Lehrstuhl für Realzeit-Computersysteme

Methoden der Metaprogrammierung zur Rekonfiguration von Software eingebetteter Systeme

Thomas Maier-Komor

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. sc. techn. (ETH) A. Herkersdorf
Prüfer der Dissertation:	1. Univ.-Prof. Dr.-Ing. G. Färber
	2. Univ.-Prof. Dr. rer. nat. Dr. rer. nat. habil. U. Baumgarten

Die Dissertation wurde am 27.6.2006, bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 8.12.2006 angenommen.

Abstract

Der Entwurf von Software für eingebettete Systeme wird sowohl durch die Systemumgebung als auch durch das System selbst stark beeinflusst. Beide Faktoren reduzieren die Wiederverwendbarkeit und die Erweiterbarkeit der Software in erheblichem Maße. Insbesondere können wirtschaftliche Überlegungen mitunter hohe Anforderungen an das Design stellen. Eine Lösung dieser Problematik kann nur mit klar definierten Abstraktionsebenen und geeigneten Schnittstellen zur Integration ermöglicht werden.

Mit MetaC wird in dieser Arbeit eine Spracherweiterung vorgestellt, die neue Konzepte bietet, um die speziellen Anforderungen querschneidender Strukturen von eingebetteter Software anzusprechen. Insbesondere werden Methoden zur Verbesserung der Wiederverwendbarkeit, Erweiterbarkeit und Abstraktion von Software für eingebettete System vorgestellt. Die zugrundeliegende Methodik basiert auf der Idee der Metaprogrammierung und benutzt spezielle Mechanismen zur Strukturanalyse und Synthese des Quelltextes. Voraussetzung für die Applikation dieses Verfahrens ist, dass als integrierende Schnittstelle für alle Teilmodule der Software die Programmiersprache C verwendet wird, die gleichzeitig die Grundlage für MetaC bildet.

Zur funktionalen Erweiterung von bestehendem Quelltext wird eine Methodik präsentiert, die die bestehende Semantik berücksichtigen kann. Durch besondere Verfahren zur Überprüfung von Randbedingungen und zur Abstraktion werden die Portabilität und die Wiederverwendbarkeit gesteigert. Dadurch wird die Applikationssoftware unabhängiger von der zugrundeliegenden Betriebssoftware. Die vorgestellten Maßnahmen sind somit geeignet, den Lebenszyklus von Software zu verlängern und außerdem die Entwicklungszeit zu verkürzen, indem eine einfache Rekonfiguration der Software für die verschiedenen Entwicklungsphasen möglich wird.

An mehreren Beispielen werden die neuen Möglichkeiten beschrieben, mit denen C Quelltexte rekonfiguriert, refaktorisiert, an neue Anforderungen anpasst und erweitert. Dabei werden die Vorteile insbesondere am Applikationsumfeld der Echtzeitsysteme herausgearbeitet. Da ein Großteil der Projekte aus der Domäne der eingebetteten Systeme C Quelltexte zur Implementierung von Software verwendet, ist die hier vorgestellte Methodik in einem breiten Spektrum einsetzbar.

Danksagung

Die vorliegende Dissertation ist Rahmen meiner Tätigkeit am Lehrstuhl für Realzeit-Computersysteme der TU-München entstanden. Mein besonderer Dank gilt *Herrn Prof. Dr.-Ing. Färber*, dem Ordinarius des Lehrstuhls. Durch sein uneingeschränktes Vertrauen in meine Ideen, die Förderung der individuellen Fähigkeiten und die Schaffung eines interdisziplinär geprägten Klimas, hat er maßgeblich zum Gelingen dieser Arbeit beigetragen. Insbesondere wusste er stets durch präzise Fragen meine Gedanken in neue Richtungen zu lenken und hat so dem Inhalt eine ausgewogene Mischung aus Theorie und Praxis gegeben.

Herrn Prof. Dr. Baumgarten danke ich sehr herzlich für die Übernahme des Korreferats, seinem Interesse an dieser Arbeit und der freundlichen Unterstützung.

Meinen Kollegen möchte ich für die gute Zusammenarbeit und die zahlreichen theoretischen und anwendungsbezogenen Diskussionen danken, die mich immer wieder auf neue Ideen gebracht haben und mir ganz nebenbei Einblick in eine Fülle von Themenkomplexe gegeben haben. Auch die logistische Unterstützung durch Verwaltung und Technik, die stets spontan und unbürokratisch die Probleme des Alltags gelöst haben, hat mir sehr geholfen und das Lehrstuhlleben angenehm gestaltet. Allen Studenten, die mich durch Ihre Mitarbeit an dieser Arbeit unterstützt haben, danke ich recht herzlich für Ihre Beiträge.

Nicht zuletzt gilt mein Dank auch meinen Eltern, die mich stets in allen Lebenslagen nach besten Kräften unterstützt haben und mir den eingeschlagenen Ausbildungs- und Lebensweg ermöglicht haben.

Thomas Maier-Komor, in München, im Jahr 2006.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AOP	Aspektorientierte Programmierung
ABC	Abstract Base Class
ABI	Application Binary Interface
API	Application Programming Interface
AST	Abstrakter Syntax Baum
BNF	Backus-Naur-Form
CAN	Controller Area Network
CASE	Computer Aided Software Engineering
EBNF	Erweiterte Backus-Naur-Form
ESP	Expression Structure Pattern
GAG	Gerichteter Azyklischer Graph
OOD	Objektorientiertes Design
OOP	Objektorientierte Programmierung
OS	Operating-System
PC	Personal Computer
POSIX	Portable Operating System Interface
QSM	Quelltext Strukturmuster
RT	Real-Time
RTOS	Real-Time Operating-System
SDL	Specification and Description Language
SSP	Statement Structure Pattern
TPU	Time Processing Unit
TSP	Type-Definition Structure Pattern
WCET	Worst-Case Execution Time

Teil I.

Einführung

1. Einführung

Der Entwurf von Software für eingebettete Systeme unterscheidet sich deutlich von dem für normale Arbeitsplatzrechner. Zum einen variiert das Zielsystem bezogen auf die eingesetzte Prozessorarchitektur und das Betriebssystem in Abhängigkeit von der Applikation hier deutlich stärker. Zum anderen legen die Systemumgebung und betriebswirtschaftliche Überlegungen oft zusätzliche Randbedingungen fest, die sich auf den Entwurf der Software auswirken.

Den gewichtigsten Einfluss auf die Entwicklung von Software für eingebettete Systeme haben die domänenspezifischen Anforderungen der jeweiligen Applikation. Diese können sich mitunter durchaus widersprechen und schwierig vereinbar sein. In Abbildung 1.1 sind Konkurrenz und Korrelationen zwischen einer Auswahl von Anforderungen vereinfacht dargestellt. Für die unterschiedlichen Applikationsfelder sind deswegen über die Jahre verschiedene Konzepte entstanden, um den Entwurf von Software zu vereinfachen, Fehler zu vermeiden und Kosten zu reduzieren. Eine wichtige Strategie ist hierbei die Abstraktion von domänenspezifischen Softwaremustern zu einfach handhabbaren Konzepten, die die darunter liegende Komplexität vor dem Entwickler verbergen.

Die wichtigsten Ansätze, um eine Abstraktion zu realisieren sind Konzepte domänenspezifischer Spracherweiterungen oder Sprachen sowie die Integration und Kapselung des Entwurfsprozesses in Computer Aided Software Engineering (CASE) Tools. Hierbei werden häufig verwendete Funktionen, Interaktionsmechanismen und wiederkehrende Softwarestrukturen als einfach zu verwendende Schlüsselworte oder graphische Symbole abstrahiert. Dadurch vereinfacht sich die Anwendung, die Implementierung wird durchgehend konsistent und Fehler könne so vermieden werden. Diese Verfahren bringen den gewünschten Erfolg bezüglich der gegebenen Anforderungen, sind aber relativ aufwändig in der Realisierung und wenig flexibel in der Anwendung. Somit ist ihr Einsatz auf die konkreten Applikationsfelder beschränkt, für die sie entworfen wurden.

Diese Arbeit stellt eine Spracherweiterung vor, die diese Lücke schliessen soll. Dazu werden Standardverfahren und -konzepte der Software Industrie gekoppelt und mit einer zusätzlichen Abstraktionsebene überlagert, die eine einfache Rekonfiguration von komplexen Systemen und Abstraktion von Softwarestrukturen erlaubt.

1.1. Hintergrund

Die Entwicklung von Software für eingebettete Systeme und Applikationen mit Echtzeitbedingungen stützt sich zu einem großen Teil auf die Programmiersprache C. Die Codebasis dieser Programmiersprache ist entsprechend ihrer Verbreitung und ihrem Alter[1)] sehr groß, wodurch sich der Wunsch und die Möglichkeit ergeben, neue Programme auf der Basis von existierenden Code zu realisieren. Gerade in diesen Anwendungsdomänen kommt eine Vielzahl von Randbedingungen hinzu, die eine Wiederverwendung

[1)] erster ANSI Standard 1989, davor mehrere nicht standardisierte Versionen wie z.B. K&R C von Kerninghan und Richie

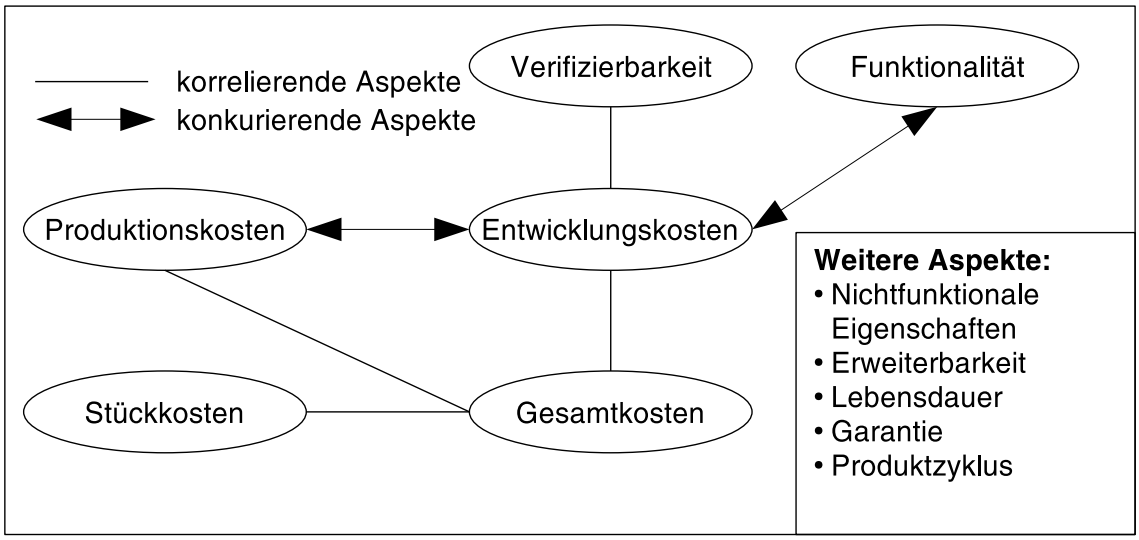


Abbildung 1.1.: Konkurierende Anforderungen beim Entwurf eingebetteter Systeme

von Quelltexten erschweren bzw. gänzlich unmöglich machen kann. Deshalb könnte eine Methodik hier anwendbar sein, die es vereinfacht Software für neue Anforderungen zu rekonfigurieren.

Entscheidend hierfür sind die verwendeten Konzepte zur Abstraktion. Schnittstellen zur vertikalen Integration können auf verschiedene Arten definiert und realisiert werden. Alle existierenden Methoden sind für den Einsatz in eigebetteten Systemen mit Echtzeitanforderungen mit Kompromissen behaftet, wie zum Beispiel die Speicheranforderung und die Skalierbarkeit sowie die Verifizierbarkeit. Insbesondere bringt der Einsatz von domänenspezifischen Entwurfswerkzeugen (Computer Aided Software Engineering Tools oder kurz CASE-Tools) einen nicht vernachlässigbaren Overhead bezüglich Verarbeitungsgeschwindigkeit und Speicherbedarf auf dem Zielsystem mit sich. Andererseits beschleunigt der Einsatz solcher Werkzeuge die Entwicklung der Software mitunter erheblich.

Allerdings bindet der Entwurfsprozess die entworfene Software beim Einsatz eines CASE-Tools an das Werkzeug, da die verwendeten Dateiformate und Codegenerierungsprozesse der unterschiedlichen Werkzeuge zueinander inkompatibel sind. Dies macht den Wechsel von einem Werkzeug zu einem anderen beinahe unmöglich. Ein solcher Umstieg von einem Tool auf ein anderes wird um so aufwendiger und schwieriger, je größer das Projekt ist. Ab einer bestimmten Größe ist ein solcher Schritt wirtschaftlich nicht mehr tragbar. Dies kann ein erhebliches Problem darstellen, wenn ein Architekturwechsel zu einer nicht-unterstützten Plattform ansteht oder die kommerzielle Unterstützung für ein CASE-Tool oder das verwendete Zielsystem aufgekündigt wird.

Dieses Problem erfordert eine Methodik, die Programme an sich ständig verändernde Randbedingungen anpassbar machen kann. Dies lässt sich nicht durch eine einzelne Methodik, eine Sprache oder ein Konzept erreichen. Vielmehr müssen verschiedene Mechanismen zusammenarbeiten, um diesem Ziel nahe zu kommen. Der Lösungsansatz der in dieser Arbeit vorgestellt wird, basiert dabei auf der Erweiterung einer existierenden Programmiersprache.

Menschliche Kommunikation	Metaprogrammierung
Jargon zur Textmodifikation	Sprache zur Metaprogrammierung (MetaC)
Natürliche Sprache	Programmiersprache (C)
Artikel	Programm
Redakteur	Metaprogrammierer
Journalist	Programmierer
Abschnitt	Modul des Programms
Absatz	Funktion
Satz	Anweisung
Nebensatz	Ausdruck
Satzzeichen	Schlüsselwort

Tabelle 1.1.: Vergleichbare Strukturen zwischen einem MetaC-Programm und einem Artikel in menschlicher Sprache

1.2. Ansatz

Um Programme automatisiert verändern zu können, muss eine Beschreibungsform geschaffen werden, die es erlaubt syntaktische und semantische Strukturen zu referenzieren, zu beschreiben und zu verändern. Die Probleme die dabei entstehen, lassen sich anschaulich durch einen Vergleich mit einem Text in menschlicher Sprache und den beteiligten Personen bei seiner Erstellung darstellen.

Den Entstehungsprozess eines Artikels bei dem ein Redakteur gegenüber einem Journalist die Zielsetzungen und geforderten Veränderungen beschreibt, kann man mit dem Ablauf vergleichen bei dem ein Programmierer ein Metaprogramm zur Veränderung eines existierenden Programms schreibt. Dabei lassen sich äquivalente Rollen zwischen der menschlichen Kommunikation und der Metaprogrammierung ausmachen (siehe Tabelle 1.1).

Der Redakteur hat hier die Aufgabe, klar zu formulieren, wie der bereits existierende Artikel verändert werden soll. Seine Rolle entspricht dabei derjenigen des Metaprogrammierers. Um seine Aufgabe zu lösen kann der Redakteur Beschreibungsformen nutzen die Textstrukturen referenzieren (z.B. die Einführung) und gewünschte Inhalte definieren. So ist es ihm möglich den Text auf vielerlei Arten zu modifizieren, indem er...

- bestimmte Stilmittel verwendet.
- die Textstruktur verändert.
- Informationen hinzufügt oder weg lässt.

Bei all diesen Formen der Modifikation kann es dazu kommen, dass er den Sinn des Textes verändert. Ob dies gewünscht ist oder nicht, sei dahingestellt; er sollte sich aber dieser Tatsache bewusst sein. Dabei liegt es in der Verantwortung des Autors ihm Rückkopplung zu seinen Änderungswünschen zu geben.

Betrachtet man einen Dialog zwischen zwei Programmierern, so nähert man sich der Metaprogrammierung noch ein wenig näher an. Durchzuführende Modifikationen und Erweiterungen werden dabei mit einer Kommunikationsform vermittelt, die sich an der grammatikalischen Struktur von C orientiert. Das Ziel ist das Selbe wie bei der Veränderung eines Prosastückes, allerdings sind die Randbedingungen und Anforderungen etwas

andere. Die Modifikationen hierbei müssen von einem Computer automatisch ausgeführt werden können. Um dies zu erreichen ist der Entwurf einer eigenen Programmiersprache notwendig.

1.3. Problemstellung

Insbesondere die wechselnden Anforderungen und die Systemumgebungen können im Umfeld der eingebetteten Systeme schnell zu einer Situation führen, bei der eine Rekonfiguration der Software notwendig wird. Andere wesentliche Ursachen für nachträgliche Änderungen an existierender Software sind ihre geforderte Wiederverwendung unter veränderten Randbedingungen und die stetig fortschreitende Entwicklung der sich schnell verändernden Hardwarebasis. Dabei sind neben den Forderungen der Kunden nach mehr Funktionalität und höherer Qualität auch wirtschaftliche Aspekte zur Reduktion der Kosten (z.B. höherer Integrationsgrad der verwendeten Schaltkreise und neuere beziehungsweise modernere Kommunikationssysteme) entscheidende Faktoren, für sich regelmäßig ändernde Systemarchitekturen.

Dazu gibt es verschiedene Lösungsansätze die teilweise in CASE-Tools integriert sind und so dem Applikationsingenieur das Leben vereinfachen sollen. Dabei werden einige wesentliche Integrationsschwellen abgesenkt, und die Entwicklungsarbeit wird beschleunigt. In diesem Bereich gibt es für die Integration und den Entwurf von Systemen sowie ihre Implementierung verschiedene, konkurrierende Ansätze. Der Grund dafür sind einerseits fehlende Standards zur Integration von Software die mit verschiedenen Werkzeugen entworfen wurden. Andererseits sind Standardisierungsprozesse langwierig, kostenintensiv und könnten in einem so vielseitigen Umfeld kaum genügend Anbieter vereinigen, um interessant für den Anwender zu sein. Dadurch fehlt der echte Nutzen für den Kunden. Denn jeder Standard ist nur so gut wie die Wahlmöglichkeiten die ein Nutzer dadurch bekommt. Kann eine Flexibilisierung dadurch nicht erreicht werden, so wirken sich die Standardisierungskosten am Ende negativ auf die Gesamtbilanz aus.

Die existierenden proprietären Integrations- und Abstraktionsverfahren sind zueinander inkompatibel. Dies macht eine Wiederverwendung von Code aus alten Projekten unter neueren Entwicklungsmethoden und Werkzeugen meist nur mit großem Aufwand realisierbar. Der dafür notwendige Aufwand ist allerdings selten wirtschaftlich tragbar. Retargierbarkeit und Wiederverwendbarkeit können dadurch zu einer schwerwiegenden Designanforderung und Kostenquelle werden, wenn die Notwendigkeit hierfür durch das Applikationsumfeld zwingendermaßen an die Software gestellt wird.

Ein anderes wesentliches Problem ist die Rekonfigurierbarkeit von Software unter dem Gesichtspunkt der unterschiedlichen Entwicklungsphasen. Für die Schritte der Entwicklung, der Simulation, den Test, die Integration und den produktiven Einsatz werden vollkommen unterschiedliche Zusatzanforderungen an die Software gestellt. So sind beispielsweise während der Entwicklung Softwarefunktionen enorm wichtig, die Beobachtbarkeit und Analysierbarkeit unterstützen und so das Debuggen und den Verifikationsprozess vereinfachen. Diese für die Entwicklung notwendige Funktionalität wird in ein fertiges Produkt auf Grund des Overheads bezüglich Ausführungszeit, Codegröße und Speicheranforderung nicht integriert. Für den Kunden sind dies Funktionen ohne Bedeutung und verteuern durch die erhöhten Anforderungen an die Systemleistung den Preis des Produktes.

Innerhalb des Lebenszyklus kann es ebenso zu wechselnden Anforderungen kommen, wenn neue Technologien eingeführt werden. So verdrängt z.B. der Funkstandard Blue-

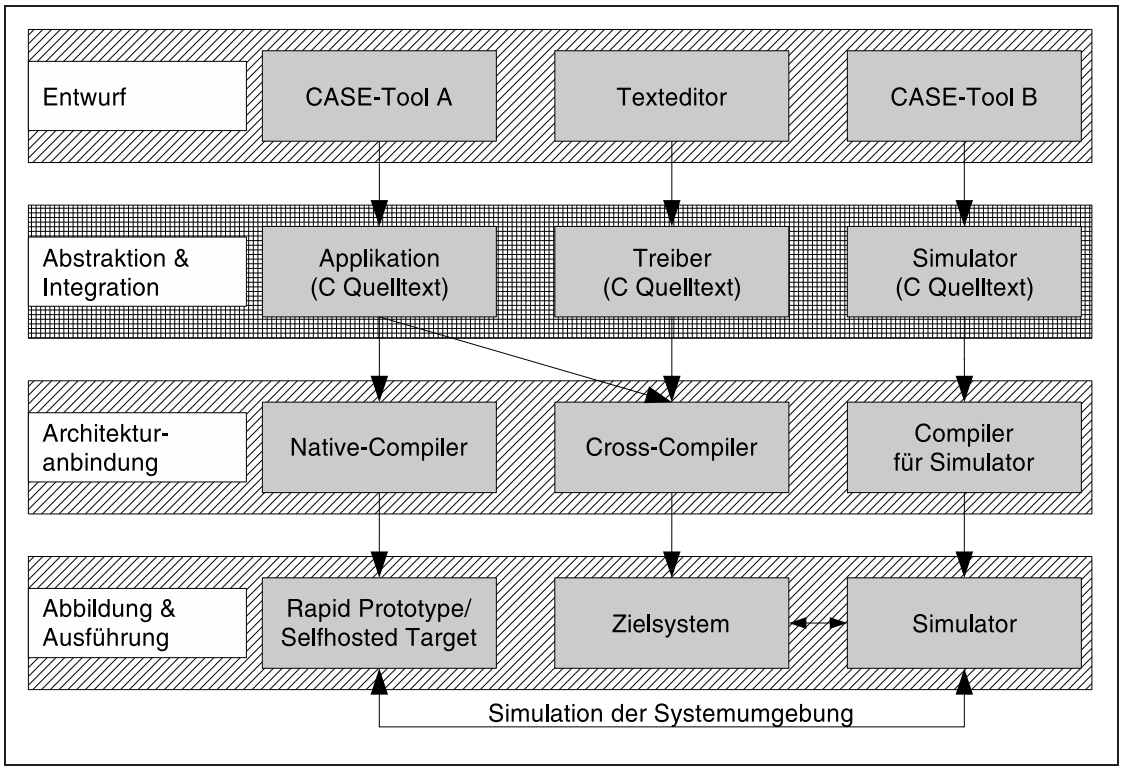


Abbildung 1.2.: Werkzeuge im Entwicklungsprozess von eingebetteter Software

tooth aktuell Infrarotschnittstellen in portablen Geräten. Diese sich ändernden Anforderungen an die Software müssen vom Design durch die Verwendung von Abstraktionsebenen vorgesehen werden. Manchmal ist ihre Notwendigkeit nicht vorab erkennbar und muss nachträglich hinzugefügt werden. Dafür gibt es verschiedene Lösungsansätze, die sich nicht systematisch testen lassen und somit eine potentielle Inkonsistenz zwischen den verschiedenen Targets einbringen.

1.4. Zielsetzung

Ziel dieser Arbeit ist die Entwicklung einer Methodik, die es erlaubt, die zuvor genannten Probleme zu bewältigen. Somit soll es möglich werden Software, die auf C Programmen basiert, grundsätzlich flexibler zu gestalten und eine einfache Rekonfiguration je nach Bedarf und Anwendungsszenario zu erreichen. Diese Methodik muss sich transparent in nahezu beliebige Entwurfs- und Entwicklungsprozesse integrieren lassen. Da die Unterschiede der verschiedenen Prozesse bei eingebetteten Echtzeitsystemen sehr groß sind, sind harte Anforderungen an die Randbedingungen zur Integration einer solchen Methodik nicht akzeptabel.

Entsprechend kann eine Integration eines Verfahrens zur Flexibilisierung des Entwicklungsprozesses nur erfolgreich sein, wenn sie auf bereits existierende Schnittstellen und Standards aufsetzt oder welche schafft, die sich einfach an vorhandene adaptieren lässt. Bei der Entwicklung von Software für eingebettete Systeme mit verschiedenen CASE Tools wird die Integration unterschiedlicher Teilmodule üblicher Weise nach der Codegenerierung durchgeführt um einheitliche Schnittstellen zur Verfügung zu haben. Dazu ist erforderlich, dass die Codegenerierung eine einheitliche Zielsprache verwendet oder

dass die unterschiedlichen Sprachen zumindest über ein kompatibles Application Binary Interface (ABI) verfügen.

Die Programmiersprache C wird von den meisten CASE Tools als Zielsprache angeboten, da es für C Compiler für nahezu alle Architekturen gibt. Dadurch erschliesst sich den Werkzeugen ein grösserer Markt als bei einer spezialisierten Generierung von Maschinencode. Der Einsatz von C ist dadurch über den gesamten Entwicklungsprozess und über verschiedene Projekte möglich und kann somit als Integrationsebene angesehen werden (siehe Abbildung 1.2). Entwurfsprozesse die auf einer Codegenerierung für andere Sprachen basieren, können von einer C basierte Methodik nicht profitieren. Software die in einem solchen, der Sprache C fremenden, Entwurfsprozess entsteht kann, jedoch mit C basierter Software integriert werden, wenn sie ein kompatibles ABI besitzen.

Die in den folgenden Kapiteln vorgestellte Spracherweiterung MetaC für die Programmiersprache C hat zum Ziel Quelltexte einfach veränderbar zu machen. Dazu werden Methoden vorgestellt, die auf dem Konzept der Metaprogrammierung basieren und die speziellen Konzepte von MetaC verwenden. So werden Lösungsmöglichkeiten für heute existierende Probleme aufgezeigt. Im nächsten Kapitel werden dazu die Konzepte von konkurrierenden Verfahren abgegrenzt die heute bereits existieren aber nur Teillösungen für Einzelprobleme liefern können. Diese Arbeit hingegen versucht eine umfassendes Verfahren zu liefern, das alle typischen Probleme eingebetteter Systeme angehen kann. Insbesondere sollen die existieren Integrationsschwierigkeiten auf dem Zielsystem und die begrenzten Optimierungsmöglichkeiten heutiger Verfahren angesprochen werden.

2. Abgrenzung zu konkurrierenden Methoden

Die Probleme die im Rahmen dieser Dissertation durch die Spracherweiterung MetaC gelöst werden, sind zum Teil auch mit existierenden Methoden und Verfahren lösbar. Jedoch behandeln die heute verfügbaren Ansätze jeweils nur Teilaspekte der Fragestellung. Hingegen kann die hier vorgestellte Methodik Problemlösungen für alle diese Bereiche bieten. Darüber hinaus weisen die neuen Verfahren Eigenschaften auf die über die Möglichkeiten konkurrierender Ansätze hinausgehen. Außerdem bieten die MetaC basierten Lösungen eine Funktionalität, wie sie nur durch den kombinierten Einsatz einer Reihe von anderen Methoden erzielt werden kann. Dieses Kapitel grenzt vorhandene Verfahren von den nachfolgend vorgestellten und konkurrierenden Konzepten ab und zeigt die Schwächen der möglichen Alternativen auf.

2.1. Metaprogrammierung

Unter dem Begriff der Metaprogrammierung werden Sprachen und Programme zusammengefasst, die es Software ermöglichen Informationen über sich selbst zu sammeln und sich entsprechend neuer Anforderungen selbst zu modifizieren. Metaprogramme sind Programme, die neben der normalen Laufzeitsemantik auch ein Kompilierzeitverhalten aufweisen. Spracheigenschaften von Programmiersprachen die es Programmen ermöglichen sich selbst zu analysieren, werden als reflexive Konstrukte bezeichnet. Diese sind insbesondere ein wichtiges Element für selbst-modifizierende Programme. Bei der Metaprogrammierung modifiziert ein Programm in der Regel nicht den Programmcode, der zur Kompilierzeit ausgeführt wird, sondern jenen, der später zur Laufzeit zum Einsatz kommt. Diese Einschränkung ist zwar nicht zwangsweise gegeben oder erforderlich, aber in dem Applikationskontext dieser Arbeit der Fall. Metaprogramme, die diese Einschränkung durchbrechen und sich selbst modifizieren, nennt man Metametaprogramme. Metametaprogramme werden auf Grund der sich ergebenden Probleme und fehlenden direkten Anwendungsmöglichkeiten in dieser Arbeit nicht näher betrachtet.

2.1.1. Begriffsdefinitionen

Das Wort *meta* kommt aus dem Griechischen und bedeutet „über etwas“. Bei Metaprogrammen handelt es sich somit um Programme die etwas über sich selbst in Erfahrung bringen oder sich selbst modifizieren. Dabei ist die Spracheigenschaft *reflection* eine wesentliche Voraussetzung für die Realisierung von Metaprogrammen.

