*destruction*" Code anzugeben. Dieser Code wird vom Framework ausgeführt, wenn die Komponente installiert beziehungsweise entfernt wird. So hat die Komponente die Möglichkeit, spezifische Änderungen am System vorzunehmen, um sich in das existierende System zu integrieren oder sich abzumelden.

Komponentenmodelle gibt es in verschiedenen Bereichen. Oft resultieren sie aus der objektorientierten Entwicklung und sind traditionell für leistungsstarke Rechner gedacht. Zu den bekanntesten kommerziellen Vertretern von lokalen Komponentensystemen gehört Microsofts *Component Object Model* (COM) [COM95] und SUNs *JavaBeans* [Sun97]. Bekannte Beispiele für verteilte Komponentensysteme sind Microsofts *Distributed COM* (DCOM) als Erweiterung von COM, SUNs *Enterprise JavaBeans* (EJB) [Sun06] und das *CORBA Component Model* (CCM) [Omg06].

Als Hauptnachteil von Komponentenmodellen gilt, dass ihre Verwendung zusätzliche Ressourcen in Anspruch nimmt. Schon bei der Entwicklung muss zusätzlicher Aufwand investiert werden, um eine saubere Aufteilung in Komponenten durchzuführen. Während das jedoch nicht unbedingt negativ zu sehen ist, sind die zusätzlichen Ressourcen, die durch ein Komponentenframework zur Laufzeit benötigt werden, meist als Nachteil einzustufen. Hier ist vor allem der Speicherbedarf zu nennen, den das Framework zur Verknüpfung der Komponenten zusätzlich benötigt, und die Kosten, welche durch die indirekte Kommunikation entstehen.

Um die Größe der Komponentenframeworks zu verringern, wurde die Infrastruktur, auf denen einige Komponentenframeworks basieren, speziell für eingebettete Systeme entwickelt oder angepasst, zum Beispiel eine leichtgewichtige CORBA-Plattform (minimal CORBA [Omg02]) oder ressourcensparende CORBA-Implementierungen (TAO [SLM98]). Dennoch sind diese Systeme nicht uneingeschränkt für eingebettete Systeme verwendbar, da sie immer noch relativ viele Ressourcen benötigen.

Im Bereich der eingebetteten Systeme stehen meist keine zusätzlichen Ressourcen zur Verfügung. Komponentensysteme, welche die Komponenten statisch zusammenbinden, kommen ohne ein spezielles Verknüpfungsframework zur Laufzeit aus. Sie benötigen daher auch weniger Ressourcen zur Laufzeit und sind für eingebettete Systeme besser geeignet.

Ein Beispiel eines solchen Komponentenmodells stellt die Programmiersprache *nesC* [GLv+03] dar. NesC wird im Rahmen von TinyOS [HSW+00], einem Betriebssystem für Sensorknoten, eingesetzt und wurde entwickelt, um die Struktur dieses Betriebssystems zu unterstützen. NesC basiert auf der Programmiersprache C und erweitert diese um die Möglichkeit, Komponenten zu definieren. Ein spezieller Compiler fügt die Komponenten zu einem Komplettsystem zusammen. Um möglichst kleinen Code zu erzeugen, werden bei diesem Vorgang globale Optimierungen angewandt. Dabei wird außerdem das wohl definierte Ablaufmodell von TinyOS zugrunde gelegt, um Datenflussanalysen durchzuführen und um vor möglichen Konflikten, die aus nebenläufigen Datenzugriffen resultieren, warnen zu können. Durch dieses Vorgehen lässt sich der resultierende Code zwar gut optimieren, jedoch ist keine dynamische Änderung zur Laufzeit möglich.

Zusammenfassend ist festzustellen, dass Komponentenmodelle eine gute Möglichkeit darstellen, Programme strukturiert und modularisiert zu entwerfen. Existierende Anwendungen mithilfe eines Komponentenmodells zu modularisieren ist jedoch äußerst aufwendig und kommt oft einer Neuim-

plementierung gleich. Außerdem benötigt die dynamische Veränderung der Zusammensetzung ein Verknüpfungsframework zur Laufzeit. Dieses benötigt zusätzliche Ressourcen und führt oft zu einer indirekten Kommunikation zwischen den Komponenten.

4.2.3 Modularisierung existierender Anwendungen

Um eine existierende Software in einzelne Module aufzuteilen, ist ein Ansatz wünschenswert, bei dem die Struktur der Anwendung im Nachhinein identifiziert wird. Ein solcher Ansatz soll hier vorgestellt werden.

Als Grenzen zur Festlegung der Module bieten sich bei einem existierenden Programm die Konstrukte an, die durch die Programmiersprache angeboten werden, um das Programm zu strukturieren. So sind die meisten Programme in einzelne Quellcodedateien aufgeteilt. Diese Aufteilung kann man zur Einteilung in Module nutzen, da hier meistens zusammengehörende Funktionen und Daten enthalten sind. Eine feinere Aufteilung ergibt sich, wenn man die einzelnen Funktionen und globalen Datenobjekte als Modulgrenzen verwendet.

Die Modularisierung kann auf Basis des Quellcodes oder auf Basis des Binärcodes geschehen. In jedem Fall müssen die Grenzen und die Abhängigkeiten der Module identifiziert werden.

4.2.3.1 Basiseinheiten der Modularisierung

Im Folgenden sollen Konstrukte betrachtet werden, die als Grenzen einer Modularisierung hergenommen werden können.

Quellcodedateien Die meisten Programmiersprachen lassen eine Unterteilung des Codes in einzelne Dateien zu. Manche Sprachen, wie beispielsweise Java, zwingen den Entwickler dazu, öffentliche Klassen in eigenen Dateien abzulegen, bei anderen Sprachen ist das ein freiwilliges Strukturelement. Gemeinsam ist jedoch, dass in einer Datei Code und Daten zusammengefasst werden, die inhaltlich zusammengehören. Diese Aufteilung bietet daher eine gute Möglichkeit der Modularisierung, die relativ einfach zu extrahieren ist.
Die Strukturierung an Dateigrenzen ist allerdings relativ grobgranular. In objektorientierten Sprachen sind in einer Datei meist alle Methoden einer Klasse enthalten, in funktionalen Sprachen werden üblicherweise auch mehrere Funktionen in einer Datei zusammengelegt, wenn sie auf denselben Daten arbeiten. Obwohl die Funktionen semantisch zusammengehören, ist es dennoch wünschenswert, einzelne Funktionen getrennt behandeln zu können. Enthält eine Datei `Thread.c` beispielsweise alle Funktionen, um den Zustand von Threads zu verändern, so möchte man hiervon möglicherweise nur die Funktion zum Erzeugen neuer Threads parken, da sie nach dem Starten des Programms kaum noch benötigt wird.

Funktionen und globale Daten Eine feinere Gliederung stellt die Unterteilung nach Funktionen[1] dar. Funktionen erfüllen üblicherweise eine mehr oder weniger wohl definierte Aufgabe und sind als Strukturelement in fast allen Programmen vorhanden. Sie sind kleiner als Quellcodedateien und erlauben eine feingranulare Aufteilung der Software. Um alle Bestandteile eines Programms zu erfassen, müssen neben den Funktionen auch globale Daten als Module dargestellt werden.
Im Vergleich zur Modularisierung an Dateigrenzen ist das Identifizieren der Modulgrenzen hier aufwendiger. Es müssen zusätzliche Maßnahmen getroffen werden, um die Funktionen und Daten zu extrahieren und einzeln erfassen zu können. Arbeitet man auf Quellcode, so müssen

[1] "Funktionen" wird hier als Oberbegriff für Prozeduren, Funktionen und Methoden verwendet

die Quellcodedateien analysiert und aufgeteilt werden. Wird Binärcode verarbeitet, so müssen diese Strukturen darin identifizierbar sein (siehe Abschnitt 4.2.5.1).
Das Identifizieren einzelner Funktionen ist jedoch zum Teil auch notwendig, wenn man die Module entsprechend den Dateigrenzen definiert. Denn für das Parken und Auslagern von Funktionalität wird eine Schnittstelle benötigt. Die Schnittstelle einer Funktion ist leicht zu erkennen und abzugrenzen. Bei einer Datei besteht die Schnittstelle aus der Vereinigung der Schnittstellen aller enthaltenen Funktionen. Je nach Verfahren kann es somit auch bei Modularisierung auf Dateiebene notwendig sein, die einzelnen Funktionen zu erfassen.

Basisblöcke Eine noch feinere Untergliederung lässt eine Aufteilung des Codes in Basisblöcke zu. Ein Basisblock ist ein Codestück, welches linear durchlaufen wird. Basisblockgrenzen sind somit Sprünge oder Sprungziele.
Vorteil dieser Aufteilung ist, dass die Module sehr klein sind und somit eine sehr flexible Konfiguration ermöglicht wird. Nachteil ist allerdings, dass sehr viele Module entstehen und es dadurch schwieriger wird, die Aufteilung nachzuvollziehen. Dies wird zusätzlich erschwert, da diese Aufteilung kaum durch Strukturelemente in den Programmiersprachen unterstützt ist, sodass die Module keine expliziten Namen durch den Entwickler zugeordnet bekommen.
Des Weiteren ist die Extraktion von Basisblöcken aus Binärcode architekturabhängig, da die Sprungbefehle identifiziert werden müssen.

Klassen Bei objektorientierter Programmierung bieten sich weiterhin Klassen als Grenze zur Modularisierung an. Diese Aufteilung ist nicht so feingranular wie auf Funktionsebene. Mit dieser Modularisierungsgranularität kann der Wunsch, nur eine Methode einer Klasse auszulagern oder zu parken, nicht erfüllt werden. Betrachtet man Klassen nur unter dem Gesichtspunkt der Strukturierung, so stellen sie hauptsächlich eine Sammlung von Funktionen dar, die auf gemeinsamen Daten arbeiten. Eine Modularisierung in Klassen lässt sich somit auf eine Modularisierung in Funktionen abbilden. Die zusätzliche Information der Klassenzugehörigkeit gibt jedoch einen Hinweis auf eine gemeinsame Datenabhängigkeit der Methoden. Dieser Hinweis kann bei der Entscheidung, ob eine Methode ausgelagert werden soll, hilfreich sein.

Zusammenfassend ist festzuhalten, dass sich, je nachdem, ob die Modularisierung auf Quellcode oder Binärcodeebene stattfinden soll, einige Strukturelemente leichter extrahieren lassen als andere. Allgemein bietet jedoch eine Modularisierung auf Ebene der Funktionen die meisten Vorteile. Es lassen sich einige andere Modularisierungen nachbilden und einen Teil des Aufwands zur Identifikation oder Extraktion muss ohnehin durchgeführt werden, um geeignete Schnittstellen zu finden. Außerdem ist diese Aufteilung durch den Softwareentwickler nachvollziehbar. Grundsätzlich ist auch eine Mischung der genannten Modelle möglich.

4.2.3.2 Ausgangsbasis: Binärcode oder Quellcode?

Möchte man eine existierende Anwendung in Teile aufteilen, stellt sich die Frage, auf welcher Ausgangsbasis man ansetzen möchte. Hier hat man die Möglichkeit den Quellcode zu analysieren oder die bereits übersetzten Dateien zu verwenden. Das generelle Vorgehen ist für beide Ausgangsformate gleich:

1. Zunächst werden die Ausgangsdateien eingelesen.
2. Anschließend werden sie analysiert, um die Module zu erstellen.

3. Gleichzeitig oder anschließend müssen auch die Abhängigkeiten der Module voneinander bestimmt und vermerkt werden.

Je nach Ausgangsformat sind die einzelnen Schritte mehr oder weniger aufwendig. Bei der Verwendung von Binärdateien muss man einige Anforderungen stellen, um die gewünschten Informationen aus dem Ausgangsmaterial extrahieren zu können:

Relokierbarkeit. Die endgültige Position eines Moduls im Speicher muss noch veränderbar sein, das heißt, der binäre Code des Moduls darf nicht bereits auf eine endgültige Adresse festgelegt sein. Ansonsten kann man einzelne Module nicht mehr flexibel hinzufügen, außerdem wäre die Aufteilung und effiziente Verwendung des vorhandenen Adressbereichs stark eingeschränkt. Folglich wäre man auch beschränkt in der Auswahl und Kombination der einzelnen Module.

Granularität. Je nach gewünschter Granularität der Modularisierung muss es möglich sein, die Strukturinformationen aus der Binärdatei zu extrahieren. Um die Struktur einer Software auf Funktionsebene zu extrahieren, benötigt man feingranular aufgeteilte Binärdateien. Diese lassen sich automatisch erstellen, jedoch sind bestimmte Einstellungen zum Übersetzen notwendig, die nicht immer standardmäßig angewandt werden. Man kann somit nicht beliebige Binärdateien verwenden, sondern muss sie speziell für diesen Einsatz vorbereiten. Details zu diesem Problem und ein Lösungsansatz werden in Abschnitt 4.2.5 vorgestellt.

Debuginformationen. Um Module als entfernten Dienst anzubieten, sind detaillierte Informationen über ihre Schnittstelle erforderlich. Für die normale Verarbeitung des Binärcodes sind diese Informationen jedoch nicht nötig, weshalb sie im Binärcode auch nicht mehr vorhanden sind. Um dennoch Informationen über die Schnittstellen zu erhalten, muss man zusätzliche Informationsquellen nutzen. Ein möglicher Weg ist die Verwendung von ausführlichen Debuginformationen, die von vielen Compilern automatisch erzeugt werden können. Sie enthalten, je nach Format, die gewünschten Typinformationen über die Schnittstellen der Funktionen. Zu beachten ist allerdings, dass die Debuginformationen zwar automatisch erzeugt werden können, dies jedoch beim Übersetzungsvorgang explizit ausgewählt werden muss.

Die Verwendung von Binärcode als Ausgangsbasis der Modularisierung stellt somit einige Anforderungen an die Eigenschaften beziehungsweise an das Format des Binärcodes und der benötigten Zusatzinformationen. Beachtet man allerdings diese Anforderungen und verwendet geeignete Binärcodedateien, so ergeben sich einige Vorteile gegenüber einem Ansatz auf Quellcodedateien:

- Besonders hervorzuheben ist die Tatsache, dass Binärcode weitestgehend sprachunabhängig ist. So ist es prinzipiell möglich, eine auf Binärcode basierende Technik anzuwenden, unabhängig davon, aus welcher Programmiersprache der Code erzeugt wurde. Will man den Quellcode analysieren, muss man für jede Sprache einen eigenen Parser erstellen.

- Im Allgemeinen können die benötigten Strukturinformationen einfacher aus dem Binärcode gewonnen werden als aus dem Quellcode. Quellcode muss man zunächst parsen, um zu einer Form zu gelangen, an der man die gewünschten Informationen ablesen kann. Das Analysieren und Parsen des Quellcodes ist jedoch relativ aufwendig, da ein Programmierer viele Freiheiten bei der Erstellung des Codes hat.
  Das Format von Binärdateien bietet hingegen weniger Freiheiten, da Binärdateien üblicherweise automatisch erzeugt werden und zur maschinellen Weiterverarbeitung gedacht sind. Binärdateien bieten somit den Vorteil, dass sie schon durch einen Compiler in eine Form überführt wurden,

die leichter zu verarbeiten ist. Daher lässt sich aus den üblichen Binärcodeformaten die Struktur ermitteln, ohne den tatsächlichen Code analysieren und parsen zu müssen. Symboltabellen informieren über die enthaltenen Elemente, deren Position und meist auch über ihre Größe.

- Die Abhängigkeiten zwischen den Modulen sind aus einer entsprechend vorbereiteten Binärdatei leichter auszulesen als aus dem Quellcode. Den Quellcode muss man zunächst analysieren, um Funktionsaufrufe extrahieren und Zugriffe auf globale Daten feststellen zu können.
  In den üblichen Binärdateiformaten sind die Abhängigkeiten schon in einer expliziten und maschinenlesbaren Form vorhanden. Symboltabellen enthalten unaufgelöste Symbole, welche die benötigten Module charakterisieren; Relokationsinformationen spezifizieren, welches Modul die Abhängigkeit besitzt.

- Bei der Erstellung eines Programms kann man auf compilerspezifische Erweiterungen zurückgreifen, die nicht im Standard der verwendeten Programmiersprache definiert sind. Bei der Verwendung des GCCs kann man beispielsweise das Schlüsselwort `asm` verwenden, um Assemblercode in C einzubetten oder mit der Anweisung `__attribute__` die Position von Code oder Daten beeinflussen. Die Bedeutung dieser Erweiterungen muss einem Werkzeug, welches auf Quellcode arbeitet, bekannt sein, um den Code entsprechend zu verarbeiten. Im Binärcode sind nur selten spezielle Erweiterungen vorhanden, die zur Ausführung wirklich relevant sind.
  Neben compilerspezifischen Erweiterungen existieren auch explizit offen gelassene Bereiche in den Standards der Programmiersprachen. So definiert zum Beispiel der C-Standard [ISO05] die Reihenfolge, in der die Bytes einer größeren Datenstruktur abgelegt werden als implementierungsspezifisch. Da einige Programme von der einen oder anderen Art der Compilereigenheiten abhängen, müsste man das Verhalten des verwendeten C Compilers nachbauen, wenn man auf Quellcodeebene arbeiten möchte [GSK04].

- Zur Erstellung eines Programms muss der Quellcode später ohnehin in Binärcode übersetzt werden. Bei der Verwendung von Binärcode wird dieser Schritt schon in die Vorverarbeitung verschoben und das Verfahren wird davon weitestgehend unabhängig.
  Außerdem sind Binärcodedateien, durch den Compiler, bereits mit architekturspezifischen Zusatzinformationen bezüglich der Positionierung im Speicher angereichert. Diese Informationen spielen zwar bei der Analyse der Struktur eine untergeordnete Rolle, jedoch sind sie für die spätere Verwendung der Module notwendig.

Die Verwendung von Binärcode als Ausgangsbasis hat allerdings auch Nachteile. Besonders zu beachten ist, dass einige Optimierungen die Struktur des Programms verändern. Hier ist vor allem das Einbetten einer Funktion in eine andere zu nennen (*Inlining*). Das Modularisieren oder Extrahieren eingebetetter Funktionen ist im Allgemeinen nicht oder nur eingeschränkt möglich. Allerdings können Debuginformationen herangezogen werden, um solche strukturverändernde Optimierungen zu erkennen. Sie sind dort vermerkt, da sie auch die Fehlersuche beeinträchtigen.

Im Rahmen dieser Arbeit wird ein Verfahren zur Modularisierung auf Basis von Binärcodedateien vorgestellt. Unter Beachtung der genannten Eigenschaften überwiegen die Vorteile dieses Ausgangsformats. Die bestehenden Nachteile können durch zusätzliche Informationen ausgeglichen werden, die man größtenteils automatisch erstellen kann, wie es beispielsweise bei Debuginformationen der Fall ist.

4.2.4 Binärdateien

Quellcode wird üblicherweise von einem Compiler und einem Assembler in eine Objektdatei übersetzt. Anschließend werden mehrere Objektdateien durch den Linker zu einem ausführbaren Programm zusammengebunden (Abbildung 4.1). Es gibt eine Reihe von unterschiedlichen Binärformaten, in denen die Objektdateien und das ausführbare Programm gespeichert sein können. Welches verwendet wird, hängt hauptsächlich von der Unterstützung des Betriebssystems ab.

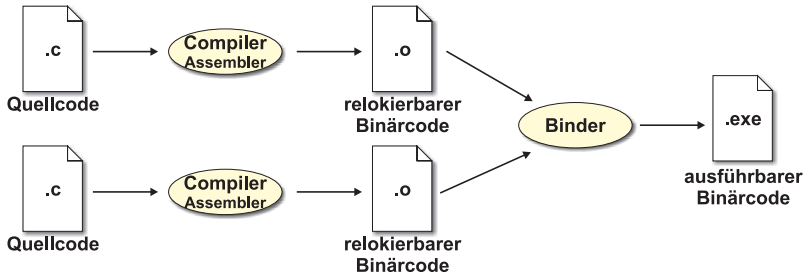


Abbildung 4.1: Ablauf beim Erstellen eines Programms
Aus zwei Quellcodedateien wird hier ein ausführbares Programm gebaut. Zunächst werden die Quellcodedateien in Objektdateien übersetzt. Anschließend werden diese zu einem ausführbaren Programm gebunden.

Dieser Abschnitt gibt einen Überblick über Binärdateien und insbesondere Objektdateien. Dabei werden zunächst die verschiedenen Dateitypen aufgezeigt, die ein Binärformat darstellen kann. Danach wird der Inhalt und generelle Aufbau von Objektdateien vorgestellt. Schließlich sollen einige bekannte Objektdateiformate betrachtet werden.

4.2.4.1 Arten von Binärdateien

Durch die meisten Binärformate lassen sich vier Dateitypen darstellen:

- Relokierbare (verschiebbare) Objektdateien (*relocatable files*)
  Diese Dateien entstehen aus einzelnen Quellcodedateien nach dem Übersetzen und enthalten den noch nicht gebundenen Binärcode. Zusätzlich sind Informationen enthalten, welche beschreiben, wie dieser Code mit anderem Code zusammengebunden werden muss und wo er im Speicher abgelegt werden darf. Dies sind Informationen für den Binder, der aus mehreren Objektdateien eine ausführbare Programmdatei oder eine gemeinsam genutzte Bibliothek erstellen kann.

- Ausführbare Programmdateien (*executable files*)
  Eine Programmdatei enthält den gebundenen Binärcode, der für die Ausführung an festgelegten Adressen vorgesehen ist. Außerdem sind Informationen enthalten, die es dem Betriebssystem ermöglichen, einen neuen Prozess zu erzeugen, der die enthaltenen Daten und Code verwendet.

- Gemeinsam benutzte Objektdateien (*shared object files*)
  Dieser Dateityp stellt eine Mischung der beiden vorherigen Dateitypen dar. Er enthält teilweise gebundenen Binärcode und Informationen zum weiteren dynamischen oder statischen Binden. Dateien dieses Typs kann man mit anderen Objektdateien zu ausführbaren Programmen zusammenbinden, alleine sind sie nicht ausführbar. Unterstützung für diesen Dateityp wurde in einigen älteren Binärdateiformaten erst nachträglich integriert.

- Speicherabbilddateien (*core files*)
  Manche Binärformate können auch eingesetzt werden, um Prozessabbilder zu speichern, wenn

das Betriebssystem einen Prozess zum Beispiel wegen eines Fehlers beendet. In diesen Dateien wird der Speicher- und Registerinhalt festgehalten. Sie dienen hauptsächlich der Fehleranalyse und sind für unsere Betrachtungen nicht relevant und daher nur zur Vollständigkeit angeführt.

Für unsere Zwecke muss der Code an verschiedenen Adressen verwendbar sein. Daher sind besonders relokierbare Objektdateien interessant. Die endgültige Position des Codes ist dort noch nicht festgelegt, somit kann er noch an verschiedene Adressen gebunden werden. Außerdem kann positionsunabhängiger Code, wie er häufig in gemeinsam genutzten Bibliotheken vorkommt, genutzt werden. Ausführbare Programmdateien sind zur Darstellung von Modulen nur geeignet, wenn sie ebenfalls positionsunabhängigen Code enthalten. Meistens ist der enthaltene Code jedoch nur an festgelegten Adressen verwendbar. Man kann sie dann allerdings nutzen, um den Grundzustand eines Knotens darzustellen, wie in Abschnitt 4.5.4.2 erwähnt wird.

4.2.4.2 Inhalt und Aufbau

Bei vielen Binärformaten sind die binären Daten in Blöcke unterteilt, welche Sektionen genannt werden. So gibt es mindestens eine Sektion, welche den Binärcode enthält (meist als `.text` bezeichnet) und eine andere Sektion, welche die initialisierten Daten enthält (meist mit dem Namen `.data` gekennzeichnet). Daneben gibt es oft noch eine Sektion, welche nicht-initialisierte Daten beschreibt (die `.bss` Sektion).

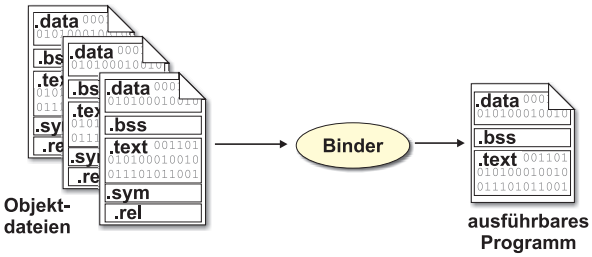


Abbildung 4.2: Ablauf beim Binden eines Programms
Die einzelnen Sektionen aus den verschiedenen Objektdateien werden zu einem ausführbaren Programm gebunden. Dabei werden die einzelnen Sektionen miteinander “verschmolzen”.

Beim Erstellen eines Programms werden die Sektionen von mehreren relokierbaren Objektdateien von einem Binder zu einer ausführbaren Datei zusammengebunden (Abbildung 4.2). Dabei wird ein Binder für die entsprechende Zielarchitektur verwendet, der fest legt, an welcher Adresse eine Sektion aus einer Objektdatei im endgültigen Programm liegen soll [Lev99].

Neben den binären Daten enthalten die Binärdateien noch zusätzliche Informationen. Ausführbare Programmdateien enthalten beispielsweise noch Informationen wie aus den Daten ein Prozessabbild zu erstellen ist, also wo die einzelnen Sektionen abzulegen sind. Da uns hauptsächlich die Objektdateien interessieren, betrachten wir diese etwas genauer. Hier sind Informationen enthalten, die das Binden des relokierbaren Codes ermöglichen. Die Art und Weise, wie diese Informationen gespeichert werden, hängt von dem verwendeten Binärdateiformat ab. Die Art der Daten ist jedoch immer sehr ähnlich. So enthalten relokierbare Objektdateien neben dem Maschinencode und den binären, initialisierten Daten vor allem Symboltabellen, Stringtabellen, Relokations- und Debuginformationen.

Symboltabellen Ein Symbol ordnet einer Position im Binärcode einen Namen zu. Dies dient vor allem der Verknüpfung zwischen verschiedenen Objektdateien. Als Ziel eines Verweises wird nicht die Position angegeben, sondern ein Symbol, da die endgültige Position (=Adresse) noch

nicht feststeht oder das Ziel in einer anderen Datei gespeichert ist. In einer Objektdatei gibt es daher auch undefinierte Symbole, die zwar einen Namen, jedoch keine Position haben. Sie beschreiben eine Position im späteren Programm, die durch eine andere Objektdatei definiert wird.

Stringtabellen Eine Stringtabelle ist lediglich eine Optimierung beim Speichern der Namen von Symbolen und Sektionen. Statt die Namen direkt in die entsprechende Struktur zu schreiben, wird dort auf einen Eintrag in einer Stringtabelle verwiesen. Doppelt vorkommende Namen müssen so nur einmal abgespeichert werden.

Relokationstabellen Enthält die Objektdatei verschiebbaren Binärcode, so können keine absoluten Zieladressen von Sprungbefehlen im Binärcode enthalten sein. Die Zieladressen werden erst während des Bindens durch den Linker bestimmt, der festlegt, wo die einzelnen Binärdaten im endgültigen Programm liegen werden. Damit der Linker weiß, an welchen Stellen er Adressen einfügen muss, enthalten verschiebbare Objektdateien eine Relokationstabelle. In dieser Tabelle werden Verweise auf Symbole gespeichert. Jeder Verweis, auch Relokation genannt, enthält das Zielsymbol, die Position, an der die Adresse des Zielsymbols in den Code eingefügt werden muss und die Art und Weise, wie sie eingefügt werden soll.

Debuginformationen Um Fehler schneller finden und leichter eliminieren zu können, besteht die Möglichkeit einer Objektdatei Debuginformationen beizufügen. Diese enthalten im Allgemeinen eine Rückabbildung von dem erzeugten Binärcode auf den ursprünglichen Quellcode. So kann im Falle eines Fehlers aus der Position im Binärcode die Stelle im Quellcode bestimmt werden. Dem Programmierer kann dann die Zeile des Quellcodes angezeigt werden, die vermutlich den Fehler verursacht hat.
Daneben können aber noch zahlreiche weitere Informationen enthalten sein, wie beispielsweise Informationen über den Aufbau der verwendeten Typen. Hierdurch kann ein Programmierer mithilfe eines geeigneten Debuggers die Datenstrukturen in einem Programmabbild untersuchen, ohne den Quellcode vorliegen zu haben. In Abschnitt 4.2.6 werden Debuginformationen genauer betrachtet und verschiedene Formate vorgestellt. Für uns sind insbesondere die Typinformationen interessant, da sie zur Definition der Schnittstelle eines Moduls genutzt werden können.

In Abbildung 4.3 werden die angesprochenen Informationen anhand eines realen Beispiels gezeigt. Dargestellt ist der Inhalt der Objektdatei eines einfachen Beispiels im ELF-Format. Anhand der Relokationstabelle kann man erkennen, welche Stellen im Binärcode positionsabhängig sind. Außerdem sind die Verweise auf die Symboltabelle zu sehen, mit deren Hilfe die endgültige Adresse bestimmt wird.

4.2.4.3 Objektdateiformate

Im Folgenden sollen kurz die wichtigsten Formate für Objektdateien vorgestellt werden. In einer Zusammenfassung werden sie abschließend kurz miteinander verglichen.

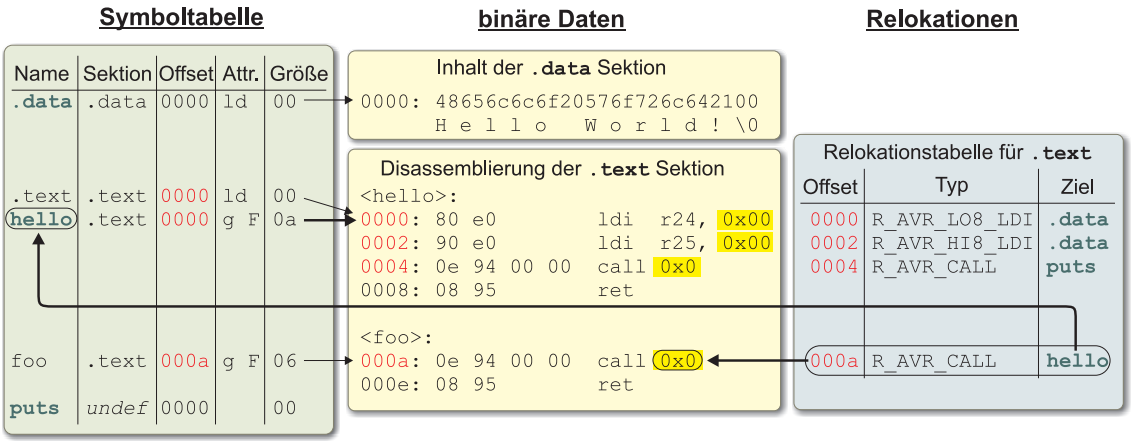
a.out

Das *assembler output* Format (kurz: *a.out*) ist ein Format für ausführbare Dateien, eingeführt mit dem ersten UNIX-System. In der ursprünglichen Form [TR71] nur für Programme und Objektdateien gedacht, wurde es später auch für Programmbibliotheken verwendet [Ber95]. Das Format wurde lange

```
void hello() {
      printf ("Hello World!\n");
}

void foo() {
      hello();
}
```

(a) Quellcode



(b) Binärdaten in ELF Datei

Abbildung 4.3: Daten in einer ELF-Datei und deren Zusammenhang

In Teilabbildung (a) ist der Quellcode eines einfachen Beispiels gegeben: eine Funktion `foo`, welche eine Funktion `hello` aufruft, welche wiederum die Funktion `printf` aufruft.

Teilabbildung (b), darunter, zeigt den Inhalt der ELF-Objektdatei für Atmel AVR Architekturen, die durch Übersetzen mit einem Compiler entsteht.

In der Mitte ist der Binärcode dargestellt. In der `.data`-Sektion erkennt man den Text “Hello World!”. In der Textsektion sieht man den Code der beiden Funktionen `hello` und `foo`. Im Code erkennt man die Funktionsaufrufe (`call`) und sieht, dass als Zieladresse für beide null angegeben ist.

Rechts ist die Relokationstabelle für die Sektion `.text` abgebildet. Die markierte Zeile bedeutet, dass an der Position `0xa` die Adresse des Symbols `hello` eingefügt werden soll. Die Position `0xa` im Binärcode entspricht der Position des Aufrufs der Funktion `hello` in der Funktion `foo`. Die Art der Einfügung ist mit dem Attribut `R_AVR_CALL` beschrieben. Dadurch weiß der Linker, dass er die Adresse des Symbols `hello` nicht direkt an der Stelle `0xa` einfügen darf, sondern dass dort ein Sprungbefehl steht und er die Position erst zwei Byte später einfügen soll.

Die Symboltabelle (links) beschreibt schließlich das Symbol `hello` genauer. In diesem Beispiel zeigt es auf die Position `0` in der Sektion `.text`, also auf den Beginn der Funktion `hello`. Man sieht auch, dass das Symbol `puts`, welches hier für die Funktionalität von `printf` verwendet werden soll, nicht definiert ist (als Sektion ist *undef* angegeben). Es muss von einer anderen Objektdatei angeboten werden.

Zeit von vielen UNIX-Systemen eingesetzt, so zum Beispiel auch von Linux bis zur Kernel Version 1.2. Inzwischen hat es jedoch kaum noch Bedeutung, da es fast überall durch Nachfolgeformate wie COFF oder ELF abgelöst wurde.

Im Laufe der Zeit existierten verschiedene Versionen des Formats. Im ursprünglichen Format (*OMAGIC*) wurde nicht zwischen initialisierten Daten und Programmcode unterschieden, das wurde erst in neueren Versionen (*NMAGIC*) eingeführt, um Programmcode schreibgeschützt zu laden. In einer weiteren Version (*ZMAGIC*) wurden dann Speicherseiten unterstützt, sodass einzelne Seiten erst bei Bedarf in den Prozess eingeladen werden konnten.

Das a.out-Objektformat ist, je nach Version, in bis zu sieben Abschnitte unterteilt. Darunter befindet sich neben einem Codeabschnitt (*text segment*) und einem Datenbereich (*data segment*) auch Relokationsinformationen (getrennt nach *text relocations* und *data relocations*) und eine Symboltabelle (*symbol table*). Symbolnamen sind im ursprünglichen a.out-Format auf eine Länge von acht Zeichen beschränkt. Durch die Einführung einer Stringtabelle können sie jedoch in neueren Versionen auch länger sein.

Eine Speicherung von Debuginformationen ist im a.out-Objektformat nicht vorgesehen. Es gibt jedoch die Möglichkeit Debuginformationen im *stabs*-Debugformat (siehe Abschnitt 4.2.6.1) als Symbole einzufügen.

COFF

Das *Common Object File Format* (*COFF*) [Gir88, Del96] ist der Nachfolger des a.out-Formats. Es wurde mit UNIX System V Release 3 (SVR3) eingeführt, um einige Schwächen von a.out auszubessern. So ist es in COFF möglich, mehrere Sektionen unterschiedlicher Typen zu definieren. Hierzu wurde im Dateikopf eine Tabelle der Sektionen eingeführt, die zu jeder Sektion auch einen Namen führt. Der Name einer Sektion ist jedoch auf höchstens acht Zeichen beschränkt. Sektionen kann der Typ Code (`.text`), initialisierte Daten (`.data`) oder nicht initialisierte Daten (`.bss`) zugeordnet werden.

Außerdem wurde ein einfaches Format definiert, um grundlegende Debuginformationen in einer COFF-Datei speichern zu können (siehe Abschnitt 4.2.6.1). Das Format der Debuginformationen ist jedoch für die Sprache C optimiert und bietet keine Erweiterungsmöglichkeiten, sodass man damit schnell an Grenzen stößt.

XCOFF Um diese Einschränkungen und die mangelnden Erweiterungsmöglichkeiten zu umgehen, hat IBM das COFF-Format für ihr UNIX-Derivat AIX erweitert. Es entstand *XCOFF* (*eXtended COFF*), welches später auch in einer 64-bit-Version definiert wurde [Ibm04]. In XCOFF wurden zusätzliche Sektionstypen eingeführt, um beispielsweise das dynamische Binden zu vereinfachen. Außerdem kann eine Sektion mit Typinformationen angelegt werden, um eine grundlegende Typprüfung beim Binden von mehreren Dateien durchzuführen. Die Behandlung von Ausnahmen (*traps*, *exceptions*) wird durch eine Sektion unterstützt, welche die Quelle und den Grund von potenziell auftretenden Ausnahmen enthält. Allgemeine Debuginformationen werden in einer eigenen Sektion im erweiterbaren *stabs*-Format abgelegt.

eCOFF Neben XCOFF gibt es noch eine andere Erweiterung namens *eCOFF* (*extended COFF*) [Com99] für Alpha und MIPS-Architekturen. eCOFF bietet vor allem verbesserte Unterstützung für dynamisches Binden. Außerdem können die Binärdaten in eCOFF komprimiert abgelegt werden, um Speicherplatz zu sparen.

PECOFF Das *Portable Executable* kurz *PE* oder *PECOFF* [Mic06] ist ebenfalls eine Erweiterung des COFF-Formats. Es wurde von Microsoft entwickelt und mit Windows NT 3.1 eingeführt. PECOFF wurde speziell für Windows Betriebssysteme angepasst. So enthalten ausführbare Dateien einen zusätzlichen PECOFF Header, um die Abwärtskompatibilität zu MS-DOS zu gewährleisten. Außerdem wurde eine Reihe neuer Sektionstypen eingeführt, um zum Beispiel die sogenannten Windows Ressourcen abzulegen oder um Informationen über exportierte Schnittstellen bei Bibliotheken zu speichern. Einige der Sektionen werden jedoch anhand ihres Namens anstatt des Typs identifiziert. Der Name von Sektionen in Objektdateien kann bei PECOFF auch länger als acht Zeichen sein, indem auf eine Stringtabelle verwiesen wird.

Auch für Debuginformationen wird in PECOFF eine spezielle Sektion angelegt. Microsoft entwickelte dafür ein eigenes Format für den gleichnamigen Debugger *CodeView* [SS+96, TIS93], der später in das Visual Studio integriert wurde. Aktuelle Microsoft Compiler erstellen jedoch keine integrierten Debuginformationen mehr, sondern legen diese Informationen in einer separaten Datei ab (*Programm Database* siehe Abschnitt 4.2.6.1).

ELF

Das *Executable and Linking Format* (*ELF*) [TIS95] ist ein Standard-Binärformat für ausführbare Dateien und gemeinsam benutzte Bibliotheken. Es wurde ursprünglich von den Unix System Laboratories (USL) als Teil des *Application Binary Interfaces* (ABI) von UNIX-Systemen entwickelt und erstmals im System V Release 4 UNIX-System (SVR4) eingesetzt, um COFF zu ersetzen. Später wurde es vom *Tool Interface Standards Komitee* (TIS) weiterentwickelt, um als portables Binärformat für Betriebssysteme auf 32-bit-Intel Architekturen zu dienen. Heute wird das ELF-Format von den meisten UNIX ähnlichen Betriebssystemen und für fast alle gängigen Prozessoren eingesetzt.

In ELF-Dateien sind die Daten, ähnlich wie in COFF, in Sektionen unterteilt. Im Gegensatz zu COFF werden zusätzliche beschreibende Daten, wie beispielsweise die Symbol- oder die Relokationstabelle, in normalen Sektionen mit entsprechendem Typ abgelegt. Dadurch kann eine ELF-Datei zum Beispiel mehrere Symboltabellen enthalten. In ELF ist kein Format zur Speicherung von Debuginformationen definiert. Üblicherweise wird jedoch zusammen mit ELF das DWARF-Debugformat (Abschnitt 4.2.6.1) eingesetzt, welches die benötigten Daten in mehreren benutzerdefinierten Sektionen ablegt.

Mach-O

Das *Mach Object* Dateiformat (*Mach-O*) wurde, wie der Name schon andeutet, ursprünglich für Mach Betriebssysteme entwickelt. Heute wird es hauptsächlich von Mac OS X eingesetzt [App06]. Auch Mach-O wurde nur als Format zur Speicherung von Objektdateien und ausführbaren Programmen entwickelt. Im Laufe der Zeit wurde jedoch auch hier Unterstützung für dynamisches Binden hinzugefügt. Der Inhalt von Mach-O Dateien ist in sogenannte Segmente aufgeteilt, die jeweils in Sektionen unterteilt sind. Eine Datenstruktur am Anfang der Datei (*load commands*) beschreibt den Inhalt der Segmente und Sektionen und wie sie in den Speicher geladen werden sollen bzw. wie der Linker sie zu einem lauffähigen Programm zusammenfügen muss.

Da es Mac OS sowohl für PowerPC (32-bit und 64-bit) als auch für Intel x86 basierende Systeme gibt, entwickelte Apple ein einfaches Archivformat als Erweiterung, welches es ermöglicht, Binärdaten für verschiedene Architekturen in einer Datei abzulegen. Diese sogenannten *PPC/PPC64binaries* bzw. *universal binaries* enthalten die kompletten Mach-O Daten für verschiedene Architekturen. Eine Tabelle am Anfang der Datei gibt über die enthaltenen Mach-O Daten Aufschluss. Das Betriebssystem ist somit in der Lage, den entsprechenden Code in Abhängigkeit der Architektur auszuwählen.

Das Speichern von Debuginformationen ist in Mach-O Dateien nicht vorgesehen. Üblicherweise werden *stabs*-Debugstrings in der Symboltabelle abgelegt.

Vergleich und Bewertung der Formate

Die meisten heute gebräuchlichen Binärformate eignen sich, um Module zu definieren. Ältere Formate wie beispielsweise das a.out sind weniger geeignet, da durch die Einschränkung auf sieben Sektionen die Struktur der Software nicht immer ausreichend beschrieben werden kann.

Das COFF-Format, in der ursprünglichen Fassung, ist ebenfalls weniger geeignet. Die Längenbeschränkung der Sektionsnamen auf acht Zeichen erschwert eine nachvollziehbare Aufteilung der Struktur unnötig. Die in COFF zusätzlich integrierten Debuginformationen sind für unser Ziel ebenfalls nicht ausreichend, da sich die Typinformationen auf einfache Standardtypen beschränken.

Besser geeignet sind die Erweiterungen des COFF-Formats. Hier ist im Speziellen Microsofts PE-COFF zu nennen. Durch die weite Verbreitung von Windows ist auch dieses Format gut unterstützt und auch im Bereich der eingebetteten Systeme verbreitet. Die extern gespeicherten Debuginformationen enthalten umfangreiche Typinformationen, jedoch ist das Format nicht offiziell beschrieben. Die angebotene Bibliothek zum Einlesen und Verarbeiten der Informationen ist nur für Windows-Betriebssysteme verfügbar.

Daher werden wir im Weiteren dieser Arbeit das ELF-Format einsetzen. Es bietet ebenfalls die Möglichkeit, beliebig viele Sektionen zu definieren und damit die Struktur der Software auszudrücken. Die Erstellung und Verarbeitung wird durch die frei verfügbare *Binary Format Description* Bibliothek (*BFD*) unterstützt. Auf dieser Bibliothek basieren viele andere Softwarewerkzeuge wie beispielsweise die *GNU Compiler Collection* (*GCC*). Die GCC ist weit verbreitet und bietet verschiedene Compiler für eine Vielzahl von Architekturen an. Zudem ist die einfache und automatische Integration von standardisierten und umfangreichen Debuginformationen (im DWARF-Format) möglich.

Mach-O wäre prinzipiell auch geeignet, da auch hier Möglichkeiten bestehen, die Struktur der Software feingranular darzustellen. Besonders interessant ist hier außerdem die Möglichkeit, binären Code für verschiedene Architekturen in einer einzigen Datei abzulegen. Das Mach-O Format wird jedoch kaum bei der Entwicklung von eingebetteten Systemen eingesetzt, daher ist die Werkzeugunterstützung auf wenige Architekturen beschränkt.

4.2.5 Erstellung der Module

Nachdem der grundsätzliche Aufbau und verschiedene Formate von Binärdateien betrachtet wurden, soll nun gezeigt werden, wie man daraus Module definieren kann. Als Ausgangsbasis werden Objektdateien im ELF-Format zugrunde gelegt.

Grober Aufbau einer ELF-Datei

Zunächst soll der Aufbau einer ELF-Objektdatei im Detail betrachtet werden. Abbildung 4.4 zeigt den Aufbau schematisch. Alle enthaltenen Daten werden in einzelnen ELF-Sektionen gespeichert. Eine Tabelle gibt Aufschluss über die Position und den Typ der einzelnen ELF-Sektionen. Der Compiler legt normalerweise mindestens zwei oder drei Sektionen mit binären Daten an. Eine für den Code, eine für die Daten und eine, welche den Platzbedarf der nicht initialisierten Daten beschreibt (`.bss`-Sektion). Zu jeder Sektion kann eine Relokationstabelle gespeichert sein, die auf Positionen innerhalb dieser Sektion wirkt. Relokationstabellen werden jeweils in eigenen ELF-Sektionen gespeichert. Daneben gibt es noch mindestens eine ELF-Sektion, die eine Symboltabelle enthält. Diese beschreibt die

öffentlichen Symbole der Sektionen und enthält die undefinierten Symbole, welche als Verweise auf andere Objektdateien dienen. Ein konkretes Beispiel des Inhalts war bereits in Abbildung 4.3 zu sehen.

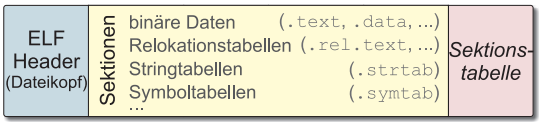


Abbildung 4.4: Schematischer Aufbau einer ELF-Objektdatei

Die Informationen in einer ELF-Datei sind in einzelnen Sektionen gespeichert. Die Sektionen sind in einer Tabelle, der sogenannten Sektionstabelle (*section header table*), beschrieben. Die Position und Länge dieser Tabelle ist wiederum im Dateikopf hinterlegt.
Es gibt verschiedene Arten von Sektionen. Einige Sektionen enthalten direkt binäre Daten, andere Sektionen enthalten beschreibende Informationen wie beispielsweise Relokations- oder Symboltabellen. Die Sektionstabelle beschreibt jede Sektion mit einem Typ und enthält weitere Informationen über die Zusammengehörigkeit. Für eine Relokationstabelle ist hier beispielsweise vermerkt, auf welche Sektion mit binären Daten sie wirkt und auf welche Symboltabelle sie sich bezieht.

Generelles Vorgehen

Aus diesen Informationen soll nun, als erster Ansatz, ein Modul pro Sektion erzeugt werden. Der Inhalt eines Moduls sind die Binärdaten der entsprechenden Sektion. Die Schnittstelle des Moduls ist durch die Symbolinformationen markiert. Jedes Symbol, das in die Sektion zeigt, stellt einen möglichen Einsprungpunkt für eine Funktion oder den Beginn einer Datenstruktur dar. Zur grundlegenden Definition der Struktur eines Moduls reichen die identifizierten Einsprungpunkte und Datenstrukturen aus. Dabei wird jedes Symbol verwendet, welches eine eindeutige Position innerhalb der Sektion beschreibt oder öffentlich sichtbar ist. Beschreiben zwei Symbole dieselbe Position und ist eines davon nur lokal, das heißt nur innerhalb der Übersetzungseinheit sichtbar, so wird nur das globale Symbol als Teil der Schnittstelle verwendet. Sind beide Symbole lokal, so enthält eines der Symbole meist mehr Informationen und beschreibt das Element an der Position genauer. Im Beispiel aus Abbildung 4.3 beschreibt das Symbol `.text` den Beginn der Sektion. Dieselbe Stelle wird auch von dem Symbol `hello` beschrieben. Das Symbol `hello` ist für unsere Aufgabe wertvoller, da es die Position als Beginn einer Funktion definiert.

Die Relokationstabelle zu der Sektion wird verwendet, um die Abhängigkeiten des Moduls zu extrahieren. Jede Relokation, deren Ziel außerhalb der Sektion liegt oder deren Ziel unbekannt ist, stellt eine Abhängigkeit zu einem anderen Modul dar. Zeigt eine Relokation auf ein lokales Symbol, welches durch den vorherigen Schritt nicht zur Schnittstelle gehört, so wird die Abhängigkeit auf das Symbol angepasst, das dieselbe Position beschreibt, aber Teil der Schnittstelle ist.

Abbildung 4.5 stellt das Ergebnis für das einfache Beispiel aus Abbildung 4.3 dar. Aus jeder der beiden Sektionen wurde hier ein Modul erstellt.