Der Begriff *reflection*[1)] wurde von Smith in [67] definiert. Seine Bedeutung lässt sich leicht aus der Übersetzung ableiten. Er bezeichnet die Spracheigenschaft die es einem

[1)] Deutsch: Reflexion

Programm erlaubt, Informationen über sich selbst zu beziehen. Das Programm kann sich dadurch gewissermaßen in einem Spiegel ansehen, betrachtet also seine Reflexion.

Der Begriff *Refaktorisierung* stammt aus dem Bereich der objektorientierten Programmierung (OOP) und wurde beispielsweise in [61] definiert. Er bezeichnet den Vorgang, einen Quelltext für neue Anforderungen derart zu verändern, dass die Funktionalität und das beobachtbare Verhalten bei verändertem Design gleich bleibt. Refaktorisierung wird immer dann durchgeführt, wenn ein Teil der Software für neue Anforderungen erweitert werden muss, das existierende Design die Integration einer solchen Erweiterung aber nur mit enormen Aufwand oder gar nicht zulässt. In diesem Fall wird die Software einem Redesign unterzogen, bei dem möglichst viel des existierenden Quelltextes weiterverwendet wird. Dabei ist das Ziel eine Softwarearchitektur zu schaffen, die den neuen Anforderungen gewachsen ist und gleichzeitig die alte Semantik erhält.

2.1.2. Template Metaprogrammierung in C++

In C++ (siehe [18]) ist eine Metaprogrammierung bei der Standardisierung der Sprache nicht vorgesehen worden. Dennoch ist es möglich in C++ Metaprogramme zu realisieren, die zur Kompilierzeit Modifikationen und Optimierungen durchführen. Dass es technisch überhaupt möglich ist in C++ ein Metaprogramm zu schreiben, zeigte Erwin Unruh in [75] anhand eines Beispiels zur Berechnung von Primzahlen.

In späteren Arbeiten (z.B. [30] und [76]) wurden die Möglichkeiten der Template-Metaprogrammierung in C++ weiter untersucht. Dabei wird die Tatsache genutzt, dass C++ eine *multi-level* oder konkreter gesagt eine *two-level* Sprache ist. Der Begriff der *multi-level language* wurde in [38] geprägt und bezieht sich auf die Art, wie ein Programm in mehreren Stufen übersetzt wird und sich dabei selbst verändert. Programmteile, die eine Veränderung bewirkt haben, werden bei diesem Prozess entfernt.

Prinzipiell lassen sich in C++ für Metaprogramme Variablen, Schleifen, alternative Ausführungspfade und Integerarithmetik realisieren. Somit scheinen alle notwendigen Konzepte für die Turing-Vollständigkeit [26] zu existieren. Allerdings ist dafür auch notwendig, dass Schleifen beliebig oft durchlaufen werden können. Das ist prinzipiell bei der C++ Template-Metaprogrammierung ein Problem, wie weiter unten erläutert wird.

Um ein Metaprogramm in C++ überhaupt realisieren zu können, werden üblicherweise folgende zwei Eigenschaften der Sprache zur Hilfe genommen: Definitionen für Typaliase können in Abhängigkeit von Templateparametern definiert und als solche referenziert werden. Gleiches gilt für Enumeratoren. Dadurch ist es möglich ganzzahlige Werte im Wertebereich der Enumeratoren und Datentypen einen Namen zuzuweisen, also über Variablen zu referenzieren.

Da hierbei eine einmal durchgeführte Definition nicht mehr geändert werden kann, gibt es somit keine Möglichkeit echte Variablen für Metadaten zu realisieren. Um trotzdem abhängige Deklarationen durchführen zu können und somit beispielsweise einen Datentyp abzuwandeln, ist ein geschickter Umgang mit den Gültigkeitsbereichen solcher Typaliasdeklarationen erforderlich. Dabei werden immer neue Gültigkeitsbereiche geschaffen und die Sichtbarkeitsregeln dazu genutzt, alte Werte einer Metavariablen, die in C++ nicht mehr geändert werden kann, zu verstecken. Dies kann zu einem enormen Overhead zur Kompilierzeit führen.

Ähnlich verhält es sich auch für den Kontrollfluss von Metaprogrammen. Um überhaupt alternative Ausführungspfade für Metaprogramme realisieren zu können, wird üblicherweise die Templatespezialisierung von C++ eingesetzt. Für die Implementierung von

Schleifen im Metacode werden rekursive Deklarationen benutzt. Hierbei kommt es zu der zuvor bereits angesprochene Einschränkung der Turing-Vollständigkeit, da die nutzbare Rekursionstiefe vom eingesetzten Compiler abhängig ist. Denn der Sprachstandard erfordert nicht, dass diese Rekursionstiefe einzig vom verfügbaren Speicher abhängen darf und dass im Compiler keine anderen Grenzen dafür existieren.

Der C++ Standard empfiehlt, dass mindestens eine Rekursionstiefe von 17 Ebenen unterstützt werden sollte, doch ist selbst dies nicht erforderlich um dem Standard gerecht zu werden. Unabhängig davon hat die Erzeugung einer Rekursion einen deutlichen Overhead in den gängigen Implementierungen von Compilern. Folglich sind der Verwendung von normalen Schleifen in C++-basiertem Metacode natürliche Grenzen gesetzt, die sich in der Anwendung schnell bemerkbar machen können. Diese können dazu führen, dass ein Compiler ein Stück Metacode ausführen kann, ein anderer jedoch nicht.

Wenn davon ausgegangen wird, dass der eingesetzte Compiler die Empfehlungen des Standards als Minimum realisiert, so können in C++ geschriebene Metaprogramme maximal 17 Iterationen zuverlässig in einer Schleife ausführen. Dies stellt eine erhebliche Einschränkung für die Praxistauglichkeit dieses Ansatzes dar, da in realen Programmen dieser Wert normalerweise nicht ausreicht. Stellt die Implementierung hinreichend viele Rekursionsebenen zur Verfügung, so ist die speicherintensive und langsame Realisierung von Schleifen mit Hilfe der Rekursion immer noch deutlich im Nachteil gegenüber einem gewöhnlichen Verfahren.

Die Turing-Vollständigkeit von Template Metaprogrammierung wurde unter anderem im Detail in [78] untersucht. Die Methodik findet beispielsweise in aktiven Bibliotheken [79] wie Blitz++ [3], der Matrix Template Library [8] und Boost [4] Verwendung. Aktive Bibliotheken zeichnen sich dadurch aus, dass sie nicht nur eine Sammlung von Algorithmen sind, sondern darüber hinaus Metaprogramme enthalten, die entsprechend den Anforderungen passende Algorithmen bzw. Implementierungen auswählen. Dieses Konzept wurde erstmals in [77] vorgestellt.

Die genannten Einschränkungen der C++ Metaprogrammierung sind in der Spracherweiterung MetaC nicht vorhanden. Insbesondere ist der Overhead für Metaprogramme in C++ in Bezug auf die Kompilierzeit und die Speicheranforderung zur Kompilierzeit ein deutliches Hindernis, das in MetaC durch entsprechende Überlegungen in der Sprachdefinition verhindert wurde.

Mit C++ existiert zwar bereits eine *two-level-language* die mit C verwandt ist und zur Metaprogrammierung und Rekonfiguration genutzt werden kann. Allerdings ist diese Nutzungsmöglichkeit kein Designziel beim Entwurf der Sprache gewesen, sondern ist erst nachträglich als zusätzliche Nutzungsvariante entdeckt worden. Entsprechend umständlich und kompliziert gestaltet sich die Syntax von Metaprogrammen in C++. Diese wird durch die Hilfe der Template Metaprogrammierung realisiert. Am Beispiel in Abbildung 2.1 ist erkennbar, dass die Syntax im Vergleich zu einer MetaC basierten Implementierung (siehe Abbildung 2.2) schwer lesbar ist und ohne entsprechendes Expertenwissen nicht erweiterbar ist. Die MetaC Variante ist zudem sehr ähnlich zu einer normalen C Funktion (siehe Abbildung 2.3), die dieses Problem zur Laufzeit statt zur Kompilierzeit löst.

Für die Realisierung in C++ werden zur Speicherung der Ergebnisse Enumeratoren verwendet, in der MetaC Variante hingegen der Datentyp **`zed`** der einen größeren Wertebereich von Integerzahlen unterstützt. Darüber hinaus wäre es in MetaC auch möglich, den Metadatentyp **`real`** zu verwenden, der die Speicherung und Verarbeitung von Gleitkommawerten zur Kompilierzeit ermöglicht. Dazu gibt es in der templatebasierten

```
template <int N, int LO = 1, int HI = N>
struct Sqrt {
    enum { mid = (LO + HI +1) / 2 };
    enum {
        result = (N<mid*mid)
        ? Sqrt<N,LO,mid - 1>::result
        : Sqrt<N,mid,HI>::result
    };
};

template <int N, int M>
struct Sqrt<N,M,M> {
    enum { result = M };
};
```

Abbildung 2.1.: C++ Template Metaprogramm zur Berechnung der Quadratwurzel

```
meta zed sqrtbase(zed N,zed LO,zed HI)
{
    zed mid = (LO + HI + 1) / 2;
    if (LO == HI)
        return LO;
    if (N < mid * mid)
        return sqrtbase(N,LO,mid-1);
    else
        return sqrtbase(N,mid,HI);
}

meta zed Sqrt(zed N)
{
    return sqrtbase(N,1,N);
}
```

Abbildung 2.2.: MetaC Funktion zur Berechnung der Quadratwurzel

```
int sqrtbase(int N,int LO,int HI)
{
    int mid = (LO + HI + 1) / 2;
    if (LO == HI)
        return LO;
    if (N < mid * mid)
        return sqrtbase(N,LO,mid-1);
    else
        return sqrtbase(N,mid,HI);
}

int Sqrt(int N)
{
    return sqrtbase(N,1,N);
}
```

Abbildung 2.3.: C Funktion zur Berechnung der Quadratwurzel mit Integerzahlen

Metaprogrammierung von C++ kein Äquivalent. Insbesondere ist bei der C++ basierten Variante auch der Einsatz der komplizierten Templatesyntax zwingend erforderlich, um das gestellte Problem lösen zu können.

Die Möglichkeiten, in C++ durch Metaprogramme Modifikationen durchzuführen, sind örtlich sehr stark begrenzt und müssen konzeptionell bereits beim Entwurf der Software vorgesehen werden. Alle diese Hürden nimmt MetaC ohne Probleme, da dies die zentralen Designziele sind. Darüber hinaus spricht noch die Tatsache für MetaC, dass es nur wenige C++ Compiler für Architekturen von eingebetteten Systemen gibt, die den Sprachschatz von C++ in einem Umfang unterstützen der notwendig ist um mit Hilfe von Template-Metaprogrammierung Softwarekonfigurationen durchführen zu können. Der Bereich der Refaktorisierung bleibt ohnehin alleine MetaC vorbehalten, da es in C++ keine Konzepte gibt um Quelltext direkt zu verändern und wieder auszugeben.

2.1.3. Offene Compiler Architekturen

Wenn die Sprache selbst keine Möglichkeiten zur Verfügung stellt um Metaprogramme zu realisieren, dann ist der Einsatz einer offenen Compiler-Architektur eine Alternative, um Spracherweiterungen einzuführen. Diese kann eingesetzt werden, um die neue dynamische Semantik transparent zur Kompilierzeit umzusetzen. Die zugehörige Funktionalität wird dem Benutzer in diesem Fall über neue Schlüsselworte angeboten.

Dieses Konzept wurde für mehrere Sprachen verwirklicht und hat den Vorteil, dass neue Ideen und Ansätze relativ einfach erprobt werden können. Offene Compilerarchitekturen zeichnen sich dadurch aus, dass sie dem Entwickler von Erweiterungen einen direkten Zugriff auf den Parsebaum gewähren. Die Idee der offenen Compilerarchitektur wurde zum Beispiel in [52] beschrieben. Damit eine solche Architektur erfolgreich eingesetzt werden kann, ist es wichtig, dass sie reflexive Konstrukte zur Verfügung stellt, die eine detaillierte Analyse des Parsebaums ermöglichen und mit deren Hilfe Modifikationen am Baum durchgeführt werden können.

Ada [14] hat ein solches Konzept bereits im Sprachstandard verankert. Dieses ist bekannt unter dem Namen Ada Semantic Interface Specification (ASIS) [16]. Diese Er-

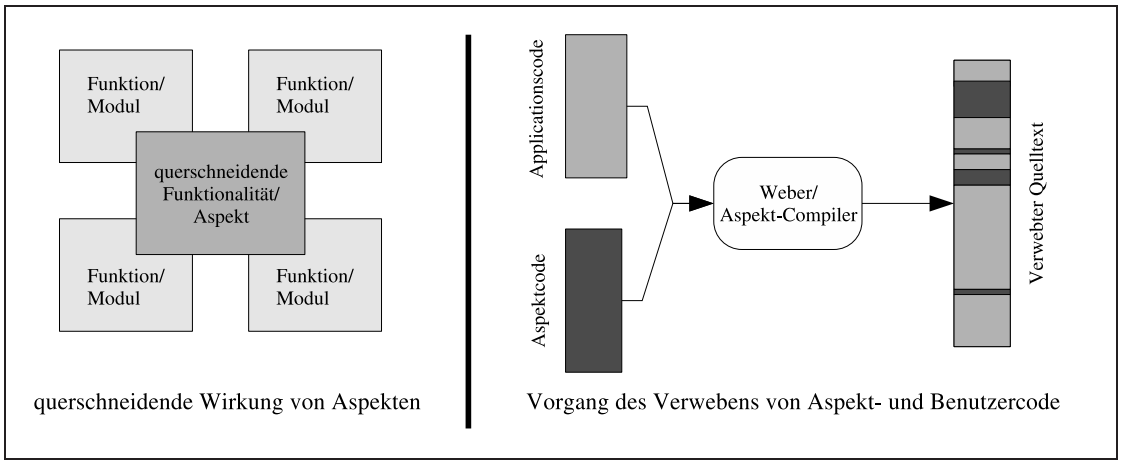


Abbildung 2.4.: Aspekte und der Prozess des Aspectweaving

weiterung bietet eine vom Compilerhersteller unabhängige Möglichkeit, Programme zur Verifikation von Ada Quelltexten zu implementieren. Dies ist eng mit der Zielsetzung verknüpft, dass die Sprache Ada für den Entwurf von sicherheitskritischer Software geschaffen wurde.

Allerdings ist die durch ASIS bereitgestellte Infrastruktur keine Schnittstelle die zur Kompilierzeit Ada Programmen zur Verfügung steht. Stattdessen ermöglicht sie den Zugriff eines bereits kompilierten Programms, das als Modul eines Kompilierers realisiert sein kann, auf ein Programm, dass noch im Quelltext vorliegt. Die Idee einer vollständig reflexiven Infrastruktur, die zur Kompilierzeit verfügbar ist, wurde dagegen in den Arbeiten rund um OpenAda [66] untersucht.

Die Konzepte von OpenAda wurden zuvor bereits in OpenJava [71] und dem C++ Meta Object Protocol [27] verwirklicht. Das C++ Meta Object Protocol wurde im Rahmen des OpenC++ Projekts entwickelt [9] und hat zum Ziel Spracherweiterungen in Form einfacher Schlüsselworte in C++ zu ermöglichen.

2.2. Aspektorientierte Programmierung

Der Begriff der aspektorientierten Programmierung wurde von Gregor Kiczales et al. in [50] geprägt. Aspekte bezeichnen querschneidende Eigenschaften oder Funktionalität in Softwaresystemen. Sprachen, die es ermöglichen solche weit verteilten Codestrukturen zur Bildung einer einzelnen Funktionalität explizit zu beschreiben, werden zu den aspektorientierten Sprachen gezählt. Einen Compiler, der Aspektbeschreibungen einer Spracherweiterung in Quelltext umwandelt, der ohne diese Spracherweiterungen auskommt, bezeichnet man als Aspectweaver. Dabei wird die Lokalität der Aspektbeschreibungen aufgelöst und seine Funktionalität in den Quelltext an vielen Stellen eingefügt (siehe Abbildung 2.4).

Die erste vollständige Umsetzung dieser Konzepte ist eine Spracherweiterung für Java zur aspektorientierten Programmierung namens AspectJ [49]. Später wurden auch Vorschläge für ähnliche Erweiterungen anderer populärer Sprachen gemacht. Für C++ wurde in [37] ein Konzept vorgelegt, das sich an dem von AspectJ orientiert und später im Projekt AspectC++ [1] umgesetzt wurde.

Die verschiedenen Ansätze der aspektorientierten Programmierung liefern Werkzeuge zur Rekonfiguration von Software, um Funktionen in Programme einzubringen, die

die existierende Softwarestruktur querschneidend durchdringen. Die Probleme die durch diese Methodik adressiert werden, können in MetaC ebenso gelöst werden, müssen dabei aber in eine andere Beschreibungsform gebracht werden. Die Beschreibungsform von MetaC orientiert sich hierfür näher an den Methoden und dem Programmierstil für C-Programme, während sich der aspektorientierte Ansatz näher am objektorientierten Design ausrichtet.

2.2.1. Sprachkonzept von AspectJ

Die Spracherweiterung AspectJ basiert auf der Programmiersprache Java, den Konzepten von [50] und wurde in [53] und [54] vorgeschlagen. Sie fügt dem Sprachschatz von Java Aspektdeklarationen mit einer eigenen Syntax hinzu, die an jene von Klassen angelehnt ist. Allerdings beinhalten Aspektdeklarationen anstatt von Methoden, so genannte *Join-Points*, *Pointcuts*, *Advices* und *Inter-type Deklarationen*.

Advices werden genutzt, um Funktionalität zu beschreiben, die an wohl definierten Stellen im Code eingebracht wird. Die Positionen im Quelltext werden dazu mit *Pointcuts* definiert und zur Laufzeit in so genannten *Join-Points* aufgelöst. *Inter-type Deklarationen* hingegen erlauben die Veränderung der Art und Weise, wie Typen miteinander interagieren. Die durchgeführten Definitionen von *Inter-type Deklarationen* werden vollständig zur Kompilierzeit aufgelöst, während die Verwendung von *Advices* zu Code führt, der durch dynamische Bindung erst zur Laufzeit aktiviert wird.

Entsprechend sind nur die Konzepte der *Inter-type Deklarationen* mit den in MetaC zur Verfügung gestellten Methoden zur Codemodifikation vergleichbar, da diese ebenso zur Kompilierzeit umgesetzt werden. Allerdings beziehen sich diese Beschreibungen auf das Klassenmodell von Java und die Interaktion zwischen Objekten über ihre Schnittstellen. Diese Konzepte existieren so in der Programmiersprache C nicht, deswegen ist ein Vergleich nicht unmittelbar durchführbar.

Die möglichen Anwendungsszenarien haben trotzdem Parallelen, die einen Vergleich erlauben. So werden in [2] ähnliche Beispiele genannt, die den Softwareentwicklungsprozess unterstützen können. Dazu gehören z.B. Pre- und Postconditions, Logging, Profiling und Tracing. Konzepte zur Abstraktion von Laufzeitumgebung und Systemumgebung hingegen machen als Erweiterung zur Sprache wenig Sinn, da diese Probleme durch die virtuelle Maschine und das zugrunde liegende Laufzeitmodell von Java bereits gelöst werden.

Der Einsatz von AspectJ für Software eingebetteter Systeme mit Realzeitanforderung hat aus mehreren Gründen deutliche Nachteile gegenüber einer MetaC basierten Lösung. Der kritischste Punkt ist, dass die Ausführungszeiten von Programmen die in Java geschrieben sind, durch den Einsatz einer dynamischen Speicherverwaltung deutlichen Laufzeitschwankungen unterworfen sind. Die Maximalwerte der Ausführungszeit können nur schwer festgelegt werden und den zugehörigen oberen Schranken liegen in der Regel extrem pessimistischen Annahmen zugrunde. Somit ist der Einsatz in RT-Systemen mit großen Problemen behaftet.

Es gibt zwar Ansätze, die Java um RT-Erweiterungen bereichern die jene Probleme beheben sollen, jedoch sind diese Erweiterungen nicht mit den aspektorientiertem Framework für Java integriert. Unabhängig davon bringt Java durch den Einsatz einer virutellen Maschine hohe Anforderungen an Rechenleistung und Speicherausstattung mit sich, die von vielen eingebetteten Systemen nicht befriedigt werden können. In solchen Fällen ist

eine C basierte Lösung, in die sich die MetaC Erweiterung und die zugehörigen Methoden nahtlos integrieren lassen, deutlich überlegen.

2.2.2. Spracherweiterungen für aspektorientierte Programmierung

Auch für die Programmiersprache C wurde an einer aspektorientierten Erweiterung gearbeitet, die aber nur konzeptionell evaluiert und nicht implementiert wurde. In [29] wurde diese Erweiterung namens AOC, die von AspectJ abgeleitet wurde beschrieben. Diese Veröffentlichung untersucht als Anwendung ein AOP basiertes Konzept um die Modularität vom Betriebssystem Code zu verbessern. Dazu wurde der prefetching Quelltext von FreeBSD [5] analysiert und festgestellt, was dieser sich über mehrere Implementierungsschichten der Betriebssystemsoftware erstreckt. Dabei werden verschiedene Strategien für das Prefetching abhängig von den Datenpfaden realisiert.

Diese querschneidenden Strukturen lassen sich explizit als Aspekte beschreiben. Die genannte Veröffentlichung beschreibt exemplarisch, wie dies mit einer aspektorientierten Beschreibung in C realisierbar wäre. Die zugehörige Implementierung wurde von Hand, also nicht mit einem Aspectweaver, realisiert und ist dadurch nur ein konzeptioneller Nachweis des Verfahrens. Wegen vielen Unterschieden zwischen Java und C bezüglich wohldefiniertem Verhalten erscheint ein vollständige Definition einer solchen Spracherweiterungung und die Implementierung eines Compilers unmöglich.

2.2.3. Applikation aspektorientierter Programmierung in Echtzeitsystemen

Einige typische Probleme die bei der Entwicklung von Software für Echtzeitsysteme auftreten, haben querschneidenden Charakter und können als solches mit Hilfe der Aspekt-Orientierten Programmierung modularisiert werden. Das heisst, diese Softwarefunktionen werden explizit und getrennt vom restlichen Applikationscode implementiert.

Ein wesentliches Problem, das immer wieder bei Echtzeitapplikationen gelöst werden muss, ist die Bestimmung der Ausführungszeiten einzelner Codepfade, die Ableitung der Worst-Case Execution Time (WCET) aus diesen Daten und die Einhaltung von Deadlines zur Laufzeit. Diese Probleme werden in den Arbeiten [73] und [74] angesprochen. Um sie zu lösen, wird ein Werkzeug vorgestellt, das die Einhaltung von Deadlines überwacht und den dazu notwendigen Code in Form von Aspektbeschreibungen in das Programm einbringt. Das Tool ist dabei auf Annotationen angewiesen, die die Informationen aus der WCET Analyse beinhalten, um die gestellte Aufgabe lösen zu können.

Solche Verfahren sind in der Praxis nur gangbar, wenn die dafür notwendigen Werkzeuge gut miteinander integriert sind und sich die erforderlichen Aspektbeschreibungen ohne großen Aufwand einbinden lassen. Da aber die Programmiersprache C ein Design erfordert, dass auf der Interaktion verschiedener Funktionen basiert, ist der Wechsel zur AOP ein großer Schritt, der nicht ohne weiteres vollzogen werden kann. Die zugrundeliegenden Konzepte sind vollkommen unterschiedlich und erfordern eine andere Form des Softwareentwurfes.

2.2.4. Aspektimplementierung ohne Spracherweiterung

In C++ gibt es eine Reihe von nativen Sprachkonstrukten die zur Implementierung von Aspekten eingesetzt werden können. Dabei kommt nicht die von AspectJ bekannte Syntax zum Einsatz, und der Grad der Separation ist oft nicht so groß, wie in AspectJ basierten Implementierungen. Trotzdem sind solche Verfahren in den Punkten Funktionsumfang, Variabilität und Parametrierbarkeit durchaus mit nativen Aspektbeschreibungen vergleichbar und sind außerdem ohne Spracherweiterung, also mit einem standardkonformen Compiler realisierbar.

Ein populäres Beispiel für eine Aspektbibliothek ist die Threads Library der Boost Bibliothekensammlung [4]. Diese stellt alle wesentlichen Primitiven für die Realisierung von parallelen Programmen zur Verfügung. Dazu werden die vom Betriebssystem bereitgestellten Primitive gekapselt und dem Benutzer in Form von Primitiven, wie sie im objektorientiertem Design (OOD) üblich sind, dargeboten. Neben den Standardprimitiven, die vom Betriebssystem üblicherweise über das POSIX API angeboten werden, wie Thread, Mutex, Semaphore und Timer, werden ihre zugehörigen Varianten als eigenständige Konstrukte und somit in einfacher nutzbarer Form bereitgestellt. Dazu gehören beispielsweise rekursive Mutexe, Barriers, Conditions und Thread Specific Storage. Eine solche Abstraktion erfordert die entsprechenden Laufzeitbibliotheken auf dem eingebetteten System. Außerdem muss das Zielsystem den Overhead von C++ Templatecode verkraften können, die diese Methodik mit sich bringt.

In [23] wird eine Methodik vorgestellt, die Aspekte erst zur Laufzeit in den Applikationscode bindet. Im Gegensatz dazu werden in AspectJ die Aspekte zur Kompilierzeit eingebunden. Die genannte Arbeit stützt sich dafür auf die in [22] und [24] vorgestellten Spracherweiterung Maya, die es ermöglicht für Java Programme zusätzliche Schlüsselworte dem Java Sprachschatz hinzuzufügen. So ist es möglich, Java um eine Schicht zur Metaprogrammierung zu erweitern, die eine Definition von so genannten Wrappern erlaubt. Diese werden genutzt, um Methodenaufrufe zur Laufzeit zu kapseln und beispielsweise Cachingstrategien zu realisieren. Die vorgestellte Methodik kann dadurch als alternatives Verfahren zur Implementierung von Aspekten in Javaprogrammen gesehen werden.

2.3. Objektorientierte Sprachen

Der Übergang vom imperativen zum objektorientiertem Design hat die Entwicklung von Software deutlich beschleunigt. Insbesondere die Verwendung von Entwurfsmustern [34] gehört heute zu den Standardverfahren beim Entwurf und der Programmierung von Software. Auch die Tatsache, dass fast alle existierende Programmiersprachen (C++ als Nachfolger von C, Ada, Perl) heutzutage objektorientierte Erweiterungen bekommen haben und dass neue Sprachen (Java, Smalltalk) in der Regel diese Konzepte auch unterstützen, ist ein Zeichen für den Erfolg der zugrunde liegenden Konzepte.

2.3.1. Objektorientierte Konstrukte zur Rekonfiguration

Die wichtigsten Sprachmittel, die eine Rekonfiguration von Software für verschiedene Anwendungsszenarien einfach ermöglichen, basieren auf den Prinzipien der Vererbung. Dafür gibt es in den verschiedenen objektorientierten Sprachen (z.B. Smalltalk, C++, Ja-

va, Ada) das Konzept der Vererbung, das nicht von allen Sprachen im gleichen Umfang unterstützt wird. Grundsätzlich lassen sich folgende Verfahren unterscheiden:

1. dynamischer Polymorphismus mit dynamischer Bindung
2. dynamischer Polymorphismus mit statischer Bindung
3. statischer Polymorphismus

Das erste Verfahren ist das klassische, das ursprünglich von Smalltalk eingeführt wurde und später unverändert von Java übernommen wurde. Dabei werden alle Methodenaufrufe abhängig vom verwendeten Objekt zur Laufzeit an die entsprechenden Codestrukturen gebunden. Der Applikationscode ruft dadurch zwar die Methode einer bestimmten Klasse auf, zur Laufzeit bestimmt hingegen das von der Applikation verwendete Objekt die Implementierungsvariante der Methode. Die anderen beiden Verfahren können als Optimierungen gesehen werden, die den Laufzeitoverhead verringern indem Teile der Entscheidungen, die sonst zur Laufzeit stattfinden auf die Kompilierzeit vorgezogen werden. Die Folge dieser Optimierung ist, dass sich die Flexibilität zur Laufzeit verringert und gleichzeitig der Aufwand zur Kompilierzeit steigt. Unabhängig von der Art des Polymorphismuses und der Bindung können alle diese Varianten dazu genutzt werden Code an bestimmten Stellen implizit einzufügen, ohne diesen modifizieren zu müssen.

Eng verbunden mit den verschiedenen Arten von Polymorphismus sind die Verfahren zum Überladen von Operatoren. Diese Konzepte werden nicht von allen objektorientierten Sprachen zur Verfügung gestellt. Es ermöglicht die existierenden Operatoren mit neuer Semanitk zu belegen und so applikationsspezifische oder domänenspezifische Erweiterungen durchzuführen. Eine weitere Möglichkeit ist die Verwendung von *Exceptions*. Dieses Sprachkonzept, das zur Behandlung von Ausnahmesituationen exisitiert, kann verwendet werden um neue, alternative Kontrollflußpfade im Quelltext einzufügen.

2.3.2. Grenzen der Anwendung von objektorientierten Sprachen

Viele Problemstellungen aus dem Bereich Rekonfiguration lassen sich durch entsprechendes Design mit den zuvor genannten, objektorientierten Konzepten lösen. Allerdings gibt es dabei folgende, wesentliche Einschränkungen im Vergleich zu der Methodik, die hier vorgeschlagen wird:

1. Die Rekonfiguration muss während dem Design des Programmes bereits vorgesehen werden, kann also nicht ohne weiteres nachträglich durchgeführt werden.
2. In vielen Applikationsgebieten von eingebetteten Systemen ist der Einsatz von objektorientierten Sprachen nicht möglich, da der Resourcenbedarf zu groß ist oder manche Konstrukte dieser Programmierparadigmen nicht in echtzeitfähigem Code umgesetzt werden.
3. Die Komplexität eines Echtzeitnachweises steigt extrem an, wenn zusätzliche Programmpfade in den Kontrollfluss eingefügt werden, wie es implizit bei einigen dieser Sprachkonstrukten der Fall ist.
4. Es gibt keine sprachlichen Konstrukte, die eine automatische Refaktorisierung von Programmcode erlaubt, um diesen für neue Anforderungen oder Randbedingungen anzupassen.

Die Implementierung objektorientierter Konstrukte ist stark vom verwendeten Compiler abhängig, trotz dass die Semantik immer die Selbe ist. Viele objektorientierte Strukturen lassen sich auf eine Vielzahl von Möglichkeiten mit unterschiedlichen Vor- und Nachteilen implementieren. Dabei gibt es Implementierugsvarianten deren Ausführungszeit großen Schwankungen unterworfen sind und deren worst-case excution time (WCET) nur schwer bestimmbar sind. Eine Analyse muss in diesen Fällen dann auf Maschinencodeebene durchgeführt werden, da ansonsten das konkrete Laufzeitverhalten nicht bestimmbar ist. Solche Konstrukte dürfen deswegen bei der Realisierung von Echtzeitsoftware nicht verwendet werden, denn dadurch ist die Nachweisbarkeit der Funktion des Gesamtsystems gefährdet.

Beim Entwurf von Echtzeitsoftware ist die Rekonfiguration vom Design her in der Regel bereits mit vorgesehen. So stellt diese Einschränkung von objektorientierten Sprachen keine wesentliche Hürde bei der Anwendung in dieser Applikationsdomäne dar. Jedoch gibt es teilweise Situationen, bei denen Konfigurationsanforderungen erst nachträglich offensichtlich werden. In solchen Fällen ist die Notwendigkeit eines Redesigns ein enormes Kostenrisiko, das nur schwer zu tragen ist und das in dieser Form mit MetaC basierten Lösungen nicht in Erscheinung tritt.

Für die Refaktorisierung liefert MetaC spezielle Konzepte, und im Kapitel 8.2.5 wird ein Applikationsbeispiel gegeben, wie durch eine metaprogrammbasierte Refaktorisierung von Quelltext die Messgenauigkeit von Ausführungszeiten von basic blocks erhöht werden kann. Da objektorientierte Sprachen dafür keine Alternativen bieten, ist hier MetaC klar im Vorteil.

2.4. Abstraktion mit Hilfe des C-Präprozessors

Da mit dem C-Präprozessor bereits eine Technik existiert um C-Quelltexte mit Freiheitsgraden zu versehen, muss MetaC Vorteile bieten, die deutlich über das Verfügbare hinausgehen. Der Präprozessor bietet drei wesentliche Techniken: das Einfügen von Quelltext aus anderen Dateien, bedingtes Weglassen von Quelltext und Quelltext-Substitution mit Hilfe von Makros. Diese Operationen sind vollständig unabhängig von der Sprache C realisiert und deshalb auch insensibel für die syntaktische Struktur und Semantik des Quelltextes, der durch diese Techniken modifiziert wird.

Das bedeutet, der Präprozessor führt eine einfache Ersetzung von Tokensequenzen durch, ohne die Auswirkung beurteilen zu können und somit dem Anwender eine Rückkopplung über die durchgeführten Modifikationen geben zu können. Der Präprozessor arbeitet also auf rein lexikalischer Ebene und ist nicht mit der Grammatik von C vertraut. Dadurch hat er auch keinen Einblick in die Semantik einer vollzogenen Modifikation. Das kann zu vielschichtigen Problemen führen, die sehr subtil in Erscheinung treten, da die Syntax eines Präprozessoraufrufes identisch mit einem Funktionsaufruf in C ist und somit leicht mit einem solchen verwechselt werden kann.

Allerdings ist das Ergebnis der Ausführung eines Präprozessormakros ein vollständig anderes als ein Funktionsaufruf. Bei dem Präprozessormakro wird an der Stelle des Aufrufs ein anderer Quelltext eingesetzt; hingegen wird bei einem Funktionsaufruf zur Laufzeit verzweigt und ein unabhängiger Sichtbarkeitsbereich geöffnet, in den die übergebenen Parameter kopiert werden (*pass-by-value* Semantik). Das daraus entstehende Problem lässt sich anschaulich mit einem kurzen Beispiel zeigen, wie es in Abbildung 2.5 dargestellt ist.

```
#include <stdio.h>
/* unbedarfte Makro Definition */
#define MIN1(a,b) a < b ? a : b
/* verbesserte Makro Definition */
#define MIN2(a,b) ((a) < (b) ? (a) : (b))

int a; /* globale Variable */

int min(int l, int r)
{
        return l < r ? l : r;
}

int f(void)
{
        return --a;
}

int main(void)
{
        int r;
  /* erster Versuch */
        a = 2;
        r = MIN1(f(),2) + 100;
  /* r = f() < 2 ? f() : 2 + 100; */
        printf("MIN1: a == %i, r == %i\n",a,r);
  /* zweiter Versuch */
        a = 2;
        r = MIN2(f(),2) + 100;
  /* r = ((f()) < 2 ? (f()) : (2)) + 100; */
        printf("MIN2: a == %i, r == %i\n",a,r);
  /* dritter Versuch */
        a = 2;
        r = min(f(),2) + 100;
        printf("min : a == %i, r == %i\n",a,r);
        return 0;
}
```

Abbildung 2.5.: Präprozessormakro mit semantischer Überraschung

```
MIN1: a == 0, r == 0
MIN2: a == 0, r == 100
min : a == 1, r == 101
```

Abbildung 2.6.: Ausgabe des Programms aus Abbildung 2.5

In diesem Beispiel wird auf drei verschiedene Varianten der kleinere von zwei ganzzahligen Werten ermittelt. Dazu werden zum einen die beiden Präprozessormakros **`MIN1`** und **`MIN2`**, sowie die C-Funktion **`min`** definiert. Das eigentliche Programm (Funktion **`main`**) verwendet die drei verschiedenen Implementierungsvarianten mit der Variable **`a`** und initialisiert diese dazu jedes mal wieder mit dem Wert 2 und speichert das Ergebnis in der Variable **`r`**.

Bei Ausführung des Programmes ergibt sich die Ausgabe die in Abbildung 2.6 dargestellt ist. Es ist zu sehen, dass sich bei allen drei Varianten unterschiedliche Ergebnisse ergeben. Ein menschlicher Leser des Quelltextes erwartet ohne Kenntnis der Tatsache, dass **`MIN1`** und **`MIN2`** Präprozessormakros sind, die normale C Laufzeitsemantik. Dies führt häufig zu subtilen und schwer aufzufindenden Fehlern in Programmen.

Weitere Probleme die bei der Verwendung des Präprozessors entstehen können sind:

- Makronamen, die Variablendefinitionen überschreiben und so entweder zu unerwarteten Typfehlern beim Kompilieren oder im Extremfall zu falsch aufgelösten Symbolen mit identischem Typ führen.

- Fehlende Klammern oder Interpunktionszeichen die durch Makros verursacht werden, können zu verwirrenden Fehlermeldungen führen. Insbesondere sind solche Probleme schwer zu beheben, wenn die Fehlermeldung auf eine andere Datei verweist, als die in der die eigentliche Ursache liegt.

Die Liste der Probleme ließe sich noch um einige Punkte erweitern. Ziel beim Entwurf von MetaC war es somit, Funktionen zur Rekonfiguration von Quelltexten bereitzustellen, die typischerweise mit dem Präprozessor realisiert werden, dabei aber zu Problemen führen können. Die weiteren Ziele werden im Detail im nächsten Abschnitt beschrieben.

2.5. Modellbasierte Ansätze

Die Object Management Group (OMG) hat eine Reihe von Technologien und ihre Interaktion in der Modell Driven Architecture (MDA) beschrieben und spezifiziert, die dazu eingesetzt werden können einmal vorgenommene Modellierungen für verschiedene Aufgaben zu verwenden. Zu diesen Technologien gehören beispielsweise die Unified Modeling Language (UML), die MetaObject Facility (MOF), der XML Metadata Interchange (XMI) und das Common Warehouse Metamodel (CWM). Die vorgestellten Methoden können auch dazu eingesetzt werden die typischen Probleme beim Entwurf von eingebetteten Systemen anzusprechen.

Insbesondere ist es durch dadurch möglich, ein und die selbe Modellbeschreibung zur Simulation und Implementierung für unterschiedliche Zielsysteme zu verwerden. Auch eine Integration mit den Projektphasen Verifikation und Validierung ist machbar. Dazu werden für jede Applikationsdomäne spezifische Modellierungskonstrukte mit wohldefinierter Semantik definiert, die ausgehend von den jeweiligen Anforderungen ausgewählt

werden. So ist eine spezielle Abstraktion der domäneneigenen Probleme möglich, und dadurch kann eine konsistente Unterstützung von Simulation und Verifikation erreicht werden.

Besonders durch die Einführung von speziellen Strukturen kann eine effiziente Modellierung ermöglicht werden. Des Weiteren kann so nicht nur das Systemverhalten spezifiziert werden, sondern ebenso die besonderen Anforderungen und Randbedingungen, die zur Generierung von Testfälle herangezogen benötigt werden. Durch den gezielten Einsatz von Werkzeugen zur Transformation dieser Informationen in verschiedene Darstellungsformen und Sichtweisen, können die unterschiedlichen Anforderung der Phasen des Entwicklungsprozesses aus einer einzelnen Beschreibung befriedigt werden. Durch diese flexiblen Transformationsmöglichkeiten kann bespielsweise erreicht werden, dass große Teile des notwendigen Programmcodes für die Verifikation aus den Modellierungen des Requirement Engineering generiert wird. So verbessert sich einerseits die Konsistenz im gesamten Projekt gegenüber herkömmlichen Methoden und gleichzeitig werden Fehler durch Inkonsistenzen vermieden. Letztendlich ist dies ein Beitrag zur Verringerund des Aufwandes in den Einzelphasen des Projektes und zur Beschleunigung der Entwicklung.

Ein Beispiel für den erfolgreichen Einsatz dieser Methodik beschreiben Mitarbeiter der Firma Kennedy-Carter in [64] am Beispiel von xUML und *Action Specification Language* (ASL). Die Sprache xUML ist eine Variante der UML, die mit Hilfe der ASL dazu eingesetzt werden kann, semantisch wohl definiertes Verhalten zu spezifizieren. Diese Beschreibungsform ist Architekturneutral und vollkommen unabhängig von der Zielsprache. Dadurch ergibt sich ein hoher Grad an Portabilität und die Modellbeschreibungen können für Simulation, Verifikation, Implementierung und Systemoptimierung herangezogen werden.

Ein Nachteil des modellbasierten Ansatzes ist, dass der Nutzen dieser Methodik bei der Verwendung von Legacycode in der Regel ausbleibt. Denn für eine effiziente Nutzung von Legazycode in einem modellgestützten Entwicklungsprozess müssten die vorliegenden Implementierungen als Modelle importiert werden können. Allerding verhindert die Beschreibungsform als reiner Quelltext die Integration von existierenden Implementierungen, da diese Darstellungsform nur zur effizienten Übersetzung in Maschinensprache ausgelegt ist. Folglich können die Werkzeuge zur Transformation in andere Darstellungsformen nicht verwendet werden, und die Realisierung zur Integration der bestehenden Teilmodule gestaltet sich schwierig.

Auch die Anpassung der Codegenerierung für applikationsspezifischen Modelle zur Abbildung von Semantik ist sehr aufwending und verlangt feste Schnittstellen zur Integration von fremden Beschreibungsformen. Jedoch widerspricht der Einsatz von harten Schnittstellen der Idee des modellbasierten Entwurfes, da solche Details abstrahiert werden sollten. Insbesondere wird in der MDA nur die Transformation vom Platform Independent Model (PIM) zum Platform Specific Model (PSM) betrachtet. Deshalb kann man von einem modellbasierten Entwicklungsprozess nur selten profitieren, wenn die existierende Codebasis einen signifikanten Anteil am Gesamtprojekt hat, da diese in einer plattformspezifischen Beschreibungsform vorliegt. Es gibt zwar Arbeiten, beispielsweise [58], die sich auch mit dem umgekehrten Weg beschäftigen, aber konkrete Lösungen zur direkten modellbasierte Integration fehlen.

Die Konzept der OMG und die Werkzeuge die auf der MDA basieren liefern die besten Ergebnisse, wenn sehr große Systeme mit vielen Teilmodulen und komplexen Abhängigkeiten entwickelt werden. Hierbei führen die verschiedenen Sichtmöglichkeiten durch die Modellbildung zu einer transparenteren Darstellung, und der zusätzliche Aufwand zur Im-

plementierung der speziellen Codegenerierung wird kompensiert. Speziell die Applikationsdomäne der eingebetteten Systeme zeichnet sich dadurch aus, dass die Projekte über viele Jahre wachsen und meistens eine große Basis an Quelltext und fertigen Teilmodulen existierent. Gerade in diese Situaton führt dazu, dass von die besonderen Konzepten der MDA und den entsprechenden Werkzeugen nur sehr eingeschränkt nutzbar sind und der Nutzen häufig den Aufwand nicht rechtfertigt. Der Hauptgrund ist, dass die vorliegenden Implementierungen nicht als Modelle importiert werden können. In diesem Fall sind Methoden, wie sie im Folgenden vorgestellt werden nutzbringender, da sie es ermöglichen bestehnden Quelltext querschneidend zu analysieren, zu modifizieren und zu erweitern.

3. Grundlagen

In diesem Kapitel werden einige grundlegenden Begriffe und Verfahren zusammengefasst die im weiteren Verlauf Verwendung finden. Die hier vorgestellten Konzepte entsprechen dem Stand der Technik und den gängigen Bezeichnungen. Da ein Großteil der zugehörigen Forschungsarbeiten im englischsprachigen Raum durchgeführt wurden und die Publikationen meist auf Englisch verfasst werden, haben sich auch in Deutschland häufig die englischen Begriffe durchgesetzt. Soweit es deutschsprachige Worte mit äquivalenter Bedeutung gibt, die zumindest teilweise in der Praxis verwendet werden, wird auch in dieser Arbeit der deutsche Begriff benutzt um die Lesbarkeit zu erhöhen. Sollte eine deutsche Übersetzung nicht eindeutig dem englischen Begriff zuzuordnen sein oder gar nicht existieren findet der Originalfachbegriff Anwendung, um das Referenzieren zu vereinfachen.

3.1. Programmierung von eingebetteten Echtzeitsystemen

Die Entwicklung von Software für eingebettete Systeme hat abhängig von der Applikation häufig besondere Aufgaben zu lösen. Der offensichtlichste Unterschied zur normalen Applikationsentwicklung ist, dass Host- und Zielarchitektur meist unterschiedlich sind. Das bedeutet, dass die Software auf einem Entwicklungsrechner (üblicherweise ein PC) entworfen wird und mit einem Crosscompiler für das Zielsystem übersetzt wird. Die Binärdatei ist dann auf dem Host (PC) nicht ausführbar und muss direkt auf dem Target (dem eingebetteten System) getestet werden. Dieser Unterschied zur normalen Softwareentwicklung fällt nicht nur unmittelbar auf, sondern wirkt sich auch bei der praktischen Tätigkeit mitunter drastisch aus.

Die dadurch entstehenden Schwierigkeiten werden auf Grund der Vorteile für die Applikation oder als notwendige Voraussetzung für ihre Realisierung gerne in Kauf genommen. Die in eingebetteten Systemen eingesetzten Prozessorarchitekturen bieten besondere Eigenschaften, die bei gewöhnlichen PCs nicht notwendig oder vernachlässigbar sind. Dazu gehören beispielsweise extrem niedrige Stückkosten, niedriger Stromverbrauch, Strahlungsfestigkeit, Unempfindlichkeit gegenüber besonders hohen und niedrigen Temperaturen oder besondere Funktionen (z.B. TPU bei der CPU32 Baureihe oder die Jazelle Erweiterung der ARM9 Kerne) und Peripherie (z.B. CAN Bus und A/D Wandler), die direkt mit auf dem Baustein integriert wird.

Die große Zahl an verschiedenen Anforderungen in den verschiedenen Applikationsfeldern hat zu einer Fülle an Architekturen geführt, die sich den Markt aufteilen. Die wichtigsten Architekturen sind Freescales [1)] 68k/Coldfire und PowerPC Reihen, die 8051, MIPS und SPARC Varianten verschiedener Hersteller, die ARM Kerne, die in Lizenz ebenfalls von verschiedenen Herstellern produziert, aber von ARM Limited entwickelt werden, die AVR Baureihe von Atmel, die im nichtprofessionellen Bereich anzusiedeln

[1)] vor July 2004 war Freescale ein Teil von Motorola

ist und die MSP430 Baureihe von Texas Instruments. Diese Liste lässt sich fast beliebig lang fortsetzen.

Die Programmiersprache C ist in diesem Umfeld etabliert und unterstützt die allermeisten Mikrokontrollerarchitekturen. Dies hat eine Reihe von Ursachen, die zum Teil in der Historie begründet sind. Dazu gehört, dass in C sehr hardwarenah und effizient programmiert werden kann und dass es, im Vergleich zu anderen Sprachen, relativ einfach ist einen Compiler zu implementieren. Das liegt vor allem daran, dass in den für fehlerhaften C Quelltext, also Programmcode der *undefined behavior* (siehe [17], Abschnitt 3.4.3) auslöst, das Verhalten der Implementierung nicht vorgeschrieben ist. Auch gibt es viele Teile in der Spezifikation, die die Definition des konkreten Verhaltens der Implementierung, also dem Compilerhersteller, überlassen.

3.1.1. Grundkonzepte der Sprache C

Die Programmiersprache C ist über die Jahre stetig erweitert worden. In den Anfängen war es nicht mehr als ein Werkzeug, um häufig auftretende Konstrukte nicht jedes Mal von Hand in Assembler ausformulieren zu müssen. Erst später kamen Variablen hinzu, die auf dem Stack alloziiert werden sowie komplexe Gleitkommaoperationen und erweiterte Datenstrukturen. Die heute verbreiteten Quelltexte basieren meist auf dem Originalstandard von 1989 des ANSI [12]. Die neuere Revision von 1999 bietet einige Erweiterungen, konnte sich aber insbesondere im Umfeld der eingebetteten Systeme noch nicht auf breiter Front durchsetzen.

Ein wichtiges Implementierungsziel von C Compilern war stets die Generierung von Maschinencode mit geringem Overhead. Dies macht die Sprache insbesondere für eingebettete Systeme interessant, da sie oft auch vollständig auf eine Laufzeitumgebung verzichten kann. Entsprechend der Historie sind in der Sprache viele Eigenschaften vorhanden, die eine systemnahe Programmierung erheblich vereinfachen. Dazu zählen:

- inline Assembler
- Zeiger auf beliebige Adressen
- Bitfelder in strukturierten Datentypen

Diese Eigenschaften sind aber nur von ihrer syntaktischen Struktur her konkret definiert. Die dynamische Semantik dieser Konstrukte ist implementierungsspezifisch, wodurch sich beim Einsatz von verschiedenen Compilern unterschiedliches Laufzeitverhalten ergeben kann. In der Regel stellt dies kein Hindernis dar, da zum einen die implementierungsspezifischen Details nur selten deutlich voneinander abweichen und zum anderen sämtliche hardwarenahen Programme ohnehin zu einem hohen Grad unportabel sind.

3.1.2. Konstrukte für Echtzeitprogrammierung

Obwohl C häufig für die Programmierung von Echtzeitapplikationen eingesetzt wird, besitzt sie keine Konstrukte in der Sprache die speziell für diese Domäne vorgesehen sind. Im Gegensatz zu anderen Sprachen, die spezielle Erweiterungen für dieses Applikationsfeld bieten, wie Ada und Java, ist das Laufzeitmodell *single-threaded*. Das bedeutet, dass der Compiler davon ausgehen darf, dass das zu kompilierende Programm durch einen einzelnen Prozessor ausgeführt wird.

Jedoch kann es ohne besondere Vorkehrungen bereits in einfachen *single-threaded* Applikationen zu nicht-deterministischer Ausführungsergebnissen also *race-conditions* kommen, wenn die Laufzeitumgebung dem Prozess ein Signal schickt. Das Konzept der Signale ist tief in POSIX verankert und betrifft somit die meisten Betriebssysteme, unabhängig davon ob es sich um einen normalen PC oder ein eingebettetes System handelt.

Zuvor wurden als Beispiele für Programmiersprachen mit Konstrukten zur vereinfachten Entwicklung von Echtzeitapplikationen Ada und Java angeführt. Beide unterstützen neben Multithreading auch Mechanismen zur Kommunikation und Synchronisation sowie Prinzipien zur Realisierung von verteilten Programmen. Alle diese Konzepte werden zusammen mit diesen und weiteren Sprachen in [25] vorgestellt und erläutert. Darüber hinaus wird auch auf das POSIX API eingegangen und an den gezeigten Beispielen wird der notwendige Mehraufwand von POSIX basierten Implementierungen deutlich.

Die Programmiersprache C bietet selbst keine nativen Konstrukte zur Unterstützung von Echtzeitapplikationen, weshalb für die Implementierung entsprechender Softwarestrukturen auf APIs wie POSIX zurückgegriffen werden muss. Die wesentlichsten Konzepte, die in C fehlen, aber in anderen Sprachen unterstützt werden sind:

- Multitasking
- Wechselseitiger Ausschluss
- Atomare Operationen
- Fehlerbehandlung
- Behandlung von und Bewußtsein für Zeit
- Synchronisationsmechanismen
- Kommunikationsmechanismen

Um solche Funktionen nutzen zu können müssen diese nachgebildet werden. Üblicherweise geschieht dies nicht über Spracherweiterungen, sondern durch Verwendung einer entsprechenden Applikationsschnittstelle (API), die durch das Betriebssystem angeboten wird. Da das relativ einfach zu bewerkstelligen ist, bietet fast jeder Betriebssystemhersteller auch ein API für C an. Diese Schnittstellen sind meist proprietär und zueinander inkompatibel, wobei teilweise auf leistungsstärkeren Prozessorvarianten (in der Regel mit einer 32 Bit Architektur) das POSIX API zumindest zum Teil unterstützt wird.

Der Nachteil dieses Konzepts ist, dass der Compiler nicht überprüfen kann, ob bei der Nutzung des APIs logische Fehler vorliegen. Dieses Problem tritt bei Sprachen mit nativer Unterstützung nicht auf. Beispiele hierfür sind die unsymmetrische Verwendung der Semaphore Operationen `obtain` und `release` oder das Versenden einer Nachricht an eine Warteschlange, die nicht initialisiert wurde. Trotzdem wird auf C wegen der großen Verbreitung, also der Verfügbarkeit von Entwicklern mit C Kenntnissen einerseits und Compilern für unterschiedlichste Zielsysteme andererseits, eingesetzt.

3.2. Grundlagen des Compilerbaus

Der Entwurf und die Programmierung von Compilern ist ein seit langem etablierter Bereich der Informatik. Als Compiler werden Programme bezeichnet, die einen Eingabedatenstrom (für gewöhnlich in textueller Form, aber nicht zwingenderweise erforderlich) in

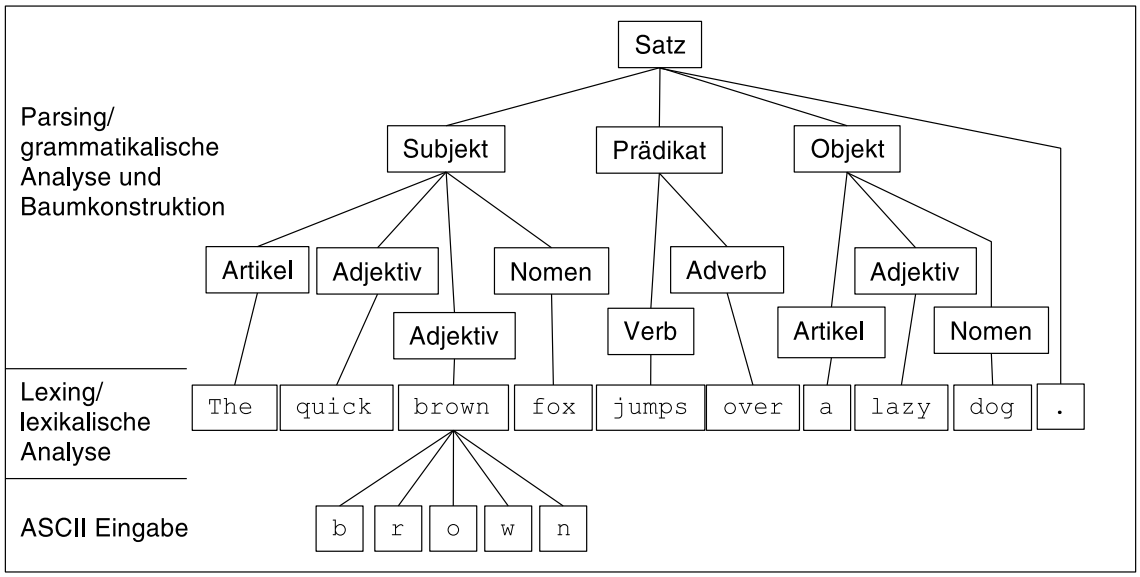


Abbildung 3.1.: Vom Text zum Parsebaum

einer Eingabesprache lesen, diesen Datenstrom transformieren und ihn abschließend in einer anderen Sprache wieder ausgeben. Heute verfügbare Compiler führen dabei verschiedene Transformationen aus, die den Ausgabestrom beispielsweise optimieren. Entsprechend ihrer konkreten Funktionsweise werden Compiler noch in weitere Unterkategorien eingeteilt, wobei hier die Grenzen oft nicht klar zu ziehen sind.

3.2.1. Konzepte und Konventionen

Die wichtigste Formalie bei der Entwicklung eines Compilers ist die Grammatik der Eingabesprache und die der Ausgabesprache. Diese werden üblicherweise in Backus-Naur-Form (BNF), Erweiterter Backus-Naur-Form (EBNF) oder einer Ableitung dieser Sprachen angegeben. Diese bestimmt die Syntax einer Sprache und damit die Obermenge der möglichen Eingaben für den Compiler.

Der Compiler arbeitet für gewöhnlich in mehreren Phasen die überlappend ausgeführt werden können. Diese sind:

1. lexikalische Analyse
2. syntaktische Analyse
3. semantische Analyse
4. Zwischencodegenerierung
5. Optimierung
6. Ausgabe

Bei Compilern deren Eingabesprache eine Obermenge der Ausgabesprache ist, werden die Phasen vier und fünf durch eine einzige Phase ersetzt, die die gewünschten Transformationen vornimmt. Eine Implementierung eines MetaC Compilers würde entsprechend ohne Phasen vier und fünf auskommen, wenn er als Ausgabesprache C verwendet.

Die Phasen eins und zwei sind in Abbildung 3.1 schematisch dargestellt. In der ersten Phase, d.h. in der lexikalischen Analyse, werden aus dem Datenstrom von Einzelzeichen die Ketten extrahiert, die als so genannte Tokens der Eingabesprache definiert sind. Das Ergebnis der ersten Phase ist ein Tokenstrom, der als Eingabe an die zweite Phase übergeben wird. Die zweite Phase führt die syntaktische Analyse durch und generiert in Abhängigkeit von den erkannten grammatikalischen Strukturen den Datenstrom, der von der nachfolgenden Phase benötigt wird.

Im Falle von MetaC entsteht an dieser Stelle ein abstrakter Syntax Baum (abstract syntax tree, AST), der die Quelltextstruktur der Eingabestrom repräsentiert. In der dritten Phase wird dieser Baum auf semantische Korrektheit überprüft. Die vierte Phase bzw. die Phasen vier und fünf führen die notwendigen Transformationen durch, um den Ausgabestrom zu erzeugen, welcher in der letzten Phase generiert wird.

3.2.2. Varianten von Parser

Ein wesentlicher Teil eines Compilers ist der Parser. Dieser führt die zweite Phase des Kompiliervorganges durch. Dabei wird der Eingabestrom syntaktisch analysiert und optional ein Parsebaum erstellt. Für die Implementierung von Parsern gibt es verschiedene Ansätze, die verschiedene Vor- und Nachteile haben. Für die gängigsten Varianten gibt es so genannte Compiler-Compiler. Das sind Programme, die aus einer Grammatikbeschreibung ein Programm generieren, das einen Eingabetext der entsprechenden Sprache syntaktisch analysieren kann.