Mithilfe des beschriebenen Vorgehens können Module in der Granularität von Sektionen erstellt werden. Das Ziel ist jedoch, jede Funktion und Datenstruktur als Modul darzustellen. Möchte man dazu eine feinere Granularität erreichen, stehen prinzipiell zwei Möglichkeiten im Raum:

1. Man extrahiert die Elemente in einer Sektion und erstellt somit mehrere Module pro Sektion.

2. Man sorgt dafür, dass der Compiler kleinere Sektionen anlegt.

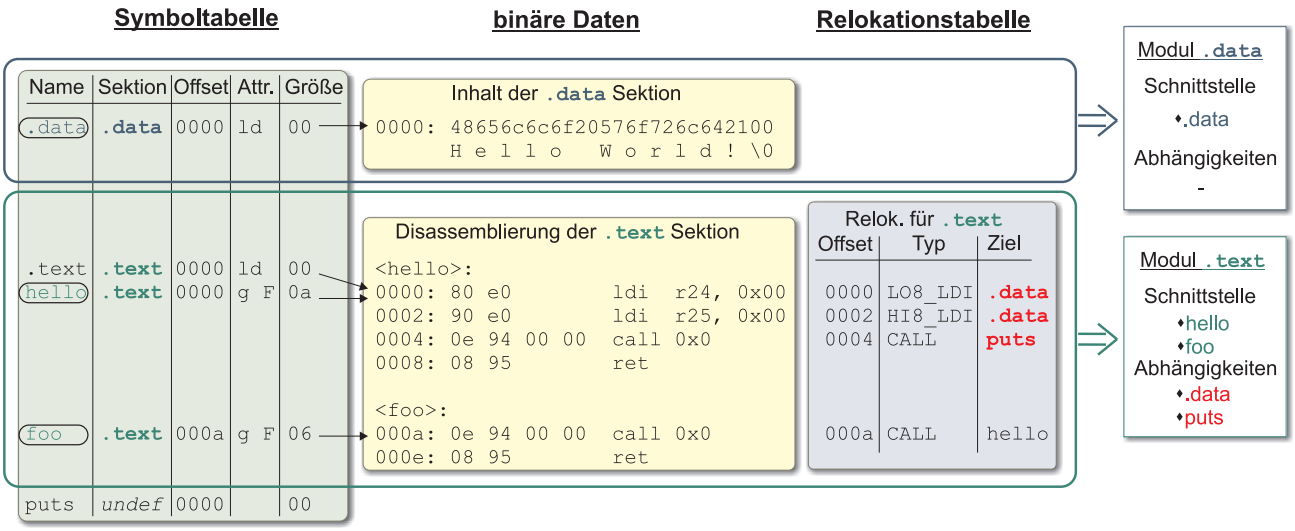


Abbildung 4.5: Erstellung von Modulen aus Text- und Datensektion

Aus den Sektionen der Objektdatei aus Abbildung 4.3 werden hier zwei Module erstellt. Oben wird aus der Datensektion ein Modul erstellt. Die Datensektion enthält genau ein Symbol, `.data`, dieses ist die Schnittstelle. Eine Relokationstabelle für die Datensektion gibt es in diesem Beispiel nicht. Daher hat das entsprechende Modul keine Abhängigkeiten. Unten entsteht das Modul für die Textsektion. In die Textsektion zeigen drei Symbole: `.text`, `hello` und `foo`. Die Symbole `.text` und `hello` zeigen jedoch auf dieselbe Position (`0`). Da `.text` sonst nicht referenziert wird und nicht öffentlich ist, trägt es nicht zur Schnittstelle bei. Das Symbol `hello` ist öffentlich sichtbar (das Attribut "`g`" bedeutet "global") und ist somit Teil der Schnittstelle. Die Schnittstelle besteht daher aus den Symbolen `hello` und `foo`. Bei der Analyse der Relokationstabelle stellen sich die Verweise auf `.data` und `puts` als externe Abhängigkeiten heraus. Der Verweis auf `hello` ist innerhalb des Moduls selbst und somit keine externe Abhängigkeit.

4.2.5.1 Extraktion von Elementen aus Sektionen

Ausgehend von den Symbolen werden Bereiche in den binären Daten identifiziert, die als Modul extrahiert werden. Das Symbol gibt die Position und die Länge der binären Daten an, die den Inhalt des Moduls bilden. Idealerweise handelt es sich dabei genau um eine Funktion oder ein Datenelement. Das Symbol selbst ist die Schnittstelle des Moduls. Alle Relokationen, die auf die ausgewählten binären Daten wirken, werden als Abhängigkeiten interpretiert.

Das Beispiel aus Abbildung 4.5 soll so in drei Module aufgeteilt werden. Die Textsektion beinhaltet zwei Symbole und lässt sich anhand dieser vollständig in die beiden Funktionen `hello` und `foo` aufteilen, wie in Abbildung 4.6 zu sehen ist.

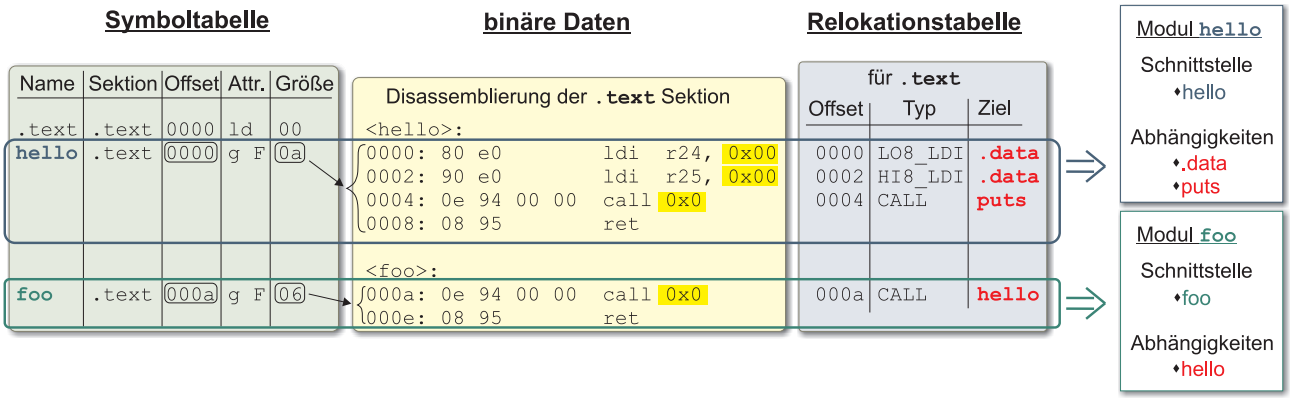


Abbildung 4.6: Teilen der Textsektion in zwei Module

Die Textsektion wird hier entsprechend der Symbole `hello` und `foo` in zwei Module aufgeteilt. Die Funktion `hello` beginnt am Anfang der Funktion und ist 10 (=`0xa`) Byte lang. Danach, an Position `0xa`, beginnt die Funktion `foo`, die sechs Byte lang ist und somit bis zum Ende der Sektion geht.
Die Relokationen werden entsprechend dieser Unterteilung den beiden Modulen zugeordnet. Das Modul `foo` ist daher von Modul `hello` abhängig, das Modul `hello` ist abhängig von `.data` und `puts`.

Die Datensektion beinhaltet nur den String für die Textausgabe und soll daher komplett als ein Modul verwendet werden. Hierbei stellt man fest, dass der Compiler kein eigenes Symbol für den Textstring erzeugt hat. Es steht nur ein Sektionssymbol zur Verfügung, das keine Datenstruktur, sondern den Beginn der Sektion beschreibt. Das Symbol enthält daher auch keine Größenangabe. Mit inhaltlichem Wissen über das dargestellte Beispiel stellt das kein Problem dar, da sich nur ein Datum in der Sektion befindet. Für die Erstellung eines automatischen Extraktionsalgorithmus sind die Daten jedoch nicht ausreichend.

Die Modularisierung der Anwendung ist nicht das ursprüngliche Ziel bei der Erstellung der Objektdateien. Informationen, welche im Quellcode noch vorhanden und für unseren Zweck notwendig sind, können bei der Übersetzung in Binärcode verloren gehen. Das Verhalten des Compilers muss daher genau betrachtet werden. Man muss feststellen, ob genügend Informationen erzeugt werden, um eine Modularisierung nach dem vorgeschlagenen Verfahren durchzuführen. Betrachten wir zwei mögliche Optimierungen eines Compilers, die das Extrahieren von Modulen auf die vorgestellte Art, erschweren oder sogar unmöglich machen.

Compiler-Optimierung 1: gemeinsames Symbol als Relokationsziel

Die erste Optimierung wurde schon am obigen Beispiel aufgezeigt. Anstatt ein Symbol für den String zu erzeugen, wird das Sektionssymbol verwendet, um die Daten zu referenzieren. Würde das Programm

mehrere Strings enthalten, so kann der Compiler sie alle mithilfe des Sektionssymbols und einem entsprechenden Versatz ansprechen und spart so die Erzeugung von zusätzlichen Symbolen. Diese Optimierung ist möglich, da die angesprochene Datenstruktur nur innerhalb der Datei verwendet werden kann. Der Compiler kann alle Verweise innerhalb einer Übersetzungseinheit mit beliebigen Symbolen darstellen, solange durch den Linker die richtige Position eingefügt wird. Für Daten und Funktionen, die über keinen anderen Weg angesprochen werden können, muss er keine zusätzlichen Symbole erzeugen. Das gilt somit nicht nur für die implizit angelegten Daten wie beispielsweise Zeichenketten, die der Compiler automatisch verwaltet, sondern auch für private (`static`) benutzerdefinierte Datenstrukturen und Funktionen, die nicht von anderen Übersetzungseinheiten erreichbar sind. Abbildung 4.7 zeigt ein einfaches Beispiel mit zwei privaten Funktionen. Beide werden mithilfe des Sektionssysmbols angesprungen. Die Auftrennung in verschiedene Module wird dabei auf zwei Arten erschwert:

1. Der Compiler erzeugt keine expliziten Symbole für private Elemente. Dadurch ist das Auffinden der Modulgrenzen, also dem Ende einer Funktion oder Datenstruktur, nicht mehr ohne zusätzliche Informationen möglich. Als zusätzliche Informationsquelle eignen sich hier Debuginformationen. Sie enthalten üblicherweise die benötigten Daten. Oft reicht auch das Übersetzen mit Debuginformationen aus, da der Compiler dann die gewünschten Symbole erzeugt, um die Fehlersuche zu vereinfachen.

2. Wenn es gelingt, die Binärdaten in einzelne Elemente zu untergliedern, müssen noch die Abhängigkeiten der potenziellen Module extrahiert werden. Viele Relokationen zeigen jedoch auf dasselbe Symbol. Es muss daher der Versatz ermittelt werden, um anschließend das Zielmodul zu bestimmen.
   Das ELF-Format sieht zwei Arten vor, Relokationen zu spezifizieren. Bei der einen Art enthält eine Relokation neben dem Zielsymbol zusätzlich einen Versatz. Bei der anderen Art kann kein Versatz mit angegeben werden. Welche Art verwendet wird, kann prinzipiell der Compiler bestimmen, es haben sich jedoch architekturspezifische Standards etabliert[2].
   Bei der ersten Art ist die Bestimmung der genauen Zielposition architekturunabhängig möglich, da der Versatz in standardisierten ELF-Datenstrukturen angegeben ist. Bei der zweiten Art ist zwar kein Versatz in den ELF-Datenstrukturen angegeben, dennoch kann mit einem Versatz gearbeitet werden. Dazu sind für einige Architekturen Relokationstypen definiert, bei denen die Zieladresse zu dem Wert an der Wirkungsposition addiert wird. Auf diese Art und Weise kann der Compiler auch hier ein Symbol für mehrere Ziele einsetzen. Um die exakte Zieladresse einer Relokation zu ermitteln und damit das entsprechende Zielmodul zu bestimmen, müssen die architekturspezifischen Relokationstypen bekannt sein und analysiert werden.

Compiler-Optimierung 2: direkte relative Sprünge

Eine zweite Optimierung betrifft die Verwendung von Sprungbefehlen. Viele Architekturen unterstützen zwei Arten von Sprungzielen. Entweder eine absolute Position, also die absolute Zieladresse, oder eine relative Position, hierbei handelt es sich meistens um eine positive oder negative Anzahl von Bytes, um die der aktuelle Befehlszähler verändert wird. Relative Sprünge benötigen oft weniger Platz, da häufig der Bereich, der angesprungen werden kann, eingeschränkt ist.

Compiler, die möglichst kleinen Code erzeugen, werden daher relative Sprünge einsetzen, wenn dies möglich ist. Innerhalb einer Sektion ist das auch häufig der Fall, da die Abstände oft klein genug sind.

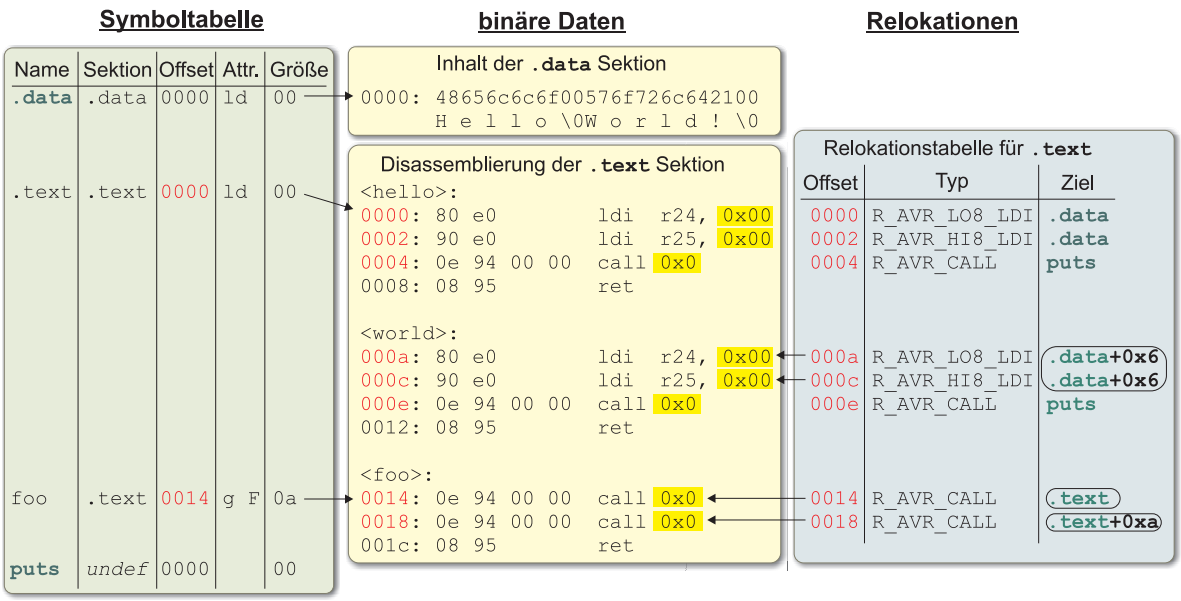
[2] Der GCC für Intel x86 Plattformen verwendet Relokationen *ohne* explizit spezifizierten Versatz. Der GCC für Renesas H8 oder Atmel AVR 8 Mikrocontroller erzeugt Relokationen *mit* Angaben zu einem möglichen Versatz.

```
static void world() {
      printf ("World!\n");
}

static void hello() {
      printf ("Hello\n");
}

void foo() {
      hello();
      world();
}
```

(a) Quellcode



(b) Binärdaten in ELF-Datei

Abbildung 4.7: Ein Symbol als Ziel mehrerer Relokationen

Dieses Beispiel enthält zwei lokale Funktionen `hello` und `world`. Beide werden von der Funktion `foo` aufgerufen. Da die beiden Funktionen jedoch lokal sind, erzeugt der Compiler keine Symbole für sie. Stattdessen verwendet er das Sektionssymbol `.text` in den Aufrufen aus der Funktion `foo`, wie anhand der Relokationstabelle zu sehen ist.
Dasselbe Vorgehen kann man auch bei der Verwendung der Datensektion beobachten. Sowohl in der Funktion `hello` als auch in der Funktion `world` wird das Sektionssymbol `.data` verwendet, um den entsprechenden String zu lokalisieren.
Die Informationen in der Symboltabelle reichen in diesem Beispiel nicht aus, um den Inhalt der Sektionen in einzelne Funktionen oder Datenelemente aufzuteilen.

Außerdem stellt eine Sektion für den Compiler eine zusammenhängende untrennbare Einheit dar, die der Binder als Ganzes im Speicher des Zielsystems oder -prozesses platziert.
Wenn der Compiler einen relativen Sprung innerhalb einer Sektion verwenden kann, so kann er zusätzlich eine weitere Optimierung durchführen. Unter der Annahme, dass eine Sektion ein zusammenhängender Block ist, sind der Abstand und die relative Position der Symbole innerhalb einer Sektion zueinander konstant. Dies erlaubt dem Compiler, das Ziel des Sprungs direkt einzutragen und keine Relokation zu verwenden. Abbildung 4.8 zeigt das Beispiel aus Abbildung 4.3 mit relativen Sprungbefehlen[3].

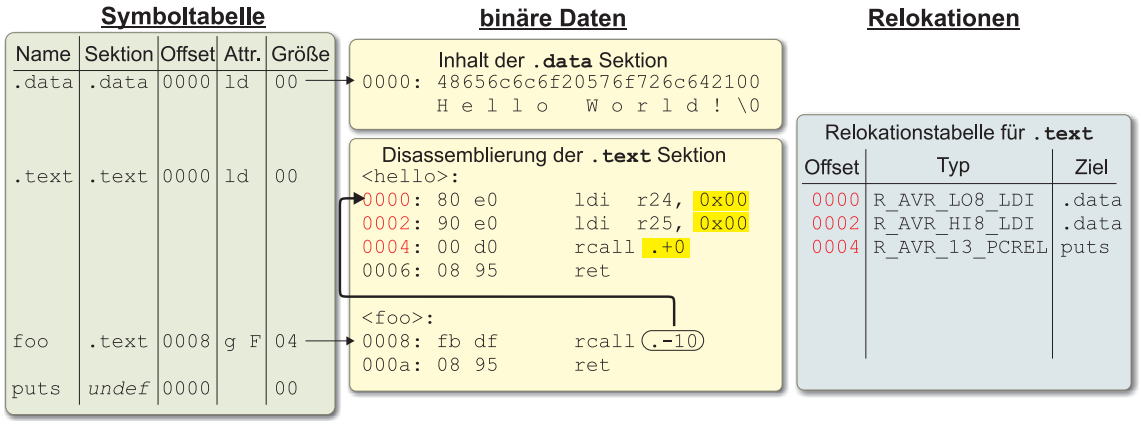


Abbildung 4.8: Sprungziel direkt als relativer Sprung innerhalb der Sektion codiert
Die Abbildung zeigt das einfache Beispiel aus Abbildung 4.3 mit der Option `-mshort-calls` übersetzt. Der Aufruf von `hello` aus der Funktion `foo` ist deshalb mithilfe eines relativen Sprungs realisiert. Da beide Funktionen innerhalb derselben Sektion liegen, geht der Compiler davon aus, dass sich ihre Position zueinander nicht verändert. Das Sprungziel ist daher immer 10 Byte vom Aufruf entfernt, der Compiler kann diese Information direkt eintragen.
Anhand der Relokationstabelle erkennt man nicht mehr, dass eine Abhängigkeit zwischen `foo` und `hello` besteht.

Da das Sprungziel nun nicht mithilfe einer Relokation aufgelöst wird, kann die Abhängigkeit der beteiligten Funktionen nicht an der Relokationstabelle abgelesen werden. Daher wird beim Erstellen der Module, nach dem oben angegebenen Verfahren, die Abhängigkeit zwischen den beiden Modulen nicht erkannt. Werden die Module anschließend unabhängig voneinander verwendet, ist nicht sichergestellt, dass die relative Position der beiden Funktionen zueinander gleich bleibt. An den Modulen lässt sich auch nicht erkennen, dass zur korrekten Funktion des einen Moduls das andere notwendig ist. Als Folge würde der Sprung an eine falsche Adresse durchgeführt werden und unvorhersehbare Auswirkungen nach sich ziehen. Um diese Optimierung zu erkennen, müsste eine Codeanalyse durchgeführt werden, um alle Sprungbefehle zu finden und deren Position mit den vorhandenen Relokationen abzugleichen. Eine solche Analyse ist jedoch architekturspezifisch und aufwendig. Als Alternative müsste man sicherstellen, dass der Compiler diese Optimierung nicht durchführt.

4.2.5.2 Kleinere Sektionen durch Compiler

Ein anderer Ansatz, Module mit der Granularität von Funktionen und Datenstrukturen zu erstellen ist, jede Funktion und jede Datenstruktur in einer eigenen Sektion abzulegen. Dadurch wird sichergestellt, dass die vorgestellten Optimierungen nicht zu Problemen führen. Da der Compiler kein Wissen über die relativen Positionen der einzelnen Funktionen und Datenstrukturen zueinander hat, kann er keine

[3]Das Beispiel ist hierbei für einen kleineren Mikrocontroller aus der Atmel AVR Familie übersetzt worden. Der Datenspeicher dieses Typs ist nur 1 KByte groß, sodass der GCC hier immer relative Sprünge generiert.

relativen Sprünge zwischen den Funktionen verwenden. Gleichzeitig kann er nicht ein Symbol für Sprünge oder Zugriffe auf mehreren Funktionen oder Daten verwenden.

Wird jedes Element in einer eigenen Sektion abgelegt, sind somit die Verknüpfungen mit anderen Modulen vollständig sichtbar und durch die Relokationstabellen beschrieben. Außerdem sind alle Einsprungpunkte durch Symbole gekennzeichnet.

Als Nachteil dieses Verfahrens kann man anführen, dass einige Optimierungen nicht mehr durchgeführt werden. Das kann jedoch teilweise entkräftet werden. So kann beispielsweise ein entsprechender Binder (ein sogenannter *relaxing Linker*) während des Bindens absolute Sprünge durch relative Sprünge ersetzen, falls es die Entfernung und der umgebende Code erlauben.

Um jedes Element in eine eigene Sektion abzulegen, gibt es zwei Möglichkeiten:

- Viele Compiler lassen sich durch spezielle Anweisungen im Quellcode steuern. Man könnte somit für jede Funktion und jede Datenstruktur im Quellcode angeben, in welcher Sektion sie abgelegt werden soll[4]. Dieser Vorgang lässt sich auch automatisieren, beispielsweise durch den Einsatz von Techniken der aspektorientierten Programmierung. Zu beachten ist allerdings, dass die Anweisungen im Quellcode compilerabhängig sind.

- Alternativ bieten die meisten Compiler von sich aus die Möglichkeit, einzelne Sektionen für jede Funktion und Datenstruktur zu erzeugen. Meistens lässt sich dieses Verhalten durch einen entsprechenden Aufrufparameter aktivieren[5]. Dieser Parameter ist zwar ebenfalls compilerabhängig, jedoch ist der Compileraufruf an sich für jeden Compiler anzupassen.

Abbildung 4.9 zeigt anhand des einfachen Beispiels, wie der Inhalt einer Binärdatei mit den entsprechenden Optionen übersetzt in einzelne Sektionen aufgeteilt ist.

4.2.5.3 Ergebnisse

In diesem Abschnitt wurde eine Möglichkeit vorgestellt, eine Software in Module aufzuteilen. Ausgangspunkt waren dabei Objektdateien, wie sie zum Zeitpunkt des Bindens vorliegen. Ausgehend von der Struktur solcher Dateien wurden Sektionen als Module abgebildet. Es wurden anschließend zwei Möglichkeiten vorgestellt, Module für einzelne Funktionen und Datenstrukturen zu erstellen.

Bei der ersten Möglichkeit werden Elemente innerhalb einer Sektion anhand der Symbole identifiziert und zu Modulen transformiert. Es wurde gezeigt, dass dieses Vorgehen nicht ohne zusätzliche Informationen durchzuführen ist. Die zusätzlich benötigten Informationen können zum Teil automatisch generiert werden, teilweise ist jedoch eine aufwendige Codeanalyse notwendig.

Beim zweiten vorgestellten Verfahren wird der Compiler angewiesen, jede Funktion in eine separate Sektion abzulegen. Anschließend kann jede Sektion in ein Modul überführt werden. Falls der Compiler

[4] Beim weit verbreiteten C-Compiler aus der GCC-Sammlung und bei Intels optimierendem Compiler für x86 (`ICC`) kann beim Prototyp einer Funktion mit dem Attribut `section` die gewünschte Sektion ausgewählt werden. Zum Beispiel:

```
void hello() __attribute__ ((section(".text.hello")));
```

Bei Microsofts Visual C/C++ Compiler wird mit einer `pragma`-Anweisung die Sektion der nachfolgenden Elemente bestimmt. Beispiel:

```
#pragma code_seg(".text.hello")
void hello() { /* Implementierung */ }
```

[5] Beim GNU C-Compiler führen die Kommandozeilenoptionen `-ffunction-sections` und `-fdata-sections` dazu, dass jede Funktion und jedes Datum in einer eigenen Sektion abgelegt wird. Andere Compiler bieten ähnliche Optionen an. Der Microsoft Visual C/C++-Compiler beispielsweise erzeugt mit der Option `/Gy` ein ähnliches Resultat.

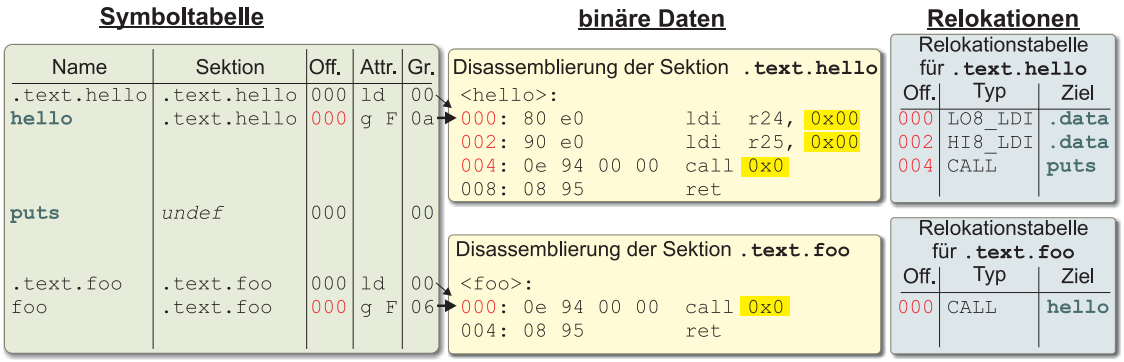


Abbildung 4.9: Eigene Sektionen für jede Funktion
Dieses Mal wurde das einfache Beispiel aus Abbildung 4.3 mit der Option `-ffunction-sections` übersetzt. Man erkennt, dass der Compiler für jede Funktion eine eigene Sektion erstellt hat. Wendet man nun das Verfahren aus Abschnitt 4.2.5 an und erzeugt aus jeder Sektion ein Modul, so erhält man Module, die genau einer Funktion entsprechen.

die automatische Aufteilung in einzelne Sektionen unterstützt, so ist dieser Weg dem ersten vorzuziehen, da er keinen zusätzlichen Aufwand verursacht.

Zur Modularisierung der Datensektion sind beide Verfahren als gleichwertig anzusehen. Um die einzelnen Daten jedoch auch ohne zusätzliche Informationen zu identifizieren, wird ebenfalls das zweite Verfahren bevorzugt.

Abschließend ist festzuhalten, dass es durch das vorgestellte Verfahren möglich ist, Module aus binären Objektdateien zu extrahieren, ohne zusätzlich manuell Informationen anzugeben. Alle benötigten Informationen können automatisch im Rahmen des Übersetzungsvorganges erzeugt werden.

4.2.6 Debuginformationen

Neben den Metainformationen, die in den Objektdateien zum Binden vorhanden sind, stellen Debuginformationen eine weitere Informationsquelle dar. Sie sind besonders für das entfernte Ausführen interessant, da sie die vorhandenen Funktionen in der Objektdatei beschreiben und somit Informationen über die Schnittstellen von Modulen zur Verfügung stellen. Hierbei sind insbesondere die Typen der erwarteten Parameter von Bedeutung. Falls es sich nicht um einfache Standardtypen handelt, ist auch Wissen über den Aufbau und die Zusammensetzung der Typen notwendig.

Bevor wir verschiedene Debugformate vorstellen, betrachten wir zunächst die Informationen, die enthalten sein können.

Zeilennummerninformationen Da Debuginformationen primär zum Auffinden von Fehlern gedacht sind, enthalten alle Formate Informationen, um dem Binärcode einzelne Zeilen des Quellcodes zuzuordnen. Bei der Verwendung einer Sprache, die zunächst in einen Zwischencode überführt wird, wie zum Beispiel Java, kann die Abbildung von Binärcode auf Quellcode auch zweistufig erfolgen (Binärcode →Java-Bytecode →Java-Quellcode).

Namen Eine wichtige Information zum Identifizieren von Elementen ist der Name. Bei manchen Binärformaten ist der Name in den Symbolinformationen gespeichert. Der Symbolnamen entspricht jedoch nicht immer dem Namen des Elements, ein Beispiel sind die um die Signatur angereicherten Symbolnamen bei der Übersetzung von C++ Code. Einige Debuginformationen enthalten daher zusätzlich den Namen von Funktionen und Variablen.

Positionen Weiterhin ist die Position jedes Elements in den Debuginformationen gespeichert. Die Position kann dabei als Ort innerhalb der Binärdatei angegeben sein oder als Adresse im laufenden Programm. Im ersten Fall müssen die Angaben entsprechend der Abbildung der binären Daten in den Speicher umgerechnet werden, um sie zur Fehlersuche zu benutzen. Im heute üblicheren zweiten Fall trägt der Binder die Position über einen Eintrag in der Relokationstabelle in die Debuginformationen ein. Das bedeutet jedoch, dass auch die Debuginformationen erst gebunden werden müssen, bevor sie gültige Adressen enthalten.

Größe Um Elemente anhand ihrer Adresse zu bestimmen, enthalten Debuginformationen neben der Position auch die Größe der Elemente. Diese Information ist zwar auch in den Symbolinformationen einiger Binärformate gespeichert, aber nicht in allen und nicht für alle Elemente.

Typinformationen Um den Typ von Variablen und Funktionsparametern zu beschreiben, enthalten Debuginformationen oft Typbeschreibungen. Zur Fehlersuche sind diese Informationen hilfreich, um sich beispielsweise den Inhalt einer Variablen darstellen zu lassen. Zur Unterstützung komplexer, zusammengesetzter Datentypen, wie zum Beispiel Strukturen, ist eine genaue Beschreibung der enthaltenen Elemente notwendig. Für jedes Element müssen neben dem Namen auch die Position und die Größe innerhalb des Datentyps beschrieben werden.

Variablen und Funktionen Mithilfe der beschriebenen Attribute werden in den Debuginformationen die Elemente einer Objektdatei beschrieben. Variablen können mit ihrem Namen, der Position, der Größe und einem Typ beschrieben werden. Zu Funktionen kann die Signatur angegeben werden, das heißt der Name der Funktion, der Typ des Rückgabewertes und die Anzahl, Namen und Typen der Parameter. Bei manchen Debuginformationen ist auch die Position der Parameter gegeben.

4.2.6.1 Debuginformationsformate

Im Folgenden sollen einige wichtige Formate zur Speicherung von Debuginformationen vorgestellt werden.

stabs

Die *stabs*-Debuginformationen [MKM06] wurden ursprünglich entwickelt, um dem a.out-Binärformat Informationen zur Fehlersuche hinzuzufügen. Da a.out nicht die Möglichkeit bietet, beliebige Zusatzinformationen abzuspeichern, werden die Informationen als Texteinträge in der Symboltabelle abgelegt. Daher stammt auch der Name *stabs*, als Abkürzung für *symbol table strings*.

Das stabs-Format war, vor allem auf UNIX-Systemen, sehr weit verbreitet, da es durch die Art der Speicherung mit fast jedem Binärformat zusammen verwendet werden kann. Das stabs-Format war ursprünglich sehr einfach, wurde jedoch mehrfach in verschiedener Weise erweitert und ist heute sogar relativ komplex. Neben Zeilennummerinformationen können Typen und Parameter von Funktionen beschrieben werden. Spezielle stabs-Versionen unterstützen sogar die Besonderheiten von objektorientierten Programmen.

Durch die verschiedenen Erweiterungen entstanden jedoch eine ganze Reihe von zueinander inkompatiblen Versionen des stabs-Formats. Die fehlende Spezifikation und die Möglichkeit, in modernen Binärformaten benutzerdefinierte zusätzliche Daten zu speichern, sind die Hauptgründe dafür, dass *stabs* heute oft durch DWARF ersetzt wird.

COFF

Beim Entwurf des COFF-Binärformats [Gir88, Del96] wurden von Anfang an mehr Informationen integriert, um eine spätere Fehlersuche zu ermöglichen. So sind die Symboltabellen erweitert und enthalten zu jedem Eintrag die Art und den Typ des beschriebenen Datums. Auch das Definieren von eigenen Typen wird in den COFF-Debuginformationen unterstützt. Zusätzlich wurde eine Tabelle mit Zeilennummerinformationen hinzugefügt.

Nachteil der COFF-Debuginformationen ist, dass das Format kaum Erweiterungsmöglichkeiten vorsieht und die angebotenen Informationen nur die grundlegenden Daten der Elemente beschreibt. So wird beispielsweise die Größe von Daten oder Funktionen nicht explizit angegeben. Außerdem werden von den Compilern oft nur Debuginformationen für global sichtbare Elemente erzeugt, nicht aber für lokale Variablen oder lokale (statische) Funktionen.

Microsoft CodeView und PDB

Microsoft verwendet zwar für die Windows-Betriebssysteme eine abgewandelte Version des COFF-Formats, um die Binärdateien zu speichern; die Debuginformationen innerhalb von COFF werden jedoch nicht verwendet. Stattdessen werden zusätzliche Informationen in einem proprietären Format erzeugt und gespeichert. Das Format hat sich dabei im Laufe der Zeit verändert. Zunächst wurden die Informationen in einer eigenen Sektion innerhalb der PECOFF-Datei im *CodeView* Format (*CV4*) [TIS93] gespeichert. Mit Visual Studio 6.0 führte Microsoft die *Programm DataBase* (*PDB*) ein, welche die Debuginformationen außerhalb der Binärdatei in einer eigenen Datei abspeichert [Mic05]. Die ursprüngliche Version (PDB 2.0) wurde in nachfolgenden Versionen des Visual Studios (Visual Studio .NET) durch eine erweiterte, neue, aber inkompatible Version (PDB 7.0) ersetzt.

Während das CodeView-Format noch offiziell dokumentiert ist [SS+96], werden für die *Programm DataBase* lediglich Bibliotheken angeboten, um auf die Daten zuzugreifen[6]. Anhand der Schnittstellen kann man jedoch schließen, dass die PDB umfangreiche Debuginformationen enthält.

DWARF

Das *DWARF*[7]-Format gibt es mittlerweile in der Version 3. Die erste Version entstand in den 1980er Jahren, zusammen mit dem ELF-Binärformat, als Format des Debuggers *sdb* im Unix System V Release 4 (SVR4). Ein großer Nachteil des Formats war jedoch, dass die Informationen nicht sehr effizient gespeichert wurden und somit unnötig viel Platz benötigten. DWARF 2 entschärfte diesen Nachteil und speichert die Daten effizienter. Dafür ist DWARF 2 vollkommen inkompatibel zu DWARF 1. Erst Ende der 1990er Jahre etablierte sich DWARF zum Standarddebugformat für Unix-Systeme, vorher wurde zum Teil ELF mit stabs-Informationen verwendet. Da die Standardisierung von DWARF 2 nie abgeschlossen wurde und Bedarf für die Unterstützung von 64-bit-Systemen entstand, wurde im Jahr 2006 DWARF 3 als kompatibler Nachfolger standardisiert [Fsg05]. Eine Einführung und Hintergründe zur Geschichte des DWARF-Formats kann man bei Michael Eager [Eag07] nachlesen.

Das DWARF-Format bietet Unterstützung zur Beschreibung der üblichen Debuginformationen. Besonders sind dabei die umfangreichen und flexiblen Möglichkeiten zu nennen, um die Position einer Variable oder eines Parameters anzugeben. In DWARF 2 wurde die Möglichkeit hinzugefügt,

[6] Zum Zugriff auf die *Programm DataBase* bietet Microsoft die Bibliothek *DbgHlp* und das umfangreichere Entwicklungspaket *DIA* (*Debug Interface Access SDK*) an.

[7] Die Bezeichnung DWARF wird häufig als Akronym für *Debugging With Attributed Record Format* gedeutet. Tatsächlich taucht diese Bezeichnung jedoch nie in einer der DWARF-Spezifikationen auf. Es wird vermutet, dass der Name DWARF als Wortspiel auf das Binärformat ELF entstanden ist [Eag07].

Informationen über C++ Programme zu speichern. In der Version 3 wurde diese Möglichkeit noch erweitert und auch Unterstützung für Java Programme integriert.

Vergleich und Bewertung der Formate

Alle vorgestellten Formate stellen die grundlegenden Informationen bereit, die zum Debuggen eines Programms notwendig sind. Für unser Ziel ist die Beschreibung der Funktionen und der Datentypen besonders wichtig. stabs und COFF beschreiben den Aufbau von zusammengesetzten Typen nur ungenau, da lediglich die Position der einzelnen Elemente innerhalb des Typs angegeben ist, nicht jedoch ihre Größe. Es gibt Erweiterungen beider Formate, die diesen Schwachpunkt ausgleichen, diese werden jedoch nicht allgemein verwendet und sind somit spezifisch für einen Compiler.

In Microsofts Debuginformationen sind die gewünschten Informationen vorhanden. Die unzureichende Dokumentation der neueren Formate und die Einschränkung auf die Windows-Plattformen schränken die Nutzung dieses Formats jedoch stark ein.

Wir konzentrieren uns daher im Folgenden auf das DWARF-Format. Es enthält ebenfalls alle benötigten Daten und ist durch die Spezifikation ausführlich beschrieben. Darüber hinaus handelt es sich um ein modernes Format mit Unterstützung für Compiler-Optimierungen, wie beispielsweise das Einbetten einer Funktion in eine andere (*Inlinig*) und objektorientierter Sprachen.

4.2.6.2 Verbesserung der Modulerzeugung durch Debuginformationen

In Abschnitt 4.2.5 wurde dargestellt, wie Module aus den Sektionen einer Objektdatei erstellt werden. Dabei wurde auch auf die Problematik hingewiesen, dass die normalen Symbolinformationen nicht immer genügend Informationen enthalten, um die Elemente zu identifizieren. Debuginformationen können hier zusätzliche Informationen liefern. Bei entsprechend detaillierten Debuginformationen sind die Positionen und Größen aller Funktionen und Datenstrukturen gegeben. Diese Informationen können dazu genutzt werden, um vorhandene Symbole zu vervollständigen oder um neue Symbole für Datenstrukturen und lokale Funktionen zu erstellen, für die keine Symbolinformationen vorhanden sind.

Darüber hinaus bieten einige Debuginformationsformate die Möglichkeit, eingebettete Funktionen zu erkennen. Falls eine Funktion nur eingebettet existiert, so wird sie komplett der umgebenden Funktion zugeschrieben und wird nicht durch ein Modul repräsentiert. Existiert die Funktion jedoch zusätzlich als normale Funktion, so wird ein unveränderlicher Verweis zwischen der Funktion und allen Funktionen, in die sie eingebettet ist, in den Moduldaten vermerkt. Anhand dessen kann das System nachvollziehen, dass Veränderungen an der eingebetteten Funktion zu Änderungen an den umgebenden Funktion führen muss.

4.2.6.3 Debuginformationen zur Datentypbestimmung

Debuginformationen können dazu genutzt werden, den Datentyp eines Datenmoduls festzustellen. Sind einzelne Datenstrukturen als Modul identifiziert, so ist es möglich, mithilfe ihrer Position einen entsprechenden Eintrag in den Debuginformationen zu suchen. Die darin enthaltenen Typinformationen sind dann dem Modul zu zuordnen.

Der Datentyp wird bei entfernten Operationen benötigt, bei denen ein Datenmodul auf einen anderen Knoten verschoben oder kopiert wird. Wenn man beispielsweise ein Funktionsmodul auf einen anderen Knoten auslagert, so werden dort die abhängigen Datenmodule benötigt. Um sie ordnungsgemäß zu übertragen, müssen der Typ und die Struktur der Daten bekannt sein. Handelt es sich bei dem

Datenmodul um einen Zeiger, so soll unter Umständen auch das Ziel des Zeigers kopiert werden. Um diese Situation überhaupt festzustellen, muss der genaue Typ des Datenmoduls bekannt sein.

Wird ein Datenmodul lediglich geparkt, so sind solche detaillierten Informationen im Allgemeinen nicht notwendig, da die Größe der Datenstruktur für diese Operation ausreicht.

4.2.6.4 Debuginformationen als Schnittstellenbeschreibung

Analog zur Typbestimmung von Datenmodulen können Debuginformationen dazu genutzt werden, Funktionsmodule mit genauen Informationen über die Schnittstelle der Funktion anzureichern. Auch hier wird durch die Position der Funktion das entsprechende Element in den Debuginformationen gesucht. Anschließend kann dem Funktionsmodul die Beschreibung der Schnittstelle zugeordnet werden.

Schnittstelleninformationen werden für das Auslagern von Funktionen auf einen entfernten Knoten benötigt. Ein Fernaufrufmechanismus muss die Parameter entsprechend transportieren und umwandeln können, hierzu benötigt man genaue Typinformationen. Für das einfachere Parken von Funktionen werden hingegen keine detaillierten Informationen über die Schnittstelle benötigt, da die Funktion auf dem fremden Knoten nicht ausgeführt wird.

4.2.6.5 Bewertung und Einschränkung bei der Verwendung von Debuginformationen

In diesem Abschnitt wurde der Informationsgehalt von Debuginformationen aufgezeigt. Die Nutzung dieser Informationen erlaubt es, den Inhalt beziehungsweise die Schnittstellen von Modulen genau zu beschreiben. Debuginformationen bieten dabei den Vorteil, dass sie automatisch erzeugt werden und es somit nicht erforderlich ist, manuell zusätzliche Informationen zu erfassen. Allerdings müssen Debuginformationen explizit erzeugt werden, was oft eine Neuübersetzung des Quellcodes nötig macht. Außerdem stößt die Verwendung von Debuginformationen an ihre Grenzen, da nur Informationen enthalten sein können, die auch im Quellcode vorhanden sind. In Abschnitt 4.5.8 wird gezeigt, dass diese Informationen nicht immer ausreichend sind und wie man zusätzliche Informationen einbringen kann, um dennoch von Debuginformationen zu profitieren.

4.2.7 Modularten

Für die Weiterverarbeitung der Module sind einige Eigenschaften besonders wichtig. Entsprechend dieser Eigenschaften kann man die Module in verschiedene Ausprägungen einteilen.

4.2.7.1 Klassifikation nach Inhalt

Je nach Inhalt des Moduls wird zwischen Funktions- und Datenmodulen unterschieden:

Funktionsmodule repräsentieren ausführbaren binären Code. Im Idealfall stellt ein Funktionsmodul genau eine Funktion dar, es können jedoch auch mehrere Funktionen in einem Funktionsmodul zusammengefasst sein. Für die Binärdaten von Funktionsmodulen gilt, dass sie von der Anwendung nur lesend verwendet werden.

Datenmodule stellen globale oder temporär auch lokale Daten des Programms dar. Auch hier soll ein Datenmodul im angestrebten Fall genau eine Variable repräsentieren. Datenmodule an sich können jedoch auch mehrere Datenstrukturen enthalten. Im Gegensatz zu Funktionsmodulen werden die binären Daten von Datenmodulen zur Laufzeit des Programms verändert. Der binäre

Inhalt eines Datenmoduls stellt somit immer nur einen zeitlich begrenzt geltenden Zustand des Programms dar.

Die Aufteilung in diese beiden Gruppen bestimmt maßgeblich die weitere Verarbeitung und die möglichen Operationen, die auf ein Modul anwendbar sind.

4.2.7.2 Klassifikation nach Granularität

Funktions- wie auch Datenmodule gibt es in verschiedener Granularität. Sie können entweder durch nur ein Symbol beschrieben sein oder durch mehrere:

Einzelsymbolmodule werden durch die Extraktion von einzelnen Funktionen oder Variablen gebildet. Ein solches Modul enthält nur einen öffentlichen Einsprungpunkt und der liegt am Anfang der enthaltenen binären Daten. Das Bestreben bei der Extraktion von Modulen ist es, solche Einzelsymbolmodule zu erzeugen. Sie stellen die feinste Aufteilung dar und können am flexibelsten verwendet werden.

Multisymbolmodule werden bei der Extraktion erzeugt, wenn die Objektdatei nicht genügend Informationen enthält, um einzelne Funktionen oder Variablen zuverlässig zu trennen. Beispielsweise wird in handgeschriebenem Assemblercode oft nur der Symbolname angegeben, ohne zusätzlich den Typ oder die Länge zu setzen. Die übersetzte Objektdatei enthält dann nur ungenaue Symbolinformationen ohne Längenangaben, wodurch eine Trennung in einzelne Funktionen unzuverlässig wird. Multisymbolmodule zeichnen sich dadurch aus, dass sie mindestens zwei Einsprungpunkte haben.

Die Aufteilung in diese beiden Typen bestimmt die Nutzbarkeit eines Moduls, da Module die kleinste Einheit darstellen und immer als Ganzes verwendet werden. Bei Multifunktionsmodulen müssen somit immer alle enthaltenen Funktionen oder Daten gemeinsam verschoben, geparkt oder ausgelagert werden.

4.3 Basismechanismen zur Modulverwaltung

In Abschnitt 4.1 wurden bereits einige Operationen zur Modulverwaltung vorgestellt. Nachfolgend werden die Basismechanismen beschrieben, die zur Realisierung dieser Operationen eingesetzt werden.

4.3.1 Installieren von Modulen

Das Installieren von Modulen wird für verschiedene Operationen benötigt. Es wird beispielsweise im Rahmen einer bedarfsgesteuerten Installation vom verwalteten Gerät angefordert oder direkt von der Kontrollschicht, bei der Umkonfiguration der Software oder während des Austauschens von Modulen, verwendet. Die Installation von Modulen erfolgt in mehreren Schritten, die in Abbildung 4.10 zusammengefasst sind.

Voraussetzung um ein Modul zu installieren ist, dass alle seine Abhängigkeiten erfüllt sind. Vor der Installation wird daher überprüft, dass alle Module, von denen ein ausgewähltes Modul abhängt, entweder bereits auf dem Zielgerät installiert sind oder ebenfalls zur Installation ausgewählt sind. Durch diese Überprüfung wird sichergestellt, dass die transitive Hülle der zu installierenden und bereits installierten Module bezüglich ihrer Abhängigkeiten in sich abgeschlossen ist.

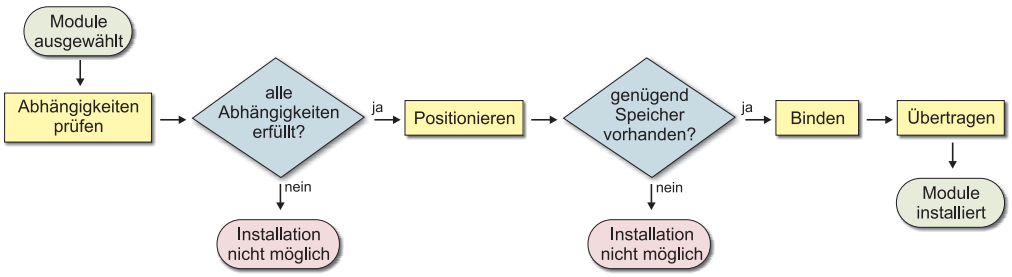


Abbildung 4.10: Vorgehen bei der Installation von Modulen

Als Folge dieser Überprüfung können nicht beliebige Module einzeln installiert werden. Nur Module, die gar keine Abhängigkeiten haben oder deren Abhängigkeiten bereits installiert sind, können einzeln installiert werden. Für alle anderen Module müssen zusätzlich die abhängigen Module zur Installation ausgewählt werden.

Bevor der Code zum verwalteten Gerät übertragen wird, muss festgelegt werden, an welcher Position die Module im Speicher des Zielsystems installiert und welche Adresse sie haben werden. Diese Aufgabe wird von der Speicherverwaltung durchgeführt (siehe Abschnitt 4.5.4). Im Rahmen dessen wird auch überprüft, ob überhaupt genügend freier Speicher zur Verfügung steht. Reicht der vorhandene freie Speicher auf dem Zielknoten nicht aus, so wird das an die Kontrollschicht gemeldet. Diese kann daraufhin zunächst andere Module von dem Knoten entfernen oder parken, um so genügend Speicherplatz freizubekommen.

Nachdem die Module positioniert wurden, können die binären Daten der Module zur Installation vorbereitet werden. Dabei löst ein architekturspezifischer Binder die Abhängigkeiten auf. Hierfür hat er Zugriff auf die Speicherverwaltung, um die Position der installierten Module zu erfahren. Es entsteht eine Version der binären Daten, welche direkt von der Zielhardware ausgeführt beziehungsweise verwendet werden kann.

Im letzten Schritt werden die Module übertragen. Dazu muss eine Verbindung zum Zielknoten existieren und der Knoten muss in der Lage sein, den Code an die zuvor festgelegte Speicherstelle zu schreiben. Details hierzu werden in Abschnitt 4.5.7 beschrieben. In den Moduldaten wird zuletzt festgehalten, dass das Modul nun installiert ist.

4.3.2 Entfernen von Modulen

Das Entfernen von Modulen ist keine Einzeloperation, sondern wird immer im Zusammenhang mit einer anderen Operation durchgeführt. Der Grund hierfür ist, dass man nur Module entfernen kann, die nicht mehr referenziert werden. Solche Module entstehen jedoch nur durch andere Veränderungsoperationen. Ein anschauliches Beispiel ist das Austauschen einer Funktion durch einen Stellvertreter. Nach der Installation des Stellvertreters werden alle Verweise auf die ursprüngliche Funktion umgeschrieben. Die ursprüngliche Funktion wird dann nicht mehr benötigt und soll im Rahmen des Austauschens entfernt werden.

Beim Entfernen von Modulen muss unterschieden werden zwischen Modulen mit veränderlichen Daten und Modulen mit konstanten Daten. Sind die Daten veränderbar, muss zunächst der aktuelle Zustand, also der Inhalt des Speichers, den das Modul repräsentiert, vom verwalteten Gerät gesichert und im Modulobjekt abgespeichert werden. Auf diese Weise ist es möglich, den aktuellen Zustand bei einer erneuten Installation wieder herzustellen. Bei konstanten Modulen, wie zum Beispiel Funktionsmodulen ist das nicht notwendig.

Das eigentliche Entfernen eines unbenutzten Moduls ist anschließend relativ einfach. Dazu muss lediglich der Speicherbereich, der durch das Modul belegt wird, der Speicherverwaltung als frei gemeldet werden. Damit steht der Speicher wieder für andere Aufgaben zur Verfügung. Falls die Speicherverwaltung vom Verwalter durchgeführt wird, ist nicht einmal eine Interaktion mit dem Gerät notwendig.