Da der Einsatz eines solchen Werkzeugs die Entwicklung eines Compilers deutlich beschleunigt, muss die Eingabegrammatik in EBNF Darstellung zunächst in die Form des Compiler-Compiler gebracht werden. Die wichtigsten Parserklassen für die Compiler-Compiler exisitieren, sind LL(k), LR(k) und LALR(1). LALR(1) ist eine Spezialform von LR(k), die sich aufgrund der reduzierten Anforderungen an Speicher und der hohen Verarbeitungsgeschwindigkeit auf breiter Front für die Entwicklung von Compilern durchgesetzt hat. Dabei sind die Kompromisse die dafür eingegangen werden müssen, vergleichsweise gering.

Für die Details und Unterschiede der Parserklassen, sei auf [65] verwiesen. Wichtig für den Entwurf der Sprache MetaC ist hierbei lediglich, dass die Grammatik so entworfen werden sollte, dass sie in eine Form übersetzt werden kann die einfach zu implementieren ist. Somit sollte sie den Anforderungen von LALR(1) genügen. Die Überprüfung dieser Anforderung ist bei der Entwicklung des MetaC Compilers durch die Verwendung des Compiler-Compilers `yacc` [47] durchgeführt worden.

3.2.3. Symboltabellen

Für eine effiziente Implementierung eines Compilers sind Symboltabellen ein wichtiger Schlüssel. Symboltabellen speichern die Namen und Typen von durchgeführten Deklarationen. Dabei sind zwei wesentliche Eigenschaften zu berücksichtigen:

- Symboltabellen sind anhand der syntaktischen Struktur, zu der sie assoziiert sind, hierarchisch gegliedert.
- Die durchgeführten Deklarationen können in unterschiedlichen Namensräumen die voneinander unabhängig sind, durchgeführt werden.

Die erste Eigenschaft hat zur Folge, dass es zum einen zu Verdeckung von Symbolen kommen kann. Das heißt, dass ein Symbol in einer oberen Hierarchieebene von einem Symbol mit gleichem Namen in einer unteren Hierarchieebene verdeckt also unsichtbar gemacht wird. Zum anderen dürfen Symbole aus einer parallelen oder darunter liegenden Hierarchieebene nicht sichtbar sein, jedoch müssen die einer übergeordneten sichtbar sein.

Die zweite Eigenschaft hat zur Folge, dass es für jeden Namensraum eine eigene Hierarchie von Symboltabellen geben muss, die unabhängig von den anderen ist. Im Falle der Sprache C ist dies beispielsweise mit den Namen der Sprungziele und Variablennamen der Fall. So darf es also ein Sprungziel geben, das genauso heißt, wie eine Variable, die in der selben syntaktischen Struktur definiert ist. Beide müssen in unterschiedlichen Symboltabellen gespeichert werden.

3.2.4. Codegenerierung

Nach erfolgreicher Transformation des AST, welches in den nachfolgenden Kapiteln noch diskutiert wird, muss ein Compiler eine Ausgabe erzeugen. Dieser Prozess wird für gewöhnlich als Codegenerierung bezeichnet. Dabei wird im Falle von Transformatoren, also Compilern die einen AST lediglich umbauen, die interne Darstellung entsprechend ihrer Grammatik wieder in einen Tokenstrom umgewandelt und dieser wiederum in einen Strom von ASCII Zeichen. Es entspricht also vom Prinzip her dem umgekehrten Prozess der Phasen eins und zwei, wobei normalerweise die Erzeugung eines Tokenstroms ausgelassen wird und somit dieser Prozess tatsächlich in einer Phase durchgeführt wird.

Compiler, die nicht unter die Klasse der Transformatoren fallen, generieren die Ausgabe in einer anderen Sprache als sie die Eingabe gelesen haben. Somit ist die Ausgabegrammatik eine andere als die Eingabegrammatik. Das ist bei Compiler für normale Programmiersprachen der Fall. Hier wird in der Regel Maschinencode in einer Assemblersprache generiert, der wiederum von einem Assembler in ein systemspezifisches Binärformat übersetzt wird. In diesem Fall weicht die interne Repräsentation des Programms vom AST ab. Entsprechend dem Zielformat sind andere Eigenschaften (z.B. Daten- oder Kontrollfluss) zum Erreichen des Ziels (z.B. schneller und effizienter Maschinencode) wichtiger, als die Konservierung der ursprünglichen Darstellung.

Bei Transformatoren besteht hingegen der Wunsch des Anwenders, dass das Ausgabeformat nicht nur von der Syntax her der Eingabe entspricht, sondern auch möglichst nahe an dem gelesenen Original liegt. Dadurch ist es beispielsweise unmöglich, Informationen die für die Semantik eines Programms keine Rolle spielen (wie z.B. Namen der Variablen, konkrete Sequenz von Ausdrücken und Anweisungen) beim Einlesen oder während der Transformation zu verwerfen.

3.3. CASE-Tools und eingebettete Systeme

Für den Entwurf von Software für eingebettete Systeme gibt es eine Reihe von CASE-Tools, die sich zur Modellierung anbieten und die gleichzeitig eine direkte Targetierung von diversen Prozessorarchitektur-Betriebssystem-Kombinationen unterstützen. Die prominentesten Beispiele hierfür sind Matlab von The Mathworks [44], Telelogics Tau Suite [72], IBMs Rational Suite [42], Aonix Software Through Pictures [20] und Esterel Technologies Esterel [35].

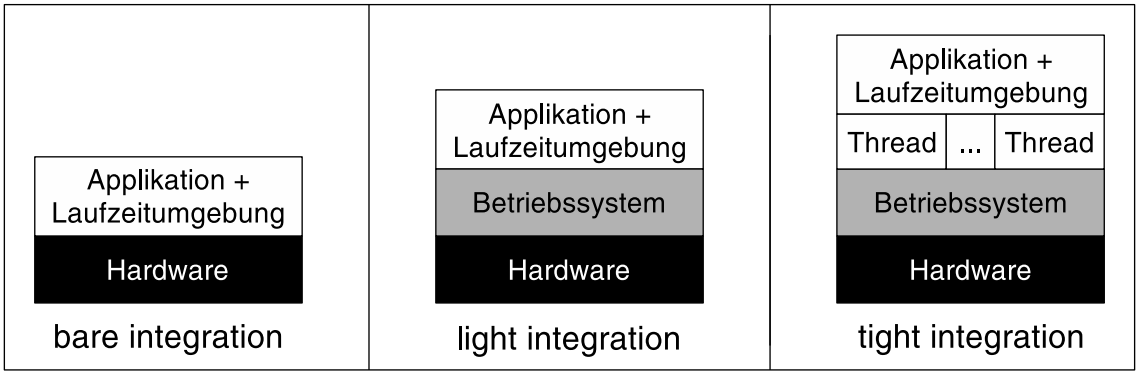


Abbildung 3.2.: Integrationsvarianten auf dem eingebetteten System

3.3.1. Integrationsvarianten mit dem Target

Software die mit Hilfe eines CASE-Tools entworfen wird, wird während des Entwurfsprozesses normalerweise in einer Simulationsumgebung auf dem Entwicklungsrechner einer Reihe von Tests unterzogen. Später muss diese Software aber auf einem eingebetteten System an die physikalische Welt gekoppelt werden, also mit Sensoren und Aktoren verbunden werden. Dazu gibt es verschiedene Verfahren, die die Applikationssoftware unterschiedlich eng an die Hardware bzw. die Betriebssystemumgebung koppeln.

Die wichtigsten gängigen Verfahren sind in [59] beschrieben und in Abbildung 3.2 konzeptionell dargestellt. Nicht alle diese Verfahren werden von gängigen CASE-Tools unterstützt, da dies entweder einen hoch flexible Codegenerierung oder einen enormen Aufwand bei der Integration der verschiedenen Zielsysteme erfordert.

Der höchste Integrationsaufwand tritt bei der *bare-integration* auf, da hier die Applikation ohne Betriebssystem auf der Hardware integriert wird. Das bedeutet, dass die Laufzeitumgebung sich nicht auf Betriebssystemprimitive stützen kann, sondern all diese Funktionalität vollständig implementiert werden muss.

Die *light-integration* erfordert einen ähnlich hohen Integrationsaufwand, da hier die Parallelität in der Applikation durch die Laufzeitumgebung realisiert wird und dabei in einer einzelnen Task oder Thread auf dem Betriebssystem läuft. Dies hat den Vorteil, dass Zusatzfunktionalität relativ einfach auf dem eingebetteten System nachträglich hinzugefügt werden kann, da hierbei das Betriebssystem einige Hürden aus dem Weg räumt. Auch bietet diese Variante Vorteile bei der Verwendung von Standardperipherie für die von Betriebssystemen Treiber angeboten werden.

Bei der *tight-integration* findet die engste Kopplung der Applikationssoftware an das Betriebssystem statt. Hierbei werden parallel laufende Pfade in der Applikation (z.B. SDL Prozesse) auf einzelne Threads abgebildet die vom Betriebssystem bereitgestellt werden. Die notwendige Abstraktionskomplexität ist bei dieser Variante am höchsten, dafür ist der relative Aufwand geringer als bei den anderen Verfahren.

Die Entscheidung für eines dieser Verfahren ist abhängig von den Anforderungen an das System. Keine Variante ist den anderen eindeutig überlegen. Durch die verschiedenen Anforderungen an den Grad der Abstraktion ist es normalerweise nicht möglich mit einem einzigen Codegenerator zu arbeiten. Abstraktionsmethoden, die auf abstrakten Datentypen und Funktionen basieren und mit Hilfe von MetaC umgesetzt werden (siehe Abschnitt 9), ermöglichen diese Lücke zu schließen ohne die Applikation in ihrer Flexibilität einzuschränken. Dies kann mit normalen Integrationsverfahren oder reinen Codegenerator gestützten Konzepten nicht erreicht werden.

3.3.2. Abstraktion von Architektur und Plattform

Das CASE-Tool Tau von Telelogic [72] unterstützt alle zuvor genannten Integrationsvarianten. Dazu werden zum einen verschiedene Codegeneratoren eingesetzt und zum anderen basiert die Codegenerierung auf einem großen Satz an Präprozessormakros, die zur Kompilierzeit entsprechend dem gewählten Zielsystem aufgelöst werden. Im Rahmen eines interdisziplinären Projekts am Lehrstuhl für Realzeit-Computersysteme der Technischen Universität München wurde die auf der Verwendung von Makros basierende Methodik eingehend untersucht und in [80] dokumentiert.

Das Konzept dieses CASE-Tools für die Codegenerierung basiert auf einem Satz von Präprozessormakros, die einfachste Realzeitprimitive des Betriebssystems nutzen. Eine mehrfache Verschachtelung der Makros ineinander erreicht die notwendige Abstraktion des darunter liegenden Betriebssystems und integriert gleichzeitig die Laufzeitumgebung.

Diese Methodik ist zwar mit angemessenen Aufwand für ein neues Target realisierbar, jedoch ist es dabei weder möglich spezielle Eigenschaften eines RTOS zu nutzen, noch das Design so abzuändern, dass dies möglich wird. Weiterhin bringt diese Methodik einen deutlichen Overhead mit sich, da redundante Operationen nicht erkannt werden können und auch von einem Compiler hinterher nicht mehr auflösbar sind. Ein MetaC basierter Ansatz könnte hingegen eine Struktur schaffen bei der einfache und komplexe RTOS Primitive gleichermaßen zum Einsatz kommen können. Dadurch wird die Implementierung schlanker und effizienter zugleich. Außerdem ist es dabei leichter möglich Erweiterungen vorzunehmen. Zusätzlich verkürzt sich die Entwicklungszeit durch die Feedback Fähigkeiten eines MetaC Compilers, der im Fehlerfall ausführliche Informationen liefert.

3.3.3. Modellierung von Software mit CASE-Tools

Entsprechend der Applikationsdomäne für die ein CASE-Tool ausgelegt ist, werden unterschiedliche Semantiken abstrahiert. Dahinter verbergen sich neben der konkreten Semantik der einzelnen graphischen Konstrukte auch Prinzipien und Konzepte, die einen Entwurf von Software für die jeweilige Problemklasse vereinfachen und robust gestalten sollen.

Unabhängig von den Ideen im Hintergrund haben alle CASE-Tools das Problem, dass die modellierte Software am Ende auf ein eingebettetes System gebracht werden muss. Da es meistens nicht wirtschaftlich ist für die verschiedenen Architekturen und Betriebssystemkombinationen direkt nativen Maschinencode zu erzeugen, wird oft der Umweg über eine Zwischensprache in Kauf genommen, der die Integration mit verschiedenen Targets vereinfacht.

Dazu gibt es verschiedene Verfahren um die Anzahl der Änderungen im Codegenerator möglichst gering, im Optimalfall gleich Null, zu halten. Jedes dieser Verfahren hat unterschiedliche Vor- und Nachteile. Die Kompromisse die dabei geschlossen werden, beziehen sich auf allgemeingültige Optimierungsziele beim Entwurf von Software, wie Codegröße, Verarbeitungsgeschwindigkeit, Beobachtbarkeit, Wartbarkeit, Turnaroundzeiten, usw. und auf die spezifischen Anforderungen der Werkzeugumgebung und der Applikationsdomäne. Diese Vielzahl von konkurrierenden Anforderungen zeigt bereits, dass es keine Universallösung für alle Problemklassen geben kann. Insbesondere entstehen häufig wesentliche Schwierigkeiten, wenn sowohl für das eingebettete System, als auch die Simulationsumgebung der selbe Codegenerator verwendet wird (siehe Abbildung 3.3).

Ein Ansatz der sehr flexibel ist, besteht in der Anbindung des Targets durch Metaprogramme. Prinzipiell können dadurch alle Integrationsvarianten unterstützt werden, und

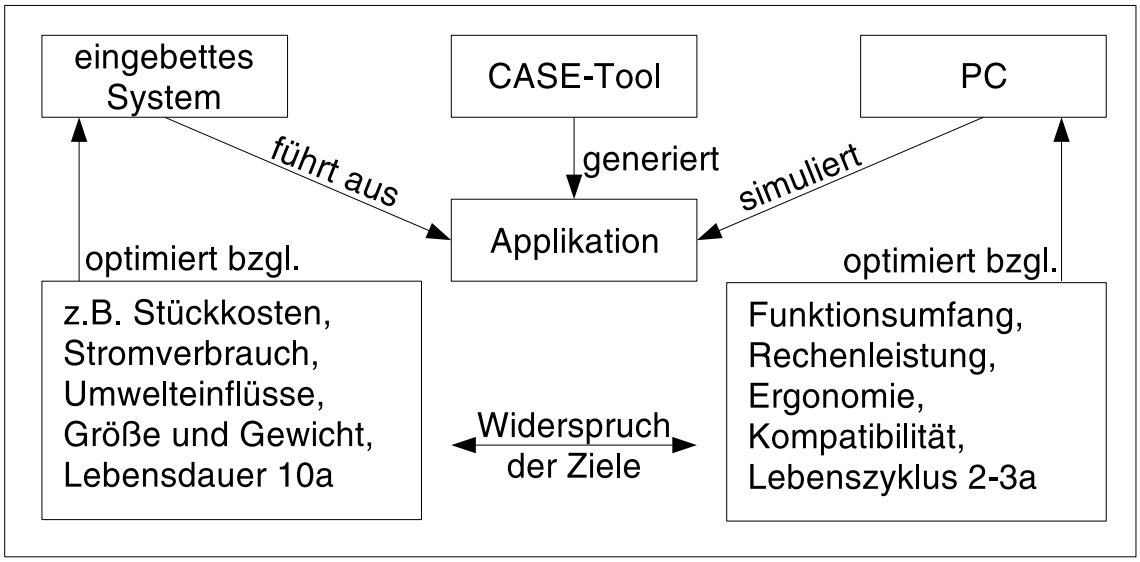


Abbildung 3.3.: Unterschiede in den Anforderungen an Simulations- und Zielsystem

bei durchdachter Konzeptionierung sind keine Änderungen am Codegenerator erforderlich. Dazu wird zum einen ein Satz an abstrakten Typen und Funktionen mit wohldefinierter Semantik spezifiziert. Diese werden abhängig vom Zielsystem und Integrationsart durch Metaprogramme realisiert. Dazu werden zur Kompilierzeit die abstrakten Datentypen durch konkrete ersetzt und die Funktionsaufrufe an das API des Zielsystems gekoppelt.

Diese Methodik wurde konzeptionell evaluiert und für ein Subset des Sprachschatzes von SDL [45],[60] im Rahmen eines Studentenprojekts (interdisziplinäres Projekt, IDP) [32] nachgewiesen. Es konnte gezeigt werden, dass die Sprachmittel von SDL durch entsprechende Konkretisierung der Semantik und Unterlagerung mit abstrakten Datentypen und Funktionen auf eine semantische Abstraktionsschicht abgebildet werden kann. Diese wiederum kann von einem Codegenerator generisch angesprochen werden und wird von Metaprogrammen anschließend zielspezifisch angebunden.

Die dafür notwendigen Konkretisierungen betreffen die Teile des SDL Sprachschatzes, die auf realen Systemen nicht ohne weiteres realisierbar sind. Das betrifft zum einen Idealisierungen, wonach keine zeitlichen Verzögerungen beim Transport von Nachrichten oder der Verarbeitungen von Befehlssequenzen auftreten. Zum anderen werden Puffergrößen als unendlich groß und Variablen von ihrem Wertebereich ausgehend als unbeschränkt modelliert. Diese Idealisierungen müssen bei der Realisierung aufgelöst werden.

Teil II.

Die Spracherweiterung MetaC

4. Konzepte und Ziele von MetaC

In diesem Kapitel wird erläutert welche Ziele beim Entwurf der Sprache MetaC verfolgt wurden und wie diese in Syntax und Semantik der Sprache zum Tragen kommen. Die beschriebenen Konzepte und Verfahren sind die Grundlagen für die Umsetzung von einer Reihe von Anwendungen, die in den Kapiteln 8 und 9 beschrieben werden.

MetaC als Spracherweiterung soll die notwendige Infrastruktur zur Lösung einer Reihe von Problemen bei der Rekonfiguration, Integration und Abstraktion von eingebetteten Echtzeitsystemen bieten. Durch die Definition einer solchen Erweiterung ergibt sich die Möglichkeit, die Hürden dieser Herausforderungen mit einer einzigen Beschreibungsform überwinden zu können. Existierende Ansätze stützen sich auf die Integration einer Reihe von Werkzeugen, die dabei unterschiedliche Probleme ansprechen. Das Zusammenspiel dieser Werkzeuge ist eine wesentliche Schwierigkeit bei der Lösung von Problemen, die die Applikationsbereiche der einzelnen Werkzeuge überspannen. Als Konsequenz ergibt sich, dass eine automatisierte Lösung einen deutlichen Overhead gegenüber einer manuellen Implementierung mit sich bringt.

Beim Entwurf einer Sprache sind eine Vielzahl von Entscheidungen zu treffen, die die Art und Weise ihrer Anwendung deutlich beeinflussen können. Auch auf die Implementierung eines Compilers können diese Entscheidungen von großer Bedeutung sein. Dies gilt ganz allgemein für Programmiersprachen, insbesondere aber bei domänenspezifischen Sprachen. Dieses Kapitel erläutert zunächst, welche Entscheidungen bei dem Entwurf von domänenspezifischen Programmiersprachen mit hohem Verbreitungsgrad getroffen wurden, und wie sich diese Entscheidungen auswirken. Danach werden die Ziele beim Entwurf von MetaC erläutert und die Prinzipien erklärt, die bei der Realisierung dieser Sprache zum Einsatz kommen.

4.1. Designentscheidungen domänenspezifischer Sprachen

Die größte Aufmerksamkeit beim Entwurf einer domänenspezifischen Sprache gilt der Abstraktion häufig auftretender Strukturen. Diese Strukturen sollen aussagekräftige Darstellungen haben und einfach zu beschreiben sein, d.h. mit möglichst wenigen Schlüsselwörtern und Parametern auskommen. Gleichzeitig besteht die Notwendigkeit, dass alle Freiheitsgrade der domänenspezifischen Konstruktion berücksichtigt werden müssen. Zum Teil können diese konkurrierenden Anforderungen gleichzeitig befriedigt werden, indem ein Teil der Freiheitsgrade eingeschränkt wird und wohldefinierte Semantik spezifiziert wird. Die Verwendung von domänenspezifischen Schlüsselwörtern und Operatoren kann außerdem die Lesbarkeit verbessern und den Abstraktionsgrad vergrößern.

4.1.1. Ada

Ein Beispiel für eine domänenspezifische Abstraktion, ist die Unterstützung von Tasks in der Programmiersprache Ada. Eine Task hat neben der Eigenschaft, dass sie parallel zu anderen Tasks verarbeitet wird oder werden kann, noch eine Reihe von weiteren Freiheitsgraden. Beispiele hierfür sind ihre Priorität, ihre Scheduling-Klasse sowie das Abbruch- und Synchronisationsverhalten. Da alle diese Eigenschaften wichtig sind, um einem Programm definiertes Verhalten geben zu können, dürfen sie nicht einfach ignoriert werden. In der Regel haben die meisten dieser Freiheitsgrade in gängigen Implementierungen den gleichen Wert. Das heißt als Schedulingverfahren wird beispielsweise für gewöhnlich First-In-First-Out (FIFO) within-fixed-priority eingesetzt.

Auch die anderen Freiheitsgrade werden meist nicht ausgenutzt, obwohl es durchaus Einzelimplementierungen gibt, die sie sich zu Nutze machen. Beim Entwurf von Ada hat man diese Tatsache unterstützt indem alle Tasks immer mit diesem voreingestellten Schedulingverfahren erzeugt werden. Dazu erhält der Anwender die Möglichkeit durch spezielle Zusätze manuell ein anderes Verfahren auszuwählen. Trotzdem wurde dieser Freiheitsgrad nicht als wichtig genug angesehen, um in der Syntax zur Erzeugung einer Task unmittelbar mit aufgenommen zu werden. Durch das voreingestellte Verhalten wird somit der Quelltext übersichtlicher und dennoch geht keine wichtige Funktionalität verloren.

Allerdings waren nicht alle Entscheidungen so glücklich, wie die zuvor genannte. Für die Programmiersprache Ada war nach ihrer Spezifikation ein großes Hindernis, dass die enorme Anzahl an Schlüsselworten mit den üblichen Rechnern zu dieser Zeit[1)] nicht ohne weiteres zu bewältigen war. Dies verhinderte eine kostengüstige Implementierung von Compilern und setzte gleichzeitig hohe Anforderungen an Entwicklungssysteme. Diese Situation verschärfte sich weiter, da eine Verwendung des Namens Ada nur erlaubt ist, wenn die strengen Vorgaben der Spezifikation eingehalten werden. Die Programmiersprache C stellt an die Implementierung keine derart harten Anforderungen, jedoch wirken sich einige der Freiheitsgrade die eine Compilerentwicklung vereinfachen negativ auf die Anwendung aus. Dazu zählen unter anderem die fehlende strenge Typkontrolle, die eine Reihe von Fehlern in Programmen unerkannt kompilieren lässt. Ebenso sind die zahlreichen Stellen in der Definition, die undefiniertes Verhalten für die Implementierung spezifizieren, eine häufige Quelle von Fehlern.

4.1.2. Fortran

Der Name Fortran ist aus der Abkürzung von *Formular Transformation* entstanden. Der Name entspricht der Anwendungsdomäne dieser Programmiersprache: sie wird benutzt um mathematische Formeln in schnellen Maschinencode umzuwandeln. Neben den gewöhnlichen mathematischen Operatoren zur Verarbeitung von Ganz- und Gleitkommazahlen verfügt Fortran auch über spezielle Operatoren um Matrizenoperationen durchführen zu können. Diese Entscheidung trägt wesentlich dazu bei, dass Fortran basierte Programme in dieser Domäne nach wie vor eine führende Position einnehmen. Dies zeigt sich sowohl an der Anzahl der Programme, die in dieser Sprache implementiert, als auch in der Geschwindigkeit der optimierten Binärdateien.

Die direkte Integration der wesentlichen Matrixoperatoren als Sprachkonstrukte, ermöglicht dem Programmierer, mit wenigen Zeilen ausdrucksstarke Programme zu schrei-

[1)] erster Standard von 1983

ben, die komplexe Probleme lösen und in anderen Sprachen ein vielfaches an Aufwand erfordern. Dies tröstet die Anwender über den Nachteil der umständlichen Syntax von Fortran hinweg, die empfindlich auf Weißzeichen (Leerzeichen, Tabulator, Zeilenumbruch) ist. Diese ungewöhnliche Syntax wurde in einer späteren Revision durch eine Alternative ersetzt. Die Mächtigkeit des Konzepts Matrizen als grundlegende Datentype zu unterstützen, lässt sich auch an der Tatsache erkennen, dass viele große Softwarepakete wie Matlab, Mathematika, usw. diesen Ansatz übernommen haben.

4.1.3. Perl

Perl (Practical Extraction and Report Language) wurde für die Vereinfachung und Automatisierung von administrativen Aufgaben im UNIX Umfeld geschaffen. Dafür besteht die Notwendigkeit, dass Analysen und Modifikationen an ASCII Textdateien einfach durchzuführen sind. In Standard UNIX Umgebungen werden dafür schon seit langem Programme wie Beispielsweise **`awk, cut, grep`** und **`sed`** bereitgestellt. Das Konzept, das allen diesen Programmen zugrunde liegt, ist der Einsatz von regulären Ausdrücken zum Suchen und/oder Ersetzen von Textbausteinen.

Entsprechend überrascht es nicht, dass auch in Perl reguläre Ausdrücke eine wichtige Rolle spielen. Zum Erfolg von Perl hat dabei mit Sicherheit wesentlich beigetragen, dass solche Konstrukte nicht nur verwendet werden können[2)], sondern dass ihr Einsatz durch Operatoren vereinfacht wird. Diese unterstützen alle wichtige Operationen durch eine kompakte Syntax, die direkt in der Sprache verankert ist.

Auch ist der einzige Typ, der in Perl für Variablen verfügbar ist, ASCII Text. Integervariablen können trotzdem verwendet werden, sind aber lediglich eine spezielle Interpretation einer Zeichenkette, die in einer Variable gespeichert ist. Entsprechend einfach fallen Aufgaben wie die Umwandlung von Zahlen in ASCII Text und umgekehrt, die in C mit erheblichen Aufwand verbunden sind. Der Aufwand bei einem C Programm liegt in der notwendigen Überprüfung von Randbedingungen, die in C manuell durchgeführt werden muss. Auch die dafür erforderliche Speicherverwaltung erfordert eine problemspezifische Implementierung.

4.2. Zielsetzungen von MetaC

Das Ziel bei der Definition der Spracherweiterung MetaC ist Möglichkeiten zu schaffen, die eine integrierte Lösung von Problemen beim Entwurf und Entwicklung von eingebetteten Echtzeitsystemen ermöglicht. Insbesondere sollen Modulgrenzen überschritten und Abstraktionsebenen überspannt werden können. Entsprechend dem vorgesehenen Applikationsszenario standen beim Entwurf von MetaC folgende Punkte im Zentrum der Aufmerksamkeit:

- Schaffung einer echten *two-level-language*
- Abwärts-Kompatibilität zu ISO-C90
- Bereitstellung von reflexiven Mechanismen zur Analyse von Quelltext
- Bereitstellung von Mechanismen zur Rekonfiguration und Refaktorisierung von Quelltext

[2)] In C ist dies auch möglich, wird aber über vergleichsweise umständliche Bibliotheksaufrufe realisiert.

- Bewusstsein für folgende wesentliche Spracheigenschaften der Sprache C:
  - Syntax
  - Typ
  - Kontrollfluss
  - Datenfluss

In MetaC gibt es Datentypen zur Referenzierung von Quelltextstrukturen und spezielle Metafunktionen, um auf Elemente der referenzierten Struktur zugreifen zu können. Darüber hinaus gibt es Möglichkeiten andere Eigenschaften zu überprüfen, wie die Relation der Strukturen untereinander und ihre Randbedingungen. Die gewählte Methodik, die die notwendige Infrastruktur für diese Analysemöglichkeiten bietet, stützt sich auf die Syntax der Sprache C.

Dabei entfällt allerdings die Möglichkeit, den Quelltext wie allgemein üblich auf Zeilenbasis zu referenzieren, da Zeilen in C keinerlei syntaktische und semantische Relevanz haben. Indirekt gibt es aber Möglichkeiten, um auf Strukturen unter Berücksichtigung von Zeileninformationen zuzugreifen. Dazu werden diese Metainformationen als zusätzliche Eigenschaften an verschiedenen Metadatentypen gehängt. So können diese Informationen während der Rekonfiguration und Refaktorisierung genutzt werden, um dem Benutzer Rückkopplung über durchgeführte Veränderungen auf einer Basis zu geben, die ihm sehr vertraut ist.

4.3. Neue Datentypen in MetaC

Neben den Standarddatentypen, die je nach Applikationsdomäne unterschiedlich sein können (z.B. Integer, String, Gleitkommawert), benötigen Metasprachen Datentypen welche zur Verarbeitung der reflexiven Eigenschaften geeignet sind. Bei der Erweiterung einer existierenden Programmiersprache, ergeben sich die notwendigen Reflexionen unmittelbar aus Syntax und Semantik der Basissprache. Dabei ist es wichtig die strukturellen und semantischen Eigenschaften, die reflektiv darstellbar sein müssen in einer geeigneten Granularität zu präsentieren. Das bedeutet, es muss Konzepte geben um passende Verallgemeinerungen durchzuführen. Dadurch ist die Sprache am Ende nicht überfrachtet mit Erweiterungen, die ähnliche Wirkung haben und deren minimale Unterschiede auch anders abstrahiert werden könnten.

Für die Erweiterung von C zu MetaC werden zur Reflexion der syntaktischen Struktur die Rekursionspunkte der Grammatik herangezogen. Unter diesen Rekursionspunkten sind die verschiedenen Ausprägungen einer gemeinsamen Struktur versammelt. Die Hierarchie beinhaltet gleichzeitig implizit die Priorisierung der Varianten untereinander.

Betrachtet man beispielsweise den Parser *expression* (siehe Anhang A.3), so ist dieser ein Rekursionspunkt, der von *primary-expression* und *expression* selbst benutzt wird. Dadurch wird jede grammatikalische Struktur, die durch einen Parser der im Rekursionspfad zwischen *expression* und *primary-expression* liegt, letztendlich auf *expression* reduziert. Gleiches gilt auch für den Parser *statement*. Dieser ist über die verschiedenen untergeordneten Parser indirekt rekursiv.

Somit bieten sich diese beiden Strukturen als abstrakte Basis zur Bildung von Metadatentypen an. Aus ihnen wurden die Metadatentypen **`expr`** und **`stmt`** abgeleitet. Diese beiden Metadatentypen sind allein noch nicht ausreichend um beliebigen C Quelltext vollständig zu abstrahieren. Eine weitere Rekursion der Syntax ist in den Parsern *declarator*

und *direct-declarator*. Allerdings ist es hier sinnvoll, nicht *declarator* als Metadatentyp zu realisieren, da dieser nur die Typerweiterung eines Basistyps ausdrückt. Besser ist es einen Metadatentyp zu realisieren, der den vollständigen Datentyp beinhaltet, also inklusive der *specifier-qualifier-list*. Dies ist mit der Definition des Metadatentyps **`type`** durchgeführt worden.

Dazu wurde der Parser *specifier-qualifier-list* mit integriert. Dieser beinhaltet unter anderem als Alternative einen *typedef-name*, der einem Typalias bestehend aus einer *specifier-qualifier-list* und *declarator* entspricht. Dadurch ist die Rekursion wieder geschlossen und beliebige C Datentypen werden darstellbar.

4.4. Metadatentypen und ihren Operatoren

Wie an den zuvor genannten Beispielen domänenspezifischer Sprachen unmittelbar erkennbar ist sind die Entscheidungen, die beim Entwurf einer Sprache in Bezug auf ihre Datentypen und zugehörigen Operatoren getroffen werden, von wichtiger Bedeutung. Die Anzahl der in C zur Verfügung stehenden Operatoren ist zum einen durch den grundlegenden Zeichensatz (7Bit ASCII) und zum anderen durch die semantischen Interpretationsmöglichkeiten seitens des Anwenders begrenzt.

Mit einer Abkehr von dieser Semantik durch eine vollkommene Neuinterpretation der verfügbaren Operatoren, würde man Gefahr laufen, die Sprache bis zur Unkenntlichkeit zu verändern. Andererseits ist auch der Einsatz einer großen Zahl an Schlüsselwörtern nicht möglich. Beide Randbedingungen können durch die Anlehnung an die in C vorhandene Standardbibliothek erreicht werden. Anstatt alle möglichen Operationen der verschiedenen Datentypen durch Operatoren zu realisieren, werden sie durch aussagekräftige Metafunktionen realisiert. Das bedeutet, dass für die neuen Metadatentypen nur die wesentlichen und intuitiven Operationen durch Operatoren unterstützt werden.

Zu den Konstruktionen die durch spezielle Syntax unterstützt werden, gehören jene Operationen, die durch Metafunktionen nicht realisierbar sind. Diese werden in MetaC unter dem Begriff des Quelltext-Strukturmusters zusammengefasst und in Kapitel 6 erläutert. Mit ihnen ist es möglich einen Quelltext zu suchen, zu ersetzten, neu einzufügen und Datentypen applikationsspezifisch zu generieren. Dazu werden lediglich ein neues Token und zwei neue Schlüsselwörter benötigt, deren konkrete Syntax in den entsprechenden Abschnitten von Kapitel 5 erläutert wird.

5. Syntax und Semantik von MetaC

Syntax und Semantik sind die Grundkonzepte mit denen jede Programmiersprache beschrieben wird. Im Vergleich zu menschlichen Sprachen repräsentiert die Syntax einer Programmiersprache die Grammatik einer menschlichen Sprachen. Die Semantik beschreibt die Bedeutung, die gebildeten Sätzen einer Sprache zugeordnet wird. Sätze die mit menschlichen Sprachen gebildet werden, sind meistens in ihrer Bedeutung von Vielseitigkeit geprägt, die abhängig von dem Interpretierer (also dem Menschen, der dem Satz eine Bedeutung zuordnet) zu vollkommen unterschiedlichen Ergebnissen führen kann. Diese Ungenauigkeit kann gewollt sein (z.B. bei einem literarischen Werk wie einem Gedicht), aber auch zu enormen Problemen führen (z.B. bei einem Gesetzestext oder der Spezifikation eines technischen Gerätes).

Da komplizierte Funktionalität nur korrekt umgesetzt werden kann wenn ein genauer Ablaufplan der durchzuführenden Schritte vorliegt, müssen Programmiersprachen so gestaltet sein, dass Mehrdeutigkeiten nicht auftreten können. Diese Forderung wird im Allgemeinen als *wohl-definiertes Verhalten* bezeichnet. Dieses Kapitel beschreibt Syntax und Semantik von *MetaC* in Abhängigkeit zur Muttersprache *C*, die in [17] definiert ist. Dazu werden zunächst einmal die pragmatischen Aspekte der Spracherweiterung betrachtet die keine semantischen Implikationen mit sich bringen. Danach folgen wesentliche grundlegende Definitionen und die syntaktischen und semantischen Erweiterungen, die erforderlich sind um die Zielsetzungen der Sprache umsetzten zu können.

5.1. Look and Feel

Um ein einheitliches Look and Feel von MetaC- und C-Programmen gewährleisten zu können, ist eine vollständige Abwärtskompatibililtät der Sprache erforderlich. Folglich darf keine der existierenden syntaktischen Strukturen verändert werden, und Erweiterungen müssen konzeptionell der *Beständigkeit* [68] des Sprachkonzepts entsprechen. Das heißt, dass neu hinzugefügte Konstrukte sich nahtlos in die bereits existierende Methodik eingliedern müssen. Darüber hinaus sollten die vorhandenen Konzepte der Sprache unangetastet bleiben und für Erweiterungen übernommen werden, damit der Anwender bei Benutzung der Erweiterungen nicht das Gefühl bekommt, etwas vollkommen neues erlernen zu müssen.

Dazu gehören bei der Programmiersprache C neben den offensichtlichen Eigenschaften, wie der Unterscheidung zwischen Groß- und Kleinschreibung und der Verwendung von Klammern statt Schlüsselwörtern wie **`begin`** und **`end`**, auch der sparsame Einsatz von kurzen Schlüsselwörtern und der Einbindung aller Operatoren in die grammatikalische Struktur für Ausdrücke. Dabei muss insbesondere die Ausdrucksform von C beachtet werden. Diese ermöglicht es unter anderem auch Zuweisungen als Operanden zu verwenden und so das Ergebnis einer Berechnung gleichzeitig zu speichern und für eine Folgeoperation zu verwenden. Darüber hinaus gibt es Ausdrücke, deren Wirkung sofort eintritt, und andere, deren Ergebnis erst am nächsten *Sequenzpunkt* sichtbar wird.

Die Forderung, den optischen Eindruck und die Wirkung von Quelltext aufrechtzuerhalten, lässt sich am einfachsten realisieren, indem die existierenden Ausdrücke und Anweisungen der Sprache C mit äquivalenter Semantik für Metaprogramme wiederverwendet werden. Äquivalent bedeutet es hierbei, dass die Laufzeitsemantik in eine Kompilierzeitsemantik übersetzt wird. Gelten dabei die gleichen Rahmenbedingungen wie für ein übersetztes C-Programm, so lässt sich daraus ableiten, dass ein Metaprogramm sich prinzipiell identisch zu einem Programm mit äquivalentem Quelltext verhält.

5.1.1. Abwärtskompatiblität

Bei der Erweiterung einer Sprache um neue Konstrukte muss unbedingt sichergestellt werden, dass die Abwärtskompatiblität gewährleistet ist. Dabei müssen neben der ursprünglichen Syntax auch die zugehörige statische und dynamische Semantik erhalten werden. Dies ist die Voraussetzung dafür, dass alte Applikationen ohne Änderungen weiterhin funktionieren. Für eine Spracherweiterung die das ausgesprochene Ziel hat alte Quelltexte für neue Applikationen nutzbar zu machen, gilt dies um so mehr. Deshalb gibt es in den syntaktischen und semantischen Erweiterungen von MetaC auch keine Definitionen, die die Abwärtskompatiblität beeinträchtigen.

Einzig auf ein neues Schlüsselwort, das die Potenzmenge der verfügbaren Bezeichner um ein einzelnes Element reduziert, kann nicht verzichtet werden. Diese Vorgehensweise entspricht gängiger Praxis bei allen Programmiersprachen. So wurde beispielsweise das Wort `restrict`, das in C89 noch als Bezeichner zur Verfügung steht, in der Revision von 1999 zu einem Schlüsselwort. Das neu eingeführte Schlüsselwort von MetaC ist `meta`. Es dient zur Aktivierung der Spracherweiterung und schaltet dafür einige zusätzliche Schlüsselwörter frei, die dann ähnlich zu Typalias Definitionen behandelt werden. Diese Funktionalität und die notwendige grammatikalische Erweiterung werden in Abschnitt 5.6.1 erläutert.

5.1.2. Strukturelle Dreiteilung

Die erweiterte Syntax von MetaC sieht eine klare Trennung zwischen normalem Programmcode und Daten sowie Metaprogrammcode und Metadaten vor. Innerhalb von Metaprogrammen werden zusätzlich Quelltext-Strukturmuster (QSM, siehe Kapitel 6) definiert. Diese haben wiederum eine andere Semantik und Namensauflösung als der sie umgebende Quelltext für Metaprogramme. Das bedeutet, dass sich der Quelltext strukturell in drei Bereiche aufteilen lässt, die durch syntaktische Merkmale unterschieden werden. Die Details der verwendeten lexikalischen Elemente zur Trennung während des Parsevorgangs werden weiter hinten beschrieben.

5.1.3. Konkretisierung der Semantik

Die statische und dynamische Semantik von MetaC sind aus der Semantik von C abgeleitet. Die Spezifikation von C beinhaltet an vielen Stellen nichtdefiniertes Verhalten. Alle diese Situationen in denen das Verhalten nicht definiert ist, beziehen sich auf Fehler im Programmablauf. Der Vorteil von nichtdefiniertem Verhalten im Standard liegt in der einfacheren Implementierung von Standard konformen Compilern. Denn diese können somit für die jeweilige Zielplattform auf die einfachste Art und Weise realisiert werden.

Allerdings soll von den Metaprogrammen keine ausführbare Binärdatei erstellt werden, sondern die Metaprogramme sollen zur Kompilierzeit ausgeführt werden. Somit entfallen die meisten Standardverfahren zur Fehlersuche und -behebung. Es ist davon auszugehen, dass auch bei der Entwicklung von Metaprogrammen fehlerhafter Code entsteht. Deshalb wird für die Implementierung eines MetaC Compilers empfohlen, dass alle Situationen in denen nichtdefiniertes Verhalten vorliegt eine Fehlermeldung erzeugen. Dies kann entweder zur Laufzeit geschehen, oder wenn der Fehler bereits zur Übersetzungszeit des Metaprogramms erkennbar ist, noch bevor das Metaprogramm zur Ausführung kommt.

5.2. Grundlegende Definitionen

In diesem Abschnitt werden grundlegende Begriffe definiert. Ihre exakte Definition und Abgrenzung voneinander erleichtert die Beschreibung von komplexen Zusammenhängen. Da MetaC eine Erweiterung von C darstellt, werden zunächst einmal die wesentlichen, bereits im C Standard festgelegten Definitionen aufgegriffen und dann die erweiterten Definitionen und Ableitungen für MetaC erläutert.

5.2.1. Übernommene Definitionen

Die folgenden Definitionen sind direkt aus dem C Standard übernommen worden, und die feststehenden Begriffe wurden zur besseren Lesbarkeit ins Deutsche übersetzt. Die englische Bezeichnung ist zum einfachen Referenzieren in Klammern gesetzt. Die Begriffe entsprechen den Definitionen in [65].

Bit Eine Einheit zur Speicherung von Daten, die eine von zwei Werten annehmen kann.

Byte Das Byte ist in C als adressierbare Einheit zur Speicherung von Daten definiert, die groß genug ist, um ein beliebiges Zeichen der grundlegenden Zeichenmenge der Laufzeitumgebung zu speichern und die als Bezugsgröße für alle anderen Größenangaben herangezogen wird. Dabei ist nicht eindeutig festgelegt wie viele Bit ein Byte hat. Üblicherweise hat ein Byte auf den gängigen Architekturen 8 Bit, und der POSIX Standard setzt dies auch als Grundbedingung voraus (siehe [43], Abschnitt 3.84). Deshalb kann bei allen Beispielen, die hier präsentiert werden von dieser Annahme ausgegangen werden. Dies ist aber keine zwingende Voraussetzung für eine korrekte Implementierung eines MetaC Compilers und somit auch keine Einschränkung von *MetaC* gegenüber C.

Objekt (*object*) Ein Objekt bezeichnet den Speicherbereich, auf den über eine bestimmte Deklaration zur Laufzeit zugegriffen werden kann. Objekte haben im Speicher immer eine feste Adresse und eine konstante Größe.

Bezeichner (*identifier*) Als Bezeichner werden lexikalische Sequenzen aus Buchstaben, Zahlen und Unterstrichen (_) bezeichnet, die nicht mit einer Zahl beginnen. Dieses Token wird verwendet um Definitionen und Deklarationen einen Namen zuzuordnen. Ein und derselbe Bezeichner kann dabei für verschiedene Definitionen und Deklarationen verwendet werden, da die Bezeichner in verschiedenen Namensräumen (*namespace*) und Gültigkeitsbereichen (*scope*) aufgelöst werden.

Verhalten (*behaviour*) Verhalten sind die von außen beobachtbaren Veränderungen, die von der Implementierung eines Programms erzeugt werden.

Undefiniertes Verhalten (*undefined behaviour*) Das Ergebnis bestimmter Operationen ist nicht definiert. Das bedeutet, dass eine Implementierung ein beliebiges Verhalten zeigen darf. Dieses Verhalten kann darüber hinaus auch jedes mal unterschiedlich sein. Ein Programm darf niemals undefiniertes Verhalten hervorrufen, da dadurch der Programmzustand in unvorhergesehener Art und Weise verändert werden könnte. Das Ergebnis der Programmausführung wird in einem solchen Fall fehlerhaft.

Undefiniertes Verhalten wird häufig von weniger erfahrenen Programmierern hervorgerufen, die ihr Programm nur mit einem Compiler auf einem System testen und sich der Tatsache mangels einer Meldung des Compilers nicht bewusst sind. Durch den Einsatz von verschiedenen Compilern und unterschiedlichen Testsystemen mit unterschiedlicher Architektur (insbesondere auch Endian) treten solche Fehler wesentlich schneller auf, da meistens ein Zielsystem dabei ist, das den Fehler mit einem Programmabsturz quittiert.

Leider bieten die meisten Compiler keine Unterstützung, um zur Laufzeit undefiniertes Verhalten mit der Meldung von Laufzeitfehlern anzuzeigen. Solche Lösungen sind aber durchaus machbar (z.B. siehe [21], [48]), wenn auch teurer in der Implementierung. Um nicht immer derart aufwändig testen zu müssen, sind aus Forschungsarbeiten Werkzeuge wie Lint [46] und Purify entstanden, sowie Erweiterungen für Laufzeitumgebungen und Debugger (z.B. libumem unter Solaris und die dedizierten Laufzeitüberprüfungen im dbx Debugger von Sun).

Unspezifiziertes Verhalten (*unspecified behaviour*) Bei unspezifiziertem Verhalten gibt es mehrere mögliche Varianten für die Implementierung, von denen aber genau eine ausgewählt werden muss, die für jede Ausführung von ein und demselben Konstrukt gleich sein muss. Im Gegensatz zum *undefinierten Verhalten* muss das Verhalten bei jeder Ausführung das Gleiche sein.

Parameter Eine Variable, die vor Ausführung des Funktionsaufrufes von der aufrufenden Funktion aus initialisiert wird. Diese Variable wird nach der Initialisierung nur von der aufgerufenen Funktion benutzt, und die aufrufende Funktion benutzt sie nach der Initialisierung nicht mehr.

Argument Der Wert, der während eines Funktionsaufrufs der aufgerufenen Funktion übergeben wird, um einen Parameter zu initialisieren.

Zugriff (*access*) Das Lesen oder Schreiben der Daten in einem Objekt.

Linkswert (*Lvalue*) Ein Linkswert ist das Ergebnis eines Ausdrucks, das auf ein modifizierbares Objekt verweist.

Rechtswert (*Rvalue*) Ein Rechtswert ist das Ergebnis eines Ausdrucks mit einem wohldefiniertem Wert, der als Operand in einem anderen Ausdruck verwendet werden kann.

Nebenwirkung (*side-effect*) Als Nebenwirkung wird das Schreiben auf ein Objekt, der Zugriff auf eine als volatil gekennzeichnete Variable und die Modifikation einer Datei bezeichnet. Auch Funktionsaufrufe, die eine der zuvor genannten Operationen ausführen, haben eine Nebenwirkung.

Abfolgepunkt (*sequence point*) Abfolgepunkte bestimmen in der Ausführung den spätesten Zeitpunkt, an dem das Ergebnis eines syntaktischen Konstrukts für nachfolgende Konstrukte in seiner Wirkung sichtbar sein muss.

Gültigkeitsbereich (scope) Der Gültigkeitsbereich legt fest, in welchen Teilen eines Programmes Deklarationen und Definitionen sichtbar sind. Die Sichtbarkeit steht aber in keinem direkten Zusammenhang mit der Lebensdauer der zugehörigen Objekte. Gültigkeitsbereiche sind hierarchisch angeordnet, und Deklarationen und Definitionen in Gültigkeitsbereichen einer niedrigeren Hierarchieebene verdecken gleichnamige in höheren Hierarchieebenen.

Namensraum (*namespace*) Für unterschiedliche Deklarationen und Definitionen werden in ein und dem selben Gültigkeitsbereich verschiedene Namensräume verwendet, in denen die zugehörigen Bezeichner aufgelöst werden. Zur Identifikation des Namenraumes wird der grammatikalische Kontext herangezogen, in dem sich der Bezeichner befindet. Folgende Namensräume und grammatikalische Zusammenhänge werden dabei unterschieden:

- Im normalen Namensraum werden alle Bezeichner aufgelöst, die sich in keinem der nachfolgend aufgeführten grammatikalischen Kontext befinden. Dazu gehören Typdefinitionen, Variablendeklarationen und Funktionsdefinitionen und Enumeratoren.
- Ist dem Bezeichner eines der Schlüsselwörter **`struct`**, **`union`** oder **`enum`** vorangestellt, so wird er im Namensraum für Datenstrukturen und Enumeratorspezifikationen aufgelöst.
- Bezeichner für Strukturelemente werden durch die vorangestellten Token Punkt (**`.`**) oder Subskriptoperator (**`->`**) als solche erkannt und in dem jeweiligen Namensraum der zugehörigen Datenstruktur aufgelöst.
- Ist dem Bezeichner das Schlüsselwort **`goto`** vorangestellt oder wird er in dem Parser *labeled-statement* als Sprungmarke erkannt, so wird er im Namensraum der Sprungmarken aufgelöst, der immer im Gültigkeitsbereich der zugehörigen Funktion liegt.

Lebensdauer von Objekten In C sind Objekte Bereiche von Speicher für die Haltung von Daten die verschiedene Werte repräsentieren können. Die Lebensdauer dieser Objekte hängt zum einen vom Gültigkeitsbereich, zum anderen von den verwendeten Qualifizierern in der zugehörigen Deklaration ab. So entspricht die Lebensdauer von Variablen mit globaler Sichtbarkeit der Lebensdauer des Programms selbst. Variablen mit lokaler Sichtbarkeit haben eine eingeschränkte Lebensdauer, die durch den *storage-type-specifier* **`static`** auf die Ausführungszeit des Programms erweitert werden kann(siehe [17], Abschnitt 6.2.4). Der Datentyp des Objekts hat dabei keinen Einfluss auf die Lebensdauer.

5.2.2. Neue Definitionen

Die nachfolgenden Definitionen sind so nicht im ISO-Standard von C enthalten, werden aber benötigt um die Konzepte, die Systematik und Realisierung von MetaC erklären zu können.

Symbol

Ein Symbol oder auch symbolischer Name ist ein Bezeichner mit seiner zugehörigen Deklaration, aus der sich der Datentyp ergibt und dem entsprechenden Gültigkeitsbereich. Mit einem Symbol ist es möglich, Variablen und Funktionen zu referenzieren.

Definition 1 *Der Begriff* Symbol *ist definiert als folgendes 3-Tupel:*

$$s = (i, c, t)$$

- i *ist ein Bezeichner (identifier)*
- c *ist der Definitionskontext (scope)*
- t *ist der Typ des Symbols (type)*

Metaobjekt

Ein Metaobjekt oder auch Metadatenobjekt ist ein Speicherbereich, dessen Inhalt den Wert einer Metadatendeklaration zur Laufzeit eines Metaprogramms hält. Die Metadatendeklaration besteht aus der zugehörigen Symboldefinition und der Lebensdauer der Metadatenvariable, die sich aus dem Gültigkeitsbereich und dem optionalen *storage-class-specifier* `static` ableitet. Wird dieses Schlüsselwort in einer Metadatendeklaration verwendet, so wird die Lebensdauer des Metaobjekts auf die Lebensdauer des Metaprogramms verlängert, und die zugehörige Deklaration wird im nachfolgenden Programmablauf nicht mehr ausgeführt.

Im Gegensatz zum Objekt der Sprachdefinition von C ist es nicht möglich, die Adresse des Metaobjekts im Speicher zu bestimmen. Die Art und Weise der Speicherallokation wird durch eine Implementierung definiert. Dabei ist es nicht erforderlich, dass ein Metaobjekt über seine gesamte Lebenszeit an der gleichen Stelle im Speicher verweilt. Im Gegensatz zu C kann es sogar bei der Benutzung von Feldern zu Situationen kommen, in denen ein Objekt im Speicher verschoben werden muss. Trotzdem muss eine Implementierung Möglichkeiten vorsehen um Zeiger auf Metaobjekte bereitstellen zu können.

Die Notwendigkeit dieser Definition wird bei den Erläuterungen zu Metadatenfeldern und Metadatenzeigern erläutert. Da es sich hierbei um eine Einschränkung der Funktionalität gegenüber der ursprünglichen Definition von *Objekt* handelt, ist es nicht notwendig, die Implikationen daraus entstehender Freiheitsgrade zu untersuchen. Stattdessen ist es erforderlich zu zeigen, dass die beschriebene Funktionalität der Sprache ohne die Definition der Adresse von Metadatenobjekten auskommt. Dies wird im Abschnitt 5.5.1 durchgeführt, in dem die dynamische Semantik von Pointern auf Metadatenobjekte untersucht wird.

Definition 2 *Ein* Metadatenobjekt *ist folgendermaßen definiert:*

$$m = (s, l, t)$$

- s *ist das Symbol, über das das Metaobjekt definiert wird*
- l *ist die Lebensdauer des Metaobjekts*
- t *ist der Metadatentyp des Metaobjekts*

5.3. Lexikalische Erweiterungen

Die lexikalische Analyse muss für die Erweiterung von C auf MetaC lediglich um ein neues Token ergänzt werden. Dieses Token ist die Sequenz der einzelnen Zeichen Doppelpunkt (`:`) und Istgleich (`=`). Diese Abfolge erzeugt in C normalerweise einen Parserfehler, da diese bestimmte Sequenz von einem C Lexer nicht als eigenständiges Symbol erkannt wird und es keine grammatikalische Struktur gibt, in der diese Sequenz einen gültigen Tokenstrom ergeben würde. Somit kann die Abfolge der beiden Zeichen ohne trennende Weißzeichen (also Leerzeichen, Tabulator oder Zeilenrücklauf) als neues Token definiert werden. Es ähnelt dabei den erweiterten Zuweisungsoperatoren, die in der syntaktischen Struktur *assignment-expression* Verwendung finden.

Darüber hinaus muss noch der Bezeichner **`meta`** als neues Schlüsselwort erkannt werden. Seine syntaktische Verwendung und die zugehörige Semantik werden im nachfolgenden Abschnitt erläutert. Die Namen der Metadatentypen (**`expr`**, **`func`**, **`ident`**, **`real`**, **`symb`**, **`scope`**, **`stmt`**, **`strg`**, **`type`** und **`zed`**) und die Schlüsselworte der neuen Ausdrücke (**`typeof`** und **`codeof`**) müssen in der lexikalischen Analyse nicht als Schlüsselworte berücksichtigt werden, da sie in gleicher Art und Weise behandelt werden können wie normale Typdefinitionen in C. Somit beschränkt sich die Erweiterung der lexikalischen Analyse auf zwei Elemente: das Token `:=` und das Schlüsselwort **`meta`**.

5.4. Datentypen für Metadaten

Programmiersprachen stellen ihrem Benutzer üblicherweise (C, C++, Java, Ada, Pascal, ...) verschiedene Datentypen zum Rechnen mit ganzen Zahlen und Gleitkommazahlen zur Verfügung. Darüber hinaus bieten sie häufig die Möglichkeit Datenstrukturen zu definieren, die aus verschiedenen einzelnen Datentypen zusammengesetzt werden. Interpretierte Sprachen, also solche, deren Programme normalerweise nicht in Maschinensprache übersetzt werden, unterstützen häufig auch Datentypen für Strings (so z.B. PERL, LISP, Java, die diversen Shells).

Die Sprache MetaC bietet sowohl Datentypen für arithmetische Operationen mit Ganzzahl und Gleitkomma Datentyp als auch Operationen zur Stringverarbeitung. Weiterhin gibt es Typen, die die syntaktische und semantische Struktur von C widerspiegeln bzw. Quelltextstrukturen referenzieren. Tabelle 5.1 zeigt eine Übersicht der verfügbaren Datentypen mit den zugehörigen Schlüsselwörtern und der Definition aus der sie abgeleitet wurden. Abbildung 5.1 stellt einige der Metadatentypen und ihre Codestrukturen exemplarisch dar. Im folgenden werden die Ableitungen der einzelen Typen im Detail erläutert.

Arrays, Structs, Unions, Funktionen und Pointer werden als abgeleitete Datentypen bezeichnet, da sie in Abhängigkeit von anderen Typen definiert werden. Entsprechend der normalen C Semantik ist es mit den MetaC Metadatentypen ebenso möglich abgeleitete Typen zu definieren. Dabei gibt es für die Verwendung solcher abgeleiteter Typen in Metaprogrammen einige Einschränkungen, die im Abschnitt 5.5 beschrieben werden. Diese Einschränkungen gelten aber nicht für normalen C Quelltext.