Die eigentliche Aufgabe liegt jedoch darin, sicherzustellen, dass das Modul tatsächlich nicht mehr erforderlich ist. Das ist der Fall, wenn das Modul auf dem Knoten nicht mehr referenziert wird. Möglichkeiten und Probleme, dies festzustellen, sind in Abschnitt 4.4 beschrieben. Da das Entfernen eines Moduls immer als Teiloperation in Zusammenhang mit einer anderen Operation eingesetzt wird, ist die Abhängigkeitsprüfung nicht die Aufgabe dieses Mechanismus. Im Rahmen von anderen Operationen wird manchmal Code entfernt, obwohl er zu diesem Zeitpunkt streng genommen noch referenziert wird. Die anfordernde Operation trägt daher die Verantwortung dafür, dass keine Abhängigkeiten verletzt werden.

4.3.3 Dynamisches Austauschen von Modulen

Das Austauschen von Modulen ist eine weitere wichtige Operation. Sie kann für vielfältige Szenarios genutzt werden. So stellt das Austauschen einer Funktion durch einen Stellvertreter die Basis für das dynamische Auslagern und Parken von Funktionen dar. Mit dem Austauschen des Stellvertreters durch die eigentliche Funktion wird auch das bedarfsgesteuerte Installieren realisiert. Des Weiteren kann man durch das Austauschen einer Funktion eine Ersetzungsoperation erstellen, die es erlaubt, Programmteile dynamisch zu aktualisieren.

Im Gegensatz zu Funktionsmodulen ist der Austausch von Datenmodulen ohne zusätzliches anwendungsspezifisches Wissen selten sinnvoll. Funktionsmodule verwalten einen konstanten Inhalt, den Code. Bei Datenmodulen gilt das nur für nur-lesbare Daten. Die meisten durch Datenmodule repräsentierten Speicherstellen werden jedoch während der Programmabarbeitung verändert. Ein Datenmodul repräsentiert somit zwar die Position und die Struktur, jedoch immer nur eine Momentaufnahme der tatsächlichen Daten. Das Austauschen von Datenmodulen wird daher nur im Rahmen des Ersetzens oder Aktualisierens (siehe Abschnitt 4.6.3) eingesetzt, da hier Wissen über die Bedeutung und Verwendung der Daten vorliegt. Dennoch können beim Austauschen von Funktionen zusätzliche Datenmodule installiert oder installierte Module vollständig entfernt werden. Dies entspricht jedoch nicht dem dynamischen Austauschen, da keine Verweise von einem Datenmodul auf ein anderes umgeschrieben werden.

Das generelle Vorgehen beim Austauschen ist in Abbildung 4.11 dargestellt. Zunächst müssen alle Verweise auf das Modul gefunden werden, um sie anzupassen. Details zum Auffinden von Anhängigkeiten und Grenzen der Verfahren werden im nächsten Abschnitt geschildert. Wenn alle Verweise identifiziert werden konnten, wird das neue Modul installiert. Anschließend erfolgt die Anpassung der Verknüpfungen, sodass Verweise auf das alte Modul nun auf das neue Modul zeigen. Schließlich kann das alte Modul entfernt werden.

Um den Speicherplatz, der durch das Entfernen des alten Moduls frei wird, gleich nutzen zu können, kann das Entfernen auch vor dem Installieren des neuen Moduls durchgeführt werden. Das Austauschen als Gesamtoperation darf dann jedoch nicht unterbrochen werden, da sich das System währenddessen in einem inkonsistenten Zustand befindet.

Bevor der Austausch allerdings begonnen wird, muss überprüft werden, ob nach dem Austausch eine konsistente Konfiguration vorliegt. Hierzu muss zunächst darauf geachtet werden, dass beim Austausch eines Moduls die Schnittstellen, welche von den noch vorhandenen Modulen verwendet werden, erfüllt sind. Das heißt, dass Abhängigkeiten von installierten Modulen zu alten Modulen,

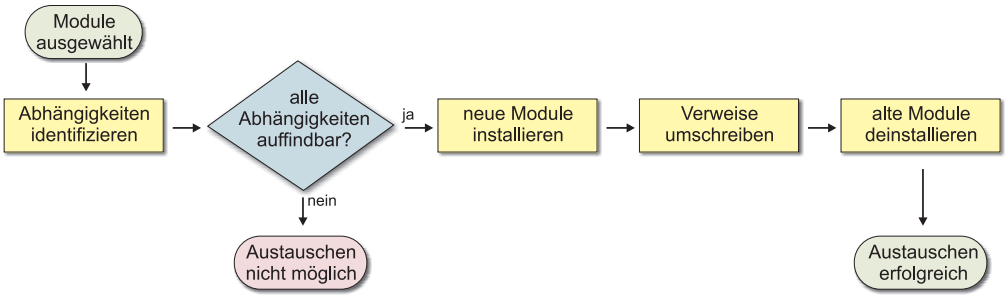


Abbildung 4.11: Vorgehen beim Austausch eines Moduls

die entfernt werden sollen, durch neue Module erfüllt werden müssen. Im einfachsten Fall wird eine Funktion durch einen Stellvertreter ausgetauscht. Dieser ist vollständig schnittstellenkompatibel, da er dieselben Symbole wie das ursprüngliche Modul anbietet und erwartet unter denselben Umständen aufgerufen zu werden. Bei Datenmodulen bedeutet die Schnittstellenkompatibilität, dass sie dieselbe Struktur besitzen. Änderungen sind daher nur begrenzt möglich. Eine kompatible Änderung stellt zum Beispiel das Hinzufügen eines zusätzlichen Elements am Ende einer Struktur dar, da hierbei die existierende Schnittstelle erhalten bleibt und lediglich erweitert wird. Das Entfernen von Elementen einer Struktur ist hingegen nicht zulässig, da die Elemente möglicherweise noch im vorhandenen Code benötigt werden.

Bei Schnittstellenkompatibilität handelt es sich somit um eine rein syntaktische Eigenschaft, die sicherstellt, dass die Software überhaupt ausgeführt werden kann. Sie sagt nichts über die Änderung der semantischen Eigenschaften des Moduls aus. Von der Infrastruktur werden hieran auch keine Forderungen gestellt. Es ist die Aufgabe der Kontrollschicht beziehungsweise des Administrators, die Kompatibilität der funktionalen und nicht-funktionalen Eigenschaften der Module zu berücksichtigen.

Sind die Schnittstellen kompatibel, so müssen die Abhängigkeiten der neuen Module analog zur Installation von Modulen überprüft und erfüllt werden. Sollte also das neue Modul Abhängigkeiten haben, die auf dem Knoten nicht erfüllt sind, so müssen zusätzliche Module zur Installation ausgewählt werden. Auf der anderen Seite kann das neue Modul auch weniger Abhängigkeiten haben. Dadurch kann es vorkommen, dass Module auf dem Gerät nicht mehr benötigt werden. Diese werden im Normalfall nach dem Austausch entfernt.

Das Austauschen verwendet somit die beiden bereits bekannten Teiloperationen “Installieren” und “Entfernen” und fügt das Anpassen der Verweise als neue Teiloperation hinzu.

4.4 Verweise auf Module

Um Module auszutauschen oder zu entfernen, müssen alle Verweise auf diese Module gefunden werden. Daraufhin können die Verweise entweder angepasst werden oder es kann sichergestellt werden, dass keine Verweise mehr existieren. Auch bei der Installation müssen die Referenzen zwischen den Modulen überprüft werden. Dieser Abschnitt beschäftigt sich daher mit den Möglichkeiten und Problemen beim Identifizieren und Modifizieren von Verweisen auf ein Modul.

4.4.1 Klassifikation von Verweisen

Es gibt verschiedene Arten, wie ein Modul in einem Programm referenziert werden kann. Im Folgenden unterteilen wir die Verweise nach der Art, wie sie entstehen.

Statisch gegebene Verweise Unter statisch gegebenen Verweisen versteht man solche Verweise, die schon vor der Laufzeit im Programmcode vorhanden sind und deren Ziel somit zum Zeitpunkt des Bindens bereits fest steht. Beispiele sind statische Funktionsaufrufe oder direkte Zugriffe auf globale Variablen. Normalerweise werden diese Verweise im Rahmen des Bindens aufgelöst, indem der Binder die Zieladresse an die entsprechende Stelle einfügt.

Dynamisch entstehende Verweise Bei dynamisch entstehenden Verweisen steht entweder das genaue Ziel eines Verweises erst zur Laufzeit fest oder der Verweis an sich wird erst zur Laufzeit erzeugt. Ein Beispiel ist ein Funktionsaufruf mittels variablem Funktionszeiger. Die Zieladresse des Sprungs steht dabei nicht zum Zeitpunkt des Bindens fest, sondern wird aus einer Speicherstelle entnommen. Analog können auch Verweise auf Daten indirekt erfolgen. Ein anderes Beispiel ist die Rücksprungadresse bei einem Funktionsaufruf, der als Verweis auf die aufrufende Funktion gewertet werden muss.

4.4.2 Identifikation von Verweisen

Das Erkennen und Auffinden der Verweise ist eine wichtige Voraussetzung, um dynamisch Veränderungen vornehmen zu können. Das Identifizieren der statisch gegebenen Referenzen ist hierbei noch relativ einfach, da sie durch Relokationen in der Relokationstabelle gekennzeichnet sind.

Anders sieht es bei Referenzen auf Code und Daten aus, die erst zur Laufzeit entstehen. Die vom Compiler generierten Informationen geben zwar Aufschluss über die Referenzen zum Startzeitpunkt des Programms, nicht jedoch, wie sie dann weiterverwendet werden. Wurde beispielsweise die Adresse einer Funktion in eine Variable gespeichert, kann sie weitergegeben werden und an einer anderen Stelle als Zieladresse eines Funktionsaufrufs dienen.

Im Folgenden sollen verschiedene Arten von Verweisen vorgestellt werden. Besondere Beachtung findet hierbei die Entstehung von dynamischen Verweisen. Dabei stehen Verweise auf Funktionen im Mittelpunkt, da der Austausch von Modulen in der Regel durch eine gewünschte Veränderung der Funktionalität ausgelöst wird. Tabelle 4.1 gibt eine kurze Aufstellung der möglichen Verweise auf eine Funktion.

	Art des Verweises	Beispiel
	Statischer Verweis	Statischer Funktionsaufruf
Dynamische Verweise	Indirekte Nutzung	Funktionsaufruf mittels Funktionszeiger
	Aktive Funktion	Instruktionszeiger des Prozessors
	Funktion auf Aufrufpfad	Rücksprungadresse auf dem Stack
	Sonstiges	Errechnen der Zieladresse

Tabelle 4.1: Verweise auf Code

In den nächsten Abschnitten sollen Möglichkeiten aufgezeigt werden, wie man solche Verweise identifizieren kann und Probleme angesprochen werden, die das Erfassen oder Verändern solcher Verweise verhindern. Es ist zu erwarten, dass es teilweise sehr aufwendig oder sogar unmöglich ist, alle dynamischen Referenzen auf ein Modul automatisch zu bestimmen und zu verändern. Ein Ziel ist es daher auch, festzustellen, ob die betrachteten Verweise zuverlässig gefunden werden können und wie man die Entstehung solcher Verweise bei der Programmierung gegebenenfalls vermeidet.

Besteht vor der Durchführung einer Veränderung nicht die Garantie, dass man alle Verweise findet, so kann die Operation nicht automatisch ausgeführt werden. Das System sollte dann Hinweise geben, welche Programmstellen beziehungsweise Module die Probleme verursacht haben. Daraufhin ist eine

explizite Änderung der Software möglich. Alternativ können zusätzliche Informationen von außen hinzugefügt werden, die das Auffinden der Verweise ermöglichen.

4.4.2.1 Statische Verweise

Wie bereits erwähnt, werden statische Verweise durch Einträge in der Relokationstabelle beschrieben. Anhand dieser Informationen kann festgestellt werden, ob ein Verweis auf ein Modul existiert. Falls ein Verweis existiert, so ist die genaue Position in den Relokationsdaten beschrieben. Eine Anpassung des Verweises kann somit leicht vorgenommen werden.

4.4.2.2 Dynamische Verweise durch indirekte Nutzung

Wird ein Modul, also eine Variable oder eine Funktion, indirekt genutzt, so bedeutet das, dass an der Stelle der Nutzung nicht direkt die Adresse des Elements durch den Linker eingefügt wird, sondern dass die Adresse bereits zu einem früheren Zeitpunkt im Speicher abgelegt wurde und nun von dort benutzt wird.

Ein einfaches, bereits genanntes Beispiel ist ein Funktionsaufruf mittels Funktionszeiger. Die Adresse der Funktion wurde vorher in einer Variablen als Funktionszeiger abgelegt. Das allgemeine Problem ist, dass man feststellen muss, wo die Adresse der Funktion überall benutzt wird, das heißt, wo überall Kopien angelegt werden. Nur dann kann man alle Verweise auf die Funktion in Erfahrung bringen, um sie anzupassen.

Lösungsansatz: Verweise aufspüren

Den Weg des Funktionszeigers kann man mithilfe einer Code- und Datenflussanalyse [Hec77] verfolgen. Man beginnt an der Stelle, an der die Adresse einer Funktion erstmals verwendet wird. Bei dem oben genannten Beispiel wird die Adresse der Funktion in einer Variablen abgelegt. Diese Position lässt sich anhand der Relokationsinformationen bestimmen. Von dort ausgehend wird der weitere Programmablauf simuliert, indem der vorliegende Binärcode teilweise interpretiert wird. Dabei wird die Position der Adresse überwacht und Zugriffe verfolgt. Mithilfe dieser Technik kann man nachvollziehen, wohin die Adresse kopiert, wie sie verwendet und wo sie abgelegt wird.

Diese Analyse hat jedoch einige Nachteile. Zum einen ist sie architekturabhängig, da Wissen über die Wirkung jedes Maschinenbefehls benötigt wird. Dies macht die Realisierung dieses Ansatzes sehr aufwendig. Hinzu kommt, dass auch die Analyse selbst aufwendig sein kann und zudem nicht immer sichergestellt ist, dass sie ein brauchbares Ergebnis liefert. Wenn sich der Programmablauf abhängig von externen Ereignissen in verschiedene Richtungen entwickeln kann, müssen alle möglichen Wege in der Simulation berücksichtigt werden. Liefern verschiedene Wege sich widersprechende Aussagen, so ist das Ergebnis der Analyse für unsere Zwecke unbrauchbar. Der Aufwand muss somit sorgfältig gegenüber dem Nutzen abgewogen werden.

Für viele Einsatzzwecke kann ein vereinfachtes Verfahren jedoch ausreichen. Dabei ist das Ziel, lediglich festzustellen, ob die Adresse einer Funktion als Zeiger Verwendung findet oder ausschließlich in Sprungbefehlen und Funktionsaufrufen verwendet wird. Um diese Unterscheidung zu treffen, reicht es aus, den Code an den Stellen zu prüfen, an denen laut Relokationsinformationen eine Referenz auf den Code eingefügt werden soll. Handelt es sich um einen Sprungbefehl, so wird die Adresse direkt verwendet. Handelt es sich nicht um einen Sprungbefehl oder wird die Adresse im Datenbereich eingefügt, dann wird sie vermutlich indirekt verwendet.

Bei manchen Architekturen kann sogar auf die Untersuchung des Codes verzichtet werden, da bereits die Art der Verknüpfung in den Relokationsinformationen klarstellt, ob die Adresse in einem Sprungbefehl verwendet wird. Ein Beispiel hierfür wurde in Abbildung 4.3 bereits angedeutet: Relokationen bei AVR-Prozessoren können den Verknüpfungstyp `R_AVR_CALL` haben. Dies ist ein sicheres Zeichen, dass die Adresse bei einem Sprungbefehl eingesetzt wird.

Stellt man nach Anwendung dieses vereinfachten Verfahrens fest, dass die Adresse ausschließlich in Sprungbefehlen verwendet wird, so können keine Verweise dynamisch entstehen und die Relokationsinformationen reichen aus, um alle Verweise zu identifizieren. Bei einem negativen Ergebnis hingegen kann die eigentliche Operation, das Löschen oder Austauchen eines Moduls, nicht ohne zusätzliches Wissen durchgeführt werden.

Für die von uns angestrebte Aufgabe, dem dynamischen Anpassen einer Laufzeitumgebung, reicht dieser Ansatz aus, da die Software bekannt ist. Will man das vereinfachte Verfahren auch bei unbekannter Software einsetzen, so muss man die Verwendung von Funktionszeigern für Funktionen, die ausgetauscht oder entfernt werden sollen, verbieten oder auf bekannte Situationen einschränken. Auch wenn diese Forderung zunächst als eine starke Einschränkung erscheint, so ist es nicht unüblich, auf den Einsatz von Zeigern zu verzichten. Sie stellen eine Fehlerquelle besonders für unerfahrene Programmierer dar. Bei der Softwareentwicklung für eingebettete Systeme wird daher der Einsatz von Zeigern häufig eingeschränkt oder verboten. Ein bekanntes Beispiel hierfür stellt Misra-C [MIS98] dar. Dabei handelt es sich um Richtlinien für den Gebrauch der Programmiersprache C in sicherheitskritischen Systemen im Bereich der Automobilindustrie, welche später für allgemeine eingebettete Systeme überarbeitet wurden [MIS04]. Die Misra-Regeln verbieten bestimmte Konstrukte in C, welche die Intension des Autors nicht eindeutig erkennen lassen und daher fehleranfällig sind. Unter anderem wird auch der Einsatz von Funktionszeigern eingeschränkt.

Zeiger auf Funktionen, die automatisch erzeugt werden, sind hingegen gut handhabbar, wenn man das Vorgehen des Generators kennt. In C++ beispielsweise werden implizit Funktionszeiger im Rahmen von virtuellen Methodenaufrufen verwendet. Die Methodentabelle wird dabei vom Compiler erzeugt. Ihr Aufbau ist bekannt, daher kann sie auf entsprechende Zeiger hin überprüft werden.

Lösungsansatz: Position erhalten

Beim Austauschen von Funktionen gibt es noch eine andere Möglichkeit, mit indirekten Verweisen umzugehen. Anstatt alle Verweise aufzuspüren und abzuändern, kann man dafür sorgen, dass die Funktion weiterhin an der ursprünglichen Position erreichbar bleibt. Existierende Verweise bleiben dadurch gültig und das Anpassen der Verweise kann entfallen.

Für diese Optimierung müssen die Einsprungpunkte des neuen Moduls an genau denselben Adressen abgelegt werden können wie die Einsprungpunkte des alten Moduls. Das bedeutet, dass die Schnittstellen des Moduls an denselben relativen Positionen innerhalb des Moduls sein müssen und, dass genügend Platz auf dem Zielgerät vorhanden sein muss, um das neue Modul an der alten Position abzulegen.

Im Idealfall lässt sich die Überprüfung reduzieren auf die Prüfung des vorhandenen Speicherplatzes. Module repräsentieren, wenn möglich, genau eine Funktion oder eine Variable. Die Schnittstelle solcher Module liegt dann immer am Anfang des Moduls und erfüllt somit die Voraussetzungen für den einfachen Austausch. Man muss somit nur noch sicherstellen, dass das neue Modul kleiner oder gleich groß ist wie das alte Modul oder dass im Programmspeicher des Zielgeräts hinter dem alten Modul genügend freier Speicher vorhanden sein, um die Größendifferenz zum neuen Modul aufzunehmen.

Kennt man alle möglichen Varianten einer Funktion, so kann man, ähnlich zum Vorgehen bei der Überlagerungstechnik (siehe Abschnitt 2.1.3), den maximal benötigten Platz anhand der größten Funk-

tion bestimmen. Um unbekannte zukünftige Versionen zu unterstützen, kann es sinnvoll sein, dennoch zusätzlichen Speicherplatz hinter der Funktion zu allozieren. Dadurch erhöht sich die Wahrscheinlichkeit, dass eine veränderte Funktion an den Platz der alten Funktion geschrieben werden kann [QL91]. Der zusätzliche Speicherplatz bleibt jedoch ungenutzt, bis er von einer neuen Funktion belegt wird. Daher ist dieser Ansatz nur geeignet, wenn man mit großer Sicherheit davon ausgehen kann, dass die Funktion in Zukunft durch eine größere ausgetauscht wird.

Lösungsansatz: Weiterleitung

Passt das veränderte Modul nicht an die ursprüngliche Position, so kann eine Weiterleitung eingerichtet werden. Eine Weiterleitung besteht im Wesentlichen aus einem Sprungbefehl, der den Kontrollfluss auf die neue Adresse der Funktion umleitet. Die veränderte Funktion kann dann an einer beliebigen anderen Adresse abgelegt werden. Hat das Modul mehrere Einsprungpunkte, so muss für jeden möglichen Einsprungpunkt eine Weiterleitung an der ursprünglichen Adresse abgelegt werden.

Wird ein Modul zu einem späteren Zeitpunkt erneut ausgetauscht und wiederum an einer anderen Position abgelegt, so kann die ursprüngliche Weiterleitung angepasst werden. Zusätzlich sind dieselben Punkte zu beachten wie bei einem normalen Austausch. Dies kann dazu führen, dass eine zweite Weiterleitung eingerichtet werden muss.

Der Nachteil bei der Verwendung von Weiterleitungen ist, dass ein Funktionsaufruf dadurch eine zusätzliche Indirektionsstufe durchlaufen muss. Dies kostet zusätzliche CPU-Zyklen. Außerdem führt eine ausgiebige Nutzung dieses Ansatzes zu einer starken Fragmentierung des vorhandenen Programmspeichers, da an vielen Orten Weiterleitungen eingerichtet werden. Diese benötigen zwar nicht viel Platz, müssen jedoch an einer festen Position im Speicher stehen. Dem gegenüber steht der Vorteil, dass keine Verweise angepasst werden müssen und somit auch das Suchen der Verweise entfallen kann.

4.4.2.3 Dynamische Verweise bei aktiven Funktionen

Bei der Suche nach Verweisen auf Funktionen ist auch der aktuelle Systemzustand zu berücksichtigen. So muss zum Beispiel das Verändern einer Funktion, die gerade ausgeführt wird, speziell betrachtet werden, da hier der Instruktionszeiger des Prozessors auf die Funktion verweist. Genauer gesagt, wird die Funktion von einer Änderungsoperation unterbrochen, wie in Abbildung 4.12 dargestellt ist. Dabei wird in den Verwaltungsstrukturen des Systems die Rücksprungadresse vermerkt, an der die Funktion später fortgesetzt werden soll. Ist nun diese Funktion von der Änderung betroffen, so muss man sicherstellen, dass die Rücksprungadresse gültig bleibt oder dass sie entsprechend angepasst wird. Ansonsten sind die Folgen nicht vorhersehbar und eine konsistente Programmausführung ist nicht gewährleistet.

Eine Garantie oder Anpassung ist unmöglich ohne genaues Wissen über Aufbau und Ablauf der Funktion vor und nach der Änderung. Der gespeicherte Zustand zum Zeitpunkt der Unterbrechung muss auch zu der veränderten Funktion passen. Das schließt die Stack- und Registerbelegung mit ein. Da der Code im Normalfall jedoch nicht manuell erzeugt, sondern durch einen Compiler erstellt wird, ist das notwendige Wissen nicht vorhanden. Daraus folgt die Forderung, dass eine Funktion nicht aktiv sein darf, wenn sie ausgetauscht oder entfernt werden soll.

Als aktive Funktion gilt nicht nur die Funktion, die durch die Änderungsoperation unterbrochen wurde, sondern alle Funktionen, die im System unterbrochen wurden und zu einem späteren Zeitpunkt fortgesetzt werden könnten. Sie entstehen insbesondere durch Aktivitätsträger, die zum Beispiel durch Verdrängung unterbrochen werden.

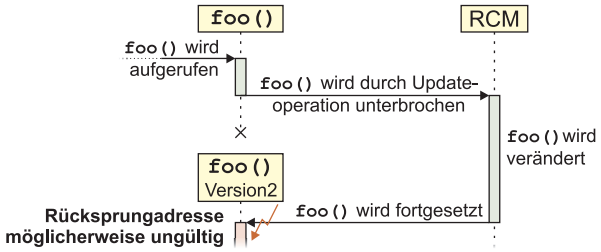


Abbildung 4.12: Verändern einer aktiven Funktion

Um eine dynamische Änderungsoperation durchzuführen, wird die aktuell ausgeführte Funktion unterbrochen. Ist nun genau die unterbrochene Funktion Gegenstand der Änderung, so kann nicht gewährleistet werden, dass die Rücksprungadresse noch sinnvoll ist. Das Fortfahren an der gespeicherten Rücksprungadresse kann zum Absturz oder anderen unvorhersehbaren Folgen führen.

Im Detail ist es systemabhängig, ob und wie viele aktive Funktionen es geben kann. In reinen ereignisgesteuerten Systemen, wie beispielsweise TinyOS [HSW+00], die nur einen Aktivitätsträger anbieten, können keine quasi-parallelen Kontrollflüsse entstehen. Nur eine Ereignisbearbeitung kann den normalen Kontrollfluss unterbrechen. Reiht man Änderungsoperationen hier als normale Aufgaben ein (siehe Abschnitt 4.5.7.2), so werden sie ausgeführt, wenn keine andere Aufgabe aktiv ist. Somit ist nur die Nebenläufigkeit der Ereignisbearbeitung zu berücksichtigen. Bietet das System hingegen ein Threadkonzept an, das die Unterbrechung der Aktivitätsträger erlaubt, so gibt es mehrere aktive Funktionen, die man identifizieren muss, um festzustellen, ob sie von der Änderungsoperation betroffen sind.

Zur Identifikation der aktiven Funktionen muss der Verwalter überprüfen, an welcher Stelle die Ausführung auf dem verwalteten Gerät durch die Änderungsoperation unterbrochen wurde. Anhand der Position kann die entsprechende Funktion bestimmt werden. Unterstützt das System mehrere Aktivitätsträger, so muss der Verwalter die aktuellen Ausführungspositionen von allen Threads überprüfen und erhält so die Menge der aktiven Funktionen. Da die Verweise auf eine aktive Funktion nicht verändert werden können, liefert die Abhängigkeitsprüfung einen entsprechenden Fehlercode zurück, falls die Abhängigkeiten von einer der aktiven Funktionen bestimmt werden sollen.

Lösungsansatz: auf Beendigung warten

Um eine solche Situation aufzulösen, bietet die Basisschicht einen einfachen Ansatz an. Der Grundgedanke ist, dass eine aktive Funktion nur eine begrenzte Zeit lange aktiv ist und man nur eine kurze Zeit warten muss, bis die Funktion nicht mehr aktiv ist. Das verwaltete System wird daher fortgesetzt, um nach einem kurzen Moment erneut zu prüfen, ob ein günstigerer Systemzustand vorliegt. Die Wahrscheinlichkeit, dass beim zweiten Versuch keine aktiven Funktionen betroffen sind, hängt vom Aufbau der Software ab.

In dem von uns hauptsächlich betrachteten Szenario werden Dienste der Laufzeitumgebung ausgetauscht oder verändert. Diese haben im Allgemeinen eine kurze Laufzeit und eine geringe Aufruffrequenz, sodass die Wahrscheinlichkeit hoch ist, dass die Funktion zu einem späteren Zeitpunkt nicht mehr aktiv ist und die Operation fortgesetzt werden kann.

Ist bei einem erneuten Versuch die Funktion immer noch oder schon wieder aktiv, so kann man das Verfahren mehrmals wiederholen. Nach einer bestimmten Zeit oder nach einer festgelegten Anzahl von Versuchen macht eine Wiederholung keinen Sinn mehr und die Abhängigkeitsprüfung schlägt endgültig fehl. Das System kann in diesem Fall die aktive Funktion und den Aktivitätsträger angeben, in dem sie

aktiv ist. Der Entwickler kann so wertvolle Hinweise erhalten, wie zukünftige Versionen der Software umgestaltet werden müssten, um die Voraussetzungen für eine erfolgreiche Abhängigkeitsanalyse zu verbessern.

Lösungsansatz: bei Beendigung benachrichtigen

Um die periodische Überprüfung zu vermeiden, kann die Software auch in geeigneter Art und Weise verändert werden, sodass beim Verlassen der aktiven Funktion automatisch der Verwalter kontaktiert wird. Das konzeptionelle Vorgehen entspricht dem Anwenden eines, aus der aspektorientierten Programmierung bekannten, *after advice*. Bei der Realisierung kann man nach zwei unterschiedlichen Verfahren vorgehen:

1. Eine Möglichkeit ist, die Rücksprungadresse zu ermitteln und diese so abzuändern, dass der Rücksprung zu einem Stück Code erfolgt, der den Verwalter kontaktiert. Dadurch wird der Verwalter beim Verlassen der Funktion sofort benachrichtigt. Die ursprüngliche Rücksprungadresse muss dabei gesichert werden, da sie benötigt wird, um das Programm nach der Operation fortzusetzen.
   Um diese Möglichkeit zu nutzen, muss es jedoch möglich sein, die Position der Rücksprungadresse zu ermitteln. Erzeugt der Compiler einen *Framepointer*[8], so kann man durch das Ablaufen des Stacks an die Rücksprungadresse gelangen. Wird kein Framepointer generiert, so ist es im Allgemeinen nicht möglich, die Position der Rücksprungadresse zuverlässig zu bestimmen.

2. Die zweite Möglichkeit basiert darauf, dass man in den vorhandenen Code an einer geeigneten Stelle einen Sprung zu der Benachrichtigungsfunktion einsetzt. So können beispielsweise die `return`-Anweisungen in der aktiven Funktion durch solche Sprünge ersetzt werden.
   Voraussetzung für dieses Vorgehen ist allerdings, dass man die Ablaufstruktur des Codes analysiert und ermitteln kann, an welchen Stellen die Funktion verlassen wird. Außerdem darf der Sprungbefehl nicht mehr Platz benötigen als der `return`-Befehl, da ansonsten die Struktur der Funktion zerstört wird[9].

Bei beiden Verfahren wird die Ausführung auf dem verwalteten Knoten nach der Veränderung fortgeführt. Sobald der Kontrollfluss die aktive Funktion verlässt, wird ein speziell vorbereitetes Codestück angesprungen, welches den Verwalter benachrichtigt. Daraufhin muss geprüft werden, ob die betrachtete Funktion immer noch aktiv ist. Um ein endloses Warten auf eine Benachrichtigung zu verhindern, muss der Verwalter aber auch bei diesem Verfahren ein Zeitfenster vorsehen, nach dessen Ablauf er die Operation abbricht, falls die Funktion nicht beendet wurde.

Bei der Wahl zwischen den beiden Verfahren ist das erste Verfahren zu bevorzugen. Der Vorteil ist, dass hier kein Code verändert werden muss, da die Rücksprungadresse im Datenbereich abgespeichert

[8] Der *Framepointer* zeigt auf den Anfang des Stackbereichs, der von der aktuellen Funktion benutzt wird. Dort ist üblicherweise auch die Rücksprungadresse abgelegt.

[9] Auf den von uns betrachteten Mikrocontrollern benötigt eine `return`-Anweisung (H8: `RTS`, AVR: `RET`) zwei Byte. Direkte Sprungbefehle mit absoluter Zieladresse (`JMP`) benötigen hier vier Byte, da nicht nur der Befehl, sondern auch die 16-bit Zieladresse codiert werden muss. Indirekte Sprungbefehle benötigen auf den betrachteten Plattformen nur zwei Byte, sind aber nicht uneingeschränkt geeignet. Für register-indirekte Sprünge (H8: `JMP`, AVR: `IJMP`/`EIJMP`) müssen zusätzliche Befehle eingefügt werden, um die Zieladresse in das entsprechende Register abzulegen. Für einen speicher-indirekten Sprung könnte der Verwalter die Zieladresse in einen unbenutzten Speicherbereich abgelegen. Der AVR bietet jedoch keinen indirekten Sprungbefehl mit Zieladresse in einer variablen Speicherstelle. Am besten eignen sich daher relative Sprünge (H8: `BRA`, AVR: `RJMP`). Sie haben allerdings den Nachteil, dass nicht der gesamte Adressraum erreicht werden kann.

wird. Dies ist besonders bei Mikrocontrollern von Vorteil, deren Code in einem Flashspeicher abgelegt ist, da dort Änderungen relativ aufwendig und zeitintensiv sind. Wie schon angedeutet, ist der Einsatz jedoch nur möglich, wenn die Position der Rücksprungadresse bestimmt werden kann. Ist das nicht der Fall, so wird die zweite Methode angewandt. Sie ist generell aufwendiger, da der Code analysiert werden muss, um die `return`-Anweisungen zu identifizieren.

4.4.2.4 Dynamische Verweise durch Funktionen auf dem Aufrufpfad

Nicht nur Funktionen, die in dem Moment aktiv sind, in dem sie verändert werden sollen, müssen besonders betrachtet werden, sondern auch solche, die sich auf dem Aufrufpfad einer aktiven Funktion befinden. Die Situation ist in Abbildung 4.13 dargestellt. Ein Beispiel ist die Funktion, welche die aktive Funktion aufgerufen hat. Sie wird zwar nicht direkt ausgeführt, ist aber dennoch in Ausführung, da der Kontrollfluss irgendwann zurückkehrt, um sie fortzusetzen.

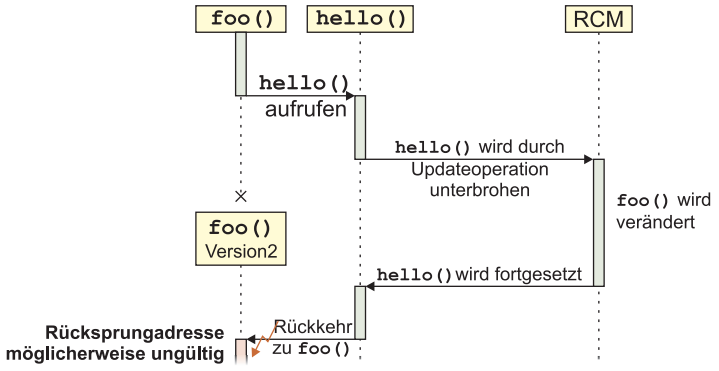


Abbildung 4.13: Verändern einer Funktion auf dem Aufrufpfad

Die Funktion `foo` ruft die Funktion `hello` auf. Während der Ausführung von `hello` wird die Funktion `foo` verändert. Da die Funktion `hello` nicht verändert wurde, kann die Ausführung nach der Änderungsoperation fortgesetzt werden. Sobald `hello` jedoch beendet ist und zu `foo` zurückkehrt, wird die Rücksprungadresse in die alte Funktion `foo` verwendet. Ob diese Adresse auch für die veränderte Funktion `foo` gültig ist, lässt sich nicht garantieren.

Die potenziellen Probleme sind grundsätzlich dieselben wie bei aktiven Funktionen, die nach einer Veränderung fortgesetzt werden sollen. Auch hier zeigt ein Verweis auf den Code, der verändert werden soll. In diesem Fall ist es die Rücksprungadresse, die verwendet wird, um von der aufgerufenen Funktion zurückzukehren. Bei Veränderung des Codes muss daher sichergestellt werden, dass die Rücksprungadresse angepasst wird und einer gültigen und erwarteten Position entspricht. Analog zur Veränderung einer aktiven Funktion muss außerdem der Zustand, das heißt der Register- und Stackinhalt, mit der neuen Funktion kompatibel sein. Für automatisch erzeugten Code kann das in der Regel nicht garantiert werden.

Allerdings ist die Situation hier etwas enger eingegrenzt, da die Funktion nicht an einer beliebigen Position unterbrochen wurde, wie das bei aktiven Funktionen der Fall ist. Es kommen nur Funktionsaufrufe innerhalb der Funktion in Frage, somit sind die Unterbrechungen auf wenige Positionen beschränkt. Für manuell erzeugten oder manuell gewarteten Code könnte man sich daher eine Unterstützung vorstellen. Ändert sich die Funktion nur in Kleinigkeiten, so kann man eine automatische Anpassung der Rücksprungadresse vornehmen. Voraussetzung ist, dass sich die Struktur der Funktion nicht verändert hat. Dann kann man den Ansatz verfolgen, dass die Rücksprungadresse des n-ten Funktionsaufrufs im alten Code durch die Position nach dem n-ten Funktionsaufruf innerhalb des veränderten Codes umgeschrieben wird.

Der Abgleich des Zustands kann nicht vollkommen automatisch vorgenommen werden. Hier ist immer externes beziehungsweise manuell erworbenes Wissen notwendig. Positiv wirkt sich jedoch aus, dass bei einem Funktionsaufruf einige Register von der aufgerufenen Funktion benutzt werden und somit keinen für die aufrufende Funktion relevanten Zustand enthalten. Der Zustand in den Registern ist somit kleiner und möglicherweise besser zu handhaben. Beim Abgleich des Stackinhalts ist kaum automatische Unterstützung möglich. Es kann lediglich der Sonderfall automatisch identifiziert werden, dass außer lokalen Variablen keine anderen Zwischenergebnisse auf dem Stack abgelegt sind.

Als generelle Lösung bietet sich daher nur der schon im Rahmen der aktiven Funktionen vorgeschlagene Weg an: das Warten auf Beendigung der Funktion. Bei Funktionen auf dem Aufrufpfad lässt sich allerdings nicht sagen, wie lange man warten muss, bis die Funktion beendet ist. Je tiefer die Funktion sich auf dem Aufrufpfad befindet, desto höher ist die Wahrscheinlichkeit einer langen Wartezeit.

4.4.2.5 Sonstige dynamisch entstehende Verweise

Neben den dargestellten Möglichkeiten, wie Verweise auf Code entstehen können, kann man sich noch verschiedene andere Arten vorstellen. So könnte die Adresse einer Funktion zum Beispiel mit Hilfe von Zeigerarithmetik aus einer Basisadresse errechnet werden. Oder eine Anwendung kann selbst den Aufrufstack analysieren, um Funktionsadressen zu extrahieren.

Solche Verweise können nicht zuverlässig erkannt und somit auch nicht automatisch angepasst werden. Man muss daher die Anforderung an den Code stellen, dass Verweise, welche dynamisch entstehen und nicht durch die oben genannten Arten gefunden werden können, durch zusätzliches externes Wissen bekannt sind. Alternativ kann man auch auf ein Regelwerk setzen, das die Entstehung solcher Verweise verhindert, wie beispielsweise die oben schon erwähnten Misra-C Regeln. Im Allgemeinen stellt diese Forderung keine große Einschränkung dar. Um nachvollziehbare und leicht verständliche Programme zu schreiben, sollte man ohnehin auf komplexe Arten, dynamisch Verweise zu erstellen, verzichten.

4.4.3 Abhängigkeitsgraph

Die Verwaltung der Verweise wird von der Basisschicht mithilfe eines Abhängigkeitsgraphen realisiert. Der Abhängigkeitsgraph wird im Rahmen einer globalen Abhängigkeitsanalyse aus den Referenzen zwischen den Modulen erstellt. Dabei gehen in erster Linie alle statischen Verweise ein. Ist ein Modul auf dem Gerät installiert, so werden auch dynamische Verweise berücksichtigt, um ein vollständiges Bild der Struktur zu erhalten. Für jedes Modul wird dabei festgestellt, welche anderen Module benötigt werden. Das Ergebnis kann man als gerichteten Graphen darstellen. Abbildung 4.14 zeigt ein einfaches Beispielprogramm mit seinem Abhängigkeitsgraphen. Die Knoten stellen die Module dar, eine Kante beschreibt eine Abhängigkeit. Zusätzlich müssen noch externe Abhängigkeiten modelliert werden. Diese werden als zusätzliche Kanten in den Graphen eingefügt und haben keinen Knoten als Quelle. Mit solchen Kanten werden beispielsweise der Startpunkt der Anwendung und Einsprungpunkte für Unterbrechungsbehandlungen gekennzeichnet. Ebenso können auch aktive Funktionen durch eine externe Abhängigkeit markiert werden.

Anhand des Abhängigkeitsgraphen lassen sich die Vorgänge und Bedingungen der Basisoperationen anschaulich darstellen:

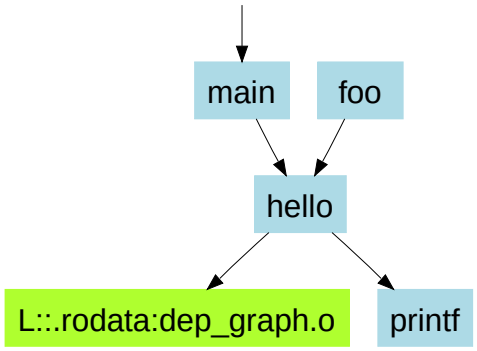
- Bei der Installation wird gefordert, dass die installierten Module in sich abgeschlossen sind. Am Abhängigkeitsgraphen bedeutet dies, dass keine Kante von den installierten Modulen zu nicht installierten Modulen geht. Werden neue Module hinzugefügt, so muss diese Bedingung weiterhin gelten.

```
void hello() {
  printf ("Hello World!\n");
}

void foo() {
  hello();
}

void main() {
  hello();
}
```

(a) Quellcode



(b) Abhängigkeitsgraph

Abbildung 4.14: Abhängigkeitsgraph eines einfachen Beispiels
Links ist der Quellcode eines einfachen Beispiels gegeben. Rechts daneben ist der automatisch erzeugte Abhängigkeitsgraph. Die Funktion `foo` wird in diesem einfachen Beispiel nicht verwendet. Im Abhängigkeitsgraphen erkennt man das daran, dass keine Kante in den Knoten der Funktion `foo` führt. Die Funktion `hello` benötigt als einziges Zugriff auf die Datensektion (`L::.rodata:dep_graph.o`). Nicht zu erkennen ist, dass Funktion `printf` aus einer anderen Datei hinzugefügt wurde.

- Möchte man ein Modul entfernen, so darf es nicht mehr referenziert werden. Demnach darf keine Kante in den entsprechenden Knoten führen. Hierbei ist zu beachten, dass diese anschaulich ausgedrückte Bedingung notwendig, aber nur dann hinreichend ist, wenn neben den statischen Verweisen auch alle dynamischen Verweise erfasst sind.
- Der Vorgang des Austauschens kann als das “Umbiegen” aller Kanten vom alten Knoten zum neuen Knoten aufgefasst werden.
  Nach dem Verändern der Kanten sollen Module identifiziert werden, die nicht benötigt werden. Knoten, in die keine Kante führt, werden nicht referenziert und werden daher auch nicht benötigt. Entfernt man sie zusammen mit ihren ausgehenden Kanten, so können weitere Knoten entstehen, in die nun keine Kanten mehr führen. Durch dieses iterative Verfahren können alle Module gefunden werden, die nicht von einer Initialkante aus erreichbar sind und somit nicht mehr benötigt werden.

Da sich mithilfe des Abhängigkeitsgraphen die Struktur der Software anschaulich darstellen lässt, dient er auch als Hilfsmittel für einen Administrator, wenn manuelle Entscheidungen getroffen werden sollen.

4.4.4 Ergebnisse

In diesem Abschnitt wurden die Verweise auf Module genauer betrachtet. Nach einer Klassifikation in statische und dynamische Verweise wurde festgestellt, dass statische Verweise leicht aufzufinden und abzuändern sind. Dynamisch entstehende Verweise hingegen sind ohne zusätzliche Informationen nur teilweise zu finden und nicht immer anpassbar.

Um sicherzustellen, dass alle Verweise identifiziert werden können, wurde für dynamische Verweise aus indirekter Nutzung eine Methode auf der Basis einer Datenflussanalyse vorgeschlagen. Für Funktionen wurde alternativ das positionserhaltende Austauschen vorgestellt, da hierdurch auf das Auffinden der Verweise verzichtet werden kann.

Anschließend wurde die Problematik der aktiven Funktionen betrachtet und herausgestellt, dass der Austausch nicht ohne Weiteres möglich und sehr aufwendig ist. Daher wurden Möglichkeiten aufgezeigt, auf die Beendigung der aktiven Funktion zu warten, um den Austausch anschließend vorzunehmen. Zuletzt wurde der Abhängigkeitsgraph vorgestellt, als ein Werkzeug zur Verwaltung der Verweise und zur anschaulichen Darstellung der Beziehungen zwischen den Modulen.

4.5 Aufbau und Architektur

In diesem Abschnitt soll die Architektur der Basisschicht beschrieben werden. Abbildung 4.15 zeigt eine grobe Übersicht der Bestandteile. Die Architektur soll dabei anhand der Dienste und Aufgaben beschrieben werden, die von der Basisschicht angeboten beziehungsweise erfüllt werden. Dabei werden Hinweise auf die Implementierung gegeben, die im Rahmen dieser Arbeit erfolgte. Als Implementierungssprache wurde Java eingesetzt.

Bevor jedoch die Dienste und die Architektur im Einzelnen betrachtet werden, soll zunächst der Aufbau eines Moduls beschrieben werden.

4.5.1 Module

Jedes Modul wird in der prototypischen Implementierung durch ein Modulobjekt dargestellt. Modulobjekte sind daher die Basis, auf der die Mechanismen der Basisschicht operieren. Mithilfe der Modulobjekte werden die binären Daten der Software zusammen mit den zugehörigen Metadaten verwaltet.

4.5.1.1 Typen

In Abschnitt 4.2.7 wurde eine Klassifikation der Module entsprechend ihrem Inhalt und ihrer Granularität vorgenommen. Diese Aufteilung in verschiedene Typen ist auch bei der Realisierung relevant, da sie die Anwendbarkeit und das Vorgehen einer Operation beeinflussen. Die verschiedenen Ausprägungen werden daher in der Implementierung durch vier verschiedene Java-Objekttypen realisiert, die in Tabelle 4.2 kurz vorgestellt werden.

	Datenmodul	Funktionsmodul
Einzelsymbolmodul	`DataObject` Das Modul beschreibt genau eine Variable.	`FunctionObject` Das Modul beschreibt genau eine Funktion.
Multisymbolmodul	`MultiDataObjects` Das Speicherobjekt des Moduls enthält mehrere Variablen.	`MultiFunctions` Das Speicherobjekt des Moduls enthält mehrere Funktionen.

Tabelle 4.2: Objekttypen von Modulen

Die verschiedenen Modultypen werden durch entsprechende Objekttypen in Java realisiert. Einzelsymbolmodule sind dabei als Spezialisierung von Multisymbolmodulen umgesetzt. Allen Modulen ist gemeinsam, dass sie einen zusammenhängenden Speicherbereich repräsentieren. Dabei können alle Elemente innerhalb eines Moduls immer nur zusammen installiert bzw. deinstalliert oder ausgelagert werden. Multisymbolmodule werden erzeugt, falls nicht alle Abhängigkeiten zwischen den Elementen erfasst werden konnten und eine Trennung daher nicht möglich ist.

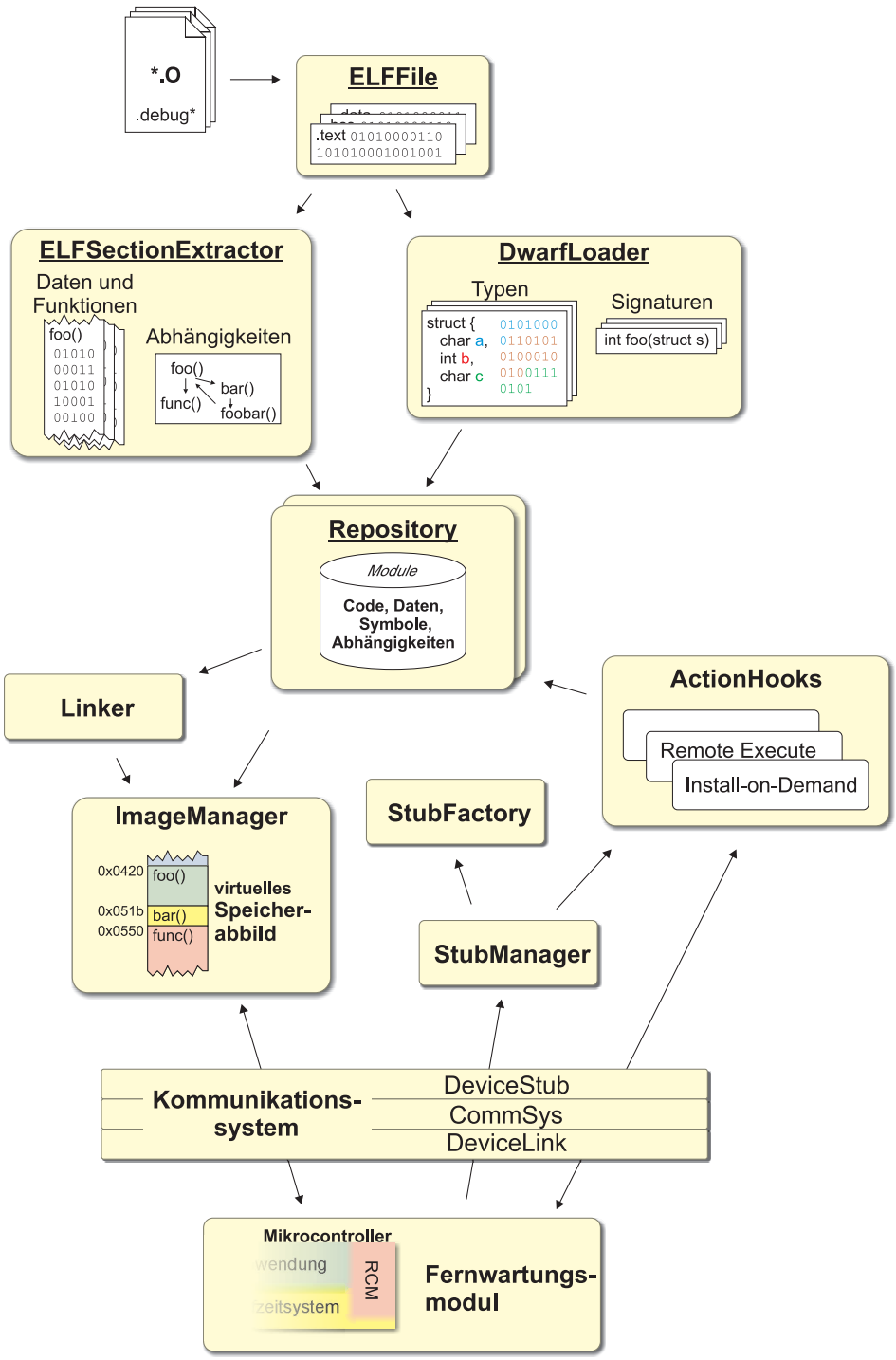


Abbildung 4.15: Architekturübersicht der Basisschicht

Die Abbildung zeigt die wichtigsten Komponenten der Basisschicht. Der `ELFSectionExtractor` modularisiert die Objektdateien und erstellt Module. Der `DwarfLoader` extrahiert dabei die Typ- und Schnittstelleninformationen aus den Debuginformationen. Der `ImageManager` verwaltet die aktuelle Konfiguration des Knotens. Ein geschichtetes Kommunikationssystem stellt die Verbindung zu dem Knoten her. Auf dem Knoten ist ein Fernwartungsmodul (*RCM*) installiert, das Befehle entgegennimmt und Unterstützungsanfragen absendet. Die Reaktion auf eine eintreffende Anfrage wird mithilfe des `StubManager`-Objekts zugeordnet und ist in einem `ActionHook` implementiert.