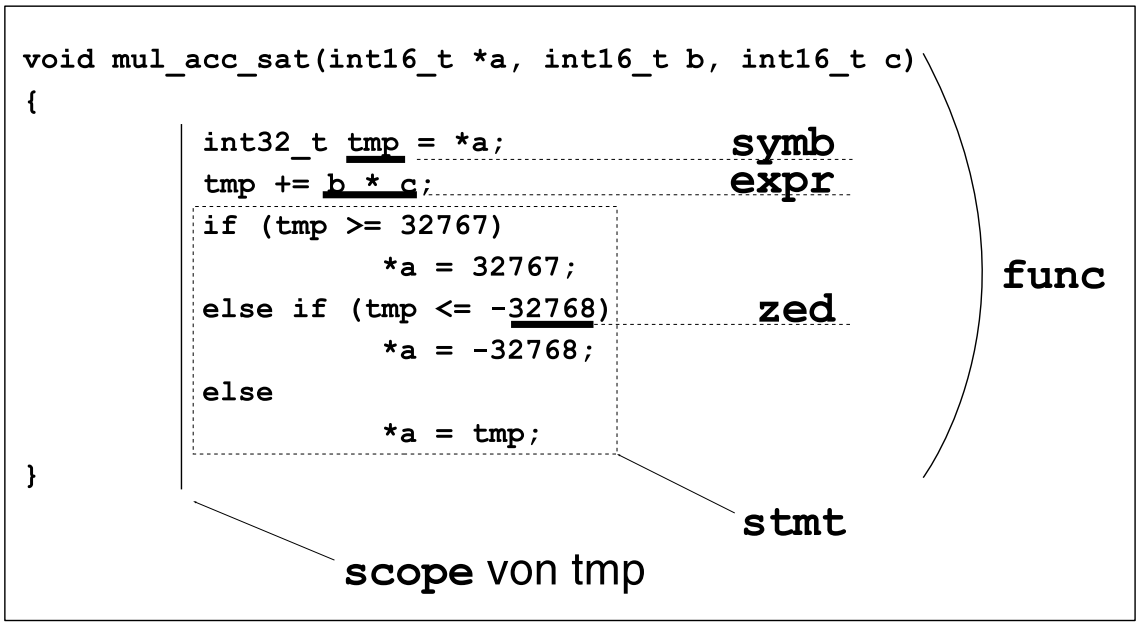


Abbildung 5.1.: Quelltextstrukturen und ihre Metadatentyp Repräsentanten

Schlüsselwort	Name	Abschnitt der Definition in ISO9899:1999
zed	set $\mathbb{Z}$	Ableitung aus *integer-constant*
expr	expression	6.5.17
func	function	6.9.1
ident	identifier	6.4.2.1
real	real	Ableitung aus *floating-constant*
scope	scope	6.2.1
stmt	statement	6.8
strg	string	6.4.5
symb	symbol	Neudefinition ohne Referenz zum ISO-Standard
type	type	6.2.5

Tabelle 5.1.: Datentypen für Metadaten

5.4.1. Überblick über die Metadatentypen

Die Schlüsselwörter der Metadatentypen erhalten erst ihre im folgenden definierte Semantik, wenn innerhalb der selben oder einer kapselnden Deklaration oder Funktions-Definition das Schlüsselwort **meta** geparst wurde. Dies stellt sicher, dass diese Bezeichner weiterhin im C Quelltext für Variablendefinitionen verwendet werden können. Es kommt somit nicht zu Kompatibilitätsproblemen bei Quelltexten, die diese Bezeichner verwenden. Daraus kann abgeleitet werden, dass diese Schlüsselwörter für den Parsevorgang in gleicher Weise behandelt werden wie Typdefinitionen, damit zwischen Typbezeichnern und normalen Bezeichnern unterschieden werden kann.

Das UML Diagramm in Abbildung 5.2 zeigt die semantischen Zusammenhänge zwischen den verschiedenen Metadatentypen. Da diese Typen verschiedene syntaktische oder semantische Strukturen in C Quelltext referenzieren, beinhalten Variablen bestimmter Metadatentypen gleichzeitig die Informationen von anderen Strukturen. Diese Zusammenhänge sind vor allem für Typumwandlungen wichtig, denn daraus ergibt sich auto-

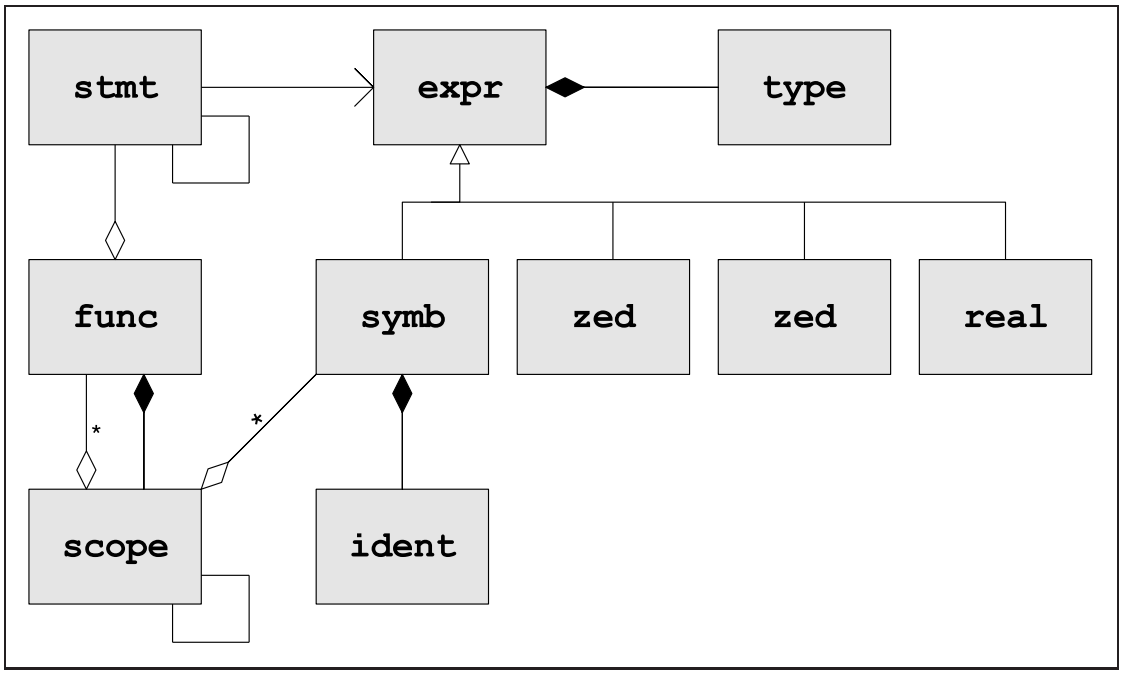


Abbildung 5.2.: Klassendiagramm der Metadatentypen

matisch, welche Wandlungen möglich sind und welche semantisch von vorn herein kein Ergebnis liefern können.

Trotzdem sind aus diesem Diagramm nicht alle theoretisch möglichen Umwandlungen ersichtlich. Denn es gibt neben den Umwandlungen, die auf der Basis der vorhandenen Assoziationen durchgeführt werden auch Wandlungen, die auf einer Reinterpretation der enthaltenen Daten beruhen. Diese werden im einzelnen für die unterschiedlichen Metadatentypen erläutert.

5.4.2. Literalzeichenfolge (`strg`)

Ein *String-Literal* kann in einer Metavariable vom Typ **`strg`** gespeichert werden. Für diesen Metadatentyp sind die Operatoren, wie in Tabelle 5.2 ersichtlich, definiert. Im Quelltext definierte Literalkonstanten die unmittelbar aufeinander folgen, werden als eine Konstante referenziert, wie es in C für Stringliterale vorgesehen ist (Übersetzungsschritt 6, siehe Abschnit 5.1.1.2 in ISO9899:1999).

Der Metadatentyp **`strg`** ist nicht nur zur Definition von String-Literalen unerläßlich, sondern wird auch benötigt um Variablen des nachfolgend beschriebenen Metadatentyps **`ident`** initialisieren zu können. Da String-Literale in Anführungszeichen aber beliebige Zeichenketten erlauben, ist eine eins-zu-eins Abbildung von **`strg`** nach **`ident`** nicht gegeben. Vielmehr ist der Wertebereich des Metadatentyp **`strg`** eine Obermenge des Metatyps **`ident`**. Entsprechend folgt, dass alle **`ident`** problemlos **`strg`**-Variablen zugewiesen werden können. Die umgekehrte Operation kann aber fehlschlagen, was sich in einer Wandlung zum Nulläquivalent von **`ident`** äußert.

5.4.3. Bezeichner (`ident`)

Das Token *Identifier* ist in [17]-6.4.2.1 definiert und wird in folgenden Parsern verwendet: *labeled-statement, primary-expression, enum-specifier, direct-declarator, identifier-*

Operator	Semantik
$=$	Zuweisung
$==$	Vergleich auf Gleichheit
$!=$	Vergleich auf Ungleichheit
$<$	Kleiner
$>$	Größer
$<=$	Kleiner-Gleich
$>=$	Größer-Gleich
$+$	Konkatenation
$+=$	Konkatenation und Linkszuweisung

Tabelle 5.2.: Semantik von Operatoren mit Metaobjekten vom Typ **`strg`**

list, jump-statement und designator. Es referenziert dabei Symbole aus unterschiedlichen Namensräumen. Der daraus abgeleitete Metadatentyp **`ident`** bezeichnet entsprechend der ursprünglichen Definition eines Bezeichners ohne weitere semantische Implikationen in Bezug auf den verwendeten Namensraum. Abhängig vom Parser, in dem ein Identifier Verwendung findet, wird ein Symbol definiert, beziehungsweise ein zuvor definiertes referenziert. Der Metadatentyp **`ident`** ist wie folgt definiert:

Definition 3 *Ein* Bezeichner *ist eine Zeichenkette aus Buchstaben, Zahlen und dem Unterstrich (_), wobei das erste Zeichen keine Zahl sein darf.*

Der Metadatentyp **`ident`** wird für die Konstruktion und den Vergleich von Namen einzelner Objekte benötigt. Durch die Unterstützung zusätzlicher Operatoren, identisch zu denen des Metadatentyp **`strg`**, können aus existierenden Namen neue Namen abgeleitet werden. Bei entsprechender Verwendung von Namenskonventionen können dadurch neuinstanziierte Variablen anhand ihres Namens mit ihrem Zweck assoziiert werden. Wie Eingangs bereits erwähnt, können Metavariablen dieses Typs mit String-Literalen durch Typkonversion initialisiert werden. Dabei ist zu beachten, dass das String-Literal den lexikalischen Anforderungen an einen Bezeichner entsprechen muss. Ansonsten wird es durch den Nullwert ersetzt. Die umschließenden Anführungszeichen werden bei der Zuweisung von **`strg`** zu **`ident`** entfernt.

5.4.4. Symbole (`symb`)

Metadaten vom Typ **`symb`** halten eine Referenz auf die Deklaration oder Definition einer Variable oder Funktion. Symbole ermöglichen im Programm bestimmte Objekte zur Laufzeit zu benutzen. Dabei beziehen sie sich zwar auf ein konkretes Objekt, jedoch ist dieser Bezug abhängig vom gegenwärtigen Kontext. So bezeichnet beispielsweise eine automatische Variable (also eine Variable, die üblicherweise auf dem Stack angelegt wird) mit dem Namen **`i`** ein konkretes Objekt; ruft sich dieselbe Funktion aber rekursiv auf, so bezeichnet dieselbe Variable nun ein anderes Objekt.

Der Metadatentyp **`symb`** bietet die Möglichkeit, auf Deklarationen eines bestimmten Gültigkeitsbereiches zuzugreifen. Dazu halten Variablen dieses Metadatentyps implizit den symbolischen Namen, den Datentyp und den zugehörigen Gültigkeitsbereich. Diese Informationen sind durch entsprechende Typumwandlungen (Typecasts) der Metadatentypen erreichbar. Verbirgt sich hinter dem Symbol eine Funktion, so ist es darüber hinaus

möglich die Metadatenvariable zu dem Typ **`func`** zu wandeln und so Zugriff auf die Eigenschaften der Funktion zu bekommen, die sich hinter dem Symbol verbergen.

Definition 4 *Der Metadatentyp* ***`symb`*** *ist definiert als folgendes Tupel:*

$$s = (i, c, t)$$

- `i` *ist der Bezeichner des Symbols*
- `c` *ist der Gültigkeitsbereich des Symbols*
- `t` *ist der Datentyp des Symbols*

5.4.5. Funktionen (`func`)

Metadaten vom Typ **`func`** halten eine Referenz auf eine Funktionsdefintion. Funktionen, die ohne Implementierung deklariert sind, werden als Vorwärtsdeklaration bezeichnet. Solche Vorwärtsdeklaration können nicht Metavariablen vom Typ **`func`** zugewiesen werden, sondern nur Metavariablen vom typ **`symb`**. Für Metaobjekte vom Typ **`func`** sind nur die Operatoren für Zuweisung (**`=`**) und Gleichheit (**`==`**) äquivalent zur Standard C Semantik definiert. Allerdings wird mit dem Gleichheit Operator die Identität überprüft und nicht der Name der Funktion.

Definition 5 *Der Metadatentyp* ***`func`*** *ist definiert als das Tupel*

$$f = (y, s, g)$$

- *y ist das Symbol der Funktion*
- *s ist das optionale* compound-statement *der Funktionsdefinition*
- *g ist der Gültigkeitsbereich der Funktion*

Da der Metatyp **`func`** weitgehend einen Spezialfall von **`symb`** repräsentiert, könnte theoretisch auf diesen verzichtet werden. Die Zusatzinformationen, die Metavariablen dieses Typs in sich tragen, sind die Referenz auf den Funktionskörper und den zugehörigen Gültigkeitsbereich. Eine Unterstützung dieses Metadatentyps vereinfacht die Implementierung von Metaprogrammen deutlich, da sich bestimmte Operationen dadurch auf einfache Typwandlungen reduzieren lassen. Dies betrifft insbesondere die angesprochenen Zusatzinformation, die bei Metavariablen vom Typ **`symb`** nicht enthalten sind. Dadurch kann auf entsprechende Spezialfunktionen zum Zugriff auf den Funktionskörper verzichtet werden, und Implementierungen müssen keine aufwändige Typanalyse bei der Verwendung von Symbolen durchführen, die Funktionen referenzieren könnten. Deshalb wird trotz der angesprochenen Redundanz nicht auf diesen Metatyp verzichtet.

5.4.6. Gültigkeitsbereiche (`scope`):

Es gibt verschiedene Arten von Gültigkeitsbereichen die hierarchisch gegliedert sind. Symbole, die in einem Gültigkeitsbereich definiert sind der innerhalb einem anderen Gültigkeitsbereich liegt, verdecken dabei Symbole, die in äußeren Gültigkeitsbereichen definiert sind. Gültigkeitsbereiche haben eine Assoziation zu der grammatikalischen Struktur, von der sie die Deklarationen speichern. Dies ist entweder das *translation-unit*, eine *function-definition* oder ein *compound-statement*.

Diese Assoziation ist Teil der Variablen, die vom Typ **`scope`** sind. Dementsprechend lassen sich Typumwandlungen nutzen, um die Zugehörigkeit eines Gültigkeitsbereiches zu bestimmen (siehe Tabelle A.10 im Anhang). Da es keinen Metadatentyp gibt, der die grammatikalische Struktur *translation-unit* repräsentiert, kann man diese Assoziation nur implizit herausfinden. Für diesen Gültigkeitsbereich gibt es noch eine Metafunktion namens **`module`**, die eine Variable vom Typ **`scope`** zurück gibt und gegen deren Ergebnis verglichen werden kann.

Darüber hinaus gibt es die Möglichkeit, die verschiedenen Typwandlungen zu nutzen, um den Zugehörigkeitsbereich eines Gültigkeitsbereiches zu ermitteln. Aus der Tabelle der Typwandlungen für **`scope`** ist ersichtlich, dass für den Gültigkeitsbereich mit Assoziation zum *translation-unit* die beiden Wandlungen zu **`func`** und **`stmt`** jeweils Null ergeben.

Definition 6 *Objekte vom Metadatentyp* ***`scope`*** *sind definiert als folgendes Tupel:*

$$c = (a, p, D, L, S)$$

- *a ist die Assoziation zur syntaktischen Umgebung (also eines aus* translation-unit, statement, function-definition*)*
- *p ist der Gültigkeitsbereich in der darüber liegenden syntaktischen Struktur*
- *D ist die Menge der Symbole im unmarkierten Namensraum*
- *L ist die Menge der Symbole im Namensraum für Sprungmarken*
- *S ist die Menge der Symbole im Namensraum für Datenstrukturen und Enumeratoren*

Neben den Standardoperatoren **`=`** und **`==`** die bei allen Metadatentypen unterstützt werden haben noch die Operatoren **`<`**, **`>`**, **`<=`** und **`>=`** eine wohldefinierte Semantik. Diese betrifft die Hierarchie, in der die Gültigkeitsbereiche angeordnet sind. Liefert also der Ausdruck **`s0 < s1`** den Wert eins als Ergebnis, und **`s0`** und **`s1`** sind vom Typ **`scope`**, so heißt dies, dass **`s1`** ein direkter oder indirekter Vater von **`s0`** in der Hierarchie der Gültigkeitsbereiche ist. Entsprechend lässt sich aus dem Ausdruck **`((s0 < s1) || (s1 < s0))`** bei einem Ergebnis von Null folgern, dass die beiden Gültigkeitsbereiche in unterschiedlichen Ästen der Hierarchie sind und dass ihre Deklarationen keiner Namensüberdeckung unterliegen.

5.4.7. Typen (`type`):

Der Metadatentyp **`type`** speichert einen vollständigen oder unvollständigen, qualifizierten C Datentyp. Ausgehend davon kann festgestellt werden, dass sowohl einfache Basistypen, als auch der Inhalt von komplexen Typdefinitionen oder unvollständige Referenzen auf noch zu definierende Datenstrukturen (z.B. Vorwärtsdeklarationen von **`struct`**, **`union`** und **`enum`** Typen) gehalten werden können. Wichtig dabei ist, dass der aufgelöste Typ gehalten werden muss, während die Assoziation zu Typaliasen verloren gehen darf. Der aufgelöste Typ ist unverzichtbar, da sonst keine Typüberprüfung durchgeführt werden kann.

Eine Implementierung kann die Information über verwendete Typdefinitionen integriert vorhalten, um sie zur besseren Lesbarkeit der Quelltexte während der Codegenerierungsphase wieder zu verwenden. Essentiell notwendig sind aber nur die aufgelösten Typen mit ihren Qualifizierern, die bei fehlenden Aliasinformationen eine identische Deklaration ermöglichen, die allerdings für den Menschen in der Regel schwieriger zu lesen ist. Entsprechend der Version des Sprachstandards, auf dessen Basis eine Implementierung realisiert wird (also ISO9899:1990 oder ISO9899:1999), sind die Qualifizierer **`const`** und **`volatile`** erforderlich und der Qualifizierer **`restrict`** optional zu speichern.

Zuweisungen zum Metadatentyp **`type`** erwarten als Rechtswert wiederum den Metadatentyp **`type`** oder die Null. Andere implizite Konvertierungen sind nicht vorgesehen. Darüber hinaus sind noch die Vergleichsoperatoren **`==`** und **`!=`** verwendbar, die auf direkte Gleichheit bzw. Ungleichheit der Werte prüfen. Da die Datentypen keiner strengen Ordnung oder Sortiermöglichkeit unterliegen, finden die restlichen Operatoren für diesen Metadatentyp keine Verwendung. Trotzdem sind mit Hilfe des unären Ausdrucks **`typeof`** beliebige Ableitungen möglich. Die Semantik dieses Ausdrucks ist in Abschnitt 5.6.2 erklärt.

Definition 7 *Der Metadatentyp* ***`type`*** *ist definiert als das Tupel*

$$t = (a, Q, t)$$

- *a ist die Art des Typs (ein Element aus der Menge der Ganzzahltypen oder der Menge der Gleitkommatypen, ein Zeigertyp, ein Funktionstyp oder ein strukturierter Datentyp oder Feldtyp)*
- *Q ist die Menge der Qualifizierer des Datentyps (eine beliebige Kombination aus* ***`const`****,* ***`volatile`*** *und* ***`restrict`****)*
- *t ist die rekursive Basis des Datentyps*

5.4.8. Anweisungen (`stmt`)

Der Metadatentype `stmt` referenziert ein Stück Quelltext, das als Statement geparst werden kann. Um mit Variablen dieses Datentyps zu arbeiten, gibt es eine Reihe von Prädikatenfunktionen, mit denen ermittelt werden kann, was für ein konkretes Statement von der Variablen referenziert wird. Variablen dieses Typs können auch einen zu Null äquivalenten Wert haben, der bedeutet, dass die Variable kein Statement referenziert.

Wie an der syntaktischen Struktur von C erkennbar, sind alle Statements Kinder von anderen Statements im Syntaxbaum. Einzige Ausnahme sind jene Compoundstatements, die Teil einer Funktionsdefinition sind und als solches keine Statements mehr als Eltern haben. Mit Hilfe der Metafunktion **`parent`**, die die übergeordnete Anweisung eines Statements im Parsebaum zurück gibt, kann man somit durch einen Vergleich mit Null ermitteln, ob eine Anweisung der Rumpf einer Funktion ist oder nur ein Element dessen.

Definition 8 *Eine Anweisung ist ein Stück Quelltext, das von dem Parser* Statement *akzeptiert wird.*

$$s = (p, D, E, S, c, q)$$

- *p ist der Vaterknoten der Anweisung im Parsebaum und ist entweder vom Typ* statement *oder* function-definition.

- *D ist die Menge der Deklarationen, die unmittelbare Kinder im Parsebaum der Anweisung sind.*
- *E ist die Menge der Ausdrücke, die direkte Kinder im Parsebaum der Anweisung sind.*
- *S ist die Menge der Anweisungen, die direkte Kinder im Parsebaum der Anweisung sind. Ihre Anzahl ist, abhängig von der Art der Anweisung, entweder Null (z.B.* expression-statement*), eins (z.B.* while-statement*) oder beliebig viele, also Null bis unendlich (z.B.* compound-statement*).*
- *c ist der Gültigkeitsbereich, der mit der Anweisung assoziiert wird oder Null.*
- *q ist die Reihenfolge der Deklarationen und Anweisungen innerhalb der Anweisung.*

5.4.9. Ausdrücke (`expr`)

Eine Variable des Metadatentyps **`expr`** hält eine Referenz auf einen Teilbaum des Parsebaums, der als *Expression* geparst wurde. Der referenzierte Baumabschnitt ist durch diese Variable nicht direkt modifizierbar, aber die Eigenschaften der Einzelknoten und ihre Struktur sind auslesbar. Um Ausdrücke applikationsspezifisch zu konstruieren, können Variablen dieses Typs Verwendung finden. Wie diese Instanziierung und die zugehörige Auflösung von Bezeichnern funktionieren ist in Kapitel 6 beschrieben.

Definition 9 *Ein Ausdruck ist ein Ast im Parsebaum eines Codefragments, der vom Parser* expression *geparst wird. Er kann optional einen oder mehrere Unterausdrücke beinhalten. Jeder Ausdruck hat einen Datentyp, der von den verwendete Operatoren und Operanden abhängt. Ein Ausdruck wird dargestellt als folgendes Tupel:*

$$e = (p, o, t, C)$$

- *p ist der Vaterknoten im Parsebaum und ist entweder vom Typ* expression *oder* statement
- *o ist der Operator oder die Art des Ausdrucks*
- *t ist der Datentyp des Ausdrucks*
- *C ist die Menge der Unterausdrücke des Ausdrucks*

Metavariablen des Typ **`expr`** können auf einen Ausdruck verweisen, dessen Vater im Parsebaum eine Anweisung ist. Alternativ kann der Vater auch selbst ein Ausdruck sein. In diesem Fall handelt es sich bei dem referenzierten Ausdruck um einen Teilausdruck. Die Information über die Position des Ausdrucks im Parsebaums muss mit gespeichert werden, damit eine Modifikation des entsprechenden Teilbaums möglich wird.

Bei Modifikationen von Ausdrücken muss beachtet werden, dass auch einzelne Teilausdrücke und komplette Ausdrücke entfernt werden könnten. Eine Referenz auf einen solchen Ast des Parsebaums sollte aber trotzdem noch einen gültigen Inhalt aufweisen, denn der Inhalt könnte als Muster für eine neue Instantiierung genutzt werden. Beim Entfernen eines Knoten, muss diese Operation durch einen besonderen Wert in der Positionsinformation vermerkt werden.

5.4.10. Integerzahlen (zed):

Die Verarbeitung von Integerzahlen wird mit dem Metadatentyp **zed** zur Kompilierzeit unterstützt. Der Name **zed** leitet sich von der mathematischen Menge Z ab, die die Menge aller ganzen Zahlen repräsentiert. Da zur Ausführungszeit eines Metaprogramms nicht zwingenderweise die Wertebereiche der integralen Datentypen feststehen, muss ein Wertebereich zur Verfügung gestellt werden der groß genug ist um alle benötigten Zahlen darstellen zu können.

Dabei wäre eigentlich eine Unterstützung der positiven Zahlen ausreichend, da negative Zahlen in der Regel im Zweierkomplement gespeichert werden. Deshalb gibt es in C auch keine negativen Zahlen als Literale, sondern nur durch eine explizite Negativwandlung durch den entsprechenden unären Ausdruck. Es vereinfacht die Implementierung von Metaprogrammen deutlich, wenn negative Werte trotzdem unterstützt werden. Die Umwandlung einer negativen Zahl in ihr Zweierkomplement erfolgt ohnehin durch den C Compiler und kann deshalb vom MetaC Compiler als positive Ganzzahl mit führendem Minus ausgegeben werden.

Die Semantik von Ausdrücken, die diesen Typ verwenden, entspricht der von äquivalenten C-Datentypen, wie sie in Abschnitten 6.2.5, 6.2.6.2 und 7.18.1.1 im ISO-Standard definiert ist. Die Operatoren zur Bitverarbeitung behalten dabei ebenso ihre Semantik, wie alle anderen Operatoren. Zusätzlich zur Semantik in C ist eine Typwandlung zum Metadatentyp **strg** möglich, die einer Operation mit der Standardfunktion **sprintf** entspricht (siehe Tabelle A.1).

Bei der Ausgabe der berechneten Werte muss beachtet werden, dass entsprechend der konfigurierten Zielumgebung notwendige Suffixe für die minimal notwendige Integertypgröße mit ausgegeben wird. Dies betrifft sowohl die Suffixe **L** und **LL** für Integerkonstanten mit erweitertem Wertebereich, sowie auch das Suffix **U** für explizit vorzeichenfreie Integer-Literale.

5.4.11. Gleitkommazahlen (real):

Der Metadatentyp **real** kann ähnlich zu **zed** dafür genutzt werden applikationsspezifische Konstanten zur Kompilierzeit zu berechnen. Dies ist insbesondere dann sinnvoll, wenn ein und der selbe Wert häufig verwendet wird und somit zur Laufzeit aufwändige Berechnungen eingespart werden können, die nicht vom C-Compiler durch Optimierung entfernt werden. Die Semantik aller Operatoren entspricht der des Datentyps **float** in C, bzw. abhängig von der eingangsseitig gewählten erweiterten Breite (d.h. **double** oder **long double**) des Datentyps, dem angepassten Wertebereich und Genauigkeit.

Entsprechend verhält es sich bei der Repräsentation der Metavariablen im Speicher äquivalent zu den Werten des Metadatentyps **zed**. Ebenso wird die Metatypumwandlung zu **strg** unterstützt (siehe Tabelle A.2), die der im Standard definierten **sprintf** Operation entspricht.

5.5. Abgeleitete Metadatentypen

Neben den grundlegenden Datentypen und Metadatentypen ist es möglich weitere Typen zu definieren, indem von den spracheigenen Typen neue abgeleitet werden. Dazu stehen verschiedene beschränkte Ableitungsmechanismen zur Verfügung, deren Einschränkungen von der Ableitungsbasis abhängen.

5.5.1. Zeiger (Pointer)

Zeiger bieten die Möglichkeit Referenzen auf existierende Objekte zu erzeugen. So kann eine Funktion die Objekte der aufrufenden Funktion modifizieren, obwohl in C alle Funktionsparameter als Wert übergeben werden und eine Übergabe als Referenz nicht explizit vorgesehen ist. Dadurch können Funktionen beliebige Objekte modifizieren oder als Datenquelle benutzen, die zur Kompilierzeit der Funktion noch unbekannt sind. Weiterhin ermöglichen Zeiger eine dynamische Verwaltung von Objekten auf dem so genannten Heap.

Bei der Verwendung von Zeigern in C ist große Sorgfalt geboten, da typische Fehler bei der Behandlung von Zeigern zu nichtdefiniertem Verhalten führen. Dies erschwert eine Fehlersuche deutlich, da falscher Umgang mit Zeigern somit nicht direkt zum Programmabbruch oder einer Fehlermeldung führt. Stattdessen kann fehlerhafter Quelltext, abhängig von der Implementierung, sogar das erwartete Verhalten zeigen. Sporadische Abweichungen von den gewünschten Wirkungen und Änderungen die durch Implementierungsvarianten (z.B. neue Compilerversion oder andere Zielarchitektur) auftreten, können dann zu äußerst schwierig zu behebenden Problemen führen.

Folgende Voraussetzungen in der Sprachdefinition von C sind die Ursache für die bereits genannten Probleme:

1. Die Möglichkeit zur Typkonvertierung von Ganzzahltypen zu Zeigern mit implementierungsspezifischem Verhalten.

2. Skalare Operatoren für die Modifikation von Zeigern zur Iteration über Felder mit undefiniertem Verhalten außerhalb der Feldgrenzen.

3. Undefiniertes Verhalten bei der Modifikation von Zeigern auf Objekte, die nicht innerhalb eines Feldes liegen.

Die Art der Implementierung von Zeigern ist in C keineswegs vorgeschrieben. Üblicherweise wird die Adresse des referenzierten Objekts im Objekt des Zeigers gespeichert. Alternative Verfahren speichern neben der Basisadresse des Zielobjekts den Typ des referenzierten Objekts und den Index des Objekts innerhalb des zugehörigen Feldes beziehungsweise Null für Objekte, die zu keinem Feld gehören. Dazu gibt es noch eine ganze Reihe von Varianten, die alle zum Ziel haben Laufzeitfehler zu erkennen und den Programmablauf somit vor undefiniertem Verhalten zu schützen (Beispiele hierfür finden sich in [48] und [21]). Zum Teil können die Überprüfungen zur Laufzeit durch statische Analyse überflüssig gemacht werden (siehe [28]).

Die speziellen Eigenschaften von C , insbesondere das laxe Typsystem, ermöglichen eine einfache Kompilierung des Quelltextes und den Einsatz von effizienten Optimierungsverfahren mit niedrigem Implementierungsaufwand. Folglich wird für das Auffinden von Fehlern, die daraus resultieren, häufig spezielle Software verwendet. Andere Sprachen bringen Einschränkungen und Anforderungen mit sich, die solche Fehler gar nicht erst entstehen lassen. So verlangt Java [40] beispielsweise Unterstützung von der Laufzeitumgebung, um die Speicherverwaltung zu automatisieren. Dafür verzichtet Java aber gänzlich auf Zeiger, wodurch Fehler verursacht durch die Pointerarithmetik, unmöglich werden. Bei interpretierten Sprachen ist solch ein Vorgehen noch verbreiteter (z.B. Perl, alle Shell Varianten, Basic, Lisp, Tcl), da Fehler im interpretierten Programm den Interpreter zum Absturz bringen könnten.

Da die bevorzugte Variante für die Ausführung des Metaprogramm-Anteils eines MetaC Programms die Interpretation ist, werden die nachfolgenden Einschränkungen empfohlen. Prinzipiell sind diese Restriktionen für eine Implementierung nicht erforderlich. Die genannten Einschränkungen betreffen dabei in keiner Weise Konstruktionen, die in der Muttersprache C erstellt sind, sondern nur den Anteil der zur Kompilierzeit ausgeführt wird, also den MetaC Anteil.

Eine Typkonvertierung vom Typ *Pointer auf ein Objekt* zu irgendeinem anderen Datentyp ist verboten. Der Grund hierfür liegt darin, dass die Ablage von Objekten im Hauptspeicher grundsätzlich implementierungsspezifisch ist. Außerdem können Felder (Arrays) zur Laufzeit automatisch wachsen und dadurch eine Verschiebung der zugehörigen Datenbereiche im Speicher notwendig werden. Dies hat zur Konsequenz, dass eine vorangegangen Typkonvertierung von Pointer nach Integer ungültig werden kann. Unabhängig davon ist die konkrete interne Repräsentanz von Zeigern implementierungsspezifisch. Dies ist möglich, da eine Wandlung von Pointer nach Integer und umgekehrt nicht definiert ist.

Damit bei der Verwendung von Pointern zur Aliasbildung keine so genannten *Wild Pointer* entstehen können, sind zusätzliche Maßnahmen notwendig. *Wild Pointer* und *Dangling References* sind Zeiger, die auf eine Stelle im Speicher zeigen, an der sich kein Objekt (mehr) befindet. Dies kann erreicht werden, indem verhindert wird, dass Zeigervariablen einen größeren Gültigkeitsbereich besitzen, als die Objekte, die sie referenzieren. Weiterhin dürfen Referenzen auf Objekte, bei der Zuweisung von Zeigern untereinander keinen größeren Gültigkeitsbereich bekommen als die Objekte, die referenziert werden.

Dieses Ziel ist nicht durch einfache, statische Analyse erreichbar, da Zeigervariablen als Funktionsparameter übergeben werden können. Außerdem kann ein und dieselbe Pointervariable dazu genutzt werden, verschiedene Objekte zu referenzieren. In beiden Fällen führt dies dazu, dass die Zeigervariable auf Objekte mit unterschiedlicher Lebensdauer verweist.

Diese Problematik lässt sich durch folgende Einschränkungen in der Behandlung von Zeigervariablen umgehen:

1. Zeigervariablen müssen eine kürzere Lebensdauer aufweisen, als die Objekte, die sie referenzieren.

2. Eine Zuweisung von einer Zeigervariablen zu einer anderen darf nur durchgeführt werden, wenn der *Lvalue* eine kleinere oder die gleiche Lebensdauer besitzt, wie die Zeigervariable von der der *Rvalue* abgeleitet wurde.

Beide Randbedingungen können in einer Implementierung wahlweise durch statische Analyse oder Laufzeitüberprüfungen sichergestellt werden. In Ada (siehe [14]) wird das Problem von ungültigen Referenzen auf Objekte, die bereits gelöscht wurden, durch Einschränkungen und sprachliche Mittel verhindert. Hierfür werden diverse syntaktische Konstrukte bereitgestellt, die dem Compiler eine detaillierte Datenflussanalyse ermöglichen. So kann bei der Implementierung des Programms weitgehend auf Laufzeitüberprüfungen verzichtet werden, wenn der Compiler über entsprechende Analyse- und Optimierungsverfahren verfügt.

In Java (siehe [6], [7]) wird das Problem dadurch gelöst, dass es keine echten Zeiger gibt, sondern nur Referenzen auf bestimmte Objekte. Hier müssen die Implementierung und die Laufzeitumgebung Zusatzcode bereitstellen, um die Gültigkeit von Objektreferenzen sicherzustellen. Da in Java der Programmierer nicht das Speichermanagement übernimmt, sondern eine so genannte Garbage Collection zum Einsatz kommt, ist dieser

Code weniger dafür verantwortlich Benutzerfehler abzufangen, als viel mehr den korrekten Zeitpunkt zu bestimmen zu dem ein Objekt gelöscht werden darf.

Dieser Ansatz bringt aber auch nicht nur Vorteile. Die Effekte der Garbage Collection können mitunter dramatische Einflüsse auf das Systemverhalten haben, da Art und Zeitpunkt zu dem sie stattfindet, implementierungsspezifisch sind. In den meisten Implementierungen wird die Garbage Collection zyklisch angestoßen, was zur kurzzeitigen Unterbrechung der normalen Programmverarbeitung führt. Für die meisten Applikationen stellt dies kein Problem dar. Jedoch kann es auf großen Applikationsservern dadurch zur verzögerten Antworten auf Anfragen kommen, was einer verringerten Dienstqualität entspricht.

Noch kritischer ist das Ganze bei Echtzeitapplikationen, bei denen eine Verzögerung zu einer Nichteinhaltung von Deadlines führen kann und somit das Programm seine Aufgabe nicht mehr korrekt erfüllt. Darum werden in Javaimplementierungen mit Echtzeitunterstützung spezielle Verfahren eingesetzt, um diese Situationen zu verhindern. In diesem speziellen Fall ist das Verfahren der Garbage Collection, welches die Entwicklung von Programmen vereinfachen soll, eine schwierig zu nehmenden Hürde und kann so den gegenteiligen Effekt erzielt.

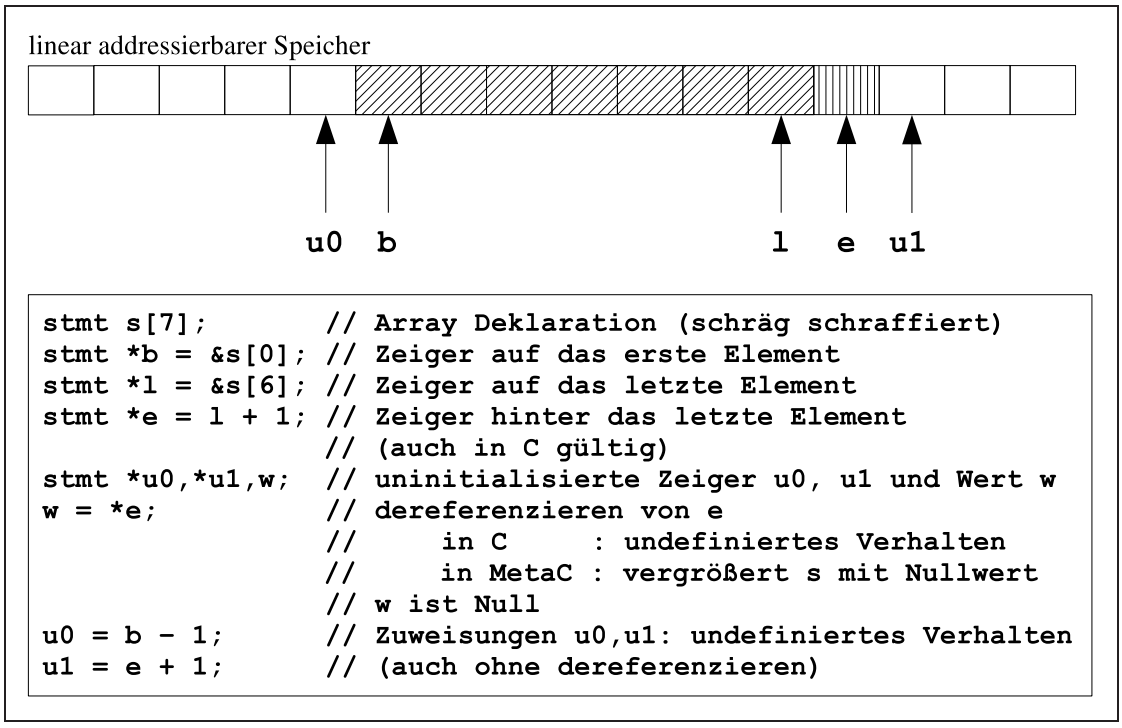
5.5.2. Felder (Arrays)

Felder sind eine Ansammlung von Objekten gleichen Datentyps, auf die über einen gemeinsamen Bezeichner via Indizierung zugegriffen werden kann. In Standard C ist über den indizierten Zugriff hinaus noch die Verwendung von Pointern definiert. Dabei können Pointer jeden Wert annehmen, der einer Referenz eines Objekts innerhalb des Feldes entspricht, sowie den Wert der genau auf das Element hinter dem letzten gültigen Objekt zeigt. Dieser Zeiger, der hinter das letzte Element zeigt, darf aber nicht dereferenziert werden, da diese Operation nicht-definiertes Verhalten hat. Des Weiteren ist das Verhalten eines Programms undefiniert, das einem Zeiger einen Wert zuweist der auf ein nichtexisitierendes Element vor dem Beginn eines Feldes verweist.

Feldvariablen, die in C deklariert werden, haben nach ihrer Initialisierung eine konstante Größe. Das heißt die Anzahl der Elemente im Feld ist fest und läßt sich nicht mehr ändern. Aus diesem Grund wird häufig mehr Speicher alloziert als notwendig ist und das Ende des genutzten Bereiches mit einem zu Null äquivalenten Wert markiert. Dieses Vorgehen ist insbesondere bei der Stringverarbeitung unumgänglich, da es in C keinen nativen Datentyp String gibt. Deshalb greift man auf Felder vom Typ `char` zurück, deren Größe ausreichend bemessen sein muss, und deren Ende durch einen Nullwert angezeigt wird.

Felder mit statischer Größe haben deutliche Vorteile bei der Implementierung gegenüber solchen, die die Anzahl der Elemente zur Laufzeit verändern können. Zum einen kann dadurch Speicher für solche Objekte direkt alloziert werden, ohne den Umweg einer indirekten Adressierung. Zum anderen können Zeiger auf Elemente eines Feldes somit durch Speichern einer einzelnen Speicheradresse realisiert werden. Direkte Zugriffe über eine einzelne Speicheradresse bieten neben der einfacheren Implementierung auch noch den Vorteil einer höheren Performance, da mindestens ein indirekter Zugriff entfällt.

Bei dynamisch veränderlicher Größe ist eine indirekte Adressierung unumgänglich, da es im Fall eines wachsenden Feldes zu der Situation kommen kann, bei der der zugewiesene Speicherbereich nicht mehr erweitert werden kann und somit die Elemente an eine neue Position kopiert werden müssen. Dadurch können Zeiger auf Elemente des Feldes




```
stmt s[7];       // Array Deklaration (schräg schraffiert)
stmt *b = &s[0]; // Zeiger auf das erste Element
stmt *l = &s[6]; // Zeiger auf das letzte Element
stmt *e = l + 1; // Zeiger hinter das letzte Element
                 // (auch in C gültig)
stmt *u0,*u1,w;  // uninitialisierte Zeiger u0, u1 und Wert w
w = *e;          // dereferenzieren von e
                 //     in C     : undefiniertes Verhalten
                 //     in MetaC : vergrößert s mit Nullwert
                 // w ist Null
u0 = b - 1;      // Zuweisungen u0,u1: undefiniertes Verhalten
u1 = e + 1;      // (auch ohne dereferenzieren)
```

Abbildung 5.3.: Konkretisierung der Semantik von Feldern von Metadaten

nicht mehr durch Speichern einer einzelnen Adresse realisiert werden, sondern es muss eine andere Form der indirekten Adressierung stattfinden [19]. Beispiele für Implementierungsvarianten finden sich bei verschiedenen Realisierungen von den Vektordatentyp der STL [10], wobei die Iteratoren die Rolle der Zeiger übernehmen.

Allerdings hat es für die Programmierpraxis viele Vorteile wenn Felder keine feste Größe besitzen. Da alle genannten Implikationen sich nur bei der Kompilierung von Binärdateien, nicht aber bei der Interpretation deutlich auswirken, kann die Semantik für MetaC durch Abwendung von undefiniertem Verhalten im C Standard so konkretisiert werden, so dass Felder in MetaC eine dynamische Größe bekommen und damit trotzdem zu 100% kompatibel zu C Feldern bleiben.

Das dynamische Verhalten von Feldern in MetaC und C ist in Abbildung 5.3 zusammengefasst. MetaC erweitert das dynamische Verhalten von C nur durch die Konkretisierung des Dereferenzierungsverhalten eines Zeigers hinter das letzte Element eines Feldes. In diesem Fall wird das zugehörige Feld um ein Element vergrössert, und der Inhalt des Elementes mit einem Nullwert initialisiert.

5.5.3. Funktionen

Die gleichen Einschränkungen und Voraussetzungen, die für Funktionen in C gelten, gelten auch für Metafunktionen (also Funktionen mit dem *storage-type-specifier* `meta`) in MetaC. Dazu gehört, dass Funktionen keine Funktionen oder Felder zurückgeben können. Auch ist ein Überladen von Funktionen nicht erlaubt, wie es in C++ möglich ist. Dabei wird derselbe Funktionsname mit unterschiedlichen Parameterlisten und verschiedenen Implementierungen versehen.

In C ist dies nur durch die Verwendung von Ellipsen (Operator `...`) möglich. Die Spracherweiterung MetaC ändert nichts an diesem Konzept und übernimmt auch nicht andere C++ Erweiterungen, wie die der automatischen Defaultparameter. Die Ellipse wird

entsprechend der Standard C Semantik unterstützt. Bei der Verwendung von Ellipsen können durch eine entsprechende Implementierung typische Laufzeitprobleme bei der Verwendung von Ellipsen erkannt werden. Zu diesen gehören falsche Typen der Argumente, falsche Parameteranzahl sowie fehlende Terminierung der Parameterlist.

Die in der Muttersprache vorhandene Einschränkung für die Rückgabewerte, könnte mit einer entsprechenden Spezifikation und Implementierung der Laufzeitumgebung aufgelöst werden. Jedoch widerspricht ein solches Vorgehen dem grundlegenden Konzept von C, in dem ein einfaches Implementierungsmodell vorgesehen ist. Deswegen wird von einer solchen Erweiterung der Semantik abgesehen.

5.5.4. Semantik von Operatoren

Operatoren haben abhängig von den verwendeten Datentypen unterschiedliche Bedeutung. So wird beispielsweise eine Multiplikation anders durchgeführt, wenn die Operanden Gleitkomma- oder Ganzzahldatentypden sind. Deshalb muss für die neu definierten Datentypen festgelegt werden, welche Operatoren verwendet werden können und welche Bedeutung eine Ausführung einer solchen Operation hat.

Der `sizeof` Operator in Metafunktionen

Der Operator **`sizeof`** bestimmt die Größe eines Objekts im Hauptspeicher zur Kompilierzeit aus der Typinformation des Ausdrucks beziehungsweise dem übergebenen Datentyp. Das Ergebnis wird in C Funktionen als **`size_t`** zurückgegeben, in Metafunktionen als **`zed`**. Der zurückgegebene Wert hat die Einheit Byte. Für Metaprogramme gibt es hierbei eine entscheidende Implikation: Da die Größe der Datentypen implementierungsspezifisch sind, können diese Daten nur ermittelt werden, wenn dem Metacompiler die Zielarchitektur bekannt ist. Ein MetaC Compiler muss deshalb eine Möglichkeit bieten die Größen der Basistypen zu konfigurieren damit sie korrekt aufgelöst werden können.

Die empfohlene Vorgehensweise hierfür ist eine enge Kopplung mit dem Präprozessor, die neben der normalen Makroersetzung die in Abschnitt 5.2.4.2 des ISO-Standards für C definierten Bezeichner für numerische Konstanten erkennt. Dadurch kann während der Übersetzungsphase vier (siehe Abschnitt 5.1.1.2 in [17]) diese für Übersetzungphasen sieben und acht notwendigen Informationen aus den System spezifischen Headerdateien ohne Benutzerinteraktion gewonnen werden.

Typkonvertierung von Ganzzahl zu Zeiger

Die Semantik der Typkonvertierung eines Integerdatentyps zu einem Zeiger ist in C implementierungsspezifisch. Jede Implementierung kann somit ihre eigene Semantik dafür vorsehen. Üblicherweise werden solche Konstruktionen dafür verwendet, Zeiger auf physikalische Speicheradressen zu erzeugen. Diese werden bei der systemnahen Programmierung dringend benötigt, um beispielsweise Interrupt Vektortabellen modfizieren zu können oder auf memory-mapped I/O-Register zugreifen zu können.

Alle diese Szenarien finden in Metaprogrammen keine Anwendung. Es ist aber nicht auszuschließen, dass es trotzdem sinnvolle Anwendungen bei der Metaprogrammierung geben könnte, bleibt die Spezifikation an dieser Stelle unverändert, also implementierungsspezifisch. Insbesondere ist die Konversion von Null zu einem Zeiger als reservierte Operation zu betrachten, die den Zeiger mit einer ungültigem Wert initialisiert. Dieser ist

storage-class-specifier:
meta
extern
static
auto
typedef

Abbildung 5.4.: Der erweiterte Parser *storage-class-specifier*

speziell dafür vorgesehen, um als Terminierung in Listen verwendet werden zu können oder ganz allgemein einen unverwechselbaren ungültigen Wert explizit zu speichern.

5.6. Neue syntaktische Strukturen

In diesem Abschnitt werden die syntaktischen Erweiterungen von C zu MetaC erläutert, die notwendig sind, um Strukturen für die Metaprogrammierung bereitstellen zu können, die vollständig abwärts kompatibel sind und die Definition von Quelltextvorlagen erlauben. Alle syntaktischen Erweiterungen wurden auf LALR(1) Konformität mit Hilfe von **yacc** ([47], [41]) überprüft.

5.6.1. Erweiterung des *storage-class-specifiers*

Storage-class-specifier bezeichnen den Platz an dem eine Variable gespeichert wird. Dies beeinflusst die Referenzierbarkeit von entsprechend qualifizierten Deklarationen. So wird Variablen kein eigener Speicherbereich zugeordnet, die mit dem Schlüsselwort **extern** gekennzeichnet sind. Stattdessen wird der Speicher verwendet, der einer Variable gleichen Typs und gleichen Namens durch eine reguläre Deklaration in einem anderen *translation-unit* zugeordnet wurde. Die zugehörigen Abhängigkeiten werden nicht vom Compiler sondern vom Linker aufgelöst.

Abhängig vom syntaktischen Kontext in dem das Schlüsselwort **static** verwendet wird, reduziert es die Sichtbarkeit einer Deklaration auf das *Translation-Unit* (in *module-declaration*) oder sorgt dafür, dass Speicher für Variablen einer Funktion statisch zugeordnet wird (in *block-item*). Somit werden solche Variablen nicht auf den Stack gelegt und ihr Inhalt ist dadurch persistent über alle Funktionsaufrufe einer Programmausführung.

Mit **typedef** gekennzeichnete Deklarationen definieren keine Variablen sondern Typaliase, die zur Verbesserung der Lesbarkeit des Quelltextes dienen. Ihre Lebensdauer ist auf die Übersetzungszeit beschränkt; danach haben sie keine Bedeutung mehr.

Nur einer der zuvor genannten *storage-type-specifier* darf gleichzeitig in einer Deklaration verwendet werden. Nur eines oder keines dieser Schlüsselworte darf gleichzeitig in einer Deklaration verwendet werden. Dies gilt für das nachfolgend beschriebene, neue Schlüsselwort **meta** nicht. Es kann mit **static** kombiniert werden und verleiht so Metadatenobjekten die gleiche Kompilierzeitsemantik, wie sie normalerweise Objekten zur Laufzeit zugeordnet wird.

Das neue Schlüsselwort **meta** beeinflusst Deklaration dahingehend, dass die zugehörigen Variablen nur zur Laufzeit des Metaprogramms sichtbar sind. Folglich können sie

unary-expression:
postfix-expression
+ *expression*
++ *expression*
- *expression*
-- *expression*
& *expression*
***** *expression*
~ *expression*
sizeof (*expression* **)**
codeof *identifier*
codeof (*identifier* **)**
typeof (*expression* **)**
typeof (*type-name* **)**
typeof (typedef *dependent-specifier* **)**

Abbildung 5.5.: Erweiterung des Parsers *unary-expression*

auch nur die für Metadaten geeigneten Typen verwenden. Gleiches gilt auch bei der Anwendung auf Funktionen: Funktionen, die mit diesem Schlüsselwort gekennzeichnet werden, sind als Metafunktionen zu interpretieren und sind dadurch nur zur Laufzeit des Metaprogramms sichtbar. Ihr Quelltext wird danach entfernt und ist somit während der Ausführung des Programms unsichtbar.

5.6.2. Neue Unäre Ausdrücke

Die Erweiterungen des Parsers *unary-expression* sind in Abbildung 5.5 dargestellt. Der dünn gedruckte Teil entspricht dem Original, und die fett gedruckten Zeilen sind Erweiterungen.

Semantik des `codeof` Operators

Das Schlüsselwort **`codeof`** wird in Funktionsdefinitionen, die mit dem *storage-class-specifier* **`meta`** qualifiziert sind, als Operator aktiviert. In normalen C Funktionen bleibt somit das Schlüsselwort als Bezeichner verfügbar und garantiert somit 100%ige Abwärtskompatibilität. Der unäre Operator **`codeof`** referenziert über seinen Bezeichner das Quelltext-Strukturmuster einer Anweisung (SSP). Der Name muss identisch sein mit dem Bezeichner, der im Parser *statement-pattern* innerhalb der selben Funktion ein SSP definiert. Der Metadatentyp dieses Operators ist immer vom Typ **`const stmt`**.

Die Anweisung die dieser Operator referenziert, wird in der erweiterten MetaC Semantik während dem Parsevorgang nicht der Symbolauflösung unterzogen und ist somit unvollständig definiert. Der Parser *statement-pattern* (siehe Abbildung 5.6) ist ein neues Element des Parser *block-item*. Dem Bezeichner der das QSM identifiziert, folgen das neue lexikalische Token **`:=`** und die Anweisung, die als Codevorlage für Instantiierungen dient. Der Name wird im Namensraum für Sprungmarken aufgelöst und über das Schlüsselwort **`codeof`** referenziert.

Die Symbolauflösung eines *statement-pattern* findet erst während der Anwendung des Musters statt (beschrieben in Kapitel 6). Somit sind so referenzierte Anweisungen ledig-

statement-pattern:
    identifier **:=** *statement*
    identifier **:=** *declaration*

block-item:
    declaration
    statement
    statement-pattern

Abbildung 5.6.: Erweiterung des Parsers *block-item*

lich als Strukturreferenz zu sehen, deren konkrete Semantik erst durch ihre Applikation definiert wird. Denn die exakte dynamische Semantik ist von den konkret verwendeten Typen abhängig.

Semantik des `typeof` Operators

Ebenso wie das Schlüsselwort **`codeof`** ist das Schlüsselwort **`typeof`** nur in Metafunktionen als solches aktiviert und steht dadurch in C Funktionen als normales Schlüsselwort zur Verfügung. Dieser neue, unäre Operator ermittelt den Datentyp des übergebenen Arguments und gibt ihn als Metaobjekt vom Typ **`const type`** zurück. Als Argumente stehen folgende syntaktische Strukturen zur Verfügung:

1. ein Ausdruck
2. ein Typalias
3. eine abhängige Typspezifikation

Der Rückgabewert dieses Operators ist ein temporäres Metadatenobjekt vom Metadatentyp **`type`**. Ist das Argument des **`typeof`** Operators ein Typalias, so wird der Datentyp des Typalias zurückgegeben. Für Ausdrücke wird entsprechend der Typ ermittelt. Diese Funktionsweise ist äquivalent zu der vom **`sizeof`** Operator.

Alternativ kann als Argument auch ein Konstrukt übergeben werden, das vom Parser *type-name* erkannt wird. Das sind Typaliase, also **`typedef`** qualifizierte Deklarationen und daraus abgeleitete Typspezifikationen. Da Typdeklarationen mit Hilfe von Metafunktionen verändert werden können, ist eine solche Konstruktion kein statisch analysierbarer Ausdruck, sondern muss zur Laufzeit des Metaprogramms interpretiert werden.

Diese letzte Variante erwartet als Argument des **`typeof`** Operators die Übergabe einer abhängigen Typdeklaration. Diese Form ist für die Funktionalität von MetaC am wichtigsten, da sie die dynamische Konstruktion von neuen Datentypen erlaubt. Dazu sind die drei neuen Parser *dependent-specifier*, *dependent-base* und *dereferencing-declarator* notwendig. Die dynamische Semantik dieses Konstrukts ist ähnlich zu den anderen Quelltext-Strukturmustern. Deshalb wird die zugehörige dynamische Semantik gemeinsam mit den anderen Strukturmustern die in Abschnitt 6.1.1 in Kapitel 6 diskutiert. Im Folgenden werden solche Konstruktionen als *type-structure-pattern* (TSP) bezeichnet.

In Abbildung 5.7 ist die Syntax der entsprechenden Argumentvariante definiert. Sie ist wie alle anderen Parser auch LALR(1) konform und ermöglicht die Erstellung von

dependent-base:
identifier
type-qualifier identifier
dependent-base type-qualifier

qualifier-modification-list:
type-qualifier
- *type-qualifier*
qualifier-modification-list type-qualifier
qualifier-modification-list **-** *type-qualifier*

dereferencing-declarator:
& *qualifier-modification-list*$_{opt}$
. *identifier qualifier-modification-list*$_{opt}$
dereferencing-declarator **&** *qualifier-modification-list*$_{opt}$
dereferencing-declarator **.** *identifier qualifier-modification-list*$_{opt}$

dereferencing-postfix:
& *type-qualifier*$_{opt}$
. *identifier type-qualifier*$_{opt}$
dereferencing-declarator **&** *type-qualifier*$_{opt}$
dereferencing-declarator **.** *identifier type-qualifier*$_{opt}$

dependent-specifier:
dependent-base abstract-declarator$_{opt}$
dependent-base dereferencing-declarator abstract-declarator$_{opt}$

Abbildung 5.7.: Erweiterung des Parsers *unary-expression*

```
meta-type-specifier:
        zed
        expr
        func
        ident
        real
        scope
        stmt
        strg
        symb
        type
```

Abbildung 5.8.: Der neue Parser *meta-type-specifier*

Datentypen. So ist es möglich auf die Typbasis eines Datentyps zuzugreifen und Qualifizierer hinzuzufügen und zu entfernen. Auch eine normale Ableitung eines neuen Datentyps kann durchgeführt werden, wie es von **`typedef`** Deklarationen her bekannt ist. Der entscheidende Unterschied hierbei ist, dass diese Konstruktion keine Kompilierzeitkonstante ergibt, wie es bei **`typedef`** Deklarationen der Fall ist. Stattdessen wird der Typ in Abhängigkeit von einem durch eine Metavariable vom Metatyp **`type`** übergebenen Datensatz errechnet.

5.6.3. Der Parser *meta-type-specifier*

Der Parser *meta-type-specifier* ergänzt den Parser *type-specifier* in Funktionsdefinitionen und Deklarationen, die durch das *storage-type-specifier* **`meta`** qualifiziert sind und in Deklarationen, die sich innerhalb einer Metafunktion befinden. Die Schlüsselwörter die in dem Parser *type-specifier* verwendet werden, können trotzdem nicht als Bezeichner verwendet werden. Dies verhindert zum einen, dass diese Schlüsselwörter in einer ungewöhnlichen Art und Weise gebraucht werden, was zu schlecht lesbarem Quelltext führen kann und die Implementierung der lexikalischen Analyse komplexer macht.

Darüber hinaus ist ihre Verfügbarkeit als Schlüsselwörter wichtig, da auch in Metafunktionen Bezug auf die normalen Datentypen genommen werden könnte. Ein Beispiel hierfür ist die Verwendung der Datentypen innerhalb eines **`sizeof`** Ausdrucks, der zur Bestimmung von Randbedingungen zur Optimierung von Quelltexten herangezogen werden könnte.

Da der Parser *meta-type-specifier* erst nach dem Schlüsselwort **`meta`** aktiviert wird, sind theoretisch Sequenzen der Form *type-specifier* **`meta`** *meta-type-specifier* möglich. Ein Beispiel hierfür ist **`int meta strg`**. Solche Abfolgen haben aber keine gültige Semantik und müssen somit von einer Implementierung abgelehnt werden. Außerdem muss der Parser *type-specifier* nach Erkennen des Schlüsselwortes **`meta`** weiterhin die Standardtypen als Schlüsselworte erkennen, da die Tokenfolge **`meta void`** eine gültige Sequenz ist, die für Funktionsspezifikationen ohne Rückgabewert benötigt wird.