4.5.1.2 Aufbau

Ein Modulobjekt beinhaltet neben den binären Daten noch Metadaten, welche zum einen den Inhalt beschreiben und zum anderen benötigt werden, um die Binärdaten an variablen Adressen im Speicher ablegen zu können. Der Aufbau der einzelnen Typen ähnelt sich stark, da im Wesentlichen die gleichen Daten abgespeichert werden. Daten- und Funktionsmodule unterscheiden sich im Grunde nur durch den Typ. Zwischen Einzel- und Multisymbolmodule sind die Unterschiede deutlicher, da sich die Metadaten im Umfang unterscheiden. Folgende Daten gehören zu einem Modulobjekt:

Name Jedes Modul ist durch einen Namen identifizierbar. Bei Einzelsymbolmodulen entspricht der Name dem Symbolnamen des Einsprungpunkts, sonst wird der Name der Sektion zusammen mit dem Dateinamen verwendet.

Binärdaten Jedes Modul enthält die binären Daten, die durch das Modul beschrieben und verwaltet werden. Binäre Daten werden dabei durch ein spezielles Objekt (`Memory`) verwaltet und repräsentiert, welches einen zusammenhängenden untypisierten Speicherbereich darstellt.
Ein Modul enthält auf diese Weise die ungebundenen Binärdaten und, nachdem das Modul für eine bestimmte Anwendung gebunden wurde, auch eine Kopie der Binärdaten in gebundener Form.
Bei Funktionsmodulen dient das Speichern der gebundenen Binärdaten lediglich der Optimierung, da alle notwendigen Informationen, um aus den ungebundenen Daten die gebundenen Daten zu erstellen, am Modul ablesbar sind. Bei Datenmodulen hingegen werden die gebundenen Daten durch einige Operationen mit den aktuellen Werten vom verwalteten Knoten aktualisiert. So wird beim Entfernen eines Datenmoduls dort der Zustand abgelegt. Dadurch kann eine Variable später wieder mit dem gespeicherten Wert installiert werden. Die ungebundenen Daten entsprechen dem Anfangswert der Variablen und werden beispielsweise bei einem erneuten Aufsetzen der Anwendung verwendet.

Symbole Jedes Modul enthält eine Liste von Symbolen, die bestimmte Stellen innerhalb der Binärdaten beschreiben. Die Symbole beschreiben dabei die Einsprungpunkte in die binären Daten. Einzelsymbolmodule enthalten nur ein öffentliches Symbol. Ein Multisymbolmodul enthält mehrere öffentliche Symbole.

Abhängigkeiten Jedes Modul stellt auch eine Liste mit Symbolen bereit, die erfüllt sein müssen, um das Modul einzusetzen. Dazu wird auch je eine Relokation gespeichert, welche beschreibt, wie und an welcher Stelle die Adresse des Symbols verwendet werden soll.

Architektur Für jedes Modul ist darüber hinaus vermerkt, für welche Architektur die enthaltenen binären Daten geeignet sind.

Zustand Ein Modul besitzt einen Zustand, der angibt, wie das Modul gerade verwendet wird. Tabelle 4.3 zeigt die möglichen Zustände. Auf die Bedeutung der Zustände und die möglichen Übergänge wird in Abschnitt 4.5.4.1 eingegangen.

4.5.2 Verarbeiten von ELF-Dateien und Erzeugen von Modulen

Die prototypische Realisierung ist in der Lage, Objektdateien im ELF-Format einzulesen und zu verarbeiten. Nach der Verarbeitung werden Modulobjekte erzeugt, die keine ELF-spezifischen Informationen mehr enthalten. Es ist somit prinzipiell auch möglich, ein entsprechendes Verfahren für andere Binärformate zu erstellen.

Zustand	Beschreibung	
UNDEFINED	Das Modul ist dem Verwalter unbekannt. Es wurde noch nicht in einen Behälter geladen.	
LOADED	Das Modul wurde in einen Behälter geladen.	
SEL4INST	Das Modul ist zur Installation ausgewählt. Hiervon gibt es zwei Unterzustände, je nachdem, ob alle Abhängigkeiten des Moduls erfüllt sind. Eine Abhängigkeit ist erfüllt, wenn ein Verweis auf ein externes Symbol innerhalb des Behälters aufgelöst werden kann und das entsprechende Zielmodul ebenfalls ausgewählt oder installiert ist.	
	ERR	Das Modul kann noch nicht installiert werden, da noch nicht alle Abhängigkeiten erfüllt sind.
	OK	Alle Abhängigkeiten sind erfüllt, das Modul kann installiert werden.
SEL4REM	Das Modul soll vom Gerät entfernt werden. Auch hiervon gibt es zwei Unterzustände, die angeben, ob das Modul noch benötigt wird oder nicht.	
	ERR	Das Modul wird auf dem Gerät noch benötigt und kann daher nicht entfernt werden.
	OK	Es existieren keine Referenzen mehr auf das Modul, es kann daher gefahrlos entfernt werden.
INSTALLED	Das Modul ist auf dem Gerät installiert.	

Tabelle 4.3: Modulzustände
Die Tabelle zeigt die Zustände, die ein Modul während der Konfiguration erhalten kann. Der Übergang zwischen den Zuständen ist in Abbildung 4.17 dargestellt

4.5.2.1 ELFFile

Das Laden einer Objektdatei wird beim Erzeugen eines Objekts vom Typ `ELFFile` durchgeführt. Beim Einlesen wird die Objektdatei analysiert und in ihre Bestandteile aufgetrennt. Jede Sektion wird dabei entsprechend ihres Typs in ein Objekt abgelegt. Die einzelnen Elemente von Symbol- und Relokationstabellen werden ebenfalls in einzelne Java-Objekte umgewandelt. Verknüpfungen zwischen den Sektionen werden dabei aufgelöst oder als Verknüpfung zwischen den Objekten dargestellt. So wird der Name von Symbolen und Sektionen aus der Stringtabelle extrahiert und die Zielsymbole der Relokationen als Referenz auf das Symbolobjekt dargestellt.

Diese Struktur stellt die Ausgangbasis für die Extraktion von Modulen dar. Dabei stehen Sektionen mit binären Daten im Mittelpunkt, die in ELF-Dateien mit dem Typ `SHT_PROGBITS` gekennzeichnet sind. Relokationen und Symbole beziehen sich auf solche Sektionen und werden daher im Rahmen des Einlesens auch den entsprechenden Sektionen zugeordnet.

4.5.2.2 ELFSectionExtractor

Nach dem Einlesen erfolgt die Aufteilung in einzelne Module. Dazu ist der `ELFSectionExtractor` zuständig, der eine Implementierung des in Abschnitt 4.2.5 vorgestellten Verfahrens darstellt. Als Grundlage für ein Modul können prinzipiell alle Sektionen dienen, die binäre Daten enthalten, also vom ELF-Typ `SHT_PROGBITS` sind. Allerdings sollen Module nur von solchen Sektionen erzeugt werden, die zum Programmabbild gehören und somit entweder Programmcode oder -daten enthalten und nicht andere binäre Zusatzdaten wie beispielsweise Debuginformationen. Diese Filterung lässt sich zum Beispiel am Namen der Sektionen festmachen. Im ELF-Format kann man dafür auch auf

die Eigenschaften von Sektionen zurückgreifen, die unter anderem angeben, ob sich der Inhalt einer binären Sektion zur Laufzeit im Speicher des Programms befindet (Flag: `SHF_ALLOC`).

Für jede der so bestimmten Sektionen werden anschließend die Symbole analysiert, die Positionen innerhalb der Sektion beschreiben. Ist nur ein Symbol vorhanden, welches darüber hinaus die gesamte Sektion abdeckt, so wird ein Einzelsymbolmodul erzeugt. Sind in einer Codesektion neben diesem Symbol noch lokale Symbole vorhanden, die weder Typ noch Größenangabe enthalten, so wird ebenfalls ein Einzelsymbolmodul erzeugt. Hierbei wird angenommen, dass es sich bei den zusätzlichen Symbolen um interne Sprungmarken einer Funktion handelt, die nicht von außen angesprungen werden. In allen anderen Fällen muss ein Multisymbolmodul erzeugt werden (Abbildung 4.16).

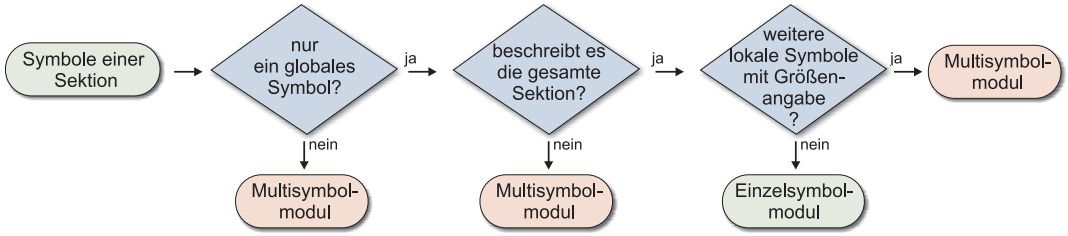


Abbildung 4.16: Einzel- oder Multisymbolmodul?

Zur Bestimmung, ob es sich um ein Daten- oder Funktionsmodul handelt, können mehrere Merkmale ausgewertet und miteinander abgestimmt werden. Der beste Weg, den Typ festzustellen, ist die Überprüfung der Typinformation in den Symbolen. Zeigen alle Symbole entweder auf Daten oder auf Funktionen, so kann hieraus der Typ des Moduls abgeleitet werden. Es gibt jedoch Compiler, die Symbole ohne diese Kennzeichnung erzeugen[10]. Außerdem kann der Inhalt einer Sektion auch durch ein allgemeineres Sektionssymbol angesprochen werden (siehe Abschnitt 4.2.5.1). In diesen Fällen kann als Merkmal der Name der Sektion herangezogen werden. Beginnt er mit `.text`, so handelt es sich üblicherweise um Code. Beginnt er hingegen mit `.data`, `.rodata` oder `.bss`, so sind Daten abgespeichert. Da der Name jedoch kein eindeutiges Merkmal darstellt, kann man die Annahme durch die Eigenschaften der Sektionen, welche in der ELF-Sektionstabelle gespeichert sind, untermauern. Eine Sektion, die Code enthält, muss als ausführbar gekennzeichnet sein und ist normalerweise nur-lesbar, wohingegen Datensektionen nicht ausführbar sind.

Nachdem ermittelt wurde, welche Art von Modul erstellt wird, werden die Symbole und Relokationen vereinheitlicht, sodass keine ELF-spezifischen Informationen mehr verwendet werden. Dabei werden die Symbole in zwei Typen unterteilt:

Endgültige Symbole (`FinalSymbol`). Symbole, deren Werte bekannt sind und entweder eine Position innerhalb des Moduls oder einen absoluten Wert repräsentieren.

Zielsymbole (`RelocationSymbol`). Symbole, die als Ziel von Relokationen eingesetzt werden. Sie enthalten keinen Wert, sondern dienen als Platzhalter und werden später mit einem endgültigen Symbol des selben Namens verbunden.

Eine Relokation enthält nach der Vereinfachung nur noch die Position innerhalb des Moduls, an der eine Änderung vorgenommen werden soll, das Zielsymbol, welches den Wert liefert der eingefügt werden soll und einen Typ, der die Art und Weise des Einfügens genauer beschreibt.

[10] Beispiel hierfür ist der GCC für H8 Architekturen. Beobachtet mit verschiedenen GCC-Versionen von `http://www.kpitgnutools.com/` (*KPIT GNU Tools and Support*). Neueste getestete Version: 4.2-GNUH8_v0703 (entspricht GCC 4.2.1).

4.5.3 Modulverwaltung

Nach der Erzeugung der Module ist es die Aufgabe der Kontrollschicht festzulegen, welche Module miteinander verbunden werden sollen. Die Basisschicht stellt hierzu *Modulbehälter* (*Repositories*) zur Verfügung, um eine Menge von zusammengehörenden Modulen miteinander zu verbinden. Für solche Behälter sind Operationen vorhanden, um Module hinzuzufügen, zu entfernen oder auszutauschen. Beim Hinzufügen neuer Module werden die Abhängigkeiten der Module innerhalb des Behälters aufgelöst, das heißt, die Zielsymbole der Relokationen werden mit endgültigen Symbolen anderer Module im Behälter verbunden. Bei anderen Operationen werden die Verbindungen entsprechend angepasst.

Durch diese Verknüpfung entsteht implizit der Abhängigkeitsgraph. So kann an jedem Modul nachgeprüft werden, ob eine Abhängigkeit offen ist oder ob ein Modul bekannt ist, welches die Abhängigkeit erfüllt. Darüber hinaus verwaltet jeder Behälter eine Liste mit Einsprungpunkten.

Modulbehälter werden zum Beispiel verwendet, um unterschiedliche Versionen einer Software zu verwalten. Die Module einer Version sollen miteinander verbunden werden, jedoch dürfen die Module unterschiedlicher Versionen nicht zusammen in einen Behälter. Auch bei der Verwaltung von Modulen für verschiedene Architekturen, wie es beispielweise bei Fernaufrufen vorkommen kann, werden Behälter eingesetzt.

4.5.4 Konfigurations- und Abbildverwaltung

Die Komponente `ImageManager` ist die Schnittstelle für die Konfiguration eines Knotens. Sie verwaltet die Module, die auf einem Knoten installiert sind. Dazu verwaltet sie Metadaten über das Speicherlayout und die Speichernutzung eines Knotens und überprüft bei einer Veränderung die Konsistenz der entstehenden Konfiguration.

4.5.4.1 Konsistenzprüfung

Die Konsistenzprüfung muss die Einhaltung der in Abschnitt 4.3 vorgestellten Bedingungen bei der Installation und beim Entfernen von Modulen sicherstellen. Dadurch wird gewährleistet, dass die Abhängigkeiten zwischen den Modulen auf einem verwalteten Gerät immer erfüllt sind und somit eine konsistente Konfiguration vorliegt.

Vor einer Operation werden (beispielsweise durch die Kontrollschicht) die Module ausgewählt, die entfernt und installiert werden sollen. Die Modulauswahl wird in der Basisschicht durch den Modulstatus gekennzeichnet. Ein Modul bekommt daher entweder den Zustand `SEL4INST` oder `SEL4REM`, je nachdem ob es installiert oder entfernt werden soll. Die Konsistenzprüfung nutzt den implizit erstellten Abhängigkeitsgraphen, um die Gültigkeit der Auswahl festzustellen. Um Probleme bei einer ungültigen Auswahl besser darzustellen und der Kontrollschicht die Möglichkeit zu geben, einen inkrementellen Prozess anzubieten, werden die Zustände in je zwei Unterzustände unterteilt:

OK Die Zustände `SEL4INST_OK` und `SEL4REM_OK` bedeuten, dass das Modul keine Probleme bereitet. Die Abhängigkeiten sind erfüllt beziehungsweise das Modul wird nicht mehr referenziert.

ERR Sind nicht alle Abhängigkeiten erfüllt oder wird das Modul auf dem Gerät noch referenziert, so erhalten die Module die Zustände `SEL4INST_ERR` beziehungsweise `SEL4REM_ERR`. Die Abhängigkeiten von beziehungsweise zu Modulen mit diesen Zuständen sollten überprüft werden, um eine gültige Auswahl herzustellen.

Die Unterzustände werden im Rahmen einer Konsistenzprüfung vergeben und aktualisiert. Die Kontrollschicht kann sie nutzen, um einem Administrator grafisch aufzuzeigen, welches Modul verantwortlich ist, wenn die aktuelle Auswahl die geforderten Bedingungen nicht erfüllt. Wird die Konsistenzprüfung nach jeder Veränderung der Auswahl aufgerufen, kann ein interaktiver inkrementeller Auswahlprozess realisiert werden. Die möglichen Zustandsübergänge werden in Abbildung 4.17 dargestellt.

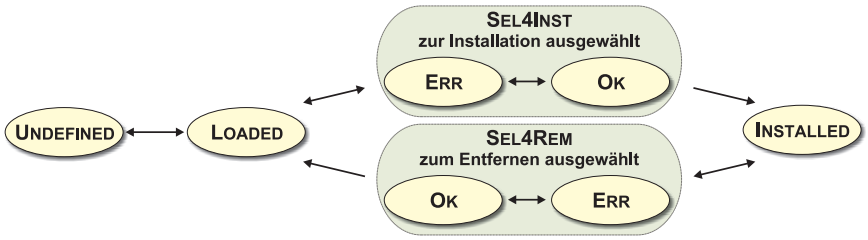


Abbildung 4.17: Übergänge zwischen den Modulzuständen

4.5.4.2 Speicherverwaltung

Ausgewählte und installierte Module werden durch ein Objekt vom Typ `Image` verwaltet. Dabei wird aus den Positionen und Größen der Module ein Modell der Speicherbelegung des Knotens erstellt und aktualisiert. Dieses kann als Grundlage für Visualisierungen und der Speicherverwaltung dienen.

Werden Module hinzugefügt, so müssen sie zunächst positioniert werden, das heißt, den binären Daten des Moduls muss eine Position im Speicher des verwalteten Knotens zugeordnet werden. Dabei muss berücksichtigt werden, dass bereits eine dynamische Speicherverwaltung vonseiten der installierten Laufzeitumgebung auf dem verwalteten Knoten vorhanden sein kann. Falls das der Fall ist, so muss die Positionierung mit ihr koordiniert werden. Bei Knoten mit Harvard-Architekturen, wie zum Beispiel AVR- oder PIC-Mikrocontrollern, betrifft das nur den Datenspeicher, da das Betriebssystem den Programmspeicher nicht verwaltet. Die Verwaltung des Programmspeichers kann dann vom Verwalter autonom durchgeführt werden.

Zur Koordinierung kann der Verwalter die Datenstrukturen der Speicherverwaltung am Gerät direkt auslesen, interpretieren und entsprechend modifizieren. Eine mögliche Realisierung besteht darin, den Code des Speicherverwaltungssystems ähnlich einer entfernten Ausführung auf dem Verwalterknoten auszuführen. Umgekehrt kann der Verwalter auch das Speicherverwaltungssystem auf dem Knoten aufrufen und die Änderungen so von dem Gerät vornehmen lassen.

Die tatsächliche Positionsbestimmung ist Teil der Speicherverwaltung und somit Aufgabe der Kontrollschicht. Sie muss daher eine entsprechende Schnittstelle anbieten, die zur Positionierung genutzt werden kann. Eine erfolgreiche Positionsbestimmung garantiert, dass genügend Speicher zur Installation des Moduls vorhanden ist.

Um den Speicherinhalt vollständig zu erfassen, kann neben den hinzugefügten Modulen ein sogenanntes *initiales Modul* verwaltet werden. Dieses entspricht dem Anfangszustand des Knotens und enthält die Symbole der Elemente, welche bereits auf dem Knoten installiert sind, wie beispielsweise das Fernwartungsmodul (RCM) und Initialisierungscode oder Funktionen im ROM des Geräts. Das initiale Modul entspricht dem Abbild, welches manuell auf dem Knoten installiert wurde und ist daher bereits gebunden. Die darin enthaltenen Elemente sind nicht relokierbar und können somit nur eingeschränkt im Rahmen der dynamischen Konfiguration verändert werden.

Ein initiales Modul ist allerdings nur dann notwendig, wenn der Ausgangszustand des Knotens nicht durch den Verwalter selbst erstellt wurde oder wenn die dazu verwendeten Informationen nicht mehr zu Verfügung stehen. Ansonsten kann das Speichermodell mit den Daten aus einer vorherigen Verwalterverbindung initialisiert werden.

4.5.5 Binder

Um den Code für die Ausführung auf dem Knoten vorzubereiten, stellt die Basisschicht einen dynamischen Binder für verschiedene Zielarchitekturen zur Verfügung. Er wird vor dem Installieren der ausgewählten Module vom `ImageManager` aufgerufen. Der Linker findet auch Verwendung, um Code im Rahmen einer Dienstleistung für die lokale Ausführung vorzubereiten. Als Eingabe erhält er neben den Ausgangsdaten eine Liste von Relokationen, die er anwenden soll. Anhand der Relokationsdaten kann der Binder die Adresse des Zielsymbols identifizieren und die entsprechende Speicherstelle anpassen. Die gebundenen Binärdaten werden in einen neuen Speicherbereich geschrieben, um die Originaldaten für ein erneutes Binden zu erhalten.

4.5.6 Interaktion mit verwalteten Knoten

Bei der Interaktion mit einem verwalteten Knoten sind verschiedene Komponenten beteiligt. Zunächst wird ein Kommunikationssystem benötigt, um Unterstützungsanfragen zu empfangen und Befehle an das Gerät zu schicken. Zur Umsetzung der Befehle wird auf dem Gerät ebenso eine Infrastruktur benötigt, wie zur Erzeugung und Weiterleitung von Anfragen an den Verwalter. Diese Komponenten werden im Folgenden betrachtet.

4.5.6.1 Kommunikation

Die Anbindung eines verwalteten Knotens wird in der Basisschicht des Verwalters durch die Komponente `DeviceStub` realisiert. Sie bietet eine einheitliche Schnittstelle an, um mit einem Knoten zu interagieren. Die wichtigste Operation dabei ist der Zugriff auf den Speicher eines Knotens. Daneben werden aber auch Möglichkeiten angeboten, Code auf dem Knoten aufzurufen oder die Ausführung an einer bestimmten Stelle auf dem Knoten fortzusetzen.

Für den tatsächlichen Zugriff auf den Speicher ist eine Kommunikation mit dem verwalteten Knoten notwendig. Diese wird durch die Komponente `CommSys` abstrahiert. Die tatsächliche Realisierung der Kommunikation kann so unabhängig vom Rest des Systems erfolgen.

Für die prototypische Implementierung wurde ein leichtgewichtiges paketorientiertes Protokoll entwickelt, welches Befehle zeichenweise über eine unzuverlässige Kommunikationsverbindung überträgt und Antwortpakete entsprechend auswertet. Das Protokoll verwendet Prüfsummen, um Veränderungen durch Übertragungsfehler festzustellen, und Identifikationsnummern, um falsche oder verlorene Nachrichten erneut anfordern zu können. Die Kommunikationsverbindung, über die diese Pakete gesendet und empfangen werden, ist durch eine einfache Schnittstelle (`DeviceLink`) vorgegeben. Abbildung 4.18 fasst den Aufbau in Form eines Klassendiagramms zusammen.

Die Trennung in drei Objekte lässt eine flexible Abstimmung auf das tatsächliche Kommunikationsmedium zu. So wurden `DeviceLink`-Objekte für Netzwerk- und serielle Verbindungen implementiert, die unabhängig vom Übertragungsprotokoll ausgetauscht werden können. Falls das Kommunikationsmedium bereits eine zuverlässige Übertragung bietet, so kann auch das Kommunikationsprotokoll durch ein einfacheres ersetzt werden. Man kann sogar eine sehr enge Kopplung mithilfe gemeinsam genutzten

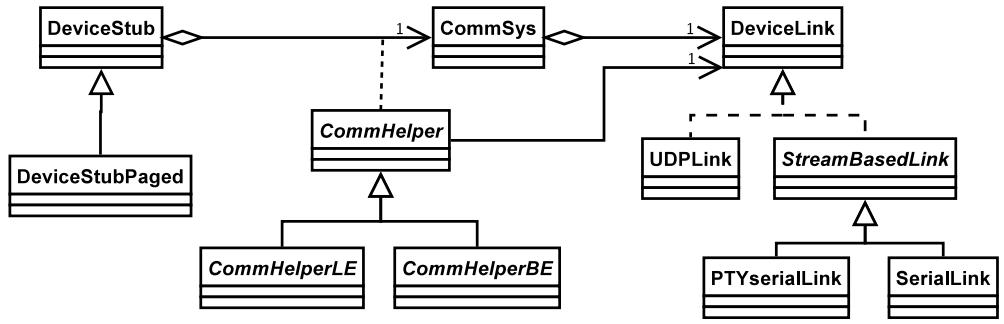


Abbildung 4.18: Klassendiagramm des Kommunikationssystems

In dem vereinfachten Klassendiagramm erkennt man die Dreiteilung in `DeviceStub`, `CommSys` und `DeviceLink`. `DeviceStub` stellt die Schnittstelle für die anderen Komponenten zur Verfügung. Neben der allgemeinen Version gibt es noch eine erweiterte Schnittstelle (`DeviceStubPaged`), welche seitenbasierte Zugriffe auf den Speicher anbietet, um Zugriffe auf Mikrocontroller mit Flashspeicher zu optimieren. Zur Übertragung wird ein `CommSys`-Objekt verwendet, welches hauptsächlich das Übertragungsprotokoll abwickelt und zur Kommunikation auf ein `DeviceLink`-Objekt zurückgreift. Die Byteordnung von Adressen, die während der Kommunikation ausgetauscht werden, passt ein `CommHelper`-Objekt an.

Speichers (*Dual-Ported-RAM*) unterstützen, indem die Speicherzugriffe direkt umgesetzt werden oder man entsprechende Kommandos erzeugt und über eine festgelegte Speicherstelle austauscht.

Das Gegenstück zum `DeviceStub` des Verwalters ist das Fernwartungsmodul (*Remote Configuration Module*, *RCM*) auf dem verwalteten Gerät. Das RCM muss immer auf dem Knoten installiert sein und ist normalerweise Teil des initialen Moduls. Es stellt den Kommunikationsendpunkt dar und empfängt und verarbeitet die Anfragen des Verwalters. Dazu implementiert es ebenfalls das oben beschriebene Kommunikationsprotokoll. Zum Zugriff auf das Kommunikationsmedium und zur Ausführung der Befehle ist das RCM in das Laufzeitsystem integriert. Der Aufbau des RCMs ist daher vom verwendeten Laufzeitsystem abhängig und wird in Abschnitt 4.5.7 diskutiert.

4.5.6.2 Befehle an einen verwalteten Knoten

Die Befehle, die ein RCM bearbeiten muss, sind sehr einfach gehalten. Zu den grundlegenden Operationen gehört das Auslesen (`peek`) und Beschreiben (`poke`) von Speicherstellen sowie das Fortsetzen der Ausführung auf dem Knoten nach einer Unterstützungsanforderung (`resume`). Daneben können noch geräte- oder laufzeitsystemspezifische Befehle angeboten werden, die oft zur Optimierung der Vorgänge dienen.

Mithilfe der `peek`- und `poke`-Operationen kann der Verwalter den Zustand des Knotens auslesen und Daten installieren. Um den Zustand des Knotens vollständig erfassen zu können, benötigt der Verwalter zum Teil auch den Inhalt einiger Register. Auf einigen Architekturen, wie beispielsweise den AVR-Mikrocontrollern, sind alle Register in den Speicher eingeblendet und können somit über die `peek`-Operation abgefragt werden. Bei anderen Architekturen ist eine zusätzliche Operation dafür notwendig[11].

Bei Harvard-Architekturen muss die Möglichkeit bestehen, auf die verschiedenen Adressräume zuzugreifen. Hierzu kann man den Adressraum in der Adresse codieren[12] oder man sieht getrennte

[11] Bei x86- und H8/300-Architekturen ist beispielsweise eine zusätzliche Operation `GetBasePointer` realisiert, die den Inhalt des Registers `EBP` beziehungsweise `r6` zurückliefert.

[12] Zur Codierung des Adressraumtyps kann man Bits verwenden, welche außerhalb des gültigen Adressbereichs liegen. Ein gesetztes Bit bedeutet beispielsweise, dass es sich um eine Adresse im Programmspeicher handelt.

`peek`- und `poke`-Operationen vor. In der prototypischen Realisierung für AVR-Mikrocontroller wurden getrennte Operationen zur Verfügung gestellt. Bei diesen Mikrocontrollern kommt hinzu, dass der Programmspeicher durch Flashspeicher ausgeführt ist. Dieser kann nur seitenweise gelesen und beschrieben werden, sodass die `peek`- und `poke`-Operationen eine entsprechend abgeänderte Schnittstelle aufweisen.

Zur Realisierung der Operationen für AVR-Mikrocontroller muss bei der Positionierung des RCMs berücksichtigt werden, dass der Code zum Verändern des Flashspeichers in einem bestimmten Adressbereich des Programmspeichers liegen muss. Nur in dieser sogenannten *Boot Loader Section* [Atm07b] sind die speziellen Befehle zum Beschreiben des Speichers wirksam.

4.5.6.3 Funktion und Erzeugung von Stellvertretern

Unterstützungsanforderungen werden durch spezielle Codestücke ausgelöst, die durch die Kontrollschicht an Stellen in den Code eingebunden wurden, an denen eine Umkonfiguration nötig wird, weil sich beispielsweise die Voraussetzungen für eine Konfiguration ändern. Diese Codestücke stellen die Stellvertreter bei einem Fernaufruf oder beim Nachinstallieren von Prozeduren dar. Sie können jedoch auch zusätzlich integriert werden, wenn eine Interaktion mit dem Verwalter notwendig ist.

Die Basisschicht stellt die Infrastruktur zur Verwaltung und Erzeugung dieser Stellvertreter bereit. Die Erzeugung wird von Stellvertreterfabriken vom Typ `StubFactory` durchgeführt. Für verschiedene Arten von Stellvertretern kann es je ein eigenes Fabrikobjekt geben. In der prototypischen Implementierung wurden zwei Fabriken entwickelt. Eine Fabrik erzeugt Codestücke, die zur allgemeinen Benachrichtigung und als Stellvertreter bei der bedarfsgesteuerten Installation verwendet werden können. Diese Stellvertreter senden neben der Unterstützungsanforderung keine weiteren Daten zum Verwalter. Die andere Fabrik erzeugt Stellvertreter, die für Fernaufrufe optimiert sind. Hier werden neben der Unterstützungsaufforderung auch die Parameter der aufgerufenen Funktion an den Verwalter gesendet.

Die generelle Funktionsweise der Fabriken unterscheidet sich nicht. Die Stellvertreter bestehen im Wesentlichen aus zwei Teilen:

- einem allgemeinen Teil, der nur einmal im System vorhanden sein muss, und
- einem individuellen Teil, der für jeden Einsatz speziell angepasst wird.

Der spezifische Teil enthält eine eindeutige Identifikationsnummer und Parameter, um die Funktion des allgemeinen Teils zu beeinflussen. Er ruft mit diesen Informationen den allgemeinen Teil auf, der eine entsprechende Nachricht erstellt und, im Rahmen dessen, die notwendigen Informationen sammelt. Anschließend greift der allgemeine Teil auf Funktionen des RCMs zurück, um die Nachricht abzuschicken. Abbildung 4.19 zeigt den Ablauf.

Durch die Aufteilung des Stellvertreters in zwei Teile kann der Speicherbedarf pro Nutzung sehr gering gehalten werden. Nur der individuelle Teil muss mehrfach vorhanden sein. Er ist allerdings sehr klein, da er nur die unbedingt notwendigen Register sichert und mit den spezifischen Werten füllt. Dann springt er sofort zu einem gemeinsam genutzten Codestück, welches die restlichen Registerinhalte sichert und das RCM aufruft. Tabelle 4.4 zeigt den Speicherbedarf der beiden Stellvertreter für verschiedene Architekturen.

Zur Erzeugung eines konkreten Stellvertreters lädt die Fabrik die beiden Teile für die jeweils benötigte Architektur. Für den allgemeinen Teil des Stellvertreters muss die Fabrik nur sicherstellen, dass er installiert wird, sobald ein Stellvertreter dieses Typs verwendet wird. Für den individuellen Teil stellt das geladene Modul eine Vorlage dar, welche Platzhalter für die spezifischen Daten enthält. Die

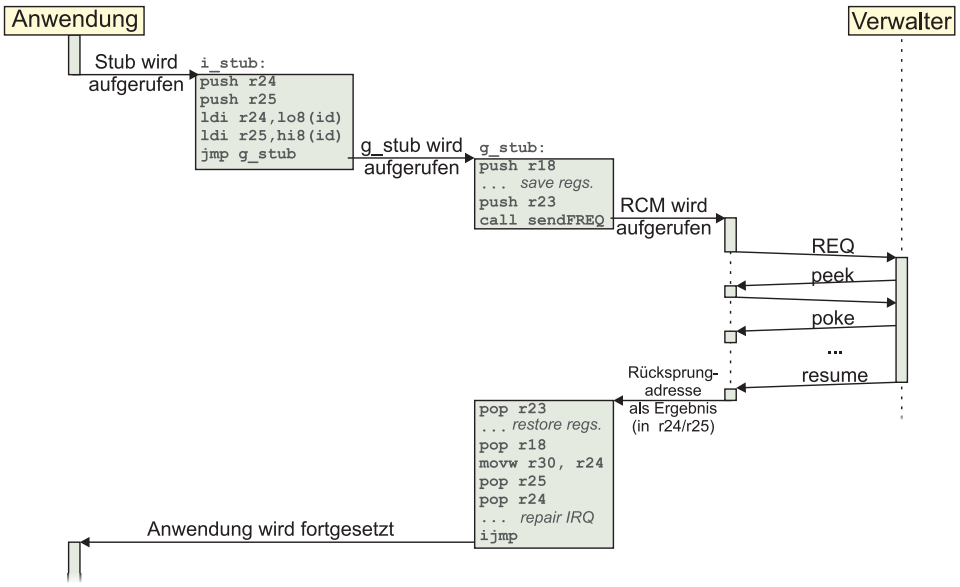


Abbildung 4.19: Ablauf bei Aktivierung des Stellvertreters

Die Abbildung zeigt das Zusammenspiel der Stellvertreter-Komponenten. Der individuelle Teil (`i_stub`) enthält die spezifischen Daten, schreibt sie in Register und springt zu einem generellen Teil des Stellvertreters (`g_stub`). Dort werden die restlichen Register gesichert und das RCM aktiviert, welches eine Unterstützungsnachricht sendet. Nach dem Beenden der Unterstützung stellt der allgemeine Stellvertreterteil die Registerinhalte wieder her und fährt mit der Ausführung in der Anwendung fort.

Architektur	Einfacher Stellvertreter	Stellvertreter für Fernaufruf
Atmel AVR ATmega	$x * 12 + 126$ Byte	$x * 52 + 206$ Byte
Renesas H8/300	$x * 10 + 88$ Byte	$x * 16 + 164$ Byte
Intel x86	$x * 10 + 66$ Byte	$x * 15 + 127$ Byte

Tabelle 4.4: Speicherbedarf der Stellvertreter

Die Tabelle zeigt den Speicherbedarf der Stellvertreter in Byte. Die Größe des spezifischen Teils ist dabei durch ein Vielfaches von x ausgedrückt. Der zweite Summand gibt die Größe des allgemeinen Teils an, der nur einmal im System vorhanden sein muss.

Der Stellvertreter für Fernaufrufe benötigt etwas mehr Speicher, da er Code enthält, um die Aufrufparameter aus den Registern oder vom Stack zu erfassen und mit der Unterstützungsanforderung zum Verwalter zu schicken.

Platzhalter sind durch Verweise auf absolute Werte implementiert. Dadurch werden sie als Relokationen im Modul sichtbar und können anhand ihres Namens identifiziert werden. Die Fabrik fertigt für jeden konkreten Stellvertreter eine Kopie der Vorlage an und ersetzt die Platzhalter durch reale Werte.

4.5.6.4 Infrastruktur zur Verarbeitung von Anfragen

Die Verwaltung der Stellvertreter wird durch ein Objekt vom Typ `StubManager` vorgenommen. Dieses stellt der Kontrollschicht eine Schnittstelle zur Registrierung einer Unterstützungsanforderung bereit. An dieser Schnittstelle wird ein Objekt vom Typ `ActionHook` übergeben, welches die Reaktion auf eine eintreffende Anforderung definiert. Zusätzlich kann man noch ein weiteres Objekt mitgeben, welches als Parameter an das `ActionHook`-Objekt übergeben wird. Üblicherweise wird hier das Funktionsmodul verwendet, welches durch den Stellvertreter ersetzt wird.

Bei der Registrierung legt der `StubManager` zunächst eine eindeutige Identifikationsnummer fest und ruft dann die entsprechende Stellvertreterfabrik auf. Die Stellvertreterfabrik wird dabei durch das `ActionHook`-Objekt bestimmt, da so ein optimierter Stellvertreter für die jeweilige Operation erzeugt werden kann, der Informationen für die Durchführung der Unterstützung sammelt. Nach der Erzeugung des Stellvertreters wird dieser zusammen mit dem Parameterobjekt und einem Verweis auf das `ActionHook`-Objekt unter der eindeutigen Identifikationsnummer abgespeichert.

Beim Eintreffen einer Unterstützungsanforderung kann so das Kommunikationssystem auf den `StubManager` zurückgreifen, um anhand der Identifikationsnummer das entsprechende `ActionHook`-Objekt zu aktivieren (siehe Abbildung 4.20).

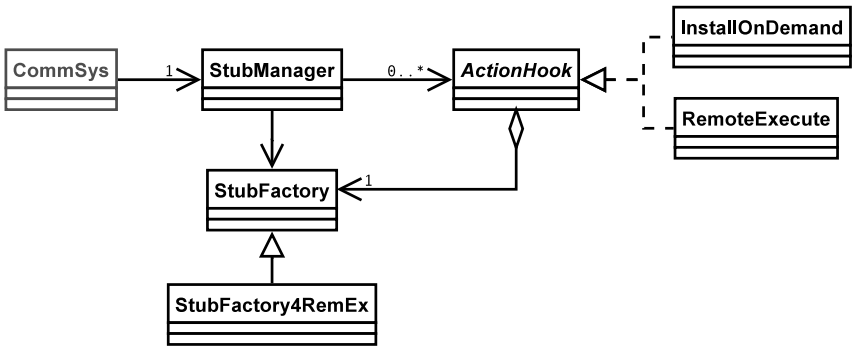


Abbildung 4.20: Klassendiagramm der Stellvertreter Infrastruktur

Das Klassendiagramm zeigt den `StubManager`, welcher zur Erzeugung von Stellvertretern eine Stellvertreterfabrik (`StubFactory`) verwendet. Die Fabrik bekommt er von einem `ActionHook`-Objekt, welches die Reaktion auf eine Unterstützungsanforderung definiert. Intern verwaltet er eine Abbildung der Stellvertreter auf die entsprechenden `ActionHook`-Objekte, sodass das Kommunikationssystem (`CommSys`) bei einer Unterstützungsanfrage das entsprechende `ActionHook`-Objekt aktivieren kann.

4.5.7 Fernwartungsmodul (RCM)

Über den genauen Aufbau des Fernwartungsmoduls (*Remote Configuration Module*, *RCM*) lassen sich kaum allgemeine Angaben machen, da es in das Laufzeitsystem, welches auf dem verwalteten Knoten verwendet wird, integriert werden muss. Besonderheiten bei der Integration der Komponenten sollen anhand der prototypischen Realisierung erläutert werden. Es lassen sich zwei Hauptaufgaben des RCMs identifizieren.

1. Zugriff auf das Kommunikationsmedium und Abwicklung des Kommunikationsprotokolls.

2. Ausführung der Operationen (`peek`, `poke`, `resume`, ...).

4.5.7.1 Kommunikation

Im einfachsten Fall kann das RCM ein eigenes Kommunikationsmedium nutzen, das nur für die Verbindung zum Verwalter verwendet wird. Dann kann die Steuerung der Kommunikationshardware vollständig durch das RCM erfolgen und die Integration in das Laufzeitsystem des Geräts ist minimal.

Bei der prototypischen Realisierung wurden jedoch Geräte verwendet, die nur eine Kommunikationsschnittstelle bereitstellen und die auch für die Anwendung zur Verfügung stehen soll. Die Erkennung und Auswertung der Paketstruktur wurde daher stromorientiert implementiert und in den Treiber der Kommunikationsschnittstelle integriert. Dadurch können Nachrichten an das RCM aus dem restlichen Nachrichtenstrom herausgefiltert werden. Die ankommenden Daten werden dazu nach einer speziellen Signatur überprüft, die anzeigt, dass es sich um ein Paket für das RCM handelt.

4.5.7.2 Ausführung der Operationen

Der nächste Schritt ist die Ausführung der Befehle. Dies muss wiederum mit dem Laufzeitsystem koordiniert werden. Die Realisierung ist davon abhängig, ob das Laufzeitsystem mehrere nebenläufige Aktivitätsträger anbietet und von welcher Seite die Interaktion gestartet wurde.

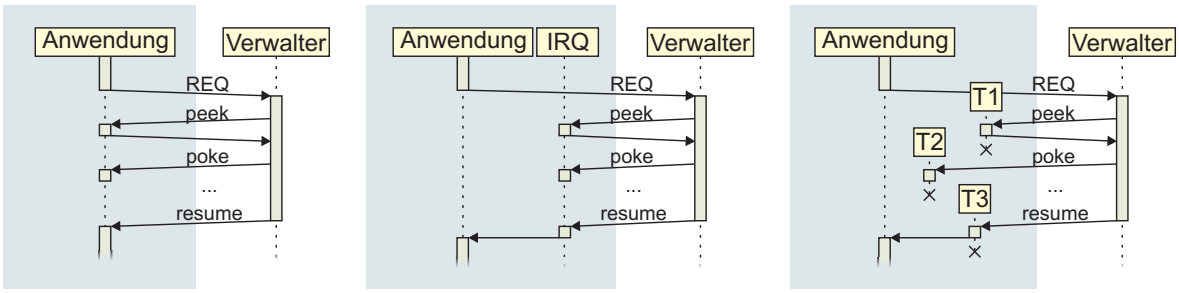
Initiiert der Verwalter die Interaktion, so stellt sie ein asynchrones Ereignis für den verwalteten Knoten dar. Der Knoten muss auf solche Ereignisse vorbereitet sein, um die Befehle des Verwalters verarbeiten zu können. Bei der Realisierung hat man verschiedene Möglichkeiten. Die Ausführung der Operationen kann beispielsweise direkt nach dem Empfangen der Nachricht gestartet werden. Sie wird dann noch im Kontext der Unterbrechungsbehandlung ausgeführt und der aktive Thread bleibt unterbrochen. Alternativ kann man auch einen neuen Thread beziehungsweise Task erstellen und diesen zur Verarbeitung einreihen, sodass er zu einem späteren Zeitpunkt vom regulären Scheduler aktiviert wird.

Wie bereits in Abschnitt 4.4.2.3 aufgezeigt, hat der zweite Ansatz in Systemen wie TinyOS Vorteile. TinyOS bietet nur einen Aktivitätsträger, daher wird bei diesem Vorgehen die Operation nicht nebenläufig zu einer aktiven Aufgabe abgearbeitet. Bietet die Laufzeitunterstützung jedoch die nebenläufige Abarbeitung von mehreren Aufgaben an, so ist dieser Vorteil nicht unbedingt gegeben. Wird außerdem eine verdrängende Einplanungsstrategie verwendet, so muss man für eine geeignete Koordinierung sorgen, sodass keine ungültigen Zwischenzustände der Veränderungen sichtbar werden.

Geht die Interaktion vom verwalteten Gerät aus, so erfolgt sie synchron im Rahmen einer Unterstützungsanforderung. Der aktuelle Thread ist daher unterbrochen und muss nach erfolgreicher Unterstützung durch den Verwalter transparent fortgesetzt werden. In Systemen wie TinyOS kann in diesem Fall die zweite der oben beschriebenen Möglichkeiten nicht angewandt werden. Da nur ein Aktivitätsträger angeboten wird, könnte die Abarbeitung der Verwaltungsoperationen durch einen neuen Task erst aktiviert werden, wenn der aktuelle Task beendet wurde. Um diese Verklemmung zu vermeiden, muss die Abarbeitung entweder direkt im Rahmen der Unterbrechungsbehandlung oder durch den aktuellen Task selbst durchgeführt werden. Bei Laufzeitsystemen, die mehrere nebenläufige Ausführungen unterstützen, ist diese Einschränkung nicht gegeben und man hat drei Varianten zur Auswahl. Abbildung 4.21 stellt diese nebeneinander dar.

4.5.7.3 Speicherbedarf

Im Rahmen der prototypischen Implementierungen zu dieser Arbeit wurde ein RCM für verschiedene Systeme entwickelt. Tabelle 4.5 zeigt den Speicherbedarf der Beispielimplementierung. Das RCM wird



(a) direkt im Task (b) im IRQ-Handler (c) durch neue Tasks

Abbildung 4.21: Abarbeitung der Befehle im Rahmen einer Unterstützungsanforderung

Teilabbildung (a) zeigt die Abarbeitung der Befehle durch den Task der Anwendung. Im Rahmen von Unterstützungsanforderungen ist dies eine sinnvolle Realisierung, da der Task auf die Fertigstellung der Operationen angewiesen ist. Die Bearbeitung von unaufgeforderten, asynchron eintreffenden Befehlen ist mit diesem Ansatz jedoch nur möglich, wenn der Task regelmäßig und aktiv die Kommunikation überprüft.
Teilabbildung (b) zeigt die Bearbeitung der Operationen direkt im Rahmen der Unterbrechungsbehandlung. Nachteil dabei ist, dass während der Abarbeitung keine anderen Unterbrechungen bearbeitet werden können. Gleichzeitig ergibt sich daraus jedoch auch der Vorteil, dass die Operationen nicht mit anderen Aufgaben koordiniert werden müssen.
In Teilabbildung (c) wird für jeden eintreffenden Befehl eine neue Aufgabe (Task) erzeugt, die nebenläufig zur Anwendung ausgeführt wird. In Systemen, die nur einen Aktivitätsträger anbieten, kann dieser Ansatz nicht bei der Bearbeitung von Unterstützungsanforderungen angewandt werden, da hierzu ein weiterer Aktivitätsträger zur parallelen Abarbeitung der Aufgaben zur Verfügung stehen müsste.

dabei aus derselben Codebasis für verschiedene Architekturen gebaut. Grundsätzlich hängt die Größe des RCMs von der Funktionalität ab. Das hier verwendetet RCM enthält das beschriebene Kommunikationsprotokoll zur zuverlässigen Übertragung und eine flexibel erweiterbare Befehlsverarbeitung, die bei der Version für AVR Mikrocontroller genutzt wurde, um zusätzliche `peek` und `poke` Befehle für den Zugriff auf den Programmspeicher anzubieten.

Architektur	.text	.rodata	.data	.bss	Total
Atmel AVR ATmega	1314	38	20	263	**1635** Byte
Renesas H8/300	1150	96	0	8	**1254** Byte
Intel x86	888	116	0	14	**1018** Byte

Tabelle 4.5: Größe eines flexiblen Fernwartungsmoduls (RCM) in Byte

Die Tabelle zeigt die Größen eines RCMs für verschiedene Architekturen in Byte. Auffällig ist, dass die Version für AVR-Mikrocontroller deutlich größer ist. Das liegt daran, dass hier separate `peek` und `poke` Befehle implementiert wurden, um auf den Programmspeicher zuzugreifen, da es sich um eine Harvard-Architektur handelt.
Da der Programmspeicher als Flashspeicher realisiert ist, wird außerdem spezieller Code benötigt, um diesen seitenweise zu beschreiben. Dieser Code nimmt mehr Platz in Anspruch als das einfache Setzen einer Speicherstelle auf den beiden anderen Architekturen. Zusätzlich wird Datenspeicher in der Größe einer Seite (hier: 256 Byte) benötigt, um die Daten vor dem Schreiben zwischenzuspeichern.

Im Vorfeld wurden auch einfachere Versionen des RCMs erstellt, die deutlich machen, wie viel Speicherplatz man einsparen kann, wenn man mit minimalem Funktionsumfang auskommt. So wurde ein Fernwartungsmodul erstellt, welches ein sehr einfaches Kommunikationsprotokoll verwendet, das keine Übertragungsfehler erkennt oder toleriert. Außerdem ist der Befehlsumfang dieses RCM fest vorgegeben und nicht flexibel erweiterbar. In Tabelle 4.6 sind die Größen für diese Implementierungen angegeben.

Architektur	.text	.rodata	.data	.bss	Total
Atmel AVR ATmega	566	0	20	0	**586** Byte
Renesas H8/300	248	20	0	0	**268** Byte
Intel x86	316	20	0	0	**336** Byte

Tabelle 4.6: Größe eines minimalen Fernwartungsmoduls (RCM) in Byte

In dieser Tabelle ist die Größe eines minimalen Fernwartungsmoduls dargestellt. Auch hier ist die Version für AVR-Mikrocontroller deutlich größer, da zusätzliche Funktionen zum Beschreiben des Flashspeichers notwendig sind. Das Zwischenspeichern einer Flashseite wird in dieser Implementierung auf dem Stack durchgeführt, sodass der Speicher hierfür nicht in dieser Tabelle auftaucht.

Im Rahmen einer Studienarbeit [Oec07] wurde außerdem ein RCM für das Betriebssystem TinyOS entwickelt. Dieses ist vom Funktionsumfang mit dem komfortablen Fernwartungsmodul aus der ersten Tabelle vergleichbar. Es wurden jedoch keine Optimierungen zur Größenreduktion durchgeführt und kann somit nicht direkt verglichen werden. Es belegt 2789 Byte Speicher.

4.5.8 Schnittstellenbeschreibung und Typinformationen

Zur Durchführung von Fernaufrufen und zum Anpassen von Datenstrukturen beim Aktualisieren und Ersetzen von Modulen müssen Typinformationen vorliegen, um binären Daten eine Struktur zuzuordnen. Diese Informationen können zum großen Teil aus Debuginformationen gewonnen werden (siehe Abschnitt 4.2.6). In der vorliegenden Arbeit werden DWARF-Debuginformationen ausgewertet, da dieses Format das Standarddebugformat für ELF-Objektdateien darstellt.

4.5.8.1 Verarbeiten von DWARF-Debuginformationen

Das Auswerten der DWARF-Debuginformationen aus den ELF-Dateien wird durch ein Objekt vom Typ `DwarfLoader` übernommen. Beim Einlesen der Objektdatei werden unbekannte Sektionen an dieses Objekt übergeben, welches prüft, ob es sich um Debuginformationen handelt. Für das Auswerten der DWARF-Informationen wird zunächst die Sektion mit dem Namen `.debug_abbrev`benötigt. Sie enthält Informationen über den genauen Aufbau der Sektion `.debug_info`. Dort wiederum sind die Informationen zu den einzelnen Programmelementen abgespeichert. Der genaue Aufbau der DWARF-Informationen ist für die weitere Arbeit nicht relevant, daher wird an dieser Stelle nicht weiter darauf eingegangen.

Das `DwarfLoader`-Objekt wird mit dem `ELFFile`-Objekt verknüpft, da sich die Debuginformationen auf den Inhalt einer Objektdatei beziehen. So stehen die Informationen bei der Modulerzeugung zur Verfügung. Der `ELFSectionExtractor`, der Module aus einer Objektdatei erzeugt, greift auf das `DwarfLoader`-Objekt zurück, um jedem erzeugten Modul Typinformationen zuzuordnen. Für jedes Funktionsmodul wird ein Objekt von Typ `Subprogramm` erstellt, welches die Schnittstellenbeschreibung der Funktion in einem DWARF-unabhängigen Format speichert. Für Datenmodule wird ein Objekt vom Typ `DataType` erstellt und zugeordnet (siehe Abbildung 4.22).

Im Rahmen dieser Zuordnung werden auch zusätzliche Debuginformationen ausgewertet. Besonders hilfreich ist beispielsweise die Information, ob eine Funktion in eine andere Funktion integriert wurde (*inlined*). In diesem Fall wird eine spezielle Abhängigkeit vermerkt, da die integrierte Funktion nicht ohne Weiteres ausgetauscht werden kann, ohne die umgebende Funktion neu zu erstellen.

Die Schnittstellenbeschreibung in Form des `Subprogramm`-Objekts liefert den Datentyp des Rückgabewerts und eine Liste mit Informationen über die Parameter als Feld von `DataComponent`-Objekten. Diese Objekte enthalten wiederum den Namen, den Datentyp und eine Ortsangabe zu einem

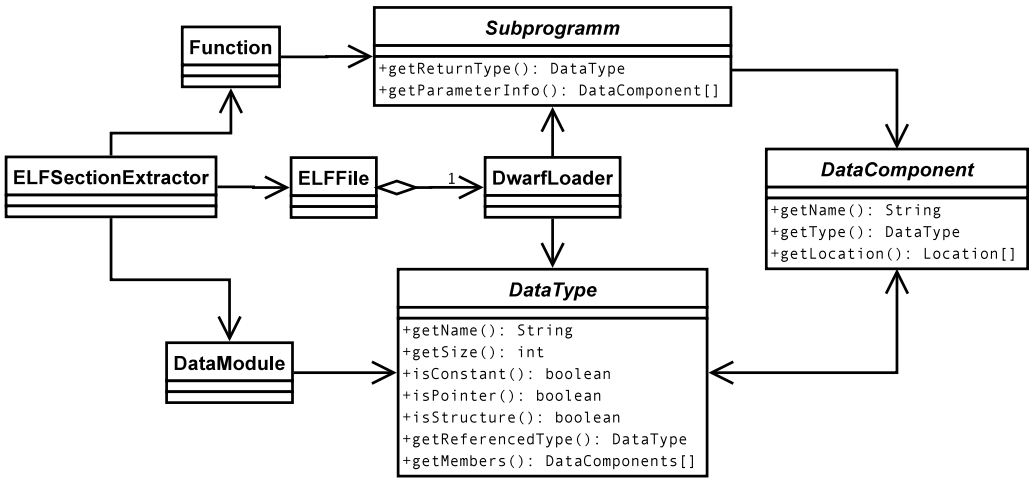


Abbildung 4.22: Klassendiagramm des DWARF-Typsystems

Das schematische Klassendiagramm zeigt den `ELFSectionExtractor`, der auf die binären Daten im `ELFFile` zugreift um `Function` oder `DataModule` Objekte zu erzeugen. Zu jedem solchen Modulobjekt wird dann mithilfe des `DwarfLoader`-Objekts ein `Subprogramm`- beziehungsweise ein `DataType`-Objekt erstellt.
Die Parameter einer Funktion werden durch `DataComponent`-Objekten ausgedrückt, welche auch die Komponenten eines zusammengesetzten Datentyps beschreiben können. `DataComponent`-Objekte beschreibt dabei immer eine Position und verweisen auf einen Typ. Das Zusammenspiel der Klassen lässt sich anhand der Methoden erahnen, die hier zum Teil angegeben sind.

untergeordneten Datum. `DataComponent`-Objekte werden auch eingesetzt, um beispielsweise die Elemente einer Struktur zu beschreiben. Datentypen werden durch Objekte vom Typ `DataType` dargestellt. Sie erlauben es abzufragen, ob es sich um einen Zeiger oder einen zusammengesetzten Datentyp handelt. Im jeweiligen Fall kann der referenzierte Typ beziehungsweise eine Liste mit den Elementen abgefragt werden.

4.5.8.2 Schnittstellenbeschreibung aus externer Quelle

Prinzipiell ist es möglich, die benötigten Typ- und Schnittstellenbeschreibungen aus einer anderen Quelle zu erhalten. So könnte man die Schnittstellen von Funktionen mithilfe einer speziellen Schnittstellenbeschreibungssprache wie CORBA-IDL [Omg04] oder Microsoft-IDL [Mic07] definieren. Ein solcher Ansatz hat jedoch einige Nachteile.

Zum einen müssen die Beschreibungen separat und manuell gepflegt werden. Änderungen am Quellcode wirken sich nicht automatisch auf die Schnittstellenbeschreibungen aus. Inkonsistenzen in den Daten können zu einem Fehlverhalten des Verwalters führen, da zum Beispiel die Parameter falsch interpretiert werden.

Ein weiterer Nachteil ist, dass mit Schnittstellenbeschreibungssprachen keine architekturspezifischen Informationen zum Aufbau der Datentypen ausgedrückt werden. Bei Debuginformationen sind beispielsweise die Positionen der Elemente einer Struktur innerhalb des Speicherbereichs, der die Struktur ausmacht, genau angegeben. In einer allgemeinen Schnittstellenbeschreibungssprache kann man nur den strukturellen Aufbau von Schnittstellen und Datentypen beschreiben. Die Repräsentation der Daten im Speicher ist dabei nicht festgelegt.

Die Verwendung einer Schnittstellenbeschreibungssprache hat aber auch Vorteile. Debuginformationen können nur das Wissen widerspiegeln, das in der Programmiersprache ausdrückbar ist. Die Datentypen werden jedoch nicht immer eindeutig verwendet. Ein Beispiel ist ein Zeiger auf ein

Zeichen in der Sprache C (`char*`). Mit diesem Datentyp kann nicht nur ein einzelnes Zeichen, sondern auch ein Feld von Zeichen referenziert werden. Die Länge des Feldes muss dabei durch zusätzliche Angaben gegeben sein oder ist durch ein spezielles Zeichen festgelegt, wie es für nullterminierte Zeichenketten üblich ist. Welche der Bedeutungen gemeint ist, muss durch zusätzliches Wissen definiert sein.

Eine allgemeine Beschreibungssprache unterliegt nicht diesen Beschränkungen. Hier kann man explizit festlegen, dass es sich beispielsweise um eine nullterminierte Zeichenkette handelt oder dass sich es sich um ein Feld variabler Länge handelt, dessen Länge im nächsten Parameter der Funktion übergeben wird [Mic07].

4.5.8.3 Verbesserung der Debuginformationen

Aufgrund der unzureichend genauen Typinformationen, mancher Programmiersprachen, ist auch bei der Verwendung von Debuginformationen vorgesehen, zusätzliches Wissen einzubinden. Dies kann durch die Verwendung einer externen Schnittstellenbeschreibung erfolgen. Dabei können die Vorteile beider Ansätze kombiniert werden. Es bleibt jedoch der Nachteil bestehen, dass die Beschreibung extern anzufertigen und zu pflegen ist.

Zur Vermeidung dieses Nachteils kann man auch den Quelltext um zusätzliche Informationen anreichern. Viele Programmiersprachen, wie beispielsweise C, lassen es zu, dass man eigene Datentypen definiert (`typedef`). Diese Datentypen sind in den Debuginformationen sichtbar. Bei der Softwareentwicklung kann man nun darauf achten, dass man eindeutige Datentypen definiert und verwendet. So kann man sich beispielsweise einen Datentyp `Zeichenkette` definieren, der technisch gesehen einem `char*` entspricht. Wird nun an allen Stellen, an denen eigentlich eine nullterminierte Zeichenkette übergeben wird, der neue Datentyp eingesetzt, so ist es möglich, aus den Debuginformationen eine genauere Beschreibung der Schnittstelle zu extrahieren. Äquivalente Datentypen können auch für Felder variabler oder fester Länge erstellt werden.

Der Nachteil dieses Ansatzes ist allerdings, dass dieses Verfahren bereits während der Entwicklung der Software berücksichtigt werden muss und sich nicht ohne Änderung auf existierende Programme anwenden lässt. Des Weiteren muss die Technik von den Entwicklern akzeptiert werden. Im Allgemeinen ist es aber gar nicht so ungewöhnlich, spezielle Typen für Zeichenketten oder Felder zu verwenden. Ein Beispiel hierfür sind die Richtlinien zur Programmentwicklung für Symbian OS [Nok04], welche die Verwendung spezieller Typen für Zeichenketten und Felder vorschlagen, um die Länge dieser Konstrukte explizit zu erfassen. Egal, ob der Grund die Übersichtlichkeit, die Vermeidung von Fehlern oder die Kennzeichnung verschiedener Datentypen ist, mit diesem Hilfsmittel enthalten die Debuginformationen genügend Informationen, um daraus eine ausreichend genaue Schnittstellenbeschreibung zu erstellen.

4.5.8.4 Datentypenkonvertierung

Auf der Basis der Typinformationen stellt die Basisschicht Funktionen zur Konvertierung der Daten zwischen verschiedenen Architekturen zur Verfügung. Die Funktionen passen die Byteordnung und die Größe primitiver Datentypen an und können rekursiv auch zusammengesetzte Datentypen umwandeln. Trifft die Funktion auf einen Zeiger, so wird eine Hilfsfunktion verwendet, die den Zeiger konvertieren muss. Sie muss situationsabhängig das Ziel des Zeigers zunächst auf den Verwalter kopieren oder den Zeiger als Integerwert interpretieren. Verwendet wird die Datentypkonvertierung bei der Vor- und Nachbereitung eines Fernaufrufs.

4.6 Operationen zur dynamischen Konfiguration eines Knotens

Aufbauend auf den in Abschnitt 4.3 beschriebenen Mechanismen, können Verwaltungsoperationen zur Unterstützung und Konfiguration von kleinen Knoten realisiert werden. Die Basisschicht stellt die grundlegende Infrastruktur zur Realisierung dieser Methoden bereit. Die Steuerung der Operationen ist jedoch Aufgabe der Kontrollschicht, da strategische Entscheidungen den genauen Ablauf beeinflussen.

In diesem Abschnitt wird die Realisierung der Operationen dargestellt. Dabei wird aufgezeigt, welche Entscheidungen von der Kontrollschicht zu treffen sind.

4.6.1 Bedarfsgesteuertes Installieren von Modulen

Das bedarfsgesteuerte Installieren wird durch ein `ActionHook`-Objekt vom Untertyp `InstallOnDemand` realisiert. Es verwendet die einfache Stellvertreterfabrik, da außer der Identifikation der Funktion keine weiteren Informationen benötigt werden.

Welche Funktionen erst bei Bedarf installiert werden sollen, entscheidet die Kontrollschicht bei der Erzeugung des initialen Abbildes. Ebenso muss die Kontrollschicht dafür sorgen, dass zu dem Zeitpunkt, an dem die Funktion installiert werden soll, genügend Speicher verfügbar ist.

Der Vorgang des Installierens wird durch den Austauschmechanismus realisiert. Abhängigkeiten des neu zu installierenden Moduls müssen dabei aufgelöst werden. Funktionen, von denen es abhängig ist, können entweder mitinstalliert oder durch Stellvertreter ersetzt werden. Datenmodule müssen immer mitinstalliert werden, wie im nächsten Abschnitt erläutert wird.

Nach Beendigung des Austausches wird die Ausführung an der Stelle der neu installierten Funktion fortgesetzt.

Bedarfsgesteuertes Installieren für Datenmodule

Bedarfsgesteuertes Installieren wird nur für Funktionsmodule angeboten, da hierbei durch einen Stellvertreter eine einfache Möglichkeit besteht, die Benutzung der Funktion auf dem verwalteten Knoten festzustellen und Unterstützung anzufordern. Für Daten funktioniert dieser Weg im Allgemeinen nicht. Um den Zugriff auf Daten festzustellen, könnte man eine der beiden folgenden Möglichkeiten nutzen:

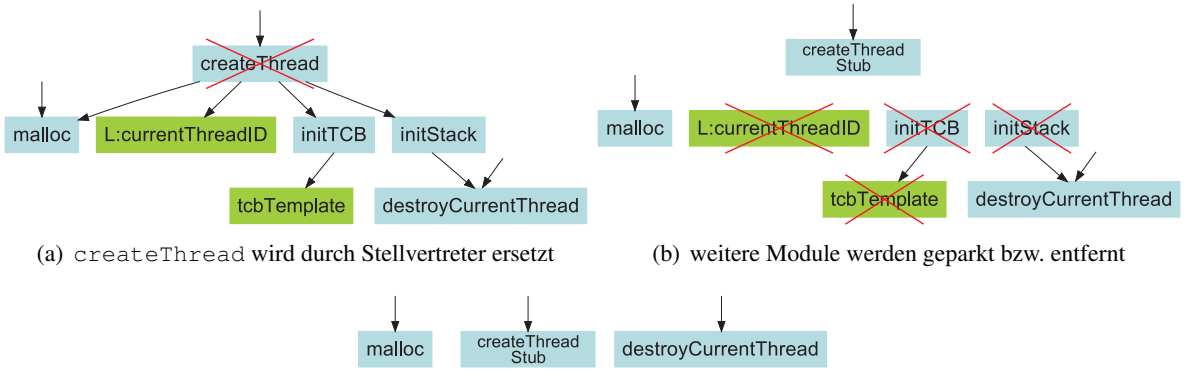
- Zugriffe auf Daten könnten auf Funktionsaufrufe abgebildet werden. Um das für die Anwendungsentwicklung transparent zu realisieren, müssen automatisch alle Zugriffe auf Daten durch entsprechende Funktionsaufrufe umgesetzt werden. Hierzu benötigt man jedoch einen entsprechenden Codetransformator oder Unterstützung vom verwendeten Compiler. Nachteil dabei ist außerdem, dass eine zusätzliche Indirektionsstufe eingeführt wird, die einen entsprechenden Mehraufwand verursacht.

- Eine Adressumsetzeinheit (*Memory Management Unit*, *MMU*) kann genutzt werden, um Zugriffe auf bestimmte Adressen abzufangen und zu melden. Kleine Mikrocontroller sind jedoch häufig nicht mit dieser Hardwareunterstützung ausgestattet.

4.6.2 Parken von Modulen

Das Parken wird von der Kontrollschicht ausgehend initiiert, um zum Beispiel Speicher für das bedarfsgesteuerte Installieren freizumachen. Beim Parken wird eine Funktion durch einen Stellvertreter ausgetauscht, der mithilfe des `InstallOnDemand-ActionHook`-Objekts erzeugt wird. Da ein

Stellvertreter im Allgemeinen keine Abhängigkeiten zu anderen Anwendungsfunktionen hat, können durch das Parken einer Funktion andere Daten- und Funktionen ebenfalls nicht mehr benötigt werden. Abbildung 4.23 zeigt das anhand eines einfachen Beispiels.



(a) `createThread` wird durch Stellvertreter ersetzt

(b) weitere Module werden geparkt bzw. entfernt

(c) Endergebnis

Abbildung 4.23: Entfernen von abhängigen Modulen beim Parken

Anhand eines Ausschnitts aus dem vereinfachten Abhängigkeitsgraphen um die Funktion `createThread`, zeigen die Teilabbildungen die Auswirkungen beim Parken dieser Funktion. Da die Funktion durch einen Stellvertreter ersetzt wurde, werden einige abhängige Module nicht mehr referenziert. Sie können daher ebenfalls geparkt werden. Ein Stellvertreter muss lediglich für die Funktion `createThread` eingesetzt werden, da sie die einzige ist, die noch vom verbleibenden Programm referenziert wird.

Funktionen, die durch das Parken einer Funktion im Abhängigkeitsgraphen nicht mehr erreichbar sind, werden daher entfernt, wenn es von der Kontrollschicht gewünscht wird. Da diese Funktionen nicht mehr referenziert werden, muss auch kein Stellvertreter installiert werden. Bei Datenmodulen wird der aktuelle Zustand gesichert, um ihn wieder herzustellen, wenn das Modul durch einen späteren Bedarf wieder installiert wird.

Das Parken von einzelnen Datenmodulen kann nur angeboten werden, wenn das bedarfsgesteuerte Installieren für Datenmodule unterstützt wird. Ansonsten werden Datenmodule nur in Abhängigkeit einer Funktion geparkt. Außerdem muss die Kontrollschicht je nach Situation damit rechnen, dass die Funktion zurzeit aktiv ist und die Operation nicht durchgeführt werden kann.

4.6.3 Ersetzen und Aktualisieren von Modulen

Das Ersetzen eines Moduls entspricht zunächst dem Austauschen eines Anwendungsmoduls durch ein anderes Anwendungsmodul. Es unterscheidet sich jedoch von den bisher vorgestellten Verwendungen der Basisoperation “Austauschen”, da bisher immer ein Stellvertreter beteiligt war, von dem bekannt ist, dass er keine Abhängigkeiten zu anderen Anwendungsmodulen hat. Beim Ersetzen hat normalerweise sowohl das ursprüngliche als auch das neue Modul Abhängigkeiten auf andere Module. Ein Beispiel eines üblichen Falles ist das Aktualisieren einer Funktion, welche auf globale Daten zugreift. Die entsprechenden Datenmodule müssen gleichzeitig mit dem Funktionsmodul angepasst oder ausgetauscht werden.

Da kein Stellvertreter direkt beteiligt ist, wird der Vorgang immer von der Kontrollschicht ausgehend gestartet. Es kann jedoch sein, dass der Anlass des Ersetzens eine Benachrichtigung durch den verwalteten Knoten war.

Für das Ersetzen muss eine Abbildung erstellt werden, welche die Änderung an der Struktur darstellt. Darin muss vermerkt sein, welche Module neu hinzukommen, welche Module durch andere ausgetauscht und welche Module komplett entfernt werden.

Das Erstellen der Abbildung kann man automatisieren, indem man die Namen der Module heranzieht. Ausgangsbasis ist je ein Behälter mit der alten und einer mit der neuen Konfiguration. Die Modulnamen werden miteinander abgeglichen, dabei wird angenommen, dass Module mit gleichem Namen aufeinander abgebildet werden sollen. Dieses automatische Vorgehen kann jedoch Sonderfälle nicht berücksichtigen, wenn beispielsweise eine Funktion aufgeteilt wurde und im Rahmen dessen einen anderen Namen erhalten hat. Daher dient die automatisch erstellte Abbildung nur als Ausgangsbasis für die Kontrollschicht, welche Änderungen vornehmen kann und die endgültige Abbildung liefert.

Das Umsetzen der Abbildung und somit das Durchführen der Ersetzung wird von der Basisschicht vorgenommen. Bei einzelnen Fällen wird die Kontrollschicht eingebunden, um anwendungsspezifische Entscheidungen zu treffen.

Im Folgenden sollen die verschiedenen Fälle anschaulich dargestellt werden. Als Beispiel soll eine Funktion durch eine neue Version ersetzt werden. Die Abhängigkeiten im Beispiel sind globale Variablen, das Vorgehen ist jedoch ähnlich für abhängige Funktionen anwendbar.

Neue Module Die neue Funktion benötigt eine zusätzliche globale Variable. In diesem Fall wird das Modul, welches die neue Variable repräsentiert, installiert.

Unbenutzte Module Im umgekehrten Fall wird eine Variable nicht mehr von der neuen Funktion genutzt. Es muss ermittelt werden, ob das entsprechende Datenmodul noch von anderen Modulen referenziert wird. Kann mit Sicherheit festgestellt werden, dass es nicht mehr verwendet wird, so wird es entfernt.

Unveränderte Module Die neue Funktion verwendet eine globale Variable, die auch von der ursprünglichen Funktion verwendet wurde. Handelt es sich um dieselbe Variable, so muss keine Anpassung vorgenommen werden.

Ersetzte Module Die neue Funktion verwendet eine Variable, die den gleichen Namen hat wie eine zuvor genutzte Variable, jedoch nicht identisch ist. Sie hat beispielsweise einen anderen Datentyp. Dann ist zwischen zwei Fällen zu unterscheiden:

- Wird die alte Variable ansonsten nicht mehr referenziert, so ist davon auszugehen, dass sie durch die neue Variable ersetzt wurde. Vor dem Austauschen des Moduls wird die Kontrollschicht beauftragt, den Zustand aus dem Speicherbereich der alten Variablen in den Speicherbereich der neuen Variablen zu transferieren und umzuwandeln. Details dazu sind im nächsten Abschnitt zu finden.
- Wird die alte Variable hingegen noch von anderen Stellen referenziert, so handelt es sich vermutlich um eine unvollständige Abbildung. Die Kontrollschicht wird daher benachrichtigt und kann in diesem Fall explizit anweisen, das Modul als neues Modul zu installieren.

Zustandsübertragung

Im Rahmen der Aktualisierung werden Datenmodule ersetzt. Dabei muss der Zustand des ursprünglichen Moduls in das Format des neuen Moduls umgewandelt werden. Dieser Vorgang ist im Allgemeinen anwendungsabhängig und kann nicht automatisch durchgeführt werden. Die Basisschicht bietet jedoch Unterstützung an.

Das Datenmodulobjekt kann mit dem aktuellen Speicherinhalt vom verwalteten Gerät aktualisiert werden. Dadurch steht der aktuelle Zustand am Verwalter zur Verfügung und kann in das neue Modulobjekt übertragen und dabei umgewandelt werden. Schließlich kann man die Modulobjekte austauschen, wodurch der neue Modulinhalt installiert wird.

Zur Umwandlung der Daten stellt die Basisschicht mit der Datentypbeschreibung und den Funktionen zur Datentypkonvertierung eine Infrastruktur zur Verfügung, die es erlaubt, einzelne Elemente aus den Speicherbereichen zu extrahieren und neu zusammenzustellen. Damit lassen sich Veränderungen im Aufbau von zusammengesetzten Datentypen bewältigen, wie zum Beispiel die Verschiebung der Elemente einer Struktur, die durch das Entfernen eines Elements auftreten kann. Behalten die anderen Elemente ihre Namen und Datentypen, so können sie automatisch zugeordnet und übertragen werden. Die Verantwortung liegt jedoch bei der Kontrollschicht, da nicht alle Fälle automatisch und ohne inhaltliches Wissen durchgeführt werden können [Sun01]. Werden beispielsweise mehrere einzelne Variablen in einer Struktur zusammengefasst, so ist es nicht möglich, dies automatisch zu erkennen.

Das Konvertieren des Zustands ist im Allgemeinen nur für veränderbare Datenmodule notwendig. Bei Code gibt es keinen Zustand anzupassen oder zu übertragen, bei konstanten Datenmodulen ist der Inhalt bekannt und kann vorab in die Erstellung der neuen Module einfließen.

4.6.4 Externe Dienste mittels Fernzugriff

Selten benutzte Funktionen können von der Kontrollschicht als externe Dienste ausgelagert werden. Aufrufe der Funktion werden dabei mittels Fernaufruf transparent zum Dienstanbieter weitergeleitet. Die Funktion kann dabei an verschiedenen Orten ausgeführt werden:

- Zum einen kann der Verwalter selbst die Funktion als Dienst zur Verfügung stellen.
- Zum anderen kann die Funktion an einem anderen verwalteten Knoten ausgelagert werden.

4.6.4.1 Verwalter als Dienstleister

Die Ausführung am Verwalterknoten wird durch ein `ActionHook`-Objekt vom Untertyp `RemoteExecute` realisiert. Zur Erzeugung der Stellvertreter kann eine der beiden in Abschnitt 4.5.6.3 vorgestellten Fabriken eingesetzt werden. Die einfache Fabrik ist besonders für Funktionen ohne Parameter geeignet. In allen anderen Fällen sollten hingegen Stellvertreter erzeugt werden, welche die Parameter mit der Unterstützungsanforderung mitsenden. Ansonsten müssen die Parameter erst durch `peek`-Operationen vom verwalteten Gerät gelesen werden.

Die Ausführung der Funktion beim Empfangen einer entsprechenden Unterstützungsanforderung ist in Abbildung 4.24 dargestellt. Der Ablauf gliedert sich in drei Phasen:

1. Erstellen einer Ausführungsumgebung am Verwalterknoten
2. Ausführen der Funktion in dieser Ausführungsumgebung
3. Abbauen der Ausführungsumgebung

Um ein Funktionsmodul durch den Verwalter auszuführen, müssen am Verwalter alle Abhängigkeiten erfüllt sein. Dies wird mithilfe einer Ausführungsumgebung organisiert und realisiert. Die Ausführungsumgebung verwaltet einen Adressraum, in dem das Funktionsmodul und alle Abhängigkeiten am Verwalter installiert werden.

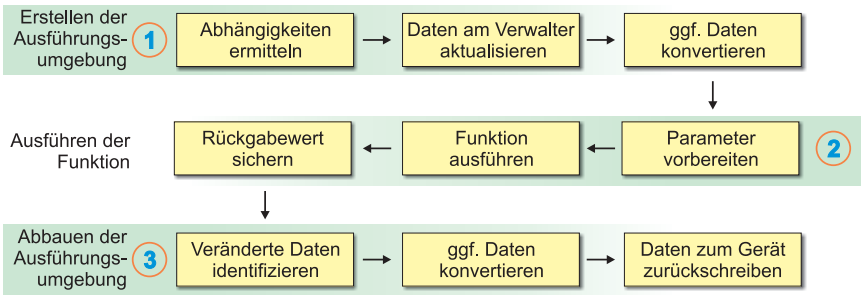


Abbildung 4.24: Ablauf einer Ausführung am Verwalter
Vor der Ausführung wird der Zustand der abhängigen Datenmodule vom Gerät geladen und in der Ausführungsumgebung installiert. Nach der Ausführung werden veränderte Daten wieder in den Speicher des Geräts geschrieben.

Bei der Bestimmung der Abhängigkeiten ist zu beachten, dass der Verwalterknoten oft eine andere Architektur besitzt als der verwaltete Knoten. Daher muss zwischen Modulen für das Gerät und Modulen für den Verwalter unterschieden werden. Zudem wird eine Abbildung benötigt, mit deren Hilfe man zu einem Modul der einen Architektur das oder die entsprechenden Module der anderen Architektur bestimmen kann. Die Abbildung wird durch die Kontrollschicht bereitgestellt.

Zur Bestimmung der Abhängigkeiten wird das Funktionsmodul für den Verwalter betrachtet. Um die hiervon referenzierten Datenmodule in der Ausführungsumgebung zu installieren, müssen sie zunächst mit den Daten von dem verwalteten Gerät initialisiert werden. Dazu werden die entsprechenden Datenmodulobjekte für das Gerät festgestellt und mit dem Speicherinhalt vom verwalteten Gerät aktualisiert. Daraufhin können die Daten in die Module für den Verwalter übertragen und gegebenenfalls konvertiert werden.

Bei referenzierten Funktionen muss die Kontrollschicht entscheiden, ob sie ebenfalls in der Ausführungsumgebung installiert oder ob sie stattdessen durch Stellvertreter ersetzt werden sollen. Ein Stellvertreter kann einen Rückruf zum Gerät veranlassen, um die Funktion dort auszuführen. Diese Möglichkeit bietet sich an, falls die Funktion besonders viele Daten referenziert und somit das Transportieren all dieser Daten sehr aufwendig ist.

Nachdem die Ausführungsumgebung aufgebaut ist, kann die Funktion aufgerufen werden. Zuvor bereitet der Verwalter allerdings noch die direkten Parameter des Funktionsaufrufs vor. Er extrahiert sie aus der Unterstützungsanforderung und wandelt sie gegebenenfalls für die vorhandene Architektur um. Unmittelbar vor dem Aufruf legt er die Parameter dann auf dem Stack ab.

Nach der Ausführung der Funktion werden die veränderten Daten zum verwalteten Gerät zurückübertragen. Das Vorgehen dabei ist analog zu der Vorbereitung der Ausführungsumgebung. Der Verwalter identifiziert veränderte Datenmodule und bestimmt die entsprechenden Datenmodule auf dem Gerät. Anschließend werden die Daten in die Zielmodule übertragen und dabei gegebenenfalls konvertiert.

Bei der tatsächlichen Realisierung lassen sich verschiedene Varianten unterschieden:

Direkt Die Möglichkeit, auf die der oben beschriebene Ablauf abgestimmt wurde, ist, dass der Verwalter die Funktion direkt ausführt. Hierbei ist oft eine Architekturgrenze zu überwinden, weswegen die Module in verschiedenen Versionen vorliegen müssen. Im einfachsten Fall kann der Quellcode für beide Architekturen übersetzt werden. Bei ähnlichen Optimierungseinstellungen lässt sich dann die überwiegende Anzahl der Module anhand des Namens automatisch aufeinander abbilden.

Simulation Um die aufwendige Zustandskonvertierung zwischen verschiedenen Architekturen zu vermeiden, kann der Verwalter um einen Simulator erweitert werden, der die Architektur des

verwalteten Knotens simuliert. Bei geschickter Adressnutzung können sogar die Adressen der Daten beibehalten werden. Die Simulation erlaubt es auch, Daten erst bei Bedarf nachzufordern. Der Simulator kann dazu den simulierten Speicher überwachen und bei einem Zugriff auf ein nicht vorhandenes Modul dessen Daten nachfordern.
Allerdings muss ein geeigneter Simulator für die Zielarchitektur vorhanden sein. Außerdem sind die Kosten zu berücksichtigen, die durch das Simulieren der Zielarchitektur entsteht.

Extern Eine weitere Möglichkeit besteht darin, dass dem Verwalter ein spezieller Knoten zugeordnet ist. Dieser hat dieselbe Architektur wie die verwalteten Knoten und wird nur zur Ausführung der Fernaufrufe der verwalteten Knoten eingesetzt. Der Knoten sollte dazu mit einer schnellen Kommunikationsverbindung an den Verwalter angebunden und besonders für dynamische Veränderungen geeignet sein, indem beispielsweise der Programmspeicher durch RAM statt Flashspeicher ausgeführt ist.
Mit solch einer Konfiguration kann der Verwalter Funktionen zur Ausführung auf diesem Knoten installieren und anschließend wieder entfernen. Man hat auch hierbei den Vorteil, dass keine Zustandskonvertierung stattfinden muss. Allerdings müssen die Kosten für den zusätzlichen Knoten berücksichtigt werden.

4.6.4.2 Verwalteter Knoten als Dienstleister

Das Auslagern einer Funktion an einen anderen verwalteten Knoten hat den Vorteil, dass das Gesamtsystem auch ohne den Verwalter lauffähig und somit für temporäre Verwalter geeignet ist. Es muss jedoch ein geeigneter Knoten zu Verfügung stehen, der über genügend Ressourcen verfügt, um die Funktion und das Fernaufrufsystem auszuführen. Zur Durchführung des Fernaufrufs muss eine Infrastruktur für den verwalteten Knoten vorliegen, die es erlaubt, dass er Fernaufrufe entgegen nimmt und verarbeitet.

Die Module der Infrastruktur können vor der Installation durch den Verwalter parametrisiert werden, ähnlich wie die Stellvertretermodule in Abschnitt 4.5.6.3. Dabei kann der Verwalter die genaue Anzahl der Aufrufparameter und die Adresse der Zielfunktion einfügen. Außerdem kann er entscheiden, ob Code zur Anpassung des Datenformats hinzugefügt werden muss, falls die Knoten von unterschiedlicher Architektur sind. Diese Aufgaben können jedoch mithilfe der bereits vorgestellten Mechanismen und Operationen durchgeführt werden, sodass hierfür kein zusätzlicher Dienst notwendig ist.

4.7 Zusammenfassung

In diesem Kapitel wurde die Basisschicht eines Verwaltungsknotens vorgestellt. Zunächst wurden die Operationen aufgezeigt, die durch Mechanismen der Basisschicht realisiert werden sollen. Als Grundlage wurde danach die Modularisierung der Software auf der Basis von relokierbaren Objektdateien vorgestellt. Anschließend wurden drei Mechanismen präsentiert, welche die Grundlage der dynamischen Konfiguration bilden. Dazu wurde auch die Problematik angesprochen, dass nicht immer alle Verweise zwischen Modulen gefunden und verändert werden können. Als nächstes wurden einige Vorgehensweisen anhand der prototypischen Realisierung aufgezeigt. Dabei wurde vermittelt, wie die Module und die Konfiguration verwaltet werden. Außerdem wurde die Anbindung eines verwalteten Knotens an den Verwalter betrachtet. Mit diesen Grundlagen wurde abschließend die Realisierung der geforderten Konfigurationsoperationen dargestellt.

5 Kontroll- und Verwaltungsschicht

Die Kontrollschicht ist der systemabhängige Teil des Verwalters. Sie ist für die Unterstützung und dynamische Konfiguration der verwalteten Knoten verantwortlich. Sie verändert die Software auf den Knoten mithilfe der Mechanismen, die von der Basisschicht zur Verfügung gestellt werden, und entscheidet, welche Module wie genutzt werden sollen. Dabei findet, technisch gesehen, keine Unterscheidung zwischen Anwendung und Laufzeitsystem statt. Für die Planung und Umsetzung einer sinnvollen Strategie ist jedoch inhaltliches Wissen notwendig, sodass sich die Strategie auf ein bekanntes Laufzeitsystem konzentrieren muss.

5.1 Konfigurationsfindung

Um zu entscheiden, welcher Mechanismus eingesetzt werden soll, sind Informationen über die verfügbaren Knoten und Ressourcen notwendig, aber auch über die Anforderungen und beschränkenden Rahmenbedingungen beim Einsatz der Module. Grundlegende Informationen über die Knoten können der Hardwaredatenbank entnommen werden. Die Bedingungen für den Einsatz der Module sind teilweise durch ihre Semantik bestimmt und daher extern vorgegeben.

Neben den inhaltlichen Gründen spielen auch die Kosten beim Einsatz der einzelnen Mechanismen eine Rolle. Sie müssen gegenüber den Vorteilen abgewogen werden. Da die Vorteile und Einsatzmöglichkeiten der einzelnen Mechanismen bereits in Abschnitt 4.1 vorgestellt wurden, stehen hier vor allen die Kosten im Mittelpunkt.

5.1.1 Kosten

Neben dem Speicherplatzbedarf für die Infrastruktur auf dem verwalteten Knoten ist vor allem die Unterbrechung beziehungsweise Verzögerung der Ausführung als Nachteil anzusehen. Wenn davon auszugehen ist, dass der Verwalterknoten über ausreichend Ressourcen und eine entsprechend schnelle CPU verfügt, so wird die Verzögerung hauptsächlich durch die Kommunikation beeinflusst[1]. Tabelle 5.1 zeigt die Zeitaufteilung bei einem Konfigurationsvorgang.

[1] Das gilt nur eingeschränkt, wenn die Kommunikation über gemeinsam genutzten Speicher erfolgt.

	Anforderung senden	Änderungen vorbereiten	Kommandos übertragen	Code installieren	Funktions-aufruf
ATmega 169	2,0 ms	20,5 ms	333,6 ms	19,5 ms	0,002 ms
H8/300	2,0 ms	10,4 ms	243,3 ms	0,2 ms	0,008 ms

Tabelle 5.1: Zeitverbrauch bei einer bedarfsgesteuerten Installation

Die Tabelle zeigt Zeiten für die Installation einer kleinen Funktion auf einem ATmega169 (AVR Butterfly) und auf einem H8/300 (Lego RCX). Die Geräte sind jeweils über eine serielle Verbindung mit 19200 Baud an den Verwalter angebunden. Die erste Spalte ist der Zeitbedarf zum Versenden des 4 Byte großen Anforderungspakets. Anschließend wartet das Gerät bis Kommandos eintreffen. Diese Zeit ist in der zweiten Spalte dargestellt und entspricht der Verarbeitung des Auftrags und der Vorbereitung der Änderungen am Verwalter. Dabei wird festgelegt, was installiert werden muss, wohin es installiert werden soll und welche Verweise anzupassen sind. Diese Zeit hängt hauptsächlich von der Leistungsfähigkeit des Verwalters ab. Die Messungen wurden auf einem Linuxsystem mit 2 GHz Pentium M durchgeführt. Im Beispiel muss eine Funktion und ein Datenmodul installiert und ein Verweis von der aufrufenden Funktion angepasst werden. In den nächsten beiden Spalten ist die Zeit dargestellt, die benötigt wird, um die Änderungen durchzuführen. Dabei sind die Zeiten für das Übertragen und das Anwenden der Änderungen getrennt dargestellt ist. Zum Vergleich enthält die letzte Spalte die Zeit für einen normalen Funktionsaufruf.
Wie leicht zu erkennen ist, benötigt das Installieren einer Funktion ein Vielfaches der Zeit eines gewöhnlichen Funktionsaufrufs. Betrachtet man die Zeit genauer, so nimmt die Übertragung mit Abstand die meiste Zeit in Anspruch. Die anderen Operationen benötigen beim AVR-Mikrocontroller etwas mehr Zeit als beim H8/300. Dies liegt daran, dass hier zwei Adressräume (Programm- und Datenspeicher) verwaltet werden und dass der Code schließlich in den Flashspeicher geschrieben werden muss.

Die Kommunikationsdauer wird durch verschiedene Parameter beeinflusst. Zunächst sind die Eigenschaften der Kommunikationshardware zu nennen: Übertragungsrate und Latenz. Da diese Parameter jedoch nicht im Einflussbereich des Verwalters liegen, sind stattdessen die Menge der zu übertragenden Daten und die Anzahl der Kommunikationsvorgänge für uns relevant. Im Allgemeinen gilt, je mehr und je öfter Daten übertragen werden, desto teurer ist die Operation.

Neben der benötigten Zeit ist auch zu beachten, dass Kommunikation im Allgemeinen relativ viel Energie verbraucht[2]. Das spielt besonders bei kleinen batteriebetriebenen Geräten eine Rolle. Neben der Möglichkeit die Kommunikationshardware abzuschalten, wenn sie nicht benötigt wird, gilt auch hier der Grundsatz, dass je weniger Daten zu übertragen sind, desto weniger Energie aufgewendet werden muss.

5.1.1.1 Kostenbeinflussende Faktoren bei dynamischer Umkonfiguration

Die Anzahl der Übertragungen bei einer dynamischen Veränderung der Konfiguration kann man durch die Wahl des Kommunikationsprotokolls beeinflussen. Im letzten Kapitel wurde die Schnittstelle zum Zugriff auf den verwalteten Knoten vorgestellt. Das dabei vorgestellte Protokoll überträgt einzelne Veränderungen. Um die Anzahl der Kommunikationsvorgänge zu reduzieren, kann man, anstatt alle Änderungen einzeln zu übertragen, ein Protokoll einsetzen, das alle Veränderungen in einer Nachricht kombiniert. Dazu ist vorgesehen, eine Umkonfiguration zunächst nur auf dem Speicherabbild am Verwalter durchzuführen. Anschließend können die Veränderungen durch einen Vergleich des Speicherabbildes vor und nach der Umkonfiguration bestimmt werden. Die Unterschiede können

[2] In [SHC+04] wurde die Stromaufnahme eines Sensorknotens vom Typ Mica2 vermessen und einzelnen Komponenten zugeordnet. Die Mica2 Knoten werden mit 3 V aus 2 Batterien (ca. 2200 mAh) versorgt und besitzen einen CC1000 Funkchip, der Daten mit bis zu 38400 kbps versenden und empfangen kann. Der Stromverbrauch zum Empfangen ist mit 7,03 mA bestimmt worden, der Stromverbrauch zum Senden liegt zwischen 3,72 mA (bei -20 dBm) bis 21,48 mA (bei +10 dBm) je nach eingestellter Übertragungsleistung. Im Vergleich dazu benötigt die CPU unter Last 8,0 mA und 3,2 mA im Ruhezustand.

dabei durch Änderungskommandos, wie sie durch übliche Vergleichsalgorithmen erzeugt werden (zum Beispiel UNIX-diff [HM76] oder Xdelta [Mac00]) ausgedrückt und dann in einer Nachricht zum verwalteten Knoten übertragen werden. Der verwalteten Knoten muss dann allerdings in der Lage sein, die Änderungskommandos auszuführen.

Die Datenmenge muss für die einzelnen Operationen getrennt betrachtet werden. Beim bedarfsgesteuerten Installieren müssen neben der aufgerufenen Funktion auch ihre Abhängigkeiten bereitgestellt werden. Hierbei besteht die Wahl, abhängige Funktionen zu installieren oder lediglich Stellvertreter zu verwenden. Durch den Einsatz von Stellvertretern wird die Datenmenge reduziert, allerdings ist dann möglicherweise ein erneutes Nachinstallieren notwendig, wodurch eine weitere Unterbrechung der Ausführung stattfindet und die Anzahl an Kommunikationsvorgängen insgesamt erhöht wird.

Beim Parken von Modulen muss der Zustand von veränderlichen Datenmodulen beim Verwalter gesichert werden. Dabei sind im Allgemeinen weniger Daten zu übertragen als beim Installieren, da Code und konstante Datenmodule nicht gesichert werden müssen.

Beim Ersetzen oder Aktualisieren der Module setzt sich die zu übertragende Datenmenge aus der Größe des alten Zustands zuzüglich den Daten der neuen Module zusammen. Neben der Datenmenge spielt hierbei aber auch die Verknüpfung des Codes eine Rolle. Wird eine Funktion beispielsweise nicht an der Adresse installiert, an der ihr Stellvertreter war und wird auch keine Weiterleitung verwendet, so müssen alle Verweise angepasst werden. Je nach eingesetztem Kommunikationsprotokoll kann das zu einer Reihe zusätzlicher Kommunikationsvorgängen führen. Das Installieren einer stark verknüpften Funktion wird daher teurer sein als der Austausch einer kaum referenzierten Funktion. Besonders stark schlägt sich das nieder, wenn auf dem verwalteten Gerät nur blockweise auf den Programmspeicher geschrieben werden kann, wie das beispielsweise bei AVR-Prozessoren der Fall ist. Bei starker Verknüpfung müssen viele Blöcke verändert werden.

5.1.1.2 Kostenbeinflussende Faktoren bei einem Fernaufruf

Bei einem Fernaufruf müssen für jeden Aufruf die Parameter übertragen werden. Die Datenmenge hängt dabei von der Anzahl und der Größe der Parameter ab, die direkt für den Aufruf mitgegeben werden. Außerdem trägt der Rückgabewert zur Datenmenge bei.

Wie in Abschnitt 4.6.4 erläutert wurde, müssen neben den expliziten Parametern auch Daten übertragen werden, um am Zielknoten eine Ausführungsumgebung aufzubauen. Daher könnte man alle Datenstrukturen, die zur Ausführung der Funktion benötigt werden, als implizite Parameter bezeichnen. Bei diesen zusätzlichen Daten ist zu unterscheiden, ob es sich um veränderbare oder um konstante Daten handelt. Konstante Daten müssen nicht übertragen werden, da am Verwalterknoten eine Kopie verwendet werden kann.

Bei den veränderbaren Daten muss man zwischen Daten unterscheiden, die ausschließlich von der Funktion benötigt werden und Daten, welche mit anderen Funktionen gemeinsam genutzt werden. In beiden Fällen müssen sie zur Ausführung der Funktion am Verwalter vorhanden sein. Allerdings müssen Daten, welche ausschließlich von der Funktion genutzt werden, nicht bei jedem Aufruf aktualisiert und bei Veränderung wieder zum Gerät zurückgeschrieben werden.

Unter Berücksichtigung dieses Umstands kann man die Datenmenge, die bei jedem Aufruf übertragen werden muss, reduzieren, indem man Funktionen, welche auf gemeinsamen Daten arbeiten, gemeinsam auslagert. Dadurch wird der Anteil der Daten, die an verschiedenen Orten benötigt werden, reduziert und die Kosten für das Erstellen der Ausführungsumgebung gesenkt. Das Auslagern einer weiteren Funktion kann somit die Kosten reduzieren.

Eine ausgelagerte Funktion kann natürlich auch weitere Funktionen aufrufen. Hierbei ist die Entscheidung zu treffen, an welchem Ort diese Funktionen ausgeführt werden sollen. Ist die Zielfunktion

nicht von Ressourcen abhängig, die ausschließlich auf dem verwalteten Gerät vorhanden sind, kann die Funktion entweder auf dem Verwalter oder mithilfe eines Rückrufs auf dem Gerät ausgeführt werden. Da ein Rückruf wieder einen Fernaufruf darstellt, gelten hier auch die genannten Kostenfaktoren. Insbesondere müssen alle Daten am Gerät aktualisiert werden.

Ein weiterer wichtiger Kostenfaktor ist die Häufigkeit der Aufrufe. Eine Funktion, die nur selten aufgerufen wird, eignet sich am Besten für eine Auslagerung, da die Kosten eines Aufrufs nur selten anfallen. Wird eine Funktion hingegen sehr häufig aufgerufen, so fallen die Aufrufkosten stark ins Gewicht. Die gesamte Ausführungszeit kann dadurch stark beeinflusst werden.

5.1.2 Metriken zur Abschätzung der kostenbeeinflussenden Faktoren

Die bisher beschriebenen kostenbeeinflussenden Faktoren können teilweise aus den Daten der Basisschicht ermittelt werden. Daraus lässt sich eine Kostenabschätzung ableiten, die zur Strategiefindung beitragen kann.

- Die Datenmenge beim Installieren, Parken oder Ersetzen von Modulen lässt sich anhand der Modulgrößen ermitteln. Zusätzlich kann anhand des Abhängigkeitsgraphen im Vorfeld festgestellt werden, welche Module genau betroffen sind. So kann beispielsweise bestimmt werden, wie viel das Installieren aller abhängigen Funktionen zusätzlich kostet. Auf dieser Basis kann entschieden werden, ob diese gleich mitinstalliert werden sollen.

- Die Verknüpfung der Funktion innerhalb der Anwendung kann anhand des Abhängigkeitsgraphen ermittelt werden. Daraus können die Kosten für das Anpassen der Verweise abgeschätzt werden.

- Die Menge der zu übertragenden Daten bei einem Aufruf einer ausgelagerten Funktion kann ebenfalls im Vorfeld abgeschätzt werden. Zur Bestimmung der direkten Parameter wird auf die Schnittstellenbeschreibung zurückgegriffen. Hieraus kann der Typ und daraus die Größe der Parameter und des Rückgabewerts ermittelt werden. Bei manchen Datentypen ist die Größe jedoch nicht konstant. Bei Feldern beispielsweise kann die Länge durch einen zusätzlichen Parameter spezifiziert sein, der erst zur Laufzeit ausgewertet werden kann.
  Die impliziten Parameter werden durch den Abhängigkeitsgraphen bestimmt. Anschließend kann mithilfe der geplanten Konfiguration ermittelt werden, welche Datenmodule gemeinsam mit anderen Funktionen auf dem Gerät benutzt werden.

- Anhand des Abhängigkeitsgraphen lässt sich die Beziehung zwischen den Funktionen ermitteln. Man kann feststellen, welche Funktionen ein Datenmodul referenzieren. Beim Auslagern einer Funktion kann man hieran prüfen, ob das Auslagern zusätzlicher Funktionen den Aufwand bei einem Aufruf reduziert.

- Die Häufigkeit eines Aufrufs lässt sich nicht statisch bestimmen. Ein Anhaltspunkt für die Wichtigkeit und damit der Aufrufwahrscheinlichkeit kann die Einbindung der Funktion in die Software sein. Das heißt: Wird die Funktion von vielen Stellen innerhalb der Anwendung referenziert, so kann das als Hinweis betrachtet werden, dass sie auch oft aufgerufen wird. Allerdings ist dieser Zusammenhang nicht zwingend gegeben.
  Ein anderes Herangehen ist das Ermitteln eines Nutzungsprofils zur Laufzeit. Durch Instrumentieren des Codes können die tatsächlichen Aufrufe gezählt werden. Die Instrumentierung kann entweder als Konfigurationsoperation durch die Kontrollschicht eingefügt und ausgewertet werden oder man führt sie bei einem separaten Testlauf durch und verwendet die Ergebnisse als

externe Datenquelle. Beachtet werden sollte allerdings, dass eine Instrumentierung zusätzliche Laufzeitkosten verursacht.

5.1.3 Kosten – Nutzen Abwägung

Der Grund, einen Knoten durch einen Verwalter zu betreuen, ist in erster Linie die Flexibilität, die man erhält, indem man die Möglichkeiten des Verwalters nutzt. Um dabei zu einer Konfigurationsentscheidung zu gelangen, müssen die Kosten gegenüber dem Nutzen abgewogen werden. Betrachtet man beispielsweise den Energieverbrauch, so wird durch die Nutzung externer Dienste nicht nur Energie verbraucht, man kann dadurch auch Energie einsparen, indem man Aufgaben verschiebt, die besonders viel Energie benötigen, weil sie beispielsweise sehr rechenintensiv sind oder viele Ein-/Ausgabeoperationen durchführen müssen [RRPK98, OH98, PLDM02].

Es kann auch Aufgaben geben, die ein Gerät ohne Unterstützung durch den Verwalter gar nicht durchführen kann. Ein solcher Fall liegt vor, wenn nicht genügend Ressourcen, wie beispielsweise Rechenleistung, zur Verfügung stehen. Muss die Aufgabe unbedingt durchgeführt werden, so sind die Kosten für die Nutzung eines entfernten Dienstes nebensächlich.

Häufig ist auch die Wahl zwischen verschiedenen Alternativen möglich. Ist es beispielsweise unumgänglich einen Teil des Codes vom verwalteten Gerät zu entfernen, da der Speicherplatz benötigt wird, so ist zu entscheiden, welche Funktion entfernt wird und wie verfahren werden soll, wenn die Funktion später wieder benötigt wird.

Die Auswahl der Funktion kann beispielsweise anhand der Kosten erfolgen, die im vorherigen Abschnitt aufgezeigt wurden. Dabei darf man das Entfernen nicht alleingestellt betrachten, sondern muss auch die Kosten zur späteren Nutzung berücksichtigen. Hierzu stehen zwei Möglichkeiten zu Verfügung: Die Funktion kann geparkt und bei Bedarf wieder installiert werden oder sie kann als externer Dienst angeboten werden.

Der Vorteil des Parkens ist, dass der Vorgang relativ einfach ist. Im Gegensatz zur entfernten Ausführung wird beispielsweise keine Version der Module für die Architektur des Verwalters benötigt. Bei der späteren Reintegration verursacht allerdings nicht nur das Installieren Kosten, sondern auch das Bereitstellen des benötigten Speicherplatzes vor der Installation, da hierzu möglicherweise erst andere Module entfernt werden müssen.

Wird die Funktion hingegen als entfernter Dienst angeboten, so kann die Funktion ohne Installation sofort genutzt werden. Allerdings fallen bei jeder Nutzung Kosten an, da zumindest die Parameter übertragen werden müssen.

Bei der Entscheidung ist somit abzuwägen, wie oft die Funktion benötigt wird und wie viele Parameter sie benötigt. Bei einer unregelmäßig und selten genutzten Funktion mit wenigen Parametern eignet sich das Auslagern als externer Dienst, da die Funktion schnell zur Verfügung steht. Bei einer phasenweisen intensiven Nutzung, beispielsweise bei Initialisierungs- und Kalibrierungscode, bietet sich das Parken an. Die einmaligen Mehrkosten des Installierens können hier günstiger sein als die Kosten bei jedem Aufruf.

Zusammenfassung

In diesem Abschnitt wurden die Kosten der durch die Basisschicht angebotenen Mechanismen angesprochen und ihr Einfluss auf die Konfigurationsstrategie. Zur Unterstützung der Strategiefindung wurden einige Möglichkeiten aufgezeigt, die Kosten der Mechanismen im Vorfeld einzuschätzen. Dabei können bereits die statisch vorhandenen Daten gute Anhaltspunkte liefern. Eine Erweiterung um Laufzeitanalysen kann diese Daten ergänzen.

5.2 Arten von Strategien

Im Folgenden sollen zwei verschiedene Arten einer Kontrollschicht vorgestellt werden. Sie verfolgen zwei unterschiedliche Konfigurationskonzepte. Zum einen eine manuelle, interaktiv unterstützte Konfiguration und zum anderen eine automatische Konfiguration.

5.2.1 Manuelle, interaktive Konfiguration

Bei diesem Ansatz enthält die Kontrollschicht keine oder nur wenig Wissen über die verwaltete Software. Stattdessen wird eine Benutzeroberfläche zur Verfügung gestellt, welche es einem Administrator erlaubt, die Konfigurationsmechanismen anzuwenden. Abbildung 5.1 zeigt die Benutzeroberfläche einer Kontrollschicht, die im Rahmen dieser Arbeit erstellt wurde.

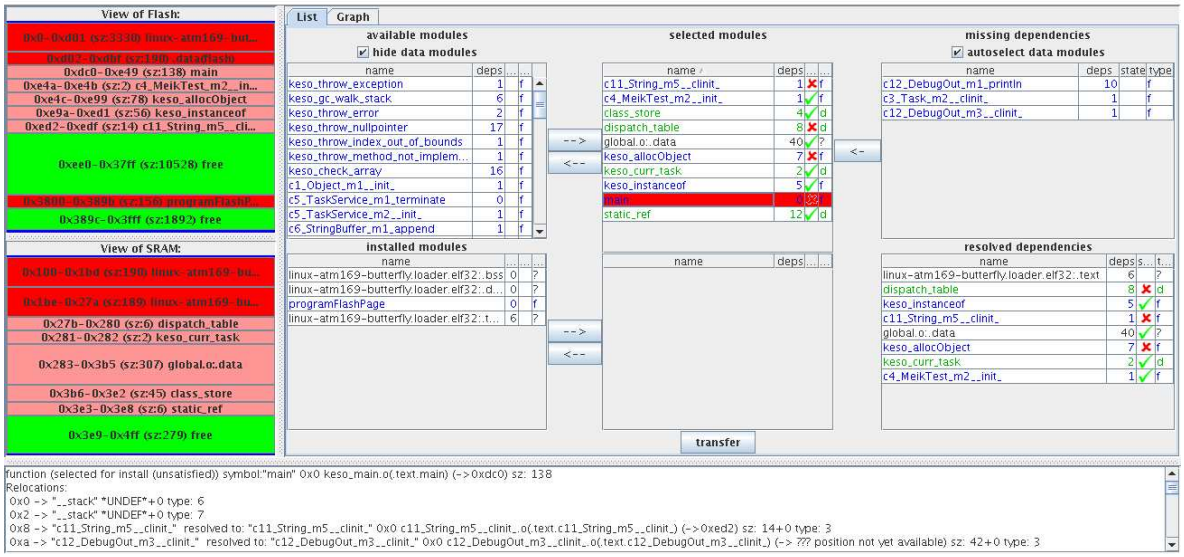


Abbildung 5.1: Benutzeroberfläche einer interaktiven Kontrollschicht

Links wird dem Administrator die aktuelle Speicherbelegung dargestellt. Rechts daneben sind die verfügbaren Module (oben) und die bereits installierten Module (unten) aufgelistet. Diese können zur Installation oder zum Entfernen ausgewählt werden. Zu einem ausgewählten Modul wird ganz rechts angezeigt, welche weiteren Module benötigt werden. Für benötigte Module können auf Wunsch Stellvertreter installiert werden. Ist eine gültige Installation erstellt, so können die Veränderungen übertragen und angewandt werden.

Dem Administrator werden alle verfügbaren Module angezeigt. Gleichzeitig sieht er die Speicherbelegung des Geräts und eine Liste der installierten Module. Der Administrator hat nun die Möglichkeit, neue Module auf dem Gerät zu installieren und zu positionieren. Er erstellt dazu eine neue Konfiguration, indem er Module auswählt. Die Kontrollschicht führt nach jeder Auswahl eine Konsistenzprüfung durch und präsentiert die Ergebnisse in der Oberfläche. Auf diese Weise bekommt der Administrator sofort eine Rückmeldung, was seine Änderung bewirkt und kann schnell erfassen, wenn die neue Konfiguration noch nicht umsetzbar ist, weil beispielsweise eine Abhängigkeit nicht erfüllt ist. Der Administrator hat hierbei auch die Möglichkeit auszuwählen, ob ein Modul erst bei Bedarf installiert oder die Funktion auf dem Verwalterknoten ausgeführt werden soll.

Weiterhin kann er auch Module vom Gerät entfernen. Dabei überprüft die Kontrollschicht, ob das Modul gerade aktiv ist und quittiert das Vorhaben gegebenenfalls mit einer Fehlermeldung. Ist das

Entfernen erlaubt, so kann der Administrator wählen, welche Art von Stellvertreter das Modul ersetzen soll.

Hat der Administrator eine gültige Konfiguration erstellt, kann er sie anwenden, woraufhin die Daten zum Gerät übertragen und in den Speicher geschrieben werden. Als nächstes hat er die Möglichkeit, den Programmablauf auf dem verwalteten Gerät an einer bestimmten Stelle zu starten oder fortzusetzen. Sind während der Bearbeitung einer Unterstützungsanfrage Entscheidungen zu treffen, muss dies wiederum vom Administrator durchgeführt werden. Reicht beispielsweise der Speicherplatz im Rahmen einer bedarfsgesteuerten Installation nicht aus, so wird der Eingriff eines Administrators benötigt.

5.2.2 Automatische, anwendungsbezogene Konfiguration

Bei der automatischen Konfiguration ist die Konfigurationsumgebung auf genau eine Laufzeitumgebung zugeschnitten. Sie kennt die Wirkungsweise der einzelnen Bestandteile und von möglichen Varianten. Daraus kann sie eine optimale Konfiguration für die vorliegende Anwendung erstellen. Der grundlegende Ablauf ist in Abbildung 5.2 dargestellt.

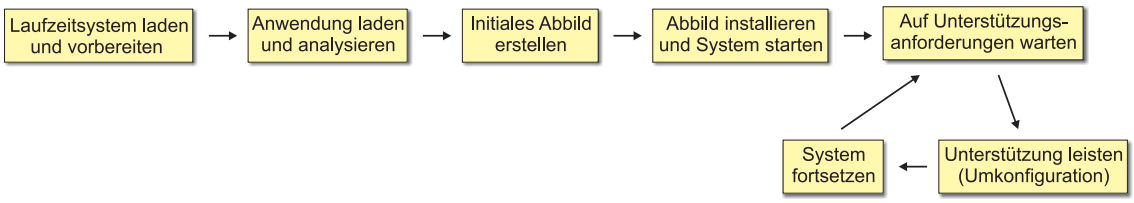


Abbildung 5.2: Grundlegender Ablauf einer automatischen Konfiguration

Zunächst wird die Kontrollschicht initialisiert, indem sie die Objektdateien des Laufzeitsystems einliest. Gibt es von einigen Komponenten verschiedene Versionen, die später zur Laufzeit dynamisch ausgetauscht werden können, so werden diese ebenfalls geladen. Dabei muss die Kontrollschicht die verschiedenen Versionen erkennen und ihnen Bedingungen zuordnen, unter denen sie eingesetzt werden sollen. Um das entfernte Ausführen auf verschiedenen Architekturen zu unterstützen, muss außerdem von den betroffenen Komponenten Code für diese Architekturen geladen werden. Beim Laden der Objektdateien werden sie durch die Basisschicht modularisiert.

Nachdem die Kontrollschicht initialisiert wurde, werden die Objektdateien der Anwendung geladen. Der nächste Schritt ist die Erstellung des initialen Abbildes für den oder die Knoten. Dazu werden die Module in einen Behälter geladen und miteinander verknüpft. Hierbei ergeben sich Abhängigkeiten zu Modulen des Laufzeitsystems. Daraus kann die Kontrollschicht ermitteln, welche Dienste von der Anwendung genutzt werden.

Die Kontrollschicht muss in diesem Schritt entscheiden, welche Module verwendet werden und welche durch Stellvertreter ersetzt werden. Für Komponenten, die in verschiedenen Versionen vorliegen, muss die Kontrollschicht auch entscheiden, welche Version sich für die aktuelle Anwendung am besten eignet. Da Varianten, die für bestimmte Spezialfälle optimiert sind, meistens nicht außerhalb dieses Spezialfalles eingesetzt werden können, muss die Kontrollschicht an den Codestellen, an denen sich die Voraussetzungen für den Einsatz ändern, eine Unterstützungsaufforderung in das Abbild einfügen. Wird diese Codestelle während der Ausführung erreicht, kann die Konfiguration dynamisch angepasst werden und der Verwalter kann die Spezialvariante gegen eine allgemeine Variante austauschen.

Ein Beispiel hierfür ist eine Variante der Threadverwaltung, welche für eine feste Anzahl an Threads optimiert ist. Die Metainformationen zu den Aktivitätsträgern können dann in einem Feld fester Länge abgelegt und müssen nicht in einer dynamisch wachsenden Liste organisiert werden. Vorteil dieser

Variante ist, dass das Auffinden der Metainformationen zu einem bestimmten Aktivitätsträger sehr schnell durchgeführt werden kann, da die Informationen an einer festen Stelle abgespeichert sind. Sobald die Anzahl an Threads jedoch den festgelegten Maximalwert überschreitet, müsste das Feld zur Aufnahme der Metainformationen vergrößert werden. Das ist jedoch nicht immer durchführbar, da nicht sichergestellt werden kann, dass genügend Speicherplatz an der Stelle vorhanden ist. Daher muss die Kontrollschicht über die Erzeugung eines neuen Threads informiert werden, sodass sie gegebenenfalls die Threadverwaltung durch eine andere Variante ersetzen kann.

Nachdem das initiale Abbild erstellt wurde, kann es auf den Knoten geladen werden. Die Kontrollschicht ist nun für die Laufzeitunterstützung verantwortlich. Je nach Art der Unterstützungsanforderung müssen Module nachinstalliert, ersetzt oder geparkt werden.

5.3 Kontroll- und Verwaltungsschicht für eine Java-Laufzeitumgebung

Als Anwendungsbeispiel sollen nun die Möglichkeiten einer Kontrollschicht anhand einer Java-Laufzeitumgebung für eingebettete Systeme betrachtet werden.

5.3.1 Motivation

Typische Netze aus eingebetteten Knoten bestehen meist aus unterschiedlichen Knoten. Um die Softwareentwicklung zu vereinheitlichen, möchte man teilweise die Heterogenität der Knoten abstrahieren. Eine virtuelle Maschine kann hier eine Abstraktionsschicht bilden und die verschiedenen Fähigkeiten der Knoten zu einem bestimmten Grad vereinheitlicht darstellen. Ein Programm wird dann für die virtuelle Maschine entwickelt. Eine Anpassungsschicht, die nur aus einem Compiler bestehen kann, meist jedoch ein Laufzeitsystem beinhaltet, sorgt für die Ausführung des Programms und leistet Laufzeitunterstützung. Je umfangreicher eine virtuelle Maschine ist, das heißt je mehr Dienste sie anbietet, desto allgemeiner, komfortabler und abstrakter kann die Umgebung sein, für die ein Programm entwickelt wird. Allerdings nimmt dann auch das Laufzeitsystem mehr Platz in Anspruch. Hier soll ein Verwalter eingreifen und kleine Knoten bei der Ausführung eines großen Laufzeitsystems unterstützen.

5.3.2 Java-Laufzeitumgebungen für eingebettete Systeme

Ein bekanntes Beispiel einer virtuellen Maschine stellt die Java-Maschine (*Java Virtual Machine*, *JVM*) dar. Sie ist für Bürocomputer entwickelt worden und durch die Vielzahl von Diensten für kleine Knoten auf den ersten Blick ungeeignet. Betrachtet man jedoch verschiedene Java-Anwendungen, so stellt man fest, dass nicht jede Anwendung den vollen Umfang an Diensten benötigt. Besonders im Bereich der eingebetteten Systeme werden die Anwendungen meist im Hinblick auf die begrenzten Ressourcen entwickelt, sodass ressourcenintensive Dienste nicht eingesetzt werden. Dieser Umstand ermöglicht es, eine Java-Umgebung für kleine Systeme anzubieten. Im Folgenden sollen zwei Techniken dazu vorgestellt werden.

5.3.2.1 Allgemeine Reduktion des Funktionsumfangs

Eine Möglichkeit ist, eine eingeschränkte Version der JVM anzubieten, die nicht alle Dienste und Funktionen einer normalen Java-Umgebung bereitstellt. Dabei sind zwei Arten von Einschränkungen möglich. Zum einen kann die Klassenbibliothek eingeschränkt werden, zum anderen kann die virtuelle Maschine selbst eingeschränkt sein.

Diesen Ansatz verfolgt SUN mit der *Java 2 Micro Edition* (*J2ME*), eine Java-Umgebung für kleine tragbare Geräte. Basis dabei bildet die sogenannte Konfiguration, welche die Fähigkeiten der virtuellen Maschine und den Umfang der Klassenbibliothek beschreibt. Darauf können dann verschiedene Profile genutzt werden, wobei Profile weitere Schnittstellen und Bibliotheken für die Anwendungen definieren.

Aktuell werden zwei Konfigurationen definiert. Die erste Konfiguration wird *CDC* (*Connected Device Configuration*) genannt [Sun02]. Sie basiert auf einer vollständigen JVM und unterscheidet sich hauptsächlich durch eine eingeschränkte Klassenbibliothek. Zielplattformen dieser Umgebung sind relativ leistungsstarke mobile Geräte wie zum Beispiel PDAs.

Für kleinere Geräte wie beispielsweise Mobiltelefone wurde die *CLDC*-Umgebung (*Connected Limited Device Configuration*) definiert [Sun03a]. Neben einer noch stärker eingeschränkten Klassenbibliothek wurde hier auch die zugrunde liegende Maschine verkleinert. Fehlerbehandlung, Threadmodell und das dynamische Nachladen von Klassen wurden eingeschränkt; die Möglichkeit Typen erst zur Laufzeit zu untersuchen, wird gar nicht angeboten.

Noch einen Schritt weiter geht Sun mit der Definition von *JavaCard* [Sun03b], eine JVM für Chipkarten. Die Java-Maschine, die hierbei definiert wird, ist noch stärker eingeschränkt. Hier ist es nicht mehr möglich Threads zu definieren oder dynamisch Klassen nachzuladen. Neben Diensten wurde auch der verarbeitete Bytecode eingeschränkt. So ist die Verarbeitung von Fließkommaoperationen nicht mehr vorgesehen. Tabelle 5.2 zeigt Größeninformationen der Zielplattform der drei Java-Umgebungen.

	Geräteklasse	CPU		RAM	(EP)ROM
CDC	PDAs	32-bit	100 MHz	2 MByte	2,5 MByte ROM
CLDC	Mobiltelefone	16/32-bit	16 MHz	192 KByte	160 KByte ROM/Flash
JavaCard	Chipkarten	8/16/32-bit	2 MHz	1,2 KByte	32 KByte ROM 16 KByte EEPROM/Flash

Tabelle 5.2: Zielplattformen für J2ME
Die Tabelle zeigt die Größen- und Leistungsanforderungen an eine Zielplattform für die verschiedenen J2ME Varianten.

J2ME verfolgt den Ansatz, eine geeignete Untermenge der vollständigen Java-Laufzeitumgebung anzubieten, welche für den Bereich der eingebetteten Systeme relevant ist. Nachteil dabei ist, dass man die Kompatibilität zwischen den verschiedenen Versionen verliert. So ist nicht gewährleistet, dass Anwendungen, welche für eine vollständige Java-Umgebung entwickelt wurden, auch auf diesen eingeschränkten Umgebungen laufen.

5.3.2.2 Anwendungsspezifische Anpassung

Statt die angebotenen Dienste einer virtuellen Maschine von vornherein einzuschränken, kann man auch eine Laufzeitumgebung bereitstellen, welche exakt die Dienste anbietet, die von der Anwendung benötigt werden. Man geht von einer vollständigen JVM aus und passt sie dem tatsächlichen Bedarf des Programms an. Die Anwendung kann ohne Berücksichtigung von Einschränkungen, gegen das vollständige Maschinenmodell entwickelt werden.

Die Anpassung erfolgt aufgrund einer Analyse des fertigen Programms. Dabei ist es möglich, den Befehlssatz auf die von der Anwendung verwendeten Befehle einzuschränken. Außerdem kann man Unterstützungsdienste, welche nicht in Anspruch genommen werden, weglassen.

Ein System auf Basis dieses Ansatzes ist SUNs ursprüngliches *EmbeddedJava*. Es wurde jedoch durch J2ME ersetzt und wird heute nicht mehr von SUN unterstützt[3]. Die Anwendung wird hierbei gegen das JDK 1.1.7 entwickelt. Anschließend wird analysiert, welche Klassen und Felder von der Anwendung benutzt werden (*JavaFilter*), um daraufhin die Klassenbibliothek auf den benötigten Umfang zu reduzieren (*JavaCodeCompact*). Dabei werden die Klassen vorab geladen, das heißt, die Datenstrukturen zur Verwaltung der Klassen werden erstellt und in C-Code ausgegeben. Anschließend können die Dateien übersetzt und zusammen mit Unterstützungsbibliotheken und einem Betriebssystem zu einem Speicherabbild für den Mikrocontroller gebunden werden.

Ein anderes Beispiel ist der VM-Baukasten VM* [KP05b]. Hier liegt die JVM in Komponenten vor, aus denen, nach der Analyse der Anwendung, eine optimale Auswahl getroffen wird.

Als letztes Beispiel ist KESO zu nennen [WSSP07]. Bei KESO wird der Java-Bytecode in C-Code übersetzt und dieser dann mit einer konfigurierbaren Betriebssystembibliothek (zum Beispiel OSEK) zusammengebunden. Im Rahmen des Übersetzens wird die Anwendung analysiert und anwendungsspezifisch um Code erweitert, der die Eigenschaften der JVM herstellt. So werden beispielsweise Typprüfungen und Überprüfungen von Feldgrenzen direkt in den entstehenden Code eingefügt. Außerdem werden die Verwaltungsstrukturen anwendungsspezifisch erzeugt, wie beispielsweise eine optimale Repräsentation der Typhierarchie.

Vorteil des anwendungsspezifischen Anpassens ist, dass keine unnötigen Dienste angeboten werden. Durch die Optimierung kann Speicherplatz eingespart werden, ohne die Anwendung zu beschränken. Zu beachten ist allerdings, dass bei diesem Ansatz die dynamischen Eigenschaften der JVM verloren gehen. Im Speziellen wird das dynamische Nachladen von Klassen nicht unterstützt, da neue Klassen Dienste benötigen könnten, die durch die anwendungsspezifische Anpassung entfernt wurden. Für Anwendungen, die auf solche dynamischen Eigenschaften angewiesen sind, kann somit dieser Ansatz nicht eingesetzt werden.

5.3.3 Dynamisch anwendungsspezifisch angepasste Laufzeitumgebung

Die beiden vorgestellten Ansätze, eine Java-VM auf einem kleinen Knoten zur Verfügung zu stellen, haben beide den Nachteil, dass die angebotene Funktionalität eingeschränkt und von Anfang an festgelegt ist. Mithilfe eines Verwalters kann man auf die vorgestellten Ansätze aufbauen, ihre Vorteile nutzen und gleichzeitig durch dynamische Adaption den vollen Funktionsumfang einer JVM zur Verfügung stellen. Man verfolgt denselben grundsätzlichen Ansatz einer anwendungsspezifischen Laufzeitumgebung, ist jedoch nicht auf die einmalige Anpassung beschränkt, sondern kann sie wiederholen, falls es zur Laufzeit notwendig wird. Man erhält eine dynamisch der Anwendung angepasste JVM, die Dienste nachträglich in die Laufzeitumgebung integriert, sobald sie benötigt werden. Zusätzlich kann man optimierte Varianten der Dienste verwenden, die dynamisch gewechselt werden, sobald sie nicht mehr den Anforderungen genügen. Die eingesparten Ressourcen können der Anwendung zur Verfügung gestellt werden.

Da das Hinzufügen von Diensten und das Ersetzen von speziellen Varianten durch allgemeinere zu einem höheren Speicherbedarf führt, kann ein Punkt erreicht werden, an dem der Speicher nicht mehr ausreicht, um alle benötigten Dienste gleichzeitig lokal zur Verfügung zu stellen. Dann kann der Verwalter Komponenten parken oder als entfernten Dienst anbieten.

[3] Die Bezeichnung *EmbeddedJava* wird heute von SUN für andere Produkte verwendet. Informationen über das ursprüngliche *EmbeddedJava* finden sich kaum noch. Eine Übersichtsseite ist jedoch noch im Internetarchiv "Wayback Machine" enthalten: `http://web.archive.org/web/20040813044332/http://java.sun.com/products/embeddedjava/overview.html`

5.4 Eine JVM als Baukasten

In diesem Abschnitt soll eine Aufteilung einer JVM in Bausteine erfolgen. Dabei sollen die Abhängigkeiten der Bestandteile verdeutlicht werden, um geeignete Bausteine zu finden, die als entfernter Dienst angeboten werden können. Die Aufteilung erfolgt hauptsächlich in funktionale Einheiten, das heißt, jeder Baustein erfüllt eine bestimmte Funktion oder bietet einen Dienst an. Neben den funktionalen Bausteinen werden aber auch einige Datenstrukturen als Bausteine eingeführt. Dabei handelt es sich hauptsächlich um Klassendaten, welche die Anwendung beschreiben. Diese Daten werden von verschiedenen anderen funktionalen Bausteinen verarbeitet und ausgewertet und stellen somit eine wichtige Abhängigkeit dar. Abbildung 5.3 zeigt eine grobe Übersicht über die Bausteine einer JVM und die Abhängigkeiten zwischen diesen. In den folgenden Abschnitten über die einzelnen Bausteine ist jeweils eine detailliertere Abbildung gegeben. Funktionale Bausteine sind dabei gelb hinterlegt, reine Daten werden durch einen weißen Hintergrund gekennzeichnet.

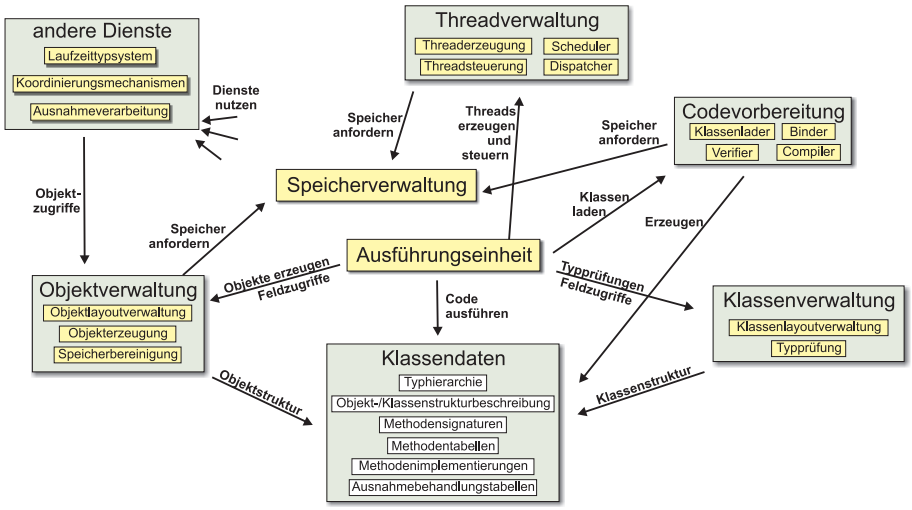


Abbildung 5.3: Bausteine einer JVM

Die Abbildung zeigt eine grobe und vereinfachte Übersicht der Bausteine einer JVM. Detailliertere Abbildungen finden sich in den folgenden Abschnitten.

Weiterhin sollen Varianten der einzelnen Dienste und Bausteine betrachtet werden. Dabei werden die Vor- und Nachteile der verschiedenen Versionen hervorgehoben und Szenarios beschrieben, in denen die Varianten sinnvoll eingesetzt werden können. Im Zusammenhang damit werden auch die Grenzen der Einsetzbarkeit einer Variante und die Ereignisse aufgezeigt, die dazu führen, dass eine bestimmte Variante nicht mehr genutzt werden kann.

5.4.1 Ausführungseinheit

Die zentrale Aufgabe einer virtuellen Maschine ist das Ausführen von Anwendungen. Als Ausführungseinheit bezeichnen wir in unserem Baukasten die Einheit, die für das Ausführen des Anwendungscodes zuständig ist. Prinzipiell existieren jedoch zwei konzeptionell unterschiedliche Ansätze, eine Anwendung für eine virtuelle Maschine auf einer echten Maschine auszuführen: die Interpretation und das Kompilieren.

Der traditionelle Ansatz ist die Interpretation des Bytecodes. Hierbei wird der Bytecode zur Laufzeit analysiert und daraufhin der Zustand der virtuellen Maschine entsprechend den Operationen im

Bytecode verändert. Die Software, die diesen Vorgang durchführt, wird *Interpreter* genannt und stellt bei diesem Ansatz die Ausführungseinheit dar.

Der größte Nachteil dieses Ansatzes ist, dass die Interpretation relativ viel Zeit und Energie in Anspruch nimmt. Um die Ausführungsgeschwindigkeit zu steigern, ist man dazu übergegangen, den Java-Bytecode in Maschinencode zu übersetzen. Dabei entsteht architekturabhängiger Code, der direkt von der CPU auf der Zielplattform ausgeführt werden kann[4]. Zum Kompilieren wird allerdings ein spezieller Compiler für die Zielarchitektur benötigt; die Portierung eines Interpreters auf eine neue Plattform ist allerdings oft weniger aufwendig als das Erstellen eines Compilers.

Da die kompilierte Anwendung direkt von der CPU der Zielplattform ausgeführt werden kann, wird kein Softwarebaustein benötigt, der die Ausführung steuert. Alle benötigten Anweisungen werden im Rahmen des Übersetzungsvorganges vom Compiler in den Code eingearbeitet.

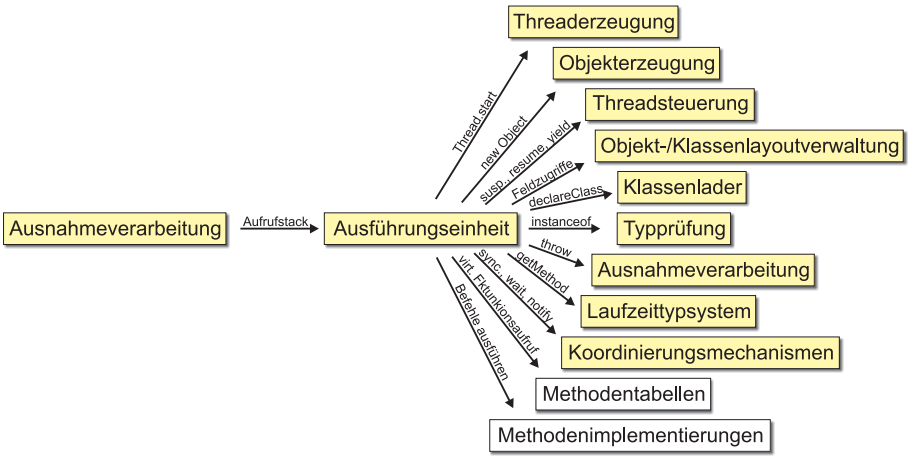


Abbildung 5.4: Abhängigkeiten der Ausführungseinheit

Bei Verwendung eines Interpreters können Ressourcen eingespart werden, indem ein anwendungsspezifischer Interpreter eingesetzt wird, der nur den Teil des Java-Maschinen-Befehlssatzes unterstützt, der in der Anwendung verwendet wird. Werden von der Anwendung beispielsweise keine Fließkommaoperationen durchgeführt, so muss der Interpreter keine Befehle, die auf Fließkommazahlen arbeiten, unterstützen. Beim Hinzufügen neuer Klassen zum System muss der Verwalter benachrichtigt werden, da der Interpreter dann möglicherweise ausgetauscht werden muss.

Bei Verwendung eines Compilers kann der Anwendungscode selbst als anwendungsspezifische Ausführungseinheit betrachtet werden. Der Compiler ist dann der Generator dieser Ausführungsumgebung. Optimierungen müssen durch ihn während der Codeerstellung durchgeführt werden. Bei der Beschreibung der anderen Bausteine werden gelegentlich noch Optimierungen angedeutet, die ein Compiler berücksichtigen kann.

5.4.2 Speicherverwaltung

Die Speicherverwaltung ist eine der grundlegenden Ressourcenverwaltungen einer virtuellen Maschine. Die hauptsächliche Aufgabe liegt dabei in der Trennung zwischen freiem und belegtem Speicher. Die Speicherverwaltung bietet Schnittstellen, um beliebig große Blöcke von Speicher zu belegen und

[4]Ein Maschinenprogramm wird etwa 10 bis 100-mal schneller ausgeführt als die Interpretation des vergleichbaren Programms in Java-Bytecode. Allerdings muss man auch die Zeit beachten, die der Übersetzungsvorgang benötigt, die, je nach Compiler die Ausführungszeit auch um das vierfache überschreiten kann [SSTP00].

wieder freizugeben. Sie ist nicht für den Inhalt des Speichers verantwortlich und entscheidet somit auch nicht selbstständig, wann ein Speicherbereich wieder als frei gilt. Benötigt ein Dienst einen zuvor angeforderten Speicherbereich nicht mehr, so meldet er das an die Speicherverwaltung, welche den Speicherbereich dann als verfügbar kennzeichnet. Eine Besonderheit stellt die Objektverwaltung dar, da die Speicherverwaltung hier eine Speicherbereinigung initiieren kann, um unbenutzten Speicher zu finden. Schlägt eine Speicheranforderung fehl, da nicht genügend Speicher frei ist, so erfolgt eine Kommunikation mit der Ausnahmebearbeitung, welche den Fehler an den Aufrufer weiterleitet.

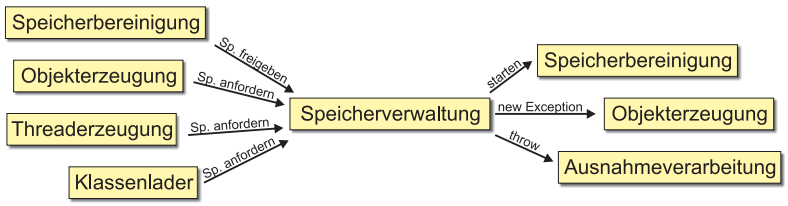


Abbildung 5.5: Abhängigkeiten der Speicherverwaltung

Die Speicherverwaltung ist ein interner Dienst, der nicht direkt von einer Anwendung aufgerufen wird, sondern nur durch andere Dienste, welche Speicher benötigen. So wird die Objektverwaltung Speicher anfordern, sobald sie ein neues Objekt anlegen will; die Klassenverwaltung benötigt Speicher, wenn eine neue Klasse geladen werden soll und die Threadverwaltung wird beim Erzeugen eines neuen Threads Speicher belegen, um den Stack und die Verwaltungsstruktur des neuen Threads abzulegen.

Von der Nutzung dieser drei Dienste hängt es demnach ab, ob die Speicherverwaltung ausgelagert werden soll. Die Objekterzeugung hat dabei den größten Einfluss auf die Entscheidung. Da es in Java nicht vorgesehen ist, Objekte auf dem Stack einer Methode zu erzeugen, müssen alle Objekte auf dem Heap erzeugt werden. Hierzu ruft die Objektverwaltung jedes Mal die Speicherverwaltung auf und ist somit der stärkste Nutzer.

Man kann den Speicherbereich nach dem Verwendungszweck in zwei Bereiche einteilen:

1. Speicher für Klassendaten und Verwaltungsstrukturen. Diese Daten werden selten gelöscht und unterliegen nicht der Speicherbereinigung, sondern werden explizit freigegeben.

2. Speicher, der durch Objekte belegt wird. Er wird häufiger wieder freigegeben.

Da Klassen- und Verwaltungsdaten weniger dynamisch sind, ist die Auslagerung dieser Speicherverwaltung leichter zu entscheiden.

Sind die möglichen Objekttypen bekannt, so kann, zur Optimierung, die Granularität der Speicherverwaltung an die möglichen Objektgrößen angepasst werden. Je nach Verfahren kann dadurch das Suchen von freien Speicherblöcken beschleunigt werden.

5.4.3 Objektverwaltung

Die Objektverwaltung ist zuständig für das Anlegen und Initialisieren neuer Objekte. Sie bestimmt außerdem das Objektlayout, das heißt die Struktur eines Objekts im Speicher. Auch das Entfernen von nicht mehr benötigten Objekten ist Aufgabe der Objektverwaltung. Entsprechend der Aufgaben kann man die Objektverwaltung in drei Komponenten aufteilen: die Objektlayoutverwaltung, die Objekterzeugung und die Speicherbereinigung.

5.4.3.1 Objektlayoutverwaltung

Die Objektlayoutverwaltung legt die Struktur eines Objekts im Speicher fest. Sie führt eine Abbildung der abstrakten Objektstrukturbeschreibung aus den Klassendaten auf einen konkreten Speicherbereich durch. Der Speicherbereich eines Objekts enthält zum einen die Nutzdaten, also die Felder eines Objekts, zum anderen Verwaltungsinformationen, welche es unter anderem ermöglichen, den dynamischen Typ des Objekts festzustellen. Die Anzahl und Größe der Felder wird durch die Objektstrukturbeschreibung, einem Teil der Klassendaten, bereitgestellt. Um den Objekttyp eindeutig zu beschreiben, kann zum Beispiel ein Index in die Typhierarchie der Klassendaten verwendet werden.

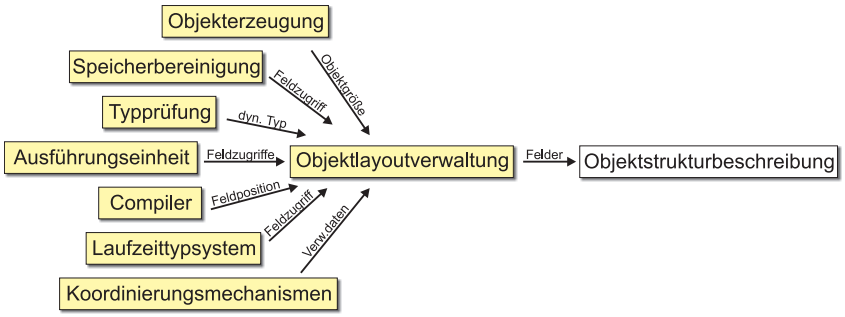


Abbildung 5.6: Abhängigkeiten der Objektlayoutverwaltung

Das Objektlayout wird von allen Bausteinen verwendet, welche die Speicherstelle eines Objektfeldes benötigen. Hauptsächlich ist das die Ausführungseinheit oder der Compiler, da hier die Feldzugriffe, welche im Anwendungscode spezifiziert sind, umgesetzt werden und dazu die tatsächliche Speicherstelle ermittelt werden muss. Auch die Speicherbereinigung greift auf die Felder eines Objekts zu, um Referenzen auf andere Objekte zu lesen.

Neben dem Zugriff auf die Objektfelder wird die Objektverwaltung auch von Bausteinen benötigt, welche auf die Verwaltungsinformationen eines Objekts zugreifen möchten. Hier ist vor allem die Typprüfung zu nennen, welche den dynamischen Typ eines Objekts auslesen muss. Aber auch die Koordinationsmechanismen können den aktuellen Zustand einer Sperre in den Verwaltungsstrukturen eines Objekts abgelegt haben (siehe Abschnitt 5.4.5).

Durch das Austauschen der Objektlayoutverwaltung kann das Objektlayout den tatsächlich benötigten Anforderungen angepasst werden. So ist es von den eingesetzten Koordinierungsmechanismen abhängig, ob Platz für die Markierung gesperrter Objekte vorhanden sein muss. Die Kennzeichnung des dynamischen Typs kann je nach Typhierarchie unterschiedlich kompakt dargestellt werden. Sind im System nur wenige Typen bekannt, so können diese Typen mit weniger Platz identifiziert werden als in einem System mit vielen Typen. Werden neue Typen hinzugefügt, so muss das Objektlayout jedoch angepasst werden. Verändert man das Objektlayout durch den dynamischen Austausch einer optimierten Objektlayoutverwaltung zur Laufzeit, so müssen gegebenenfalls alle aktiven Objekte in das neue Layout umgewandelt werden.

Beim Austausch der Objektlayoutverwaltung ist zu beachten, dass sie oft nicht als expliziter Dienst realisiert ist. Stattdessen ist sie implizit gegeben und in die Dienste integriert, welche die Speicherstelle eines Objektfeldes benötigen. Der Binder kann so beim Auflösen von Objektfeldern direkt den Abstand des Feldes vom Beginn des Objekts im Speicher vermerken. Ein Interpreter hat somit direkt die nötige Information um den Ort des Feldes zu bestimmen. Wird ein Compiler verwendet, so kann das Layout des Objekts in den erzeugten Code einfließen und Feldzugriffe werden direkt in Zugriffe auf die entsprechende Speicherstelle umgesetzt. Zugriffe auf die Verwaltungsdaten eines Objekts können

nach gleichem Muster zum Beispiel in die Typprüfung integriert werden. Anstelle eines Aufrufs der Objektlayoutverwaltung wird eine feste Position innerhalb der Objektdaten verwendet.

Das Austauschen der Objektlayoutverwaltung bedeutet somit, dass alle Bausteine, welche die Abbildung verwenden, ausgetauscht werden müssen. Ist bei kompiliertem Code das Layout der Objekte im erzeugten Code festgehalten, so muss auch der Code neu erstellt werden.

5.4.3.2 Objekterzeugung

Die Objekterzeugung ist für die Erstellung von Objekten zuständig. Sie arbeitet eng mit der Speicherverwaltung zusammen, da für jedes neu erzeugte Objekt Speicher angefordert werden muss. Die Größe des benötigten Speicherplatzes bestimmt sich aus Anzahl und Typ der Felder und der Größe der zusätzlichen Verwaltungsstrukturen. Daher wird auf die Objektlayoutverwaltung zurückgegriffen, um die initiale Struktur des Objekts zu erstellen.

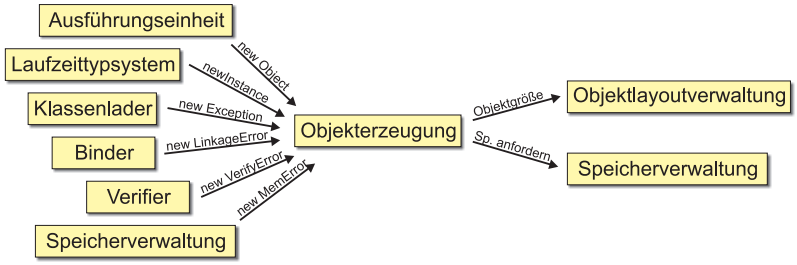


Abbildung 5.7: Abhängigkeiten der Objekterzeugung

Die Objekterzeugung wird von allen Bausteinen benötigt, welche Objekte anlegen möchten. Hauptsächlich ist das die Ausführungseinheit, welche während der Interpretation der Anwendung entsprechend Objekte anlegt. Außerdem werden von Systemdiensten Objekte erzeugt, um Fehler mithilfe des Ausnahmemechanismus an die Anwendung weiterzuleiten. Beispiele hierfür sind die Typprüfung (`ClassCastException`) oder die Speicherverwaltung (`OutOfMemoryError`).

Durch gezieltes Design einer Anwendung kann die Erzeugung von Objekten auf die Startphase der Anwendung beschränkt werden. Dann ist eine Auslagerung der Objekterzeugung nach dieser Phase sinnvoll, zumal gleichzeitig auch das Auslagern der Speicherverwaltung in Betracht zu ziehen ist. Das Identifizieren solcher Phasen kann mithilfe einer Programmflussanalyse erfolgen.

5.4.3.3 Speicherbereinigung

Die Speicherbereinigung, auch als *Garbage Collection* bezeichnet, wird benötigt, um Objekte zu finden, welche nicht mehr referenziert werden. Diese Objekte können nicht mehr erreicht und somit gelöscht werden. Um unerreichbare Objekte zu finden, muss die Speicherverwaltung die Felder der Objekte kennen; diese Information bekommt sie von der Klassenverwaltung. Enthält ein erreichbares Objekt eine Referenz, so muss die Speicherbereinigung das referenzierte Objekt von der Objektverwaltung anfordern. Objekte, die nicht erreichbar sind, werden freigegeben. Dazu wird der Speicher, der durch diese Objekte belegt war, der Speicherverwaltung gemeldet.

Die Speicherbereinigung wird von der Speicherverwaltung aufgerufen, wenn der verfügbare Speicher einen bestimmten Grenzwert unterschreitet oder wenn eine Anforderung nicht erfüllt werden kann. Die Speicherverwaltung kann auch periodisch aktiviert werden oder während das System im Leerlauf ist.

Da es in Java keine Möglichkeit gibt, Objekte explizit zu löschen und somit Speicher explizit wieder frei zugeben, ist eine Speicherbereinigung immer notwendig, wenn dynamisch Objekte angelegt

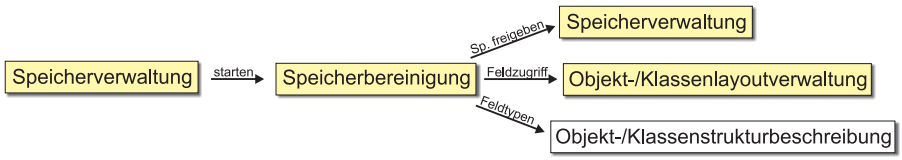


Abbildung 5.8: Abhängigkeiten der Speicherbereinigung

werden. Es gibt verschiedene Verfahren zur automatischen Speicherbereinigung, mit spezifischen Vor- und Nachteilen. Daher ist ein Wechsel zwischen diesen verschiedenen Verfahren wünschenswert.

Im Folgenden soll eine Charakterisierung der verschiedenen Ansätze vorgenommen werden, um die Vor- und Nachteile abschätzen zu können. Dabei werden nicht einzelne Algorithmen vorgestellt, sondern Kriterien, um mögliche Verfahren einzuteilen.

direkt — indirekt Eine Unterscheidung lässt sich treffen nach der Art und Weise, wie man unbenutzte Objekte erkennt. Bei den sogenannten *direkten* Ansätzen kann man für jedes Objekt direkt feststellen, ob es noch benötigt wird. Das kann zum Beispiel dadurch realisiert sein, dass man zu jedem Objekt einen Referenzzähler speichert, der die Anzahl der Verweise auf das Objekt enthält [Col60]. Vorteil hierbei ist, dass man sehr leicht erkennen kann, ob ein Objekt noch verwendet wird. Sobald der Referenzzähler auf Null ist, kann das Objekt gelöscht werden. Nachteil ist, dass der Referenzzähler immer aktuell zu halten ist. Je nach Vorgehen ist das mit relativ viel Aufwand verbunden.

Wenn nicht direkt im Objekt vermerkt ist, ob es noch benötigt wird, dann muss man das *indirekt* feststellen. Dies kann zum Beispiel dadurch erfolgen, dass man einen Erreichbarkeitsgraphen aufbaut. Das heißt, ausgehend von einer initialen Menge an Objekten, *root set* genannt, werden alle erreichbaren Objekte abgelaufen. Objekte, die man dabei nicht besucht, können gelöscht werden, da sie nicht erreichbar sind. Eines der bekanntesten Beispiele dieser Speicherbereinigungsart ist der *mark-and-sweep* Algorithmus, welcher bereits 1960 das erste Mal von John McCarthy im Rahmen von LISP beschrieben wurde [McC60]. Vorteil der indirekten Verfahren ist, dass zur normalen Ausführungszeit kein zusätzlicher Aufwand entsteht. Dafür ist allerdings das Erstellen des Erreichbarkeitsgraphen relativ aufwendig.

kopierend/verschiebend — nicht-verschiebend Eine andere Charakterisierung beurteilt die Verfahren nach der Art, wo die verwendeten Objekte nach der Speicherbereinigung angeordnet sind. Hier besteht die Möglichkeit, dass die Objekte an derselben Position liegen wie vor der Speicherbereinigung oder dass sie an eine andere Position kopiert wurden. Kopierende Speicherbereinigung bedeutet dabei eigentlich, dass alle benötigten Objekte von einem Speicherbereich in einen anderen kopiert werden [Che70, FY69]. Man kann auch noch solche Verfahren dazu zählen, welche die lebenden Objekte zusammenfassen [HW67]. Vorteil ist hauptsächlich, dass die Fragmentierung des freien Speichers vermieden wird. Nachteil kopierender oder verschiebender Verfahren ist, dass Referenzen auf verschobene Objekte angepasst werden müssen.

inkrementell — blockierend Eine weitere Unterscheidung kann nach der Fähigkeit zur nebenläufigen Ausführung der Speicherbereinigung vorgenommen werden. Einige Verfahren können nur exklusiv ausgeführt werden. Das heißt, dass das System während der Speicherbereinigung angehalten wird. Diese Verfahren werden *blockierend* genannt.

Von *inkrementellen* Verfahren spricht man, wenn die Speicherverwaltung beliebig oft unterbrochen und wieder fortgesetzt werden kann. Dabei wird jedes Mal ein Teil des Speichers durchsucht. Diese Verfahren können dann im Wechsel mit der normalen Programmausführung abgearbeitet werden und führen somit zu einer quasi parallelen Speicherbereinigung. Für viele blockierende Verfahren ist es möglich, eine inkrementelle Version zu erstellen [GLS75, HGB78].

Damit indirekte Verfahren inkrementell durchgeführt werden können, darf sich der Erreichbarkeitsgraph während der Suche nach verwendeten Objekten nicht verändern. Im Speziellen darf keine Referenz von einem unbesuchten Objekt in einen bereits abgelaufenen Zweig eingehängt werden. Denn wenn gleichzeitig alle anderen Referenzen auf das unbesuchte Objekt gelöscht werden, so wird das Objekt nicht als referenziert erkannt und fälschlicherweise gelöscht.

Um zu verhindern, dass Referenzen in bereits besuchte Objekte eingefügt werden, müssen Speicherzugriffe während einer inkrementellen Speicherbereinigung mithilfe von sogenannten *Barrieren* überwacht werden. Inkrementelle Verfahren sind durch dieses Vorgehen deutlich aufwendiger als blockierende Verfahren und benötigen insgesamt mehr Ressourcen für einen vollständigen Durchlauf. Der Vorteil vom inkrementellen Verfahren liegt jedoch auf der Hand: mit ihnen kann man eine bessere Verfügbarkeit des Systems erreichen.

Durch Austauschen der Speicherbereinigung kann man beispielsweise den schnellen nicht-verschiebenden Garbage Collector durch einen verschiebenden auswechseln, wenn der Speicher zu stark fragmentiert ist. Beim Austausch muss man allerdings beachten, dass neben dem Code auch Datenstrukturen erzeugt oder angepasst werden müssen. Für einen direkten Garbage Collector müssen zum Beispiel zunächst die Anzahl der Referenzen auf jedes Objekt ermittelt werden.

Ist eine Speicherbereinigung nur selten notwendig, so ist das Auslagern der Speicherbereinigung als externer Dienst interessant. Verwendet man dazu den Standardmechanismus der Basisschicht, so wird allerdings jede untersuchte Referenz separat zum Verwalter übertragen. Außerdem setzt das voraus, dass der Garbage Collector über festgelegte Schnittstellen auf die Objekte zugreift.

Als Optimierung kann man auch den gesamten relevanten Speicherinhalt vor der Speicherbereinigung zum Verwalterknoten transportieren. Dann kann auch ein Speicherbereinigungsalgorithmus eingesetzt werden, der die Objektstruktur implizit kennt und direkt auf dem Speicher arbeitet. Bei verschiebenden oder kopierenden Verfahren muss der Speicher nach der Bereinigung wieder zurücktransportiert werden. Verändert das Verfahren die Position von aktiven Objekten nicht, so reicht es aus, die Speicherverwaltung über den frei gegebenen Speicher zu informieren.

Bei der Verwendung einer nicht-verschiebenden Speicherbereinigung nach dem *mark-and-sweep* Verfahren bringt eine Auslagerung den Vorteil mit sich, dass der Knoten während der Speicherbereinigung nicht angehalten werden muss. Da der Garbage Collector auf einer Kopie des Speicherinhalts arbeiten kann, verändert sich der Erreichbarkeitsgraph der Objekte nicht. Alle Objekte, die zur Zeit des Abbildes nicht mehr erreichbar waren, können auch während der Speicherbereinigung nicht erreicht werden.

Da jedoch während der externen Speicherbereinigung neue Objekte entstehen können, kann nicht der gesamte Speicher, der nicht erreichbar war, als frei markiert werden. Stattdessen darf man in der sweep-Phase nur solchen Speicherplatz freigeben, der auch durch Objekte belegt war.

Das externe Ausführen der Speicherbereinigung kann sich speziell für kleine Knoten mit sehr wenig Speicher und Ressourcen lohnen. Hier muss nur sehr wenig Speicher transportiert und der Aufwand der Speicherbereinigung muss nicht lokal erbracht werden.

5.4.4 Threadverwaltung

Die Threadverwaltung ist zuständig für das Anlegen neuer und die Verwaltung existierender Threads. Sie lässt sich in vier Bausteine aufteilen: die Threaderzeugung, die Threadsteuerung, die Einplanung (Scheduler) und die Einlastung (Dispatcher).

5.4.4.1 Threaderzeugung

Die Threaderzeugung wird aufgerufen, wenn ein neuer Aktivitätsträger benötigt wird. Beim Erstellen eines neuen Threads muss ein Speicherbereich angefordert werden, der dem neuen Aktivitätsträger als Stack zugeordnet wird. Außerdem wird eine Verwaltungsstruktur erzeugt und initialisiert, welche zum Beispiel einen Verweis auf den Stack enthält und Speicher, um den Inhalt der Register aufzunehmen, während der Aktivitätsträger nicht ausgeführt wird.

Die Threaderzeugung wird hauptsächlich von der Anwendung beziehungsweise von der Ausführungseinheit aufgerufen. Neben der Anwendung kann es auch einige interne Systemdienste geben, die unabhängig von der Anwendung laufen und daher auch einen Aktivitätsträger erzeugen. Ein Beispiel hierfür ist die Speicherbereinigung, welche zwar nicht unbedingt parallel zur Anwendung ausgeführt wird, diese jedoch unterbricht und daher meist einen eigenen Ausführungskontext besitzt.

Auf die Threaderzeugung kann vollständig verzichtet werden, wenn zur Laufzeit keine neuen Aktivitätsträger erzeugt werden. Das lässt sich durch statische Analyse der Anwendung feststellen. Enthält die Anwendung keinen Code, um `Thread`-Objekte zu erzeugen, so nutzt die Anwendung nur einen Aktivitätsträger. Die Analyse gilt jedoch nur für den Zustand der Anwendung zum Zeitpunkt der Überprüfung. Wird später Code hinzugefügt, so können dort neue Threads erzeugt werden. Dann muss die Threaderzeugung zum System hinzugefügt werden.

Da Threads häufig nur während des Startens einer Anwendung erzeugt werden, bietet es sich an, die Threaderzeugung nach der Startphase als externen Dienst auszulagern. Auf diese Weise muss der Code zur Erzeugung und Initialisierung neuer Threads nicht auf jedem Knoten vorhanden sein. Dennoch können neue Threads erzeugt werden, falls es die Anwendung benötigt.

5.4.4.2 Threadsteuerung

Die Threadsteuerung bietet Möglichkeiten, den Zustand eines Aktivitätsträgers zu beeinflussen. Die wichtigsten Aktionen sind daher das Anhalten eines Threads und das Fortsetzen eines zuvor angehaltenen Threads. Daneben können noch einige weitere Parameter eingestellt werden, wie beispielsweise die Priorität eines Aktivitätsträgers. Die Threadsteuerung verwaltet den Zustand der Threads und kommuniziert mit dem Scheduler, um lauffähige Threads der Einplanung zur Verfügung zu stellen.

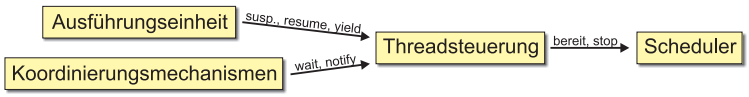


Abbildung 5.9: Abhängigkeiten der Threadsteuerung

Verwendung findet die Threadsteuerung in der Realisierung der Klasse `java.lang.Thread`. Daher ist die Anwendung und somit die Ausführungseinheit der hauptsächliche Nutzer dieses Dienstes. Die Threadsteuerung wird auch noch indirekt über die Koordinierungsmechanismen benötigt, da in diesem Zusammenhang ebenfalls Aktivitätsträger blockiert und fortgesetzt werden.

Da die Threadsteuerung hauptsächlich als direkter Dienst von der Anwendung genutzt wird, ist sie nur erforderlich, wenn die Anwendung sie explizit referenziert. Aufrufe an die Threadsteuerung

lassen sich durch den Abhängigkeitsgraphen der Basisschicht feststellen. Wie auch schon bei der Threaderzeugung kann die Threadsteuerung nachträglich notwendig werden, wenn Code nachgeladen wird.

Wie bereits in Abschnitt 5.2.2 geschildert, können die Verwaltungsstrukturen auf die Anzahl der vorhandenen Aktivitätsträger angepasst werden. Um dabei die Flexibilität zu erhalten, muss der Verwalter über die Erzeugung neuer Threads informiert werden, um gegebenenfalls die Threadsteuerung austauschen zu können.

5.4.4.3 Scheduler

Sind mehrere lauffähige Aktivitätsträger im System vorhanden, muss vor jeder Threadumschaltung ausgewählt werden, welcher der möglichen Kandidaten nun tatsächlich als nächstes die CPU verwenden darf. Diese Einplanung der Aktivitätsträger wird vom Scheduler vorgenommen, der somit die Strategie zur Auswahl des nächsten Threads darstellt.

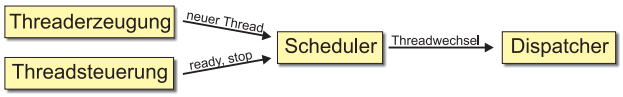


Abbildung 5.10: Abhängigkeiten des Schedulers

Der Scheduler wird vor jeder Threadaktivierung beziehungsweise -umschaltung aktiviert. Dies ist notwendig, wenn der aktuelle Aktivitätsträger zum Beispiel blockiert, durch ein Zeitscheibenereignis verdrängt wird oder den Prozessor freiwillig abgibt (`Thread.yield()`). Der Scheduler wird somit nie direkt durch die Anwendung aktiviert, sondern immer nur indirekt über die Threadsteuerung.

Je nachdem, wie viele Aktivitätsträger im System sind, lassen sich optimierte Varianten des Schedulers erstellen. Ist nur ein Thread vorhanden, wird gar keine Einplanungsstrategie benötigt, da es keine Wahlmöglichkeit gibt. Die Komplexität der Einplanungsstrategie nimmt mit zunehmender Threadanzahl zu. Bei nur zwei Aktivitätsträgern kann immer der jeweils andere ausgewählt werden, bei deutlich mehr Threads kann eine komplexe Strategie oder sogar ein generischer Scheduler mit austauschbarer Strategie eingesetzt werden. Ist die Anzahl der vorhandenen Threads konstant oder die maximale Anzahl bekannt, so kann man eine optimierte Implementierung verwenden, die im Gegensatz zu einer generischen Implementierung, weniger Overhead mit sich bringt, dafür aber nicht beliebig viele Threads verwalten kann.

Um automatisch eine Optimierung auswählen zu können, muss die Anzahl der vorhandenen Threads festgestellt werden. Einige spezielle Fälle können zur Kompilierzeit festgestellt werden. Beispielsweise kann eine Codeanalyse feststellen, dass in der Anwendung kein `Thread`-Objekt erstellt wird. Somit ist nur ein Aktivitätsträger vorhanden. Wird jedoch Code zur Erstellung eines `Thread`-Objekts gefunden, so ist es nicht immer möglich zu sagen, wie oft er durchlaufen wird.

Erfolgversprechender ist, zur Laufzeit festzustellen, wie viele Threads aktuell existieren und anhand dieser Information die optimale Threadverwaltung auszuwählen. Dazu meldet der verwaltete Knoten dem Verwalter, sobald sich die Anzahl der existierenden Threads verändert. Dieser kann dann einen Scheduler installieren, der optimal auf die entsprechende Threadanzahl abgestimmt ist.

Die Auslagerung des Schedulers als externer Dienst ist nur sinnvoll, wenn die Bestimmung des nächsten Threads sehr aufwendig ist. Oft ist die eingesetzte Strategie jedoch relativ einfach.

5.4.4.4 Dispatcher

Der Dispatcher ist für die tatsächliche Umschaltung zwischen verschiedenen Aktivitätsträgern zuständig. Seine Aufgabe ist das Wiederherstellen eines Threadkontexts und gegebenenfalls das Sichern des aktuellen Threadzustands. Diese Aufgabe ist stark architekturabhängig, da hier Registerinhalte gesichert und restauriert werden müssen.

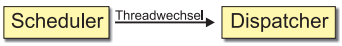


Abbildung 5.11: Abhängigkeiten des Dispatchers

Vor der Einlastung von Threads muss erst bestimmt werden, welcher Thread aktiviert werden soll. Daher wird der Dispatcher nur vom Scheduler aufgerufen, wenn dieser eine Umschaltung auf einen anderen Aktivitätsträger entscheidet.

Ist nur ein Thread im System und wird zur Behandlung von Unterbrechungen kein manueller Kontextwechsel benötigt, so kann auf einen Dispatcher verzichtet werden. Falls ein Dispatcher benötigt wird, muss er jedoch lokal vorhanden sein. Die Auslagerung ist nicht möglich, da zur Einlastung eines Threads lokale, hardwarespezifische Operationen notwendig sind, die nicht entfernt durchgeführt werden können.

Für einige Prozessoren können jedoch verschiedene Varianten eines Dispatchers angeboten werden, welche beispielsweise einige Spezialregister nur unter bestimmten Bedingungen sichern. So müssen die Register der Fließkommaeinheit bei einem Kontextwechsel nicht gesichert werden, wenn keine Thread Fließkommaoperationen durchführt. Sobald sich das jedoch ändert, muss der Dispatcher ausgetauscht werden.

5.4.5 Koordinierungsmechanismen

Zur Koordinierung nebenläufiger Vorgänge in einer Anwendung steht in Java ein Monitor Konzept [Han72] zur Verfügung, welches in Form eines `synchronized` Blocks um einen kritischen Abschnitt angewandt werden kann. Neben dem Schutz eines kritischen Abschnitts bietet das Monitor Konzept auch die Möglichkeit einer Benachrichtigung mittels `Object.wait()` und `Object.notify()`, um aktives Warten vermeiden zu können.

Zur Unterstützung dieser Koordinierungsdirektiven muss die virtuelle Maschine entsprechende Mechanismen anbieten. Neben einer Schlossvariablen, die angibt, ob der Monitor gerade besetzt ist, wird für jeden gesperrten Monitor eine Warteschlange benötigt, in die Threads eingereiht werden, die darauf warten, den Monitor betreten zu können. Gibt ein Thread, bei der Blockierung an einem Monitor, die CPU auf, so wird der Zustand mithilfe der Threadsteuerung entsprechend geändert. Bei Reaktivierung eines Threads sieht die Java-Spezifikation keine spezielle Abstimmung mit der Einplanungsstrategie von Threads vor; es wird ausdrücklich betont, dass ein zufälliger Thread aktiviert wird [LY99].

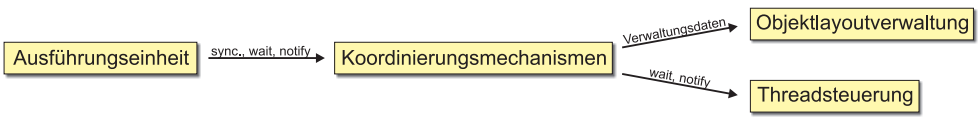


Abbildung 5.12: Abhängigkeiten der Koordinierungsmechanismen

Die Koordinierungsmechanismen stellen einen Dienst der virtuellen Maschine für die Anwendung dar. Sie werden daher nur durch die Anwendung beziehungsweise durch die Ausführungseinheit aktiviert. Der Aufruf erfolgt, sobald ein mit `synchronized` geschützter kritischer Abschnitt betreten oder verlassen wird.

Vor dem Betreten eines Monitors muss festgestellt werden, ob der Monitor gesperrt oder frei ist. Diese Information kann auf verschiedene Weise gespeichert werden. Eine Möglichkeit besteht darin, es in dem Monitor-Objekt selbst zu vermerken. Dann ist der Zustand sofort an den Verwaltungsinformationen des Objekts ablesbar. Da jedoch jedes Java-Objekt potenziell als Monitor-Objekt genutzt werden kann, muss in jedem Objekt Platz für diese Information vorgesehen werden. Das kann zu einer unverhältnismäßig starken Vergrößerung aller Objekte führen. Alternativ kann eine zusätzliche Liste geführt werden, welche die Objekte enthält, die aktuell als Monitor genutzt werden. Dadurch wird nicht in allen Objekten Platz benötigt, jedoch ist der Aufwand größer, die entsprechende Variable zu einem Objekt zu finden.

Welche Art sinnvoller ist, hängt davon ab, wie viele und wie häufig Objekte als Monitor genutzt werden. Diese Informationen können nur zur Laufzeit ermittelt werden. Mit einem Verwalter kann man dann dynamisch die Koordinierungsmechanismen und das Objektlayout austauschen.

Das Auslagern der Koordinierungsmechanismen ist zum Teil möglich. So können zum Beispiel die Warteschlangen an den Monitoren extern realisiert werden. Da das Nachfragen, ob der kritische Abschnitt betreten werden darf, dann jedoch relativ viel Zeit in Anspruch nimmt, ist die Auslagerung nur selten sinnvoll.

5.4.6 Ausnahmeverarbeitung

Die Ausnahmeverarbeitung ist der Teil der JVM, der die Behandlung einer Ausnahme (*Exception*) anstößt, sobald eine Ausnahme aufgetreten ist beziehungsweise geworfen wurde. Die Ausnahmeverarbeitung wird durch eine `throw`-Anweisung aktiviert. Ihre erste Aufgabe ist die Bestimmung des Typs des Ausnahmeobjekts, um anschließend zu prüfen, ob für diesen Typ im aktuell ausgeführten Code eine entsprechende Ausnahmebehandlung registriert ist. Dafür wird die aktuelle Codeposition mithilfe der Ausführungseinheit ermittelt. Anschließend können der Code der Ausnahmebehandlungen und die Wirksamkeit anhand der Ausnahmebehandlungstabelle bestimmt werden. Wird keine passende Ausnahmebehandlung gefunden, so wird die aufrufende Methode mithilfe der Ausführungseinheit bestimmt und dort nach einer geeigneten Ausnahmebehandlung gesucht.

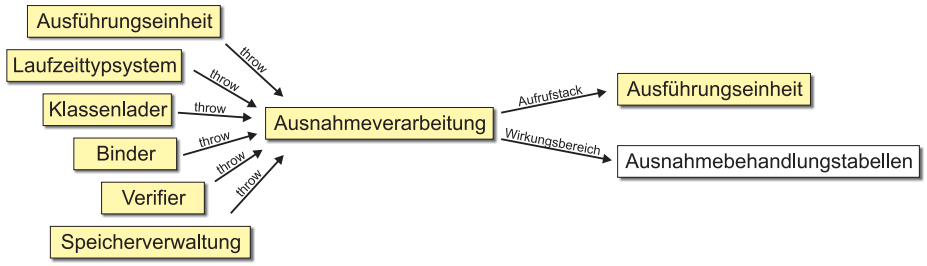


Abbildung 5.13: Abhängigkeiten der Ausnahmeverarbeitung

Die Ausnahmeverarbeitung wird von allen Diensten benötigt, die einen Fehlerzustand an die Anwendung melden möchte. Hiervon macht zum Beispiel die Speicherverwaltung gebrauch, wenn nicht mehr genügend Speicher zur Verfügung steht, um eine Anforderung zu erfüllen. Am meisten wird sie jedoch von der Ausführungseinheit benötigt. Zum einen, um entsprechende explizite Anweisungen in

der Anwendung umzusetzen, aber auch, um Fehlerzustände bei der Interpretation der Anwendung zu melden (`RuntimeException`).

Wird in einer Anwendung keine Ausnahme abgefangen, so muss auch kein Baustein zur Ausnahmeverarbeitung vorhanden sein. Ob Ausnahmen abgefangen werden, kann automatisch bestimmt werden, da geprüft werden kann, ob eine Ausnahmebehandlung vorhanden ist.

Das Auslagern der Ausnahmeverarbeitung ist sinnvoll, wenn nur wenige Ausnahmen geworfen werden. Dies sollte im Allgemeinen der Fall sein, denn der Ausnahmemechanismus soll auf eine Abweichung von der normalen Ausführung hindeuten und einen möglichen Fehlerzustand anzeigen. Durch die Auslagerung können auch die Ausnahmebehandlungstabellen der Klassenverwaltung ausgelagert werden, da sie nur von der Ausnahmeverarbeitung gebraucht werden.

5.4.7 Codevorbereitung

In der Codevorbereitung sind alle Bausteine zusammengefasst, die notwendig sind, um neue Klassen verwenden zu können.

5.4.7.1 Klassenlader

Java bietet Anwendungen die Möglichkeit, zur Laufzeit neue Klassen zu laden und damit neue Typen zu definieren. Diese Funktionalität wird durch den Klassenlader realisiert. Er bekommt eine Klassendatei als Eingabe und verändert die internen Datenstrukturen der virtuellen Maschine entsprechend, sodass die neue Klasse verwendet werden kann. Dabei werden neue Typen in die Typhierarchie eingefügt, neue Objektstruktur-, Klassenstruktur- und Methodenbeschreibungen erzeugt. Außerdem werden neue Ausnahmebehandlungstabellen angelegt und der Code der Methoden bereitgelegt. Im Rahmen dessen wird auch Speicher für die Klassenvariablen der Klasse angefordert. Der benötigte Speicherplatz wird mithilfe der Klassenlayoutverwaltung bestimmt.

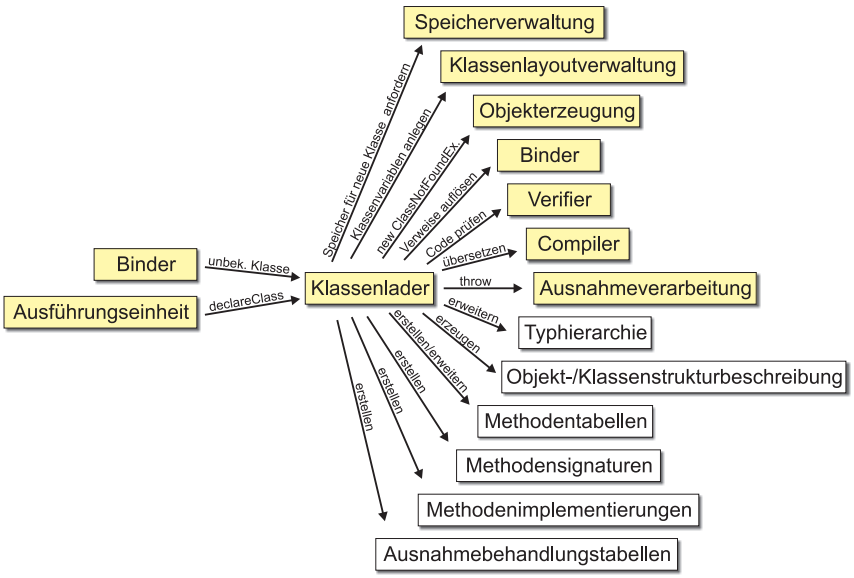


Abbildung 5.14: Abhängigkeiten des Klassenladers

Das Laden neuer Klassen kann explizit in der Anwendung angestoßen werden. Darüber hinaus kann es beim Binden einer neu geladenen Klasse dazu kommen, dass weitere Klassen nachgeladen werden

müssen. Ob die Anwendung dynamisches Laden von Klassen einsetzt, kann durch Analyse des Codes festgestellt werden.

Da eine laufende Anwendung im Normalfall selten zusätzliche Klassen benötigt, bietet sich das Auslagern des Klassenladers an. Lediglich zur Startzeit werden viele Klassen neu in die VM geladen. Aber auch hier kann ein externer Klassenlader sinnvoll sein. Speziell dann, wenn die Klassendaten nicht lokal vorliegen, sondern erst zum Knoten übertragen werden müssen. Durch externes Ausführen des Klassenladers entfällt das Laden und Analysieren der Klassendaten vor Ort. Stattdessen können direkt die vorbereiteten Datenstrukturen zum Gerät übertragen werden.

Ist die Laufzeitumgebung aus optimierten Bausteinen aufgebaut, so muss der Verwalter ohnehin darüber informiert werden, dass dem System neuer Code und neue Typen hinzugefügt werden. Der Verwalter kann dann die Voraussetzungen der optimierten Bausteine überprüfen und sie gegebenenfalls durch andere Varianten ersetzen, die unter den neuen Gegebenheiten optimal arbeiten.

5.4.7.2 Verifier

Der Verifier prüft, ob eine geladene Klasse die JVM-Spezifikation [LY99] erfüllt. Dabei wird zum Beispiel geprüft, ob alle Befehle gültig sind und ob das Ziel eines Sprungbefehls der Anfang und nicht die Mitte eines anderen Befehls ist. Neben vielen weiteren Überprüfungen ist besonders die Typprüfung wichtig. Dabei wird festgestellt, dass für jede Operation der Typ der Daten zu der darauf angewandten Operation passt.

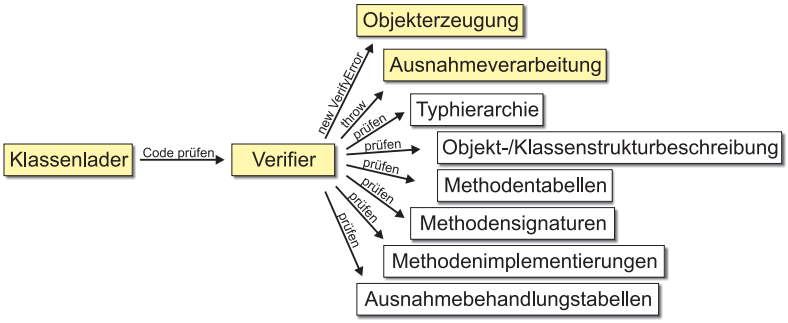


Abbildung 5.15: Abhängigkeiten des Verifiers

Der Verifier wird vom Klassenlader nach dem Laden der binären Klassendefinition aufgerufen. Erst wenn die Klasse erfolgreich überprüft wurde, fährt der Klassenlader mit dem Installieren der Klasse fort.

Der Verifier ist hauptsächlich von den konstanten Klassendaten abhängig. Er kann daher gut als externer Dienst angeboten und separat vor dem Laden aktiviert werden. Das Auslagern des Verifiers zusammen mit dem Klassenlader ist besonders sinnvoll, da dann die gesamte Vorverarbeitung am Verwalter durchgeführt werden kann.

5.4.7.3 Binder

Beim Binden werden die symbolischen Referenzen innerhalb einer Klassendefinition aufgelöst. Symbolische Referenzen sind zum Beispiel Referenzen auf Typen bei der Definition von Feldern oder der Name von Methoden bei Methodenaufrufen. Werden während des Bindens Klassen oder Typen referenziert, die noch nicht geladen sind, so wird der Klassenlader beauftragt, diese Klassen zu laden.

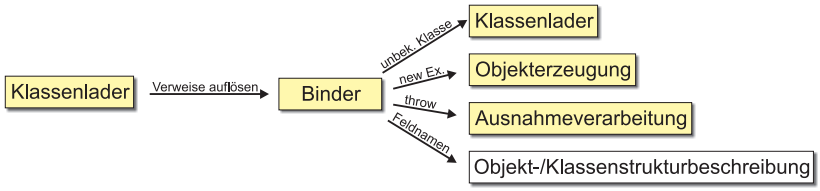


Abbildung 5.16: Abhängigkeiten des Binders

Das Auflösen einer Referenz muss vor deren Nutzung durchgeführt werden. Der Binder kann daher direkt nach dem Laden einer Klasse durch den Klassenlader aufgerufen werden oder erst zur Ausführungszeit von der Ausführungseinheit. Um Bausteine auszulagern, ist es jedoch sinnvoll, den Wirkungsbereich eines Bausteins auf wenige definierte Zeiträume einzuschränken. Daher eignet sich für dieses Ziel ein vollständiges Binden zum Ladezeitpunkt am besten. Der Code kann dann vom Verwalter vorbereitet und gebunden werden, bevor er zum Gerät übertragen wird.

Dieses Vorgehen kann sowohl beim Starten der Anwendung als auch beim Nachladen von Klassen angewandt werden. Besonders sinnvoll ist diese Lösung, wenn gleichzeitig der Klassenlader und der Verifier ausgelagert werden.

5.4.7.4 Compiler

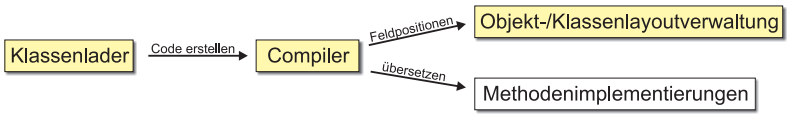


Abbildung 5.17: Abhängigkeiten des Compilers

Soll der Bytecode direkt von der Zielplattform verarbeitet werden, so wird zusätzlich zur Ausführungseinheit ein Compiler benötigt, der den Bytecode in Maschinencode übersetzt. Je nach dem Zeitpunkt des Übersetzens unterscheidet man beim Kompilieren zwischen zwei verschiedenen Ansätzen. Wird der Bytecode erst in Maschinencode übersetzt sobald er benötigt wird, so spricht man von einem *Just-In-Time* Compiler (JIT-Compiler); wird der Code zu einem früheren Zeitpunkt übersetzt, so spricht man von einer *Ahead-Of-Time* Übersetzung (AOT-Compiler).

Das erste Verfahren kann einen Java-Bytecode Interpreter ergänzen oder vollständig ersetzen. Meist ist dabei ein Zwischenspeicher vorgesehen, der kompilierte Methoden oder Teile davon vorhält, bis sie wieder benötigt werden, um den Compiler nicht bei jedem Aufruf einer Methode erneut aufrufen zu müssen.

Bei der Ahead-Of-Time Übersetzung wird die gesamte Anwendung in Maschinencode übersetzt, unabhängig davon, ob jede einzelne Methode auch aufgerufen wird. Dies stellt zwar möglicherweise unnötigen Übersetzungsaufwand dar, dafür wird der Übersetzungsvorgang meist vor dem Starten der Anwendung durchgeführt, wodurch zur Laufzeit keine Übersetzung mehr stattfinden muss. Das dynamische Nachladen von Klassen zur Laufzeit ist mit diesem Ansatz jedoch nicht ohne Weiteres möglich. Meist wird hierzu zusätzlich ein Interpreter oder ein JIT-Compiler eingesetzt.

Der Compiler eignet sich gut, als externer Dienst ausgelagert zu werden. Er arbeitet nur auf den Klassendaten und hat sonst keine Abhängigkeiten zu anderen Diensten. Der Kompilierungsdienst erhält dann den Java-Bytecode und liefert die übersetzte Version zurück. Bei Systemen, die zur Laufzeit kompilieren, wird häufig das Ziel verfolgt, die Ausführung der Anwendung durch Kompilieren des

Codes zu beschleunigen. In diesem Fall ist die Nutzung eines Kompilierungsdienstes nur eingeschränkt sinnvoll, da durch den zeitaufwendigen Aufruf des Dienstes der Geschwindigkeitsgewinn gegenüber der reinen Interpretation zunichtegemacht wird [PLDM02].

Für Systeme, welche die Anwendung vor der Ausführung übersetzen, ist ein Kompilierungsdienst aber gut einsetzbar, da die Zeit des Übersetzens häufig weniger ausschlaggebend ist. Der Compiler wird nur in der Startphase der Anwendung benötigt; zur Laufzeit ist auf dem Gerät nur der übersetzte Code vorhanden. Solche Systeme werden häufig für Anwendungen verwendet, die sich zur Laufzeit nicht mehr verändern. Somit benötigen sie weder Compiler noch Interpreter vor Ort und sind deutlich kleiner.

5.4.8 Klassenverwaltung

Die Klassenverwaltung ist für das Verwalten der Klassen- und Typdaten zuständig. Sie stellt unter anderem Schnittstellen zur Verfügung, um die vorhandenen Klassendaten abzufragen.

5.4.8.1 Klassenlayoutverwaltung

Die Klassenlayoutverwaltung entspricht der Objektlayoutverwaltung, jedoch für Klassen. Definiert eine Klasse statische Variablen (sogenannte Klassenvariablen), dann müssen diese im Speicher abgelegt werden. Die Klassenlayoutverwaltung legt die Struktur dieses Speicherbereichs fest: Sie bildet somit die Strukturinformationen einer Klasse auf einen Speicherbereich ab.

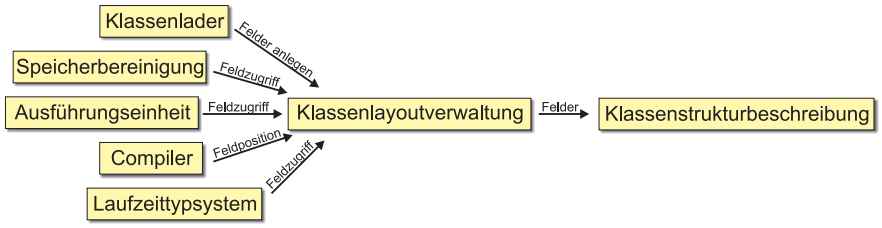


Abbildung 5.18: Abhängigkeiten der Klassenlayoutverwaltung

Die Klassenlayoutverwaltung wird von allen Bausteinen verwendet, welche die Speicherstelle einer Klassenvariablen benötigen. Das ist vor allem die Ausführungseinheit oder der Compiler, wenn Zugriffe der Anwendung umgesetzt werden. Außerdem muss die Speicherbereinigung die Klassenvariablen untersuchen, um Referenzen auf andere Objekte zu erkennen. Die Abbildung der Struktur einer Klasse auf einen Speicherbereich ist allerdings aus Effizienzgründen häufig implizit gegeben und in die Implementierung anderer Bausteine integriert.

5.4.8.2 Typüberprüfung

Die Typüberprüfung gleicht den Typ eines Objekts mit einem gegebenen Typ ab. Hat das Objekt einen kompatiblen Typ, so ist die Typprüfung erfolgreich. Die Typprüfung stellt somit eine Schnittstelle zur Typhierarchie dar.

Die Typüberprüfung wird bei allen auftretenden Typvergleichen verwendet, beispielsweise wenn eine Anwendung explizit einen Typ prüft (mittels `instanceof`) oder wenn eine Umwandlungsoperation (Cast-Operation) zur Laufzeit durchgeführt werden soll.

Unter der Annahme, dass keine Klassen zur Laufzeit nachgeladen werden, kann die Typhierarchie und damit auch die Typprüfung optimiert werden. An einigen Stellen ist dann sogar überhaupt keine

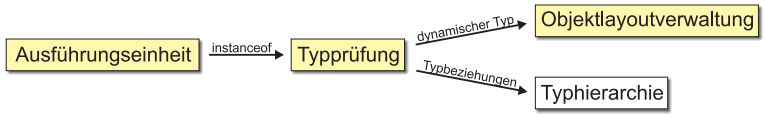


Abbildung 5.19: Abhängigkeiten der Typprüfung

Überprüfung des Typs mehr notwendig. In Abbildung 5.20 wird ein einfaches Beispiel konstruiert. Ein Schnittstellentyp *A* wird hier nur von der Klasse *B* implementiert. Da von der Schnittstelle selbst kein Objekt instanziiert werden kann, sind alle Objekte welche den Typ *A* darstellen tatsächlich von Typ *B*. Typumwandlungen von Typ *A* in Typ *B* müssen daher nicht geprüft werden.

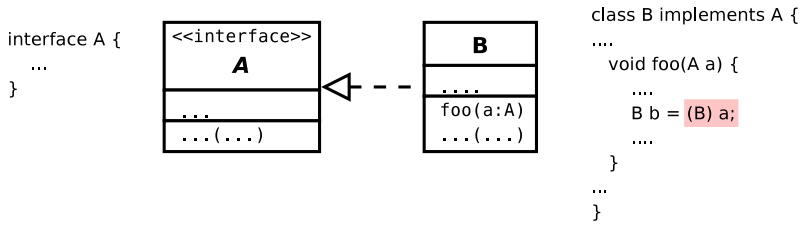


Abbildung 5.20: Einfaches Optimierungsbeispiel bei globaler Sicht

Eine Schnittstelle *A* wird **nur** von der Klasse *B* implementiert. Typumwandlungen von Objekten vom statischen Typ *A* in den Typ *B* sind dann immer erfolgreich. Die markierte Umwandlungsoperation muss somit nicht geprüft werden, da alle Objekte, welche den Typ *A* implementieren, vom Typ *B* sind.

Wird dann jedoch eine Klasse *C* nachgeladen, welche ebenfalls die Schnittstelle *A* implementiert, so ist die Optimierung ungültig. Nun ist eine Typprüfung notwendig, um Objekte vom Typ *B* von denen des Typs *C* zu unterscheiden. Die Typprüfung muss angepasst oder ausgetauscht werden.

Bei einem compilerbasierten System kann die Typüberprüfung in den Code eingearbeitet werden. Sie ist dann nicht mehr als eigenständiger Dienst sichtbar, sondern im Code der Anwendung enthalten. Bei Änderungen der Typhierarchie muss dann jedoch der betroffene Code ausgetauscht werden. Im oben betrachteten Beispiel muss beispielsweise der Code der Klasse *B* neu generiert werden, obwohl die Klasse selbst nicht verändert wurde.

5.4.9 Laufzeittypsystem

Java bietet die Möglichkeit, dass eine Anwendung zur Laufzeit Informationen über die vorhandenen Typen abfragen kann. Die Anwendung erhält dabei Objekte, welche Klassen oder Methoden beschreiben. Mithilfe solcher Meta-Objekte können dann Details der Elemente, wie beispielsweise Typen der Parameter abgefragt werden. Darüber hinaus können neue Objekte erzeugt oder existierende Objekte verändert werden. So kann beispielsweise der Inhalt eines Objektfeldes verändert oder eine Methode aufgerufen werden, deren Name während der Implementierung noch unbekannt war.

Das Laufzeittypsystem ist dafür zuständig, die entsprechenden Meta-Objekte zu erstellen und mit den Informationen aus den Klassendaten zu initialisieren. Außerdem müssen verändernde Zugriffe über diese Meta-Objekte an die entsprechenden Bausteine wie beispielsweise die Objekterzeugung weitergeleitet werden.

Die Anwendung kann das Laufzeittypsystem direkt nutzen. Ein Teil der Funktionalität wird auch von der Speicherbereinigung genutzt, um alle Referenzen zu finden.

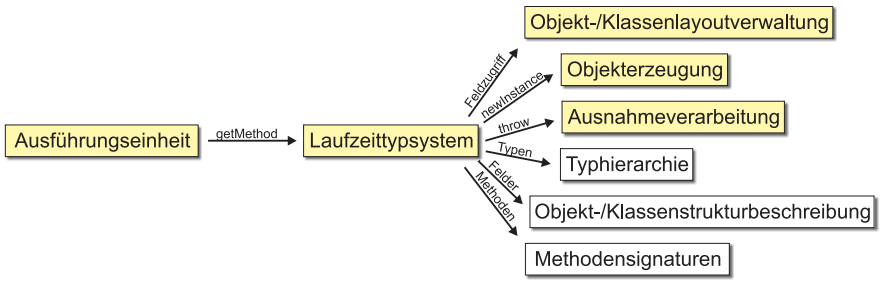


Abbildung 5.21: Abhängigkeiten des Laufzeittypsystems

Die Möglichkeit, die Struktur eines Objekts oder einer Klasse abzufragen, macht es nötig, dass die entsprechenden Informationen in Form der Objekt- und Klassenstrukturdaten sowie der Methodensignaturen zur Verfügung stehen. Da diese Daten jedoch von kaum einem anderen Dienst benötigt werden, ist es sinnvoll das Laufzeittypsystem als externen Dienst anzubieten, wenn es selten genutzt wird.

5.4.10 Klassendaten

Die Klassendaten entsprechen zum größten Teil den Informationen aus den Klassendateien der geladenen Klassen. Sie werden durch den Klassenlader erzeugt und verändert, wenn neue Klassen und Typen definiert werden.

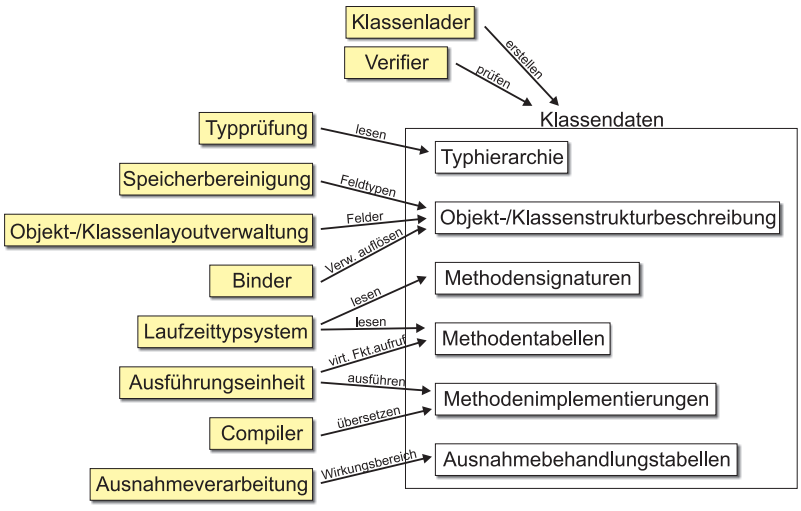


Abbildung 5.22: Abhängigkeiten der Klassendaten

Ein großer Teil der Klassendaten wird nur zur Codevorbereitung benötigt. Lagert man die Bausteine der Codevorbereitung aus, so werden viele Daten kaum noch zur Laufzeit benötigt und können mit ausgelagert werden. Einige Daten können auch durch optimierte Dienste überflüssig oder beispielsweise bei der Verwendung eines Compilers in den erzeugten Code eingearbeitet werden. Bei Veränderungen der Klassendaten muss dann die Gültigkeit überprüft und gegebenenfalls Code und Dienste ausgetauscht werden.

Da sich die Klassendaten nur beim Hinzufügen neuer Klassen verändern und das im Allgemeinen nur selten stattfindet, kann eine Kopie der Klassendaten am Verwalterknoten gehalten werden, die bei einer Veränderung aktualisiert wird. Dadurch kann man flexibel Dienste verschieben, ohne jedes Mal auch die Daten verschieben zu müssen. Generelles Ziel sollte es sein, viele Daten nur auf dem Verwalterknoten vorzuhalten.

Der genaue Aufbau der Daten ist von der tatsächlichen Realisierung der virtuellen Maschine abhängig. Dennoch kann man sie in grundlegende Bausteine einteilen. Diese sollen im Folgenden vorgestellt werden. Dabei wird auf die Nutzung und Varianten eingegangen.

5.4.10.1 Typhierarchie

Die Typhierarchie gibt Auskunft über die Beziehungen zwischen den Klassen und Typen. Sie wird von der Typprüfung verwendet, um Untertypen zu identifizieren. Daher orientiert sich der tatsächliche Aufbau der Typinformationen an diesem Dienst. Veränderungen und Optimierungen der Typprüfung haben somit Auswirkungen auf den Aufbau der Typhierarchie und umgekehrt. Eine optimierte Typprüfung kann die Typhierarchie kompakt speichern, da zur Laufzeit nicht der vollständige Typenbaum nachvollziehbar sein muss.

5.4.10.2 Objekt- und Klassenstrukturbeschreibung

Die Objektstrukturbeschreibung gibt für jede Klasse die Felder eines Objekts dieser Klasse an. Jedes Feld wird durch seinen Namen, Typ und Zugriffsrechte beschrieben. Die Klassenstrukturbeschreibung ist die entsprechende Beschreibung für Klassen. Hier sind Namen, Typ und Zugriffsrechte der Klassenvariablen abgelegt.

Die Struktur wird von der Objekt- beziehungsweise Klassenlayoutverwaltung verwendet, welche festlegen, wie die einzelnen Felder im Speicher repräsentiert werden. Die Typen der Felder werden zum Beispiel von der Speicherbereinigung benötigt, um Referenzen in Objekten zu identifizieren.

Da die Layoutverwaltung meistens implizit vorliegt, werden die Strukturbeschreibungen zur Laufzeit nicht direkt von der Anwendung benötigt. Die Informationen, um auf Objekt- und Klassenvariablen zuzugreifen, können bereits während des Bindens in den Code eingefügt werden. Es gibt jedoch einige Dienste, welche die Strukturbeschreibungen zur Laufzeit benötigen: die Speicherbereinigung und das Laufzeittypsystem. Wird das Laufzeittypsystem nicht verwendet, so können die Strukturbeschreibungen zusammen mit der Speicherbereinigung ausgelagert werden.

5.4.10.3 Methodentabellen

Die Methodentabellen enthalten Informationen, um festzustellen, welche Implementierung beim Aufruf einer Methode verwendet wird. Dies ist besonders bei Objekten von abgeleiteten Klassen notwendig, da hier erst zur Laufzeit anhand des dynamischen Typs entschieden werden kann, welche Implementierung verwendet werden muss.

Der tatsächliche Aufbau der Methodentabelle hängt stark von der Realisierung der Ausführungseinheit ab. Eine einfache Realisierung listet für jede Klasse die dazugehörenden Methodenimplementierungen auf, wobei Methoden mit derselben Signatur denselben Index in der Tabelle bekommen.

Ist das Nachladen neuer Klassen zur Laufzeit nicht vorgesehen, so können die Methodentabellen weitestgehend vereinfacht werden, da sie nur für Typen benötigt werden, welche überschriebene Methoden besitzen. Für Methoden, die nicht überschrieben werden, kann der Binder direkt die richtige und einzige vorhandene Implementierung wählen. Wird jedoch die Typhierarchie verändert, so müssen die Methodentabellen und der Code neu erstellt beziehungsweise gebunden werden.

5.4.10.4 Methodenimplementierungen

Die Methodenimplementierungen stellen den Code der Methoden dar. Je nach Art der Ausführungseinheit kann der Code in verschiedenen Formaten vorliegen. Für einen Interpreter muss der Java-Bytecode verfügbar sein. Bei Verwendung eines AOT-Compilers wird der Bytecode bereits in architekturspezifischem Maschinencode übersetzt vorliegen. Wird ein JIT-Compiler verwendet, so kann eine Mischung aus beidem abgelegt sein.

Bei einem compilerbasierten Verfahren können selten benutzten Methoden geparkt werden. Bei einem interpreterbasierten Verfahren ist das nicht möglich, da die Methodenimplementierungen nicht direkt ausgeführt werden, sondern Daten des Interpreters darstellen.

5.4.10.5 Methodensignaturen

Die Methodensignaturen stellen Typinformationen zu den Methoden bereit. Sie enthalten zu jeder Methode den Namen, die Anzahl und Typen der Parameter sowie den Typ des Rückgabewertes. Auch Zugriffsrechte sind hier vermerkt. Die Signaturen werden hauptsächlich vom Binder benötigt, um Aufrufe von Methoden auflösen zu können. Wenn das Binden nicht zur Laufzeit, sondern schon vor dem Starten durchgeführt wird, werden die Signaturen nur noch durch das Laufzeittypsystem benötigt, falls eine Methode mit einem zur Laufzeit ermittelten Namen angesprochen werden soll.

5.4.10.6 Ausnahmebehandlungstabellen

Die Ausnahmebehandlungstabellen beschreiben die in den Methoden enthaltenen Ausnahmebehandlungen. Für jede Methode, in der eine Ausnahmebehandlungsfunktion implementiert ist, sind hier Informationen zu finden, welcher Ausnahmetyp behandelt wird und welcher Codeabschnitt dadurch abgedeckt ist. Die Ausnahmeverarbeitung benötigt diese Tabellen, um beim Auftreten einer Ausnahme die entsprechende Behandlungsfunktion zu suchen. Da diese Tabellen von keinem anderen Dienst benötigt werden, können sie zusammen mit der Ausnahmeverarbeitung ausgelagert werden.

Darüber hinaus kann man Teile der Ausnahmebehandlungstabelle entfernen, wenn man die Implementierung von Methoden auslagert, da die Informationen zum Auffinden von Behandlungsfunktionen nur für aktive Funktionen benötigt werden.

5.4.11 Ergebnisse

In diesem Abschnitt wurden verschiedene Komponenten eine JVM als Bausteine identifiziert. Dabei wurde untersucht, ob sie als entfernter Dienst erbracht werden können.

Besonders zur Auslagerung geeignet sind der Klassenlader und die Codevorbereitung, da zum einen Klassen selten hinzugefügt werden und zum zweiten die Klassendateien meist ohnehin von außen zugeführt werden. Demnach können die Daten auch extern vorbereitet werden, um sie anschließend im passenden Format ohne weitere Änderung in das System zu integrieren.

Außer der Codevorbereitung bieten sich auch die Threaderzeugung und die Ausnahmebehandlung zur Auslagerung an. Diese Dienste werden ebenfalls eher selten genutzt und ohne Ausnahmebehandlung müssen auch die Ausnahmebehandlungstabellen nicht auf dem Knoten vorhanden sein. Als letzter vielversprechender Baustein wurde die Speicherbereinigung identifiziert, die durch das Auslagern parallel zum Ablauf auf dem Knoten durchgeführt werden kann.

Weiterhin wurden in diesem Abschnitt Bausteine auf mögliche Varianten untersucht. Für die Speicherbereinigung wurden verschiedene Vorgehen mit ihren Vor- und Nachteilen vorgestellt. Auch können optimierte Versionen von Threadsteuerung und Scheduler eingesetzt werden.

Zudem wurden Optimierungen angesprochen, die durch eine globale Sicht auf die vorhandenen Typen vorgenommen werden können. Unter der Annahme, dass dem System kein neuer Code hinzugefügt wird, lassen sich das Objektlayout, die Typprüfung und die Methodentabellen optimieren.

5.5 Anwendungsbeispiel KESO

Als nächster Schritt sollen einige der Unterstützungsmöglichkeiten auf ein reales Java-System angewandt werden, um die Anwendbarkeit der vorgestellten Konfigurationsmöglichkeiten zu demonstrieren. Im Rahmen dieser Arbeit wurden nicht alle Unterstützungsmöglichkeiten vollständig realisiert. Die Umsetzung einiger Beispiele wurde sorgfältig geprüft und soweit durchgeführt, dass entweder auftretende Probleme identifiziert werden konnten oder man mit ausreichender Sicherheit feststellen konnte, dass eine Realisierung möglich ist. Hier sollen einige Ideen und Ergebnisse vorgestellt werden.

Die grundsätzliche Herangehensweise wurde bereits in Abschnitt 5.3.3 vorgestellt: Man nutzt eine statisch optimierte und angepasste Java-Umgebung und erweitert sie mithilfe eines Verwalters um dynamische Fähigkeiten. Man erhält ein ressourcensparendes System, welches keine Bausteine zur Codevorbereitung auf dem Gerät selbst enthält und trotzdem die Möglichkeit hat, dynamisch Klassen nachzuladen oder Informationen über die vorhandenen Klassen zur Laufzeit abzufragen.

Die anwendungsspezifische Java-Umgebung KESO

Als Basis eignet sich hier das Java-System KESO [WSSP07]. Wie bereits in Abschnitt 5.3.2.2 dargestellt wurde, handelt es sich dabei um eine anwendungsspezifische Java-Laufzeitumgebung, die den Java-Bytecode in C-Code umwandelt, um diesen anschließend mit einem herkömmlichen Compiler zu übersetzen und mit einer Betriebssystembibliothek zusammenzubinden.

Bei der Umsetzung in C-Code wird zusätzlich zur Anwendung auch der Code der virtuellen Maschine erzeugt. Dabei wird von einem statischen System ausgegangen, zu dem keine Klassen dynamisch hinzugeladen werden können. Daher kennt der Compiler alle Klassen, aus denen die Anwendung aufgebaut ist, und kann auf dieser Basis globale Optimierungen anwenden, die bei der Betrachtung einzelner Klassen nicht möglich wären.

So werden zum Beispiel konstante Klassendaten direkt in den generierten Code eingefügt. Beim Aufruf der Objekterzeugung muss dann nicht erst die Größe des Objekts aus den Klassendaten ausgelesen, sondern kann direkt als konstanter Parameter für den Aufruf verwendet werden. Wenn die Objektgröße ansonsten nicht mehr benötigt wird, ist es möglich die Klassendaten entsprechend zu reduzieren. KESO wendet noch weitere Optimierungen an, die jedoch nicht Bestandteil dieser Arbeit sind und daher auch nicht im Detail betrachtet werden sollen.

Als Ausgangsbasis der Verwaltung dient der erzeugte C-Code. Hier sind bereits alle Optimierungen des KESO-Compilers enthalten und sowohl die Anwendung als auch die Betriebssystembibliothek liegen in einem einheitlichen Format vor. Der Verwalter kann nun vorhandene Funktionalität dynamisch verändern und anpassen, er kann aber auch komplett neue Funktionalität anbieten.

Neue Funktionen durch einen Verwalter

Mithilfe eines Verwalters kann das dynamische Nachladen von Code unterstützt werden. Das KESO-System verfügt über keinen Klassenlader, da das Hinzufügen neuer Klassen dem statischen Ansatz widersprechen würde. Der Verwalter kann diese Aufgabe jedoch übernehmen. Beim Hinzuladen neuer Klassen muss allerdings geprüft werden, ob die Optimierungen noch gültig sind, gegebenenfalls muss betroffener Code ausgetauscht werden. Der grundlegende Ablauf ist daher wie folgt: Zunächst

wird mithilfe des KESO-Compilers neuer Anwendungscode erzeugt, der die neue Klasse enthält und berücksichtigt. Anschließend untersucht der Verwalter, welche Teile des Codes sich verändert haben. Dann wird der alte Code durch den neuen ausgetauscht. Der Zustand des Systems bleibt dabei zunächst erhalten und wird in einem zweiten Schritt angepasst.

Durch den statischen Ansatz sind viele Datenstrukturen des Systems konstant und werden wie Code ausgetauscht. Um Veränderungen an der Objektstruktur festzustellen, wird der Aufbau der Datentypen untersucht. Hat sich die Struktur der Objekte verändert, so werden alle Objekte angepasst. Für die Anpassung der Objektstruktur muss eine spezielle Konvertierungsfunktion am Verwalter vorhanden sein, in der spezifisches Wissen über die Vorgehensweise des KESO-Systems enthalten ist.

Neben solchen grundlegenden Änderungen kann das System auch um Dienste erweitert werden, die KESO nicht anbietet. Beispielsweise bietet KESO keine Möglichkeit an, Informationen über die Typen zur Laufzeit abzufragen. Ein Laufzeittypsystem lässt sich jedoch gut als externer Dienst bereitstellen. Da nicht alle benötigten Klassendaten explizit im erzeugten C-Code vorhanden sind, kann der KESO-Compiler zusätzliche Tabellen mit den vollständigen Klasseninformationen erstellen. Auf dieser Basis können Anfragen über den Aufbau der Typen vom Verwalter beantwortet werden.

Anpassen existierender Funktionen

Am KESO-System kann auch das dynamische Austauschen von Bausteinen vorgestellt werden. Als Beispiel dient hierbei der Scheduler, der von der zugrunde gelegten Betriebssystembibliothek bereitgestellt wird. Zur Demonstration wurde die Betriebssystembibliothek JOSEK betrachtet, die Teil des KESO-Projekts ist und als Anpassungsschicht an andere Betriebssysteme dient. Diese Schicht enthält einen generischen, prioritätengesteuerten Scheduler, zu dem als Alternative ein spezialisierter Scheduler erstellt wurde, welcher nur zwei Threads unterstützt. Durch die Optimierung kann die Verwaltung der Prioritäten stark eingeschränkt werden, da die beiden Threads immer nur abwechselnd laufen können. Daher wurde auch die Threadsteuerung entsprechend vereinfacht. Der Verwalter integriert nun bei der Erstellung des initialen Abbildes eine Unterstützungsfunktion, die ihn benachrichtigt, sobald ein dritter Task aktiviert wird. Zu diesem Zeitpunkt wird die spezialisierte Version gegen die generische Version ausgetauscht. Bei der Installation des generischen Schedulers muss darauf geachtet werden, dass die Datenstrukturen mit dem aktuellen Zustand initialisiert werden. Dazu wurde eine spezielle Funktion am Verwalter implementiert, welche die Bedeutung der Datenfelder kennt und ihren Wert entsprechend dem aktuellen Zustand setzt. Tabelle 5.3 zeigt den Speicherbedarf des ursprünglichen Schedulers und der Spezialversion.

KESO Scheduler	.text	.rodata	.data	.bss	Total
für zwei Threads optimiert	780	280	61	0	**1121** Byte
generisch, prioritätengesteuert	1402	392	62	20	**1876** Byte

Tabelle 5.3: Speicherbedarf von verschiedenen Threadverwaltungen für KESO
Die Tabelle zeigt die Größen des Schedulers und der Threadsteuerung für KESO in zwei Varianten. Die Werte wurden an einer x86-Version von KESO ermittelt, da sich die Version für AVR-Mikrocontroller zum Zeitpunkt der Messungen (Mai 2008) noch im Aufbau befindet.

Ein weiteres Beispiel ist die Speicherbereinigung. KESO bietet zwei Varianten an. Einen einfachen "mark-and-sweep" Garbage Collector, der exklusiv laufen muss und eine inkrementelle Version, die mithilfe von Schreibbarrieren ein konsistentes Bild des Objektgraphen aufbaut. In Tabelle 5.4 sind die Größen der beiden Varianten aufgezeigt.

KESO Speicherbereinigung	.text	.rodata	.data	.bss	Total
einfacher GC (*coffee*)	775	15	1	1069	**1860** Byte
inkrementeller GC (*irr*)	2244	62	3	539	**2848** Byte

Tabelle 5.4: Speicherbedarf von verschiedenen Speicherbereinigungen für KESO
Die Tabelle zeigt die Größen der Speicherbereinigung für KESO in zwei Varianten. Die Werte wurden ebenfalls an der x86-Version von KESO ermittelt. Bei der einfachen Version ist ein relativ großer Speicherbereich vorgesehen, um die markierten Objekte aufzunehmen.

Anhand der Speicherbereinigung kann auch das Auslagern eines Dienstes demonstriert werden. Beim Auslagern können der gesamte Code und die Verwaltungsdaten vom Gerät entfernt werden. Allerdings muss die Speicherbereinigung etwas modifiziert werden, sodass sie am Verwalter auf einer Kopie des Speicherinhalts arbeiten kann. Die Speicherbereinigungsfunktion baut den Objektgraphen ausgehend von einigen festgelegten Startobjekten auf und verfolgt die enthaltenen Referenzen. Da der Speicher am Verwalter allerdings eine andere Basisadresse hat, können bei der Verfolgung von Objektreferenzen nicht direkt die Adressen verwendet werden. Die Speicherbereinigungsfunktion wurde daher angepasst und verwendet einen Korrekturwert, der zu den Adressen addiert wird. Der Korrekturwert entspricht dem Unterschied zwischen der ursprünglichen und der neuen Basisadresse.

Anhand der genannten Beispiele konnte gezeigt werden, dass die angesprochenen Möglichkeiten auch eingesetzt werden können.

5.6 Zusammenfassung

Dieses Kapitel widmete sich einigen Aspekten bei der Erstellung einer Kontrollschicht. Zunächst wurden Anhaltspunkte betrachtet, die zur Strategiefindung herangezogen werden können. Im Mittelpunkt standen dabei die Übertragungskosten beim Einsatz der Verwaltungsoperationen. Anschließend wurden zwei grundlegende Konzepte einer Kontrollschicht vorgestellt. Zur Verdeutlichung der Möglichkeiten eines Verwalters wurden anschließend die Komponenten einer Java Laufzeitumgebung betrachtet und die Anwendung verschiedener Konfigurationen anhand des Java-Systems KESO verdeutlicht.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde ein Unterstützungs- und Konfigurationssystem für Mikrocontroller vorgestellt, wobei kleine Mikrocontroller von größeren unterstützt werden. Dementsprechend werden den Knoten Rollen zugewiesen: verwaltete Knoten und Verwalter. Es wurden verschiedene Szenarios vorgestellt, in denen Verwalter entweder nur zeitweise zur Umkonfiguration oder dauerhaft für umfassende Unterstützung verfügbar sind. Ein Hauptaugenmerk dieser Arbeit lag dabei auf der Infrastruktur, die zur Verwaltung und Umkonfiguration benötigt wird. Zur Erfüllung der Dienste wird die verwaltete Software in kleine Einheiten, sogenannte Module, aufgeteilt. Dieser Prozess erfolgt universell und automatisch auf Basis von relokierbaren binären Objektdateien. Aus diesen Modulen können Softwarekonfigurationen erstellt werden. Dabei wird festgelegt, welche Module installiert und welche durch einen Fernaufruf genutzt werden. Weiterhin kann, durch ein Unterstützungsmodul auf dem Mikrocontroller, die Konfiguration auch zur Laufzeit verändert und angepasst werden. Besonderer Wert lag dabei auf einer minimalen Infrastruktur auf den verwalteten Knoten.

Die vorgestellten Mechanismen sind universell anwendbar, dennoch wird inhaltliches Wissen benötigt, um sie sinnvoll einzusetzen. Diesem Punkt widmete sich der zweite Teil der Arbeit. Hier wurde kurz auf die Kosten der Mechanismen eingegangen. Außerdem wurden zwei grundsätzlich verschiedene Vorgehensweisen beschrieben, wie die Umkonfiguration vonstattengehen kann: manuell oder automatisch. Anschließend wurden verschiedene Konfigurationsmöglichkeiten anhand einer Java-Laufzeitumgebung vorgestellt. Dabei wurden Varianten für einige Komponenten vorgeschlagen und herausgearbeitet, unter welchen Umständen sich eine Komponente zum Ausführen auf einem entfernten Knoten eignen könnte.

6.2 Erreichte Ziele

Zu Beginn der Arbeit wurden einige Anforderungen an das System gestellt. Nun soll betrachtet werden, wie sie von dem vorgestellten System erfüllt werden. Die Infrastruktur erlaubt die Anpassung und Reaktion auf veränderte Anforderungen auf verschiedene Weisen:

- Das System kann als Fernwartungssystem dienen, um Umstrukturierungen an kleinen Knoten vorzunehmen oder um ein Softwareupdate einzuspielen. Der betroffene Knoten nimmt dabei eine passive Rolle ein, das heißt, der Vorgang wird von außen, beispielsweise durch einen Administrator, angestoßen.
- Die Umstrukturierung kann auch aktiv in die Ablaufplanung eines kleinen Knotens mit einbezogen werden. Das verwaltete Gerät muss dann nicht für alle möglichen Situationen oder Ereignisse vorbereitet sein. Solange Kontakt zu einem Verwalter besteht, kann dieser bei Bedarf von dem Gerät beauftragt werden eine Umstrukturierung vorzunehmen und den kleinen Knoten so mit dem benötigen Code versorgen.
- Zusätzliche Unterstützung wird durch den Fernaufrufmechanismus geboten. Dieser erlaubt es, Aufgaben auf den Verwalter auszulagern und als externen Dienst zu nutzen. Mithilfe des dynamischen Fernwartungssystems können Aufgaben auch flexibel zur Laufzeit von einem Knoten auf den Verwalter verschoben werden.

Neben der Möglichkeit zur dynamischen Anpassung an veränderte Bedingungen wurden dabei auch einige Eigenschaften gefordert, die ebenfalls erfüllt wurden:

Ressourcenschonend Die Belastung und der Ressourcenverbrauch auf den betroffenen Knoten sollen minimal sein. Dieser Forderung wird durch eine asymmetrische Realisierung der Mechanismen nachgegangen: Alle nötigen Vorbereitungen, Anpassungen und Umwandlungen werden soweit möglich vom Verwalter durchgeführt. Sowohl der Fernaufruf als auch die Fernwartungsmechanismen benötigen daher nur minimale generische Funktionen auf dem verwalteten Knoten.

Unabhängigkeit Durch das Aufsetzen auf einen eigenen Modularisierungsprozess wurde zudem die Unabhängigkeit von speziell vorbereiteter Software erreicht. Das System arbeitet auf der Basis eines weitestgehend sprach- und architekturunabhängigen Objektdateiformats. Für die grundlegenden Operationen können alle notwendigen Informationen daraus gewonnen werden.

6.3 Wissenschaftlicher Beitrag

Im Rahmen dieser Forschungsarbeit wurden einige Verfahren entwickelt und untersucht, um die sich ergebende Problemstellungen anzugehen:

- Es wurde gezeigt, dass es möglich ist, aus relokierbaren Objektdateien eine hinreichend gute Modularisierung einer Anwendung vorzunehmen, die als Grundlage einer Verteilung dienen kann. Der Ansatz hat dabei den Vorteil, dass er auf beliebige existierende Software angewandt werden kann. Die Software muss nicht speziell vorbereitet werden. Zudem ist das Verfahren sprach- und weitestgehend architekturunabhängig.
- Es wurde ein asymmetrischer Fernaufruf- und Fernwartungsmechanismus entwickelt, bei dem die Vorbereitung und Nachbereitung der Daten und Parameter komplett durch den Dienstanbieter durchgeführt werden. Somit werden minimale Anforderungen an den Dienstnehmer gestellt und das Vorgehen eignet sich besonders für ressourcenbeschränkte Geräte.
- Es wurde gezeigt, dass Debuginformationen als Quelle für Typinformationen im Rahmen eines Fernaufrufs dienen können. Der Informationsgehalt der Daten hängt zwar von der verwendeten Programmiersprache ab, für die meisten Fälle sind die Informationen jedoch ausreichend.

- Es wurde eine Infrastruktur zur architekturübergreifenden Zustandsübertragung entwickelt, welche die Darstellung der Daten automatisch anpasst. Zur Realisierung werden Debuginformationen genutzt, welche detaillierte Angaben über die architekturspezifische Darstellung der Typen im Speicher enthalten.

6.4 Weiterführende Arbeiten

Als weiterführenden Arbeiten könnten einige Ergänzungen an dem existierenden System vorgenommen werden:

Automatische Bestimmung der Kosten Im Rahmen dieser Arbeit lag der Schwerpunkt auf den funktionalen Aspekten der Infrastruktur. Bei einem realen Einsatzszenario spielen jedoch die nicht-funktionalen Eigenschaften eine mindestens ebenso große Rolle. Als Grundlage zur realistischen Abschätzung der tatsächlichen Kosten und Auswirkungen beim Einsatz eines Verwalters müssen daher Daten zur Bestimmung der nicht-funktionalen Eigenschaften erfasst werden. Besonders relevant sind hier der Energie- und Zeitverbrauch einer Umkonfiguration. Genaue Zahlen sind jedoch immer von der eingesetzten Hardware abhängig, sodass sie für das jeweilige Szenario spezifisch erfasst werden müssen. Somit ist die Entwicklung eines Evaluierungs- oder Benchmarksystems zu prüfen, welches vor dem Einsatz eines Verwalters automatisch entsprechende Daten liefern kann.

Unterstützung für Klassen Als Erweiterung wäre auch eine verbesserte Unterstützung für objektorientierte Sprachen wie C++ denkbar. Im Rahmen dieser Arbeit wurde hauptsächlich mit Objektdateien gearbeitet, die aus C-Code erzeugt wurden. Bei der Verarbeitung von C++ sind keine grundsätzlichen Änderungen notwendig. Die Verwaltungsoperationen und der dynamische Binder können unverändert eingesetzt werden. Allerdings würde eine explizite Unterstützung des Klassenkonzepts die Verwendung des Systems erleichtern. Beachtet werden müssen beispielsweise der Aufruf der Klassenkonstruktoren beim Installieren neuer Klassen und die Anpassung der Funktionstabellen beim Austausch von Code. Realisieren könnte man die Unterstützung durch eine Zwischenschicht, welche die Mechanismen der Basisschicht nutzt und eine klassenorientierte Sicht für die Kontrollschicht anbietet.

Verwaltete Verwalter In dieser Arbeit wurden Knoten oft entweder als Verwalter oder als verwalteter Knoten dargestellt. Dabei wurde zu wenig auf einen fließenden Übergang zwischen diesen Klassen eingegangen. In Abschnitt 4.6.4.2 wird der Fall beschrieben, dass ein verwalteter Knoten durch einen Verwalter mit der Fähigkeit ausgestattet wird, spezifische Fernaufrufe zu beantworten. Bei der Realisierung wurde hierzu jedoch keine generische Infrastruktur verwendet, sondern speziell angepasste Module. Als Erweiterung könnte hierfür eine generische Infrastruktur entwickelt werden. Außerdem sollte man weitere Möglichkeiten betrachten, wie ein relativ kleiner Knoten teilweise Verwaltungsaufgaben übernehmen kann. Die Nutzung der vorhandenen Ressourcen könnte somit verbessert werden.

Zugriffsschutz In der vorgestellten Arbeit wurde nicht auf die möglichen Sicherheitsrisiken eingegangen, die das System birgt. Ohne geeignete Vorkehrungen könnten Angreifer als bösartige Verwalter den Speicherinhalt eines verwalteten Knotens auslesen oder verändern. Hiergegen sollten Maßnahmen untersucht werden, die beispielsweise einen Autorisierungsprozess, signierte Module und Nachrichten sowie verschlüsselte Verbindungen beinhalten könnten.

Neben Erweiterungen des Systems könnte der Ansatz auch in anderen Einsatzgebieten evaluiert werden:

Unterstützung mobiler Knoten Durch die Kooperation und Vernetzung mehrerer Verwalter könnte ein Unterstützungsangebot für mobile, verwaltete Knoten erstellt werden. Möchte man beispielsweise seinen PDA als generische Fernbedienung verwenden, so müssen immer die passenden Funktionen und Mechanismen vorhanden sein, um die Geräte in der Umgebung anzusprechen. Das Integrieren der aktuell benötigten Software könnte jeweils ein lokaler Verwalter durchführen. Dazu müsste das Abspeichern und Austauschen der aktuellen Konfiguration sowie der Transport dieser Daten zwischen den Verwaltern untersucht werden.

Partitionierung und Verteilung existierender Anwendungen Es sollte überprüft werden, inwieweit sich der Ansatz zur transparenten Partitionierung und Verteilung beliebiger Software nutzen lässt. Die Mechanismen und Operationen bieten das Potenzial zur Realisierung eines solchen Systems. Die Verteilung könnte durch einen genauen Verteilungsplan oder durch entsprechende Verteilungsregeln beschrieben werden [Bec03]. Eine generische Kontrollschicht könnte die Umsetzung der Verteilung realisieren und die Einhaltung der Regeln überwachen.

Entwicklung verteilter Anwendungen Als System zur transparenten Partitionierung könnte ein Verwalter auch ein wertvolles Werkzeug bei der Entwicklung verteilter Anwendungen sein. So können relativ einfach verschiedene Verteilungen evaluiert und die jeweiligen Kosten untersucht werden. Da dieser Vorgang während der Entwicklung erfolgt, ist genügend Wissen über den Aufbau der Software und die Abhängigkeiten der Module vorhanden, um beliebige Konfigurationen zu erstellen und zu testen. In diesem Rahmen könnte untersucht werden, ob eine Integration dieses Ansatzes in den Entwicklungsprozess Vorteile mit sich bringt.

Aspektweber Es könnte evaluiert werden, ob sich der Ansatz als dynamischer Aspektweber im Bereich der aspektorientierten Programmierung eignet. In diesem Bereich müssen Codestücke, sogenannte *Aspekte* in existierende Software eingefügt werden. Oft wird dies statisch vor der Übersetzung durchgeführt, es gibt jedoch auch Ansätze, Aspekte dynamisch zur Laufzeit hinzuzufügen beziehungsweise zu aktivieren [Gil07]. Meistens wird hierfür bereits zum Übersetzungszeitpunkt an den Stellen, an denen Aspekte wirken sollen, Code eingefügt, der in die Aspekte verzweigt. Es sollte untersucht werden, ob durch die Verwendung eines Verwalters Aspekte ohne solche Vorbereitung eingebracht werden können.

6.5 Abschließende Bemerkung

In der heutigen Zeit verändern sich die Anforderungen an technische Geräte sehr schnell. Daher ist die Möglichkeit nachträglich und dynamisch Änderungen vorzunehmen umso wichtiger. Nur so können Systeme wirtschaftlich den veränderten Anforderungen angepasst werden. Der vorgestellte Ansatz bietet Unterstützung beim Betrieb, der Wartung und sogar der Verwaltung von Mikrocontrollernetzen und ist daher ein Schritt in diese Richtung.

Literaturverzeichnis

[App06] APPLE COMPUTER, INC. (Hrsg.): *Mac OS X ABI Mach-O File Format Reference*. Cupertino, CA, USA: Apple Computer, Inc., Oktober 2006
(Zitiert auf Seite 47)

[ARM06] ARM LTD. (Hrsg.): *Cortex-A8 Technical Reference Manual*. Revision: r1p1. ARM Ltd., 2006
(Zitiert auf Seite 10)

[Atm07a] *WWW-Seiten von Atmel*. `http://www.atmel.com/`, September 2007. – ; Parametric Product Table of AVR 8-Bit RISC
(Zitiert auf Seite 1)

[Atm07b] ATMEL CORPORATION (Hrsg.): *ATmega128(L) Datasheet*. Dokumentnummer 2467, Revision P. San Jose, CA, USA: Atmel Corporation, August 2007
(Zitiert auf Seite 83)

[Bar81] BARRON, David W. (Hrsg.): *Pascal: The Language and Its Implementation*. New York, NY, USA : John Wiley & Sons, Inc., 1981. – ISBN 0471278351
(Zitiert auf Seite 10)

[Bec03] BECKER, Ulrich: *Verteilung von objektorientierten Anwendungen auf der Basis des Entwurfs-Modells*. Erlangen, Germany, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2003
(Zitiert auf Seite 132)

[Ber95] BERKELEY SOFTWARE DESIGN, INC.: a.out - format of executable binary files. In: *BSD Programmer's Manual*. 1995, Kapitel 5
(Zitiert auf Seite 44)

[BHA+05] BAUMANN, Andrew ; HEISER, Gernot ; APPAVOO, Jonathan ; SILVA, Dilma D. ; KRIEGER, Orran ; WISNIEWSKI, Robert W. ; KERR, Jeremy: Providing Dynamic Update in an Operating System. In: *Proceedings of the 2005 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2005, S. 279–291
(Zitiert auf Seite 15)

[BHS03] BOULIS, Athanassios ; HAN, Chih-Chieh ; SRIVASTAVA, Mani B.: Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In: *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (Mobisys '03)*. New York, NY, USA : ACM Press, Mai 2003, S. 187–200
(Zitiert auf Seite 12)

[CEE+01] CERPA, Alberto ; ELSON, Jeremy ; ESTRIN, Deborah ; GIROD, Lewis ; HAMILTON, Michael ; ZHAO, Jerry: Habitat Monitoring: Application Driver for Wireless Communications Technology. In: *ACM SIGCOMM Computer Communication Review* 31 (2001), April, Nr. 2 (supplement), S. 20–41. – DOI 10.1145/844193.844196. – ISSN 0146–4833
(Zitiert auf Seite 2)

[CES04] CULLER, David ; ESTRIN, Deborah ; SRIVASTAVA, Mani: Overview of Sensor Networks. In: *Special Issue in Sensor Networks, IEEE Computer* 37 (2004), August, Nr. 8, S. 41–49
(Zitiert auf Seiten 2 und 4)

[Che70] CHENEY, C. J.: A Nonrecursive List Compacting Algorithm. In: *Communications of the ACM* 13 (1970), November, Nr. 11, S. 677–678. – DOI 10.1145/362790.362798. – ISSN 0001–0782
(Zitiert auf Seite 112)

[CK06a] CHESHIRE, Stuart ; KROCHMAL, Marc ; APPLE COMPUTER, INC. (Hrsg.): *DNS-Based Service Discovery*. (draft-cheshire-dnsext-dns-sd-04.txt). Apple Computer, Inc., August 2006. – Internet-Draft
(Zitiert auf Seite 30)

[CK06b] CHESHIRE, Stuart ; KROCHMAL, Marc ; APPLE COMPUTER, INC. (Hrsg.): *Multicast DNS*. (draft-cheshire-dnsext-multicastdns-06.txt). Apple Computer, Inc., August 2006. – Internet-Draft
(Zitiert auf Seite 30)

[Col60] COLLINS, George E.: A Method for Overlapping and Erasure of Lists. In: *Communications of the ACM* 3 (1960), Dezember, Nr. 12, S. 655–657. – DOI 10.1145/367487.367501
(Zitiert auf Seite 112)

[COM95] MICROSOFT CORPORATION AND DIGITAL EQUIPMENT CORPORATION (Hrsg.): *The Component Object Model Specification*. Version 0.9. Microsoft Corporation and Digital Equipment Corporation, Oktober 1995. – MSDN Library, Specifications
(Zitiert auf Seite 37)

[Com99] COMPAQ COMPUTER CORPORATION (Hrsg.): *Object File / Symbol Table Format Specification*. Version 5.0. Compaq Computer Corporation, Juli 1999
(Zitiert auf Seite 46)

[Cro03] CROSSBOW TECHNOLOGY, INC. (Hrsg.): *Mote In-Network Programming User Reference*. Version 20030315. Crossbow Technology, Inc., 2003
(Zitiert auf Seite 13)

[Del96] DELORIE SOFTWARE: DJGPP'S COFF Specification. In: *DJGPP documentation*. 1996. – `http://delorie.com/djgpp/doc/coff/`
(Zitiert auf Seiten 46 und 58)

[DEMS00] DEBRAY, Saumya K. ; EVANS, William ; MUTH, Robert ; SUTTER, Bjorn D.: Compiler Techniques for Code Compaction. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22 (2000), Nr. 2, S. 378–415. – DOI 10.1145/349214.349233. – ISSN 0164–0925
(Zitiert auf Seite 11)

[DFEV06] DUNKELS, Adam ; FINNE, Niclas ; ERIKSSON, Joakim ; VOIGT, Thiemo: Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks. In: *Proceedings of the 4th ACM International Conference on Embedded Networked Sensor Systems (SenSys'06)*. New York, NY, USA : ACM Press, November 2006. – ISBN 1–59593–343–3, S. 15–28
(Zitiert auf Seite 14)

[DGV04] DUNKELS, Adam ; GRÖNVALL, Björn ; VOIGT, Thiemo: Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In: *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*. Washington, DC, USA : IEEE Computer Society Press, November 2004. – ISBN 0–7695–2260–2, S. 455–462
(Zitiert auf Seiten 14 und 15)

[DR98] DIMITROV, Bozhidar ; REGO, Vernon: Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. In: *IEEE Transactions on Parallel and Distributed Systems* 9 (1998), Mai, Nr. 5, S. 459–469. – DOI 10.1109/71.679216. – ISSN 1045–9219
(Zitiert auf Seite 17)

[Eag07] EAGER, Michael J.: *Introduction to the DWARF Debugging Format*. `http://www.dwarfstd.org/Debugging%20using%20DWARF.pdf`. Version: Februar 2007
(Zitiert auf Seite 58)

[Fsg05] FREE STANDARDS GROUP (Hrsg.): *DWARF Debugging Information Format, Version 3*. Free Standards Group, Dezember 2005
(Zitiert auf Seite 58)

[FY69] FENICHEL, Robert R. ; YOCHELSON, Jerome C.: A LISP Garbage-Collector for Virtual-Memory Computer Systems. In: *Communications of the ACM* 12 (1969), November, Nr. 11, S. 611–612. – DOI 10.1145/363269.363280. – ISSN 0001–0782
(Zitiert auf Seite 112)

[Gil07] GILANI, Wasif: *A Family-Based Dynamic Aspect Weaver*. Erlangen, Germany, Friedrich-Alexander-Universität Erlangen-Nürnberg, Diss., 2007
(Zitiert auf Seite 132)

[Gir88] GIRCYS, Gintaras R.: *Understanding and Using COFF*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., November 1988. – ISBN 0–937175–31–5
(Zitiert auf Seiten 46 und 58)

[GLS75] GUY L. STEELE, Jr.: Multiprocessing Compactifying Garbage Collection. In: *Communications of the ACM* 18 (1975), September, Nr. 9, S. 495–508. – DOI 10.1145/361002.361005. – ISSN 0001–0782
(Zitiert auf Seite 113)

[GLv+03] GAY, David ; LEVIS, Philip ; VON BEHREN, Robert ; WELSH, Matt ; BREWER, Eric ; CULLER, David: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*. New York, NY, USA : ACM Press, Juni 2003. – ISBN 1–58113–662–5, S. 1–11
(Zitiert auf Seite 37)

[GSK04] GONDOW, Katsuhiko ; SUZUKI, Tomoya ; KAWASHIMA, Hayato: Binary-Level Lightweight Data Integration to Develop Program Understanding Tools for Embedded Software in C. In: *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*. Washington, DC, USA : IEEE Computer Society Press, 2004. – ISBN 0–7695–2245–9, S. 336–345
(Zitiert auf Seite 41)

[Han72] HANSEN, Per B.: Structured Multiprogramming. In: *Communications of the ACM* 15 (1972), Juli, Nr. 7, S. 574–578. – DOI 10.1145/361454.361473. – ISSN 0001–0782
(Zitiert auf Seite 116)

[HC04] HUI, Jonathan W. ; CULLER, David: The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In: *Proceedings of the 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*. New York, NY, USA : ACM Press, November 2004. – ISBN 1–58113–879–2, S. 81–94
(Zitiert auf Seite 13)

[Hec77] HECHT, Matthew S.: *Flow Analysis of Computer Programs*. New York, NY, USA : Elsevier Science Inc., September 1977 (Programming Language Series). – ISBN 0444002162
(Zitiert auf Seite 66)

[Her87] HERRTWICH, Ralf G.: *Fernwartung verteilter Applikationen im Masseneinsatz*. Berlin, Germany, Technische Universität Berlin, Diss., Januar 1987
(Zitiert auf Seite 17)

[HGB78] HENRY G. BAKER, Jr.: List Processing in Real Time on a Serial Computer. In: *Communications of the ACM* 21 (1978), April, Nr. 4, S. 280–294. – DOI 10.1145/359460.359470. – ISSN 0001–0782
(Zitiert auf Seite 113)

[Hic01] HICKS, Michael: *Dynamic Software Updating*. Philadelphia, PA, USA, University of Pennsylvania, Diss., August 2001
(Zitiert auf Seite 12)

[HKS+05] HAN, Chih-Chieh ; KUMAR, Ram ; SHEA, Roy ; KOHLER, Eddie ; SRIVASTAVA, Mani: A Dynamic Operating System for Sensor Nodes. In: *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (Mobisys '05)*. New York, NY, USA : ACM Press, Juni 2005. – ISBN 1–931971–31–5, S. 163–176
(Zitiert auf Seite 14)

[HM76] HUNT, J. W. ; MCILROY, M. D.: An Algorithm for Differential File Comparison / Bell Telephone Laboratories. 1976 (41). – Forschungsbericht
(Zitiert auf Seite 99)

[HSW+00] HILL, Jason ; SZEWCZYK, Robert ; WOO, Alec ; HOLLAR, Seth ; CULLER, David ; PISTER, Kristofer: System Architecture Directions for Networked Sensors. In: *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. New York, NY, USA : ACM Press, November 2000. – ISBN 1–58113–317–0, S. 93–104
(Zitiert auf Seiten 13, 37 und 69)

[HW67] HADDON, Bruce K. ; WAITE, William M.: A Compaction Procedure for Variable Length Storage Elements. In: *Computer Journal* (1967), August, Nr. 10, S. 162–165. – ISSN 0010–4620
(Zitiert auf Seite 112)

[Ibm04] INTERNATIONAL BUSINESS MACHINES CORPORATION (Hrsg.): *AIX 5L Version 5.2 Documentation — Files Reference*. 7. Auflage. International Business Machines Corporation, August 2004
(Zitiert auf Seite 46)

[IEE98] IEEE (Hrsg.): *1451.2-1997, IEEE Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*. New York, NY, USA: IEEE, September 1998
(Zitiert auf Seite 29)

[Inf05] INFINEON TECHNOLOGIES AG (Hrsg.): *TriCore 1 User's Manual, Volume 2: v1.3 Instruction Set*. München, Germany: Infineon Technologies AG, Februar 2005
(Zitiert auf Seite 10)

[ISO05] ISO/IEC: *ISO/IEC 9899:1999: Programming languages – C*. Januar 2005
(Zitiert auf Seite 41)

[JC04] JEONG, Jaein ; CULLER, David: Incremental Network Programming for Wireless Sensors. In: *Proceedings of the 1st IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON '04)*, IEEE Computer Society Press, Oktober 2004. – ISBN 0–7803–8796–1, S. 25–33
(Zitiert auf Seite 13)

[JKB03] JEONG, Jaein ; KIM, Sukun ; BROAD, Alan: Network Reprogramming / University of California at Berkeley. Berkeley, CA, USA, August 2003. – Forschungsbericht
(Zitiert auf Seite 13)

[JOW+02] JUANG, Phlio ; OKI, Hidekazu ; WANG, Yong ; MARTONOSI, Margaret ; PEH, Li-Shiuan ; RUBENSTEIN, Daniel: Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. New York, NY, USA : ACM Press, Oktober 2002. – ISBN 1–58113–574–2, S. 96–107
(Zitiert auf Seiten 2 und 3)

[KKMS97] KIROVSKI, Darko ; KIN, Johnson ; MANGIONE-SMITH, William H.: Procedure Based Program Compression. In: *Proceedings of the 30th annual ACM/IEEEInternational Symposium on Microarchitecture (MICRO 30)*. Washington, DC, USA : IEEE Computer Society Press, 1997. – ISBN 0–8186–7977–8, S. 204–213
(Zitiert auf Seite 11)

[KP05a] KOSHY, Joel ; PANDEY, Raju: Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In: *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, IEEE Computer Society Press, 2005. – ISBN 0–7803–8801–1, S. 354–365
(Zitiert auf Seite 13)

[KP05b] KOSHY, Joel ; PANDEY, Raju: VM*: Synthesizing Scalable Runtime Environments for Sensor Networks. In: *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys'05)*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–054–X, S. 243–25
(Zitiert auf Seiten 12, 14 und 106)

[LC02] LEVIS, Philip ; CULLER, David: Maté: A Tiny Virtual Machine for Sensor Networks. In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. New York, NY, USA : ACM Press, Oktober 2002. – ISBN 1–58113–574–2, S. 85–95
(Zitiert auf Seite 12)

[LCP+05] LUA, Eng K. ; CROWCROFT, Jon ; PIAS, Marcelo ; SHARMA, Ravi ; LIM, Steven: A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. In: *IEEE Communications Surveys & Tutorials* 7 (2005), Nr. 2, S. 72–93. – ISSN 1553–877X
(Zitiert auf Seite 29)

[Lev99] LEVINE, John R.: *Linkers and Loaders*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., Oktober 1999. – ISBN 1–55860–496–0
(Zitiert auf Seite 43)

[LRW+05] LIU, Hongzhou ; ROEDER, Tom ; WALSH, Kevin ; BARR, Rimon ; SIRER, Emin G.: Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. In: *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (Mobisys '05)*. New York, NY, USA : ACM Press, 2005. – ISBN 1–931971–31–5, S. 1–5
(Zitiert auf Seiten 12 und 17)

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *The Java Virtual Machine Specification Second Edition*. Addison-Wesley, 1999. – ISBN 0–201–43294–3
(Zitiert auf Seiten 116 und 119)

[Mac00] MACDONALD, Joshua P.: *File System Support for Delta Compression*, University of California at Berkeley, Dep. of Electr. Eng. and Comp. Sc., Diplomarbeit, 2000
(Zitiert auf Seite 99)

[McC60] MCCARTHY, John: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. In: *Communications of the ACM* 3 (1960), April, Nr. 4, S. 184–195. – DOI 10.1145/367177.367199. – ISSN 0001–0782
(Zitiert auf Seite 112)

[MER06] Jet Propulsion Laboratory: *NASA Mars Team Teaches Old Rovers New Tricks to Kick Off Year Four*. Pasadena, CA, USA : Pressemitteilung, Dezember 2006. – Nr. 2006-152
(Zitiert auf Seite 3)

[MGL+06] MARRÓN, Pedro J. ; GAUGER, Matthias ; LACHENMANN, Andreas ; MINDER, Daniel ; SAUKH, Olga ; ROTHERMEL, Kurt: FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In: *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)* Bd. 3868. Berlin, Germany : Springer-Verlag, Februar 2006 (Lecture Notes in Computer Science). – ISBN 3–540–32158–6, S. 212–227
(Zitiert auf Seite 14)

[Mic05] MICROSOFT CORPORATION (Hrsg.): *Description of the .PDB files and of the .DBG files*. Revision 3.0. Microsoft Corporation, August 2005. – KB121366
(Zitiert auf Seite 58)

[Mic06] MICROSOFT CORPORATION (Hrsg.): *Microsoft Portable Executable and Common Object File Format Specification*. Revision 8.0. Microsoft Corporation, Mai 2006
(Zitiert auf Seite 47)

[Mic07] MICROSOFT CORPORATION (Hrsg.): *Microsoft Interface Definition Language*. MSDN. Microsoft Corporation, Januar 2007
(Zitiert auf Seiten 89 und 90)

[MIS98] MISRA (Hrsg.): *MISRA-C:1998 - Guidelines For The Use Of The C Language In Vehicle Based Software*. MISRA, April 1998
(Zitiert auf Seite 67)

[MIS04] MISRA (Hrsg.): *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. MISRA, Oktober 14, 2004
(Zitiert auf Seite 67)

[MKM06] MENAPACE, Julia ; KINGDON, Jim ; MACKENZIE, David ; FREE SOFTWARE FOUNDATION (Hrsg.): *The "stabs" debug format*. Boston, MA, USA: Free Software Foundation, November 2006
(Zitiert auf Seite 57)

[MLM+05] MARRÓN, Pedro J. ; LACHENMANN, Andreas ; MINDER, Daniel ; HÄHNER, Jörg ; SAUTER, Robert ; ROTHERMEL, Kurt: TinyCubus: A Flexible and Adaptive Framework for Sensor Networks. In: *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, IEEE Computer Society Press, Januar 2005, S. 278–289
(Zitiert auf Seite 14)

[Muc97] MUCHNICK, Steven S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., August 1997. – ISBN 1–55860–320–4
(Zitiert auf Seite 10)

[Nea02] NEABLE, Craig: The .NET Compact Framework. In: *IEEE PERVASIVE computing* 1 (2002), Oktober, Nr. 4, S. 84–87
(Zitiert auf Seite 6)

[Nel81] NELSON, Bruce J.: *Remote procedure call*. Pittsburgh, PA, USA, Carnegie Mellon University, Diss., 1981. – Tech. Report Nr.: CMU-CS-81-119
(Zitiert auf Seite 16)

[Nok04] NOKIA CORPORATION (Hrsg.): *Symbian OS: Coding Conventions in C++*. Version 1.0. Nokia Corporation, Mai 2004
(Zitiert auf Seite 90)

[NRC01] NATIONAL RESEARCH COUNCIL, Commitee on Networked Systems of Embedded C. (Hrsg.): *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. Washington, DC, USA : National Academy Press, Oktober 2001. – ISBN 0–309–07568–8
(Zitiert auf Seite 2)

[Oec07] OECHSLEIN, Benjamin: *Realisierung eines dynamischen Code-Installationssystems für AVR Mikrocontroller*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Inst. f. Informatik 4, Studienarbeit, März 2007. – SA-I4-2007-02
(Zitiert auf Seite 88)

[OH98] OTHMAN, Mazliza ; HAILES, Stephen: Power Conservation Strategy for Mobile Computers Using Load Sharing. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 2 (1998), Januar, Nr. 1, S. 44–51. – DOI 10.1145/584007.584011. – ISSN 1559–1662
(Zitiert auf Seite 101)

[Omg00] OBJECT MANAGEMENT GROUP (Hrsg.): *Mobile Agent Facility Specification*. Version 1.0. Object Management Group, Januar 2000. – formal/2000-01-02
(Zitiert auf Seite 16)

[Omg02] OBJECT MANAGEMENT GROUP (Hrsg.): *Minimum CORBA Specification*. Version 1.0. Object Management Group, August 2002
(Zitiert auf Seite 37)

[Omg04] OBJECT MANAGEMENT GROUP (Hrsg.): *CommonObject Request Broker Architecture: Core Specification*. Version 3.0.3. Object Management Group, März 2004
(Zitiert auf Seite 89)

[Omg06] OBJECT MANAGEMENT GROUP (Hrsg.): *CORBA Component Model Specification*. Version 4.0. Object Management Group, April 2006. – formal/06-04-01
(Zitiert auf Seite 37)

[OSE05] OSEK/VDX STEERING COMMITTEE (Hrsg.): *OSEK/VDX Operating System Specification 2.2.3*. OSEK/VDX steering committee, Februar 2005
(Zitiert auf Seite 3)

[Pan68] PANKHURST, R. J.: Operating Systems: Program Overlay Techniques. In: *Communications of the ACM* 11 (1968), Februar, Nr. 2, S. 119–125. – DOI 10.1145/362896.362923. – ISSN 0001–0782
(Zitiert auf Seite 11)

[PE03] PITZEK, Stefan ; ELMENREICH, Wilfried: Configuration and Management of a Real-Time Smart Transducer Network. In: *Proceedings of the IEEE Conference on Emerging Technologies and and Factory Automation (ETFA '03)*, IEEE Computer Society Press, 2003. – ISBN 0–7803–7937–3, S. 407–414
(Zitiert auf Seite 29)

[PLDM02] PALM, Jeffrey ; LEE, Han ; DIWAN, Amer ; MOSS, J. Eliot B.: When to Use a Compilation Service? In: *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'02)*. New York, NY, USA : ACM Press, Juni 2002. – ISBN 1–58113–527–0, S. 194–203
(Zitiert auf Seiten 101 und 121)

[QL91] QUONG, Russell W. ; LINTON, Mark A.: Linking Programs Incrementally. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1991), Januar, Nr. 1, S. 1–20. – DOI 10.1145/114005.102804. – ISSN 0164–0925
(Zitiert auf Seite 68)

[RL03] REIJERS, Niels ; LANGENDOEN, Koen: Efficient Code Distribution in Wireless Sensor Networks. In: *Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–764–8, S. 60–67
(Zitiert auf Seite 13)

[RMM+02] REICHARDT, Dirk ; MIGLIETTA, Maurizio ; MORETTI, Lino ; MORSINK, Peter ; SCHULZ, Wolfgang: CarTALK 2000 — Safe and Comfortable Driving Based Upon Inter-Vehicle-Communication. In: *Proceedings of the IEEE Intelligent Vehicle Symposium* Bd. 2, 2002, S. 545–550
(Zitiert auf Seite 2)

[RRPK98] RUDENKO, Alexey ; REIHER, Peter ; POPEK, Gerald J. ; KUENNING, Geoffrey H.: Saving Portable Computer Battery Power through Remote Process Execution. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 2 (1998), Januar, Nr. 1, S. 19–26. – DOI 10.1145/584007.584008. – ISSN 1559–1662
(Zitiert auf Seite 101)

[SCK06] SEKAR, Kiren ; CHESHIRE, Stuart ; KROCHMAL, Marc ; SHARPCAST, INC. (Hrsg.): *DNS Long-Lived Queries*. (draft-sekar-dns-llq-01.txt). Sharpcast, Inc., August 2006. – Internet-Draft
(Zitiert auf Seite 30)

[Sea00] SEAL, David (Hrsg.): *ARM Architecture Reference Manual*. Second Edition. Addison-Wesley, Dezember 2000. – ISBN 0201737191
(Zitiert auf Seite 9)

[SGGB99] SIRER, Emin G. ; GRIMM, Robert ; GREGORY, Arthur J. ; BERSHAD, Brian N.: Design and Implementation of a Distributed Virtual Machine for Networked Computers. In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*. New York, NY, USA : ACM Press, Dezember 12–15, 1999. – ISBN 1–58113–140–2, S. 202–216
(Zitiert auf Seite 17)

[SHC+04] SHNAYDER, Victor ; HEMPSTEAD, Mark ; CHEN, Bor rong ; ALLEN, Geoff W. ; WELSH, Matt: Simulating the Power Consumption of Large-Scale Sensor Network Applications. In: *Proceedings of the 2nd ACM International Conference on Embedded Networked Sensor Systems (SenSys'04)*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–879–2, S. 188–200
(Zitiert auf Seite 98)

[SLM98] SCHMIDT, Douglas C. ; LEVINE, David L. ; MUNGEE, Sumedh: The Design of the TAO Real-Time Object Request Broker. In: *Computer Communications* 21 (1998), April, Nr. 4, S. 149–157
(Zitiert auf Seite 37)

[SPMC04] SZEWCZYK, Robert ; POLASTRE, Joseph ; MAINWARING, Alan ; CULLER, David: Lessons from a Sensor Network Expedition. In: *Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN 2004)* Bd. 2920. Heidelberg : Springer, Januar 2004 (Lecture Notes in Computer Science). – ISBN 978–3–540–20825–9, S. 307–322
(Zitiert auf Seite 3)

[SS+96] SMITH, Steve ; SPALDING, Dan u. a. ; MICROSOFT CORPORATION (Hrsg.): *VC5.0 Symbolic Debug Information Specification*. Revision 5. Microsoft Corporation, September 1996
(Zitiert auf Seiten 47 und 58)

[SSTP00] SMITH, Todd ; SRINIVAS, Suresh ; TOMSICH, Philipp ; PARK, Jinpyo: Practical Experiences with Java Compilation. In: *High Performance Computing – HiPC 2000*. Berlin, Germany : Springer-Verlag, 2000 (Lecture Notes in Computer Science). – ISBN 978–3–540–41429–2, S. 149–157
(Zitiert auf Seite 108)

[Sun97] SUN MICROSYSTEMS: *JavaBeans API Specification*. Version 1.01. Mountain View, CA, USA, August 1997
(Zitiert auf Seite 37)

[Sun01] SUN MICROSYSTEMS: *Java Object Serialization Specification*. Revision 1.5.0. Palo Alto, CA, USA, 2001
(Zitiert auf Seite 94)

[Sun02] SUN MICROSYSTEMS: *Connected Device Configuration Specification (CDC)*. Version: 1.0a. Santa Clara, CA, USA, März 2002. – `http://java.sun.com/products/cdc/`
(Zitiert auf Seite 105)

[Sun03a] SUN MICROSYSTEMS: *Connected Limited Device Configuration (CLDC) Specification*. Version: 1.1. Santa Clara, CA, USA, März 2003. – `http://java.sun.com/products/cldc/`
(Zitiert auf Seite 105)

[Sun03b] SUN MICROSYSTEMS: *Java Card Platform: Virtual Machine Specification*. Version 2.2.1. Santa Clara, CA, USA, Oktober 2003
(Zitiert auf Seite 105)

[Sun04] SUN MICROSYSTEMS: *Java Remote Method Invocation Specification*. Revision 1.10. Santa Clara, CA, USA, 2004
(Zitiert auf Seite 16)

[Sun06] SUN MICROSYSTEMS: *JSR 220: Enterprise JavaBeans*. Version 3.0. Santa Clara, CA, USA, Mai 2006
(Zitiert auf Seite 37)

[Szy98] SZYPERSKI, Clemens: *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Dezember 1998. – ISBN 0–201–17888–5
(Zitiert auf Seite 36)

[Tan78] TANENBAUM, Andrew S.: Implications of Structured Programming for Machine Architecture. In: *Communications of the ACM* 21 (1978), März, Nr. 3, S. 237–246. – DOI 10.1145/359361.359454. – ISSN 0001–0782
(Zitiert auf Seite 10)

[TIS93] TIS COMMITTEE: Microsoft Symbol and Type Information. In: *Tool Interface Standard (TIS) Formats Specification for Windows*. Version 1.0. 1993
(Zitiert auf Seiten 47 und 58)

[TIS95] TIS COMMITTEE (Hrsg.): *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. TIS Committee, Mai 1995
(Zitiert auf Seite 47)

[TM07] THOMPSON, Donald ; MILES, Rob: *Embedded Programming with the Microsoft .NET Micro Framework*. Redmond, WA, USA : Microsoft Press, Juni 2007. – ISBN 0–7356–2365–1
(Zitiert auf Seite 6)

[TR71] THOMPSON, Kenneth ; RITCHIE, Dennis M.: a.out – assembler and link editor output. In: *Unix Programmer's Manual*. 1. Auflage. 1971, Kapitel 5
(Zitiert auf Seite 44)

[TS03] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Verteilte Systeme*. Pearson Studium, April 2003. – ISBN 3–8273–7057–4
(Zitiert auf Seite 16)

[Tur02] TURLEY, Jim: *The Two Percent Solution*. embedded.com, Dezember 2002
(Zitiert auf Seite 1)

[UPn03] UPNP FORUM (Hrsg.): *UPnP Device Architecture 1.0*. UPnP Forum, Dezember 2003
(Zitiert auf Seite 30)

[Wal99] WALDO, Jim: The Jini Architecture for Network-Centric Computing. In: *Communications of the ACM* 42 (1999), Juli, Nr. 7, S. 76–82. – DOI 10.1145/306549.306582. – ISSN 0001–0782
(Zitiert auf Seite 30)

[WC92] WOLFE, Andrew ; CHANIN, Alex: Executing Compressed Programs on an Embedded RISC Architecture. In: *Proceedings of the 25th annual International Symposium on Microarchitecture (MICRO 25)*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1992. – ISBN 0–8186–3175–9, S. 81–91
(Zitiert auf Seite 11)

[Whi01] WHITE, James: An Introduction to Java 2 Micro Edition (J2ME); Java in Small Things. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*. Washington, DC, USA : IEEE Computer Society Press, 2001. – ISBN 0–7695–1050–7, S. 724–725
(Zitiert auf Seite 6)

[WSSP07] WAWERSICH, Christian ; STILKERICH, Michael ; SCHRÖDER-PREIKSCHAT, Wolfgang: An OSEK/VDX-based Multi-JVM for Automotive Appliances. In: *Embedded System Design: Topics, Techniques and Trends*. Boston, MA, USA : Springer-Verlag, Mai 2007 (IFIP International Federation for Information Processing). – ISBN 978–0–387–72257–3, S. 85–96
(Zitiert auf Seiten 106 und 126)