Die Schlüsselwörter beziehen sich auf die verwendbaren Metadatentypen, die in Metaprogrammen zur Referenzierung von Quelltextstrukturen und für allgemeine Berechnungen herangezogen werden können. Die Details der Semantik dieser Datentypen werden im nächsten Abschnitt erläutert.

5.7. Sprachelemente von C ohne Semantikerweiterung

Die Sprache C verfügt über Sprachmittel, die in der Erweiterung zu MetaC keine Semantik erhalten haben und somit nicht für die Definition von Metaprogrammen genutzt werden können, aber trotzdem für den Anteil von MetaC zur Verfügung stehen, der nicht zur Kompilierzeit ausgeführt wird. Im Folgenden wird beschrieben, warum diese Sprachelemente keine erweiterte Semantik erhalten haben.

5.7.1. Das Schlüsselwort `volatile`

Mit dem Schlüsselwort **`volatile`** kann man Variablen qualifizieren, deren Wert aus verschiedensten Gründen als volatil zu betrachten ist. Somit wird dem Compiler mitgeteilt, dass diese Daten sich jederzeit ändern können und aus diesem Grund einzelne Lese- und/oder Schreibzugriffe nicht herausoptimiert werden dürfen.

Beispiele für eine Anwendung dieses Qualifizierers sind Variablen, auf die von zwei verschiedenen Tasks aus zugegriffen werden kann oder Hardwareregister, deren Inhalt durch externe Ereignisse beeinflusst wird. In beiden Fällen möchte der Programmierer immer mit dem aktuellen Wert der Variablen arbeiten. Deswegen muss es möglich sein den Compiler daran zu hindern, eine falsche Optimierung auszuführen, die einen veralteten Wert aus einem Register bereitstellt.

Für die Metaprogrammierung hat ein solcher Qualifizierer keinen Nutzen, da hier weder auf Hardwareregister zugegriffen wird, noch mehrere parallel arbeitende Instanzen eines Programms auf gemeinsamen Daten arbeiten können. Deswegen ist der Einsatz dieses Schlüsselworts durch eine Warnung zu quittieren, die den Programmierer auf diesen Umstand hinweist. Trotzdem bleibt das Schlüsselwort auch in Metafunktionen und Metadeklarationen ein reserviertes Wort, das nicht als normaler Bezeichner fungieren kann.

5.7.2. Einschränkungen bei strukturierten Metadatentypen

Datenstrukturen die mit dem Schlüsselwort **`union`** definiert werden, zeichnen sich dadurch aus, dass ihre Elemente ein und denselben Speicherbereich nutzen. Das bedeutet, dass zugehörige Objekte dieses Typs die Größe des größten Elements besitzen. Ausdrücke die auf Einzelelemente zugreifen, ordnen den Daten des Objekts jeweils eine unterschiedliche Bedeutung zu.

Da die Repräsentation von Objekten im Speicher der Metadatentypen **`expr`**, **`func`**, **`ident`**, **`scope`**, **`stmt`**, **`strg`**, **`symb`** und **`type`** implementierungsspezifisch ist, kann für **`union`** basierte Datenstrukturen kein wohldefiniertes Verhalten festgelegt werden, die auf diesen Typen basieren. Denn die Repräsentation der Daten im Speicher, also die Ablage einer bestimmten Anzahl von Bits mit wohldefinierter Bedeutung, ist die Voraussetzung dafür, dass einem Bitmuster unterschiedliche Interpretationen zugeordnet werden können.

In C ist die Art und Weise, wie die Werte von Ganzzahl- und Gleitkommavariablen im Speicher abgelegt werden, wohl definiert. Die Semantik der Reinterpretation von Bitmustern an einer bestimmten Adresse kann dafür genutzt werden, ein und denselben Speicherbereich in Abhängigkeit des Programmzustandes für verschiedene Aufgaben zu nutzen oder die sich ändernde Bedeutung von Hardwareregistern in Abhängigkeit von Systemkonfiguration in C zu modellieren. Alle diese Applikationen sind keine Problemklassen, die mit Metaprogrammen gelöst werden sollen.

Für Metavariablen gibt es jedoch keine wohldefinierte Repräsentanz der Metadatentypen im Speicher. Dies würde der Implementierung eines MetaC Compilers zu große Einschränkungen auferlegen, ohne einen nutzbaren Mehrwert zu bieten. Deswegen gibt es für Metavariablen keine **`union`** basierte Metadatentypen.

5.8. Interne Funktionalität des MetaC Compilers

Die bisher vorgestellten Erweiterungen von Syntax und Semantik stellen lediglich die Infrastruktur bereit, um den Quelltext zu abstrahieren. Es ist mit diesen Erweiterungen aber nicht möglich, ein Metaprogramm zu schreiben, das das einbettende Programm analysiert oder Modifikationen daran vornimmt. Dafür ist eine zusätzliche Funktionalität notwendig, die mit den existierenden Operatoren für die neuen Metadatentypen überladen werden könnten, um die gewünschten Semantiken zu erzielen. Die wichtigsten Spracheigenschaften von MetaC für diesen Zweck werden im nächsten Kapitel vorgestellt. Darüber hinaus gibt es noch Erweiterungen, die sich transparent integrieren lassen und als einfache Erweiterungen in einem MetaC Compiler bereitgestellt werden. Diese werden im Folgenden erläutert.

5.8.1. Operatoren und Metafunktionen

Für einige der notwendigen Operationen gibt es Operatoren, die sinnvollerweise dafür eingesetzt werden können. Beispielsweise könnte das Aneinanderhängen von einzelnen Anweisungen zu einem *CompoundStatement* durch den binären Additionsoperator (**+**) ausgedrückt werden. Für die Ermittlung von Vorgänger und Nachfolger könnte eine Addition mit einem Kardinalwert verwendet werden. Andererseits ist ein ähnlicher Ansatz für die Kontroll- und Datenflussanalyse denkbar.

Da nicht ein und derselbe Operator für semantisch unterschiedliche Funktionen auf den selben Operandentypen verwendet werden kann, muss eine Lösung gefunden werden, die dieses Problem nicht hat. Die Einführung weiterer Schlüsselworte ist nicht wünschenswert, da eine solche tiefgreifende grammatikalische Erweiterung das Look&Feel der Sprache verändern würde. Weiterhin müsste sichergestellt werden, dass die zusätzlichen Parser konfliktfrei in die existierende EBNF integrierbar sind.

Da somit kein einheitliches, auf Operatoren und Schlüsselworten basierendes Konzept realisiert werden kann, um die gewünschte Funktionalität in der Sprache zu verankern, muss auf eine Alternative ausgewichen werden. Dafür ist das Naheliegendste die Einführung von Metafunktionen. Dieser Ansatz bietet neben seiner prinzipiellen einfachen Realisierbarkeit noch weitere Vorteile, die unabhängig von den konzeptionellen Schwierigkeiten mit Operatoren Nutzen bieten, die über das operatorbasierende Konzept hinausgehen.

Am wichtigsten ist die fehlende Einschränkung der Parameteranzahl auf die vom Operator festgelegte Anzahl. Darüber hinaus ist eine Erweiterung der so gebotenen Funktionalität nicht auf die Anzahl der verfügbaren Operatoren beschränkt und ist zudem durch die verwendeten Namen in den Funktionsaufrufen in seiner Semantik leichter zu verstehen. Die internen Metafunktionen folgen dabei der Namenskonvention, die erfordert, dass ihr erstes Zeichen das **$** Symbol ist. Dies hat den Vorteil, dass sie in einem standardkonformen C Quelltext kollisionsfrei eingebunden werden können, da diese Zeichen normalerweise nicht erlaubt ist. Für eine exemplarische Liste von Metafunktionen, sei an dieser Stelle auf die Tabellen im Anhang A.1 verwiesen.

5.8.2. Spezialsymbole zur Präprozessorintegration

Die beiden Bezeichner **__FILE__** und **__LINE__** sind spezielle, interne Präprozessormakros, die vom Präprozessor zu dem Dateinamen der Eingabedatei bzw. zur Zeile, in der das Token vorkommt, aufgelöst wird. Auch diese Symbole können von Metaprogrammen verwendet und instantiiert werden. Dazu sind besondere Vorkehrungen notwendig, da auch der MetaC Quelltext, bevor er vom MetaC Compiler verarbeitet wird, durch den C Präprozessor gefiltert werden muss. Dies ist erforderlich, um benutzerdefinierte Makros auflösen zu können und verwendete Headerdateien einzubinden.

Weiterhin ist dieser Schritt unverzichtbar, um eine Rückwärtskompatibilität gewährleisten zu können, da viele Definitionen die zunächst eine einfache Variablendeklaration waren, im Laufe der Zeit durch Makros ersetzt wurden. Ein Beispiel hierfür ist der Bezeichner **errno**, der laut POSIX Standard eine Variable vom Datentyp **int** ist und von POSIX konformen Funktionen dazu verwendet wird, Fehlercodes an die aufrufende Funktion zurückzuliefern. Die Benutzung einer solchen globalen Variable erlaubt aber nur die Implementierung von Applikationen mit einem einzigen Thread, da es ansonsten zu nichtdeterministischen Ausführungsergebnissen durch *race-conditions* kommen kann. Deshalb wurde mit der Einführung von Threads mit dem POSIX Standard 1003.4 (später umbenannt in 1003.1b) die Variable **errno** durch ein Makro mit dem gleichen Namen ersetzt. Das Makro gewährleistet die Rückwärtskompatibilität des APIs und wird zu einem Funktionsaufruf aufgelöst, der die Referenz der passenden threadlokalen Variable zurückgibt.

Da die Art der Implementierung nicht festgelegt ist, ist nicht bekannt ob einzelne Bezeichner durch ein Makro zu einem komplexeren Ausdruck aufgelöst werden. Deshalb kann auf den Einsatz des Präprozessors nicht verzichtet werden, und es müssen die systemspezifischen Headerdateien verwendet werden. Das hat aber wiederum zur Konsequenz, dass die Bezeichner **__FILE__** und **__LINE__** bei Einsatz in Metaprogrammen und insbesondere in deren QSM durch den Präprozessor aufgelöst werden und somit nicht erst am Zielort ersetzt werden.

Deshalb müssen die beiden Bezeichner als vordefinierte Symbole in den Symboltabellen bereitgehalten werden und im Code Generator des MetaC Compilers ersetzt werden. Der Zugriff auf die beiden Makros kann somit über die Metafunktion **$get_symbol** erfolgen, wobei die zurückgegebenen Symbolvariablen den passenden Typ haben müssen. **__FILE__** ist vom Typ **char const *const**, und **__LINE__** ist vom Typ **int**. Dadurch ist auch eine unmittelbare Typprüfung zum Zeitpunkt der Instantiierung möglich.

6. Quelltext-Strukturmuster

Das dynamische Verhalten eines Programmes, also die beobachtbaren Eigenschaften zur Laufzeit, ergeben sich aus der Struktur des Quelltexts, den verwendeten Variablen und ihren Datentypen. Blendet man die verwendeten Datentypen bei der Betrachtung eines Stück Programmcodes aus, so kann keine Aussage mehr über seine konkrete Semantik gemacht werden. Trotzdem enthält diese Darstellungsform noch genügend Informationen, damit ein erfahrener Programmierer einige Aussagen über ein solches Programmfragment machen kann. Dazu gehören neben dem Kontrollfluss auch Implikationen für die Ableitungen von Datentypen, die sich aus der Quelltextstruktur ergeben.

Die Struktur eines Quelltextes lässt sich noch stärker verallgemeinern, wenn neben den Datentypen auch die Variablennamen nicht als konkret betrachtet werden. Das heißt, dass ihre Namen durch beliebige andere ersetzt werden können, und dass die verwendeten Namen in einem solchen Quelltext nur zur Unterscheidung der einzelnen Variablen untereinander verwendet werden.

Andererseits ist diese Darstellungsform durch die Bindungsfreiheit der Namen in vielen Fällen zu allgemein, um noch genügend Semantik zu transportieren. So kann es unter Umständen hilfreich sein, dem Namen `min` die Semantik des mathematischen Minimum zuzuordnen. Dabei könnten die Parameter der Funktion mit definiertem Namen sowohl typfrei gehalten werden als auch an konkrete Typen gebunden werden. Beides hat seine Vor- und Nachteile und somit eine Berechtigung.

Diese Betrachtungsweisen eines Ausschnittes eines Quelltextes sind hervorragend geeignet, um allgemeine Diskussionen über Implementierungen durchzuführen oder Prinzipien beim Design eines Teiles der Software zu kommunizieren. Um diese Flexibilität der Kommunktionsmöglichkeiten zwischen menschlichen Programmierern in MetaC nachzubilden, gibt es in MetaC die so genannten Quelltext-Strukturmuster (QSM), die es ermöglichen mit Hilfe von Metaprogrammen Quelltextstrukturen zu suchen, zu analysieren und zu modifizieren. In diesem Kapitel wird zunächst der Begriff Quelltext-Strukturmuster definiert, Implikationen diskutiert und anschließend die verschiedenen Anwendungsmöglichkeiten der zugehörigen Sprachkonstrukte im Detail erläutert.

6.1. Begriffsdefinition und statische Semantik

Quelltext-Strukturmuster benötigen eine eigene Syntax, die es ermöglicht sie explizit in normalen Programmcode zu deklarieren. Dabei sollte es möglich sein, solche Deklarationen *inline* durchzuführen; trotzdem muss eine klare Trennung vom restlichen Quelltext erkennbar sein. Denn die unterschiedliche, statische und dynamische Semantik muss sowohl für den Programmierer offensichtlich als auch für einen Compiler differenzierbar sein. Dafür ist neben den erwähnten syntaktischen Strukturen eine konkrete Semantik erforderlich, die festlegt, wie diese sprachlichen Konstrukte in den unterschiedlichen Phasen der Kompilierung behandelt werden.

Zur Definition von QSM sind syntaktische Konstruktionen vorgesehen, die sich nahtlos in die existierenden einfügen. Diese unterscheiden sich von der restlichen Grammatik

dahingehend, dass sie in der semantischen Analyse nicht unmittelbar einer Typprüfung unterzogen werden. Eine Typprüfung von Quelltext-Strukturmustern darf nicht durchgeführt werden, da die verwendeten Symbole einen unvollständigen Typ haben sollen. Dies liegt daran, dass die Bindung der Symbole an Deklarationen zum Parsezeitpunkt eines QSM nicht feststeht. Außerdem könnten die Namen einzelner Knoten in dem Muster noch ersetzt werden. Dadurch bestimmen die QSM nur die Struktur von Quelltext, der auf ihrer Basis instantiiert oder gesucht wird, nicht aber die konkrete Semantik. Diese ergibt sich erst durch die Art der Anwendung und der dabei verwendeten Variablen.

Quelltext-Strukturmuster können für zwei grundlegend verschiedene Anwendungen verwendet werden: zum Vergleich mit existierendem Quelltext und als Vorlage zur Erzeugung von neuem Quelltext. In beiden Fällen wird die Position der Applikation innerhalb des Quelltextes des Programms durch das Metaprogram zu seiner Laufzeit entschieden. Dabei sind grundsätzlich folgende syntaktische Strukturen zu unterscheiden:

- *expression*
- *statement*
- *declaration*
- *function-definition*

Diesen Strukturen sind unterschiedliche Parser zugeordnet. Somit kann das Nichtterminal *expression* beispielsweise nicht durch Quelltext ersetzt werden der eine Anweisung ist oder umgekehrt. Dadurch müssen für die unterschiedlichen Arten von Knotenpunkten innerhalb des Parsebaums entsprechende Strukturmuster anwendbar sein. Entsprechend werden folgende Strukturmuster unterschieden:

- Anweisungs Stuktur Muster (statement structure pattern oder SSP)
- Ausdrucks Stuktur Muster (expression structure pattern oder ESP)
- Typ Struktur Muster (type structure pattern oder TSP)

Um QSM für ein bestimmtes syntaktisches Konstrukt definieren zu können, muss ein Parser erstellt werden, der eben dieses syntaktische Konstrukt als Nichtterminal enthält. Für Deklarationen und Funktionsdefinitionen sind keine eigenen Parser zur Definition von QSM erforderlich, da sie sich aus einer Kombination der anderen Strukturen erstellen lassen. Wie durch einen kombinierten Einsatz Deklarationen beschreiben und instantiiert werden können, ist in Abschnitt 6.3.4 beschrieben.

In der Standard C Syntax treten Ausdrücke als Elemente von Anweisungen auf, nicht aber umgekehrt. Weiterhin ist das Nichtterminal *expression* das erste Element von *expression-statement*. Diese beiden Tatsachen zusammen führen zu der Notwendigkeit, für beide Typen von Strukturmustern unterschiedliche Grammatiken bereitzustellen. Da die Sprache C sich dadurch auszeichnet besonders wenige Schlüsselwörter zu besitzen, sollen die grammatikalischen Erweiterungen nicht einfach durch zusätzliche Schlüsselwörter eingeführt werden. Stattdessen ist die existierende Grammatik für die Definition von Ausdrucksmuster mit zusätzlicher, kontextabhängiger Semantik überlagert worden. Für Anweisungsmuster wurde ein neues lexikalisches Token eingeführt, das eine nahtlose Erweiterung der existierenden Syntax ermöglicht.

QSM müssen nicht nur definierbar sondern auch referenzierbar sein. Das heißt, wenn mehrere Muster innerhalb des gleichen Kontexts definiert werden, muss es für die Verwendung von QSM eindeutig beschreibbar sein, welches Codefragment für diesen Vorgang verwendet werden soll. Um dies sicherzustellen kommen nur zwei verschiedene Ansätze in Frage; entweder bekommt jede Vorlage einen eindeutigen Namen, der über qualifizierende Schlüsselwörter in einen alternativen Namensraum abgelegt wird. Andererseits wird jede Vorlage mit einer normalen Variable durch eine Art von Zuweisung assoziiert, und kann so indirekt über die Variable verwendet werden.

Beide Verfahren finden in MetaC Verwendung. Ersteres kann für alle Arten von QSM verwendet werden. Letzteres wird nur bei der Deklaration von SSP unterstützt, da diese typischerweise aus längeren Codeabschnitten bestehen und deshalb nicht unmittelbar in eine Zuweisung eingebettet werden sollen, damit die Lesbarkeit des Quelltextes nicht darunter leidet.

6.1.1. Muster von Datentypen

Die Syntax des **`typeof`** Operators wurde bereits in Abschnitt 5.6.2 erläutert. Dabei wurde nur die statische Semantik betrachtet und die Funktionalität der dynamischen Semantik kurz beschrieben. Die dynamische Semantik gehört mit zu den Quelltext-Strukturmustern, da hierbei in Abhängigkeit von den Metadaten eines Metaprogramms zu seiner Laufzeit der Typ errechnet wird. Dabei werden die im Quelltext eingebetteten syntaktischen Strukturen des **`typeof`** Operators genutzt, um diese Berechnung durchzuführen. Dieses Verfahren, das hier diskutiert wird, definiert die dynamische Semantik des *type-structure-pattern* (TSP).

Die Syntax dafür verwendet das vorangestellte Schlüsselwort **`typedef`**, um anzuzeigen, dass der nachfolgende Bezeichner kein Typalias und kein Ausdruck ist, sondern eine Metavariable vom Typ **`type`**. Ihr Wert wird daraufhin äquivalent zu einer Typdefinition zur Laufzeit behandelt. Ohne dieses Schlüsselwort wird die Metavariable als Beginn eines Ausdrucks gesehen (der immer den Metadatentyp **`type`** liefern würde), und die nachfolgenden Token würden zu einem Syntaxfehler führen.

Die abhängigen Typdefinitionen werden im Hinblick auf die übergebene Variable vom Metadaten **`type`** konstruiert. Dazu definieren die beiden Parser *abstract-declarator* und *dereferencing-declarator* die dafür gültige Grammatik. Der Parser *abstract-declarator* ist bereits in C definiert und ermöglicht die Deklaration von abgeleiteten Datentypen, wie Funktionen, Felder, Strukturen und Zeiger. Dies stellt aber keine Mittel bereit, um aus einer Typdefinition den Typ der Basis oder eines Elements abzuleiten.

Deshalb wird der Parser *dereferencing-declarator* neu eingeführt. Dieser bietet zwei syntaktische Konstrukte mit folgender Semantik:

1. Das Token **`&`** dereferenziert einen Zeiger oder ein Feld auf seinen Basistyp.

2. Die Tokenkette **`.`** *identifier* extrahiert den Datentyp eines Strukturelements.

Zusätzlich bietet der Parser *qualifier-modification-list* die Möglichkeit, die Qualifizierer eines Typs zu verändern. Dabei zeigt das Substraktionstoken **`-`**, wenn es einem Qualifizierer vorangestellt wird, dass dieser Qualifizierer vom referenzierten Typ zu entfernen ist. Dabei wirkt diese Modifikation immer auf die höchste Ableitung und nie auf den Basistyp einer Ableitung. Um also die Qualifizierer eines Basistyps verändern zu können, muss erst die Ableitung entfernt werden und nachträglich wieder hinzugefügt werden. Die

```
type t1 = typeof(volatile struct mach_reg *const);
type t2 = typeof(typedef t1 &const -volatile *);
somit ist folgender Ausdruck wahr:
t2 == typeof(const struct mach_reg *)
```

Abbildung 6.1.: Anwendungsbeispiel zum Parser *dereferencing-declarator*

```
typedef volatile struct mach_reg *const t1;
typedef boost::add_pointer<
          boost::add_const<
            boost::remove_volatile<
              boost::remove_pointer<t1>::type
            >::type
          >::type
        >::type t2;
```

Abbildung 6.2.: Lösungsvariante mit Boost type-traits in C++ zur Abbildung 6.1

dafür notwendigen Analysen und Modifikationen lassen sich in Metaprogrammen imperativ, unter Verwendung geeigneter Metafunktionen, für beliebige unbekannte Datentypen beschreiben.

Bei der Verwendung dieser syntaktischen Struktur ist auch eine Kombination mit einem *abstract-declarator* erlaubt. Eine solche Kombination bietet die Möglichkeit, Qualifizierer von einem Datentyp zu entfernen und/oder neue hinzuzufügen. Ein Beispiel hierzu ist in Abbildung 6.1 dargestellt. In diesem Beispiel abstrahiert der Typ **`t1`** einen Zeiger auf einen volatilen Datenbereich der Struktur **`mach_reg`**. Typischerweise werden solche Konstruktionen verwendet, um Hardwareregister, die in den Hauptspeicher eingeblendet sind, in Software zu adressieren. Da jeder Zugriff auf ein solches Register aber unterschiedliche Werte liefern kann, erstellt man üblicherweise eine Kopie des Inhaltes des Registers, bevor Operationen auf den Inhalt angewendet werden.

Da sich die Adresse des Registers zur Laufzeit typischerweise nicht ändert, wird der Zeiger auf das Register als konstant deklariert. Dies wird mit der Konstruktion des Typs **`t2`** erledigt. Dazu wird der Qualifizierer **`volatile`** durch den Qualifizierer **`const`** ersetzt und der konstante Zeiger durch einen normalen ersetzt.

Die *Boost* Biliothekensammlung [4] bietet für C++ mit den so genannten *type-traits* einen ähnlichen Mechanismus an, der jedoch deutlich komplizierter zu verwenden ist. Eine zur MetaC Variante äquivalente Lösung in C++ ist in Abbildung 6.2 dargestellt. Die hierfür verwendeten Verfahren für die Implementierung stehen in C nicht zur Verfügung. Dieses einfache Beispiel zeigt, dass die Syntax von MetaC deutlich einfacher zu benutzen ist als die umständliche Realisierung in C++ mit Hilfen von Templateprogrammierung. Noch deutlicher ist der Unterschied, wenn für die Konstruktion des Typs Kontrollfluss notwendig ist. Dieser entspricht in MetaC direkt der Syntax von C und kann in C++ nur unter Zuhilfenahme spezieller Bibliotheken und komplizierter Programmierverfahren erreicht werden.

6.1.2. Muster von Ausdrücken

Strukturmuster von Ausdrücken (ESP) können dazu eingesetzt werden, in existierendem Quelltext Ausdrücke zu suchen, zu ersetzen oder einzufügen. Dabei erhält der Ausdruck seine Laufzeitsemantik erst durch die konkrete Bindung der verwendeten Symbole an Variablen im Gültigkeitsbereich des Instantiierungsortes. Während dieses Bindungsprozesses an die Deklarationen im Gültigkeitsbereich der Zielposition werden die konkreten Datentypen bestimmt und die Typüberprüfung durchgeführt.

Das Verfahren zur Symbolauflösung ist abhängig von der Anwendung und wird entsprechend in den Abschnitten 6.2.1 und 6.3.2 im Detail beschrieben. Aus beiden Methoden ergibt sich eine enorme Flexibilität und Variabilität für die Anwendung von QSM. Einerseits können dadurch Algorithmen typunabhängig gesucht und implementiert werden; andererseits sind Modifikationen realisierbar, die beispielsweise Ausdrücke neu verketten. So könnte der Operand einer Operation durch einen Funktionsaufruf ersetzt werden, die den alten Operand als Argument verwendet.

Für die Definition von QSM von Ausdrücken gibt es in MetaC keinen eigenständigen Parser. Bei Ausdrücken kann auf diese zusätzliche Komplexität verzichtet werden, da es möglich ist, den Parser für Typumwandlungen (*cast-expression*, beschrieben in Abschnitt 6.5.4 in [17]) semantisch derart zu überladen, dass er diese Zusatzaufgabe bewältigen kann. Dieser Parser erwartet die folgende Sequenz aus Terminalen (fett gedruckt) und Nichtterminalen (kursiv gedruckt):

(*type-name* **)** *cast-expression*

Der Parser *type-name* kann dabei eine beliebige Typkonstruktion sein, die auch auf Typaliasdefinitonen basieren darf. Das Nichtterminal *cast-expression* besteht alternativ aus einer *unary-expression* und diese wiederum aus einer *postfix-expression* und im nächsten Schritt aus einer *primary-expression*. Somit kann rechter Hand vom Nichtterminal *type-name* ein beliebiger Ausdruck stehen, da der Parser *primary-expression* einen beliebigen geklammerten Ausdruck akzeptiert.

Weiterhin ist eine explizite Typumwandlung von einem Metadatentyp zum Typ **`expr`** überflüssig, da alle kompatiblen Typen diese Wandlung automatisch vollziehen. Dieses Verhalten entspricht der automatischen Typkonversion von einem Integer- zu einem Gleitkommatyp und umgekehrt in C. Folglich ist es möglich, die Typumwandlung zum Metadatentyp **`expr`** semantisch für die Deklaration von ESP (expression structure pattern) zu überladen. Dadurch wird die notwendige Syntax zur Definition von Ausdrucksmustern geschaffen, ohne einen zusätzlichen Parser definieren zu müssen.

In Abbildung 6.3 sind hierzu zwei Beispiele dargestellt, die die Anwendung einer solchen Konstruktion anhand von Deklarationen zeigen. Die Deklaration in der ersten Zeile definiert die Variable **`p0`** vom Metadatentyp **`expr`** und initialisiert diese mit dem nachfolgenden, trivialen ESP. Dieses besteht nur aus einer einzelnen Bezeichnerreferenz, dem Bezeichner **`e`**.

Die zweite Zeile zeigt eine kompliziertere Konstruktion, die insgesamt vier Bezeichner referenziert. Die syntaktische Struktur des ESP von **`p1`** hat folgende Implikaitonen: Der

```
expr p0 = (expr) e;
expr p1 = (expr) (a * b - x[c(d)]);
```

Abbildung 6.3.: Beispiele für Strukturmuster von Ausdrücken

Bezeichner **x** muss letztendlich zu einem Ausdruck vom abgeleiteten Typ Pointer oder Array und **c** zu einer Funktion aufgelöst werden. Der Datentyp von **d** muss mit dem Datentyp des ersten und einzigen Parameters, der zu **c** zugehörigen Funktion kompatibel sein.

Die Bezeichner im *expression-structure-pattern* unterliegen während dem Parsevorgang keiner Symbolauflösung. Diese wird erst bei der Verwendung des Musters durchgeführt und folgt dabei während einer Instanziierung der in Abschnitt 6.3.2 beschriebenen Methodik. Die Bezeichner, die in QSM Verwendung finden, müssen somit zum Parsezeitpunkt nicht zu bereits durchgeführten Deklarationen aufgelöst werden können, wie das normalerweise der Fall ist. Dieser Vorgang wird deshalb beim Parsen von ESPs vollständig unterdrückt.

6.1.3. Muster von Anweisungen

Für die Assoziation eines Anweisungs-Struktur-Musters (SSP) mit einer Variablen können die existierenden Operatoren nicht verwendet werden. Dies ist nicht möglich, da eine Anweisung innerhalb eines Ausdrucks zu einem unlösbaren Problem während der Syntaxanalyse führen würde. Es wäre nicht entscheidbar, ob die gelesene Sequenz zu einem Ausdruck oder weiter zu einer Anweisung reduziert werden muss. Mit den existierenden Terminalen kann auch kein Nichtterminal eingeführt werden, das eine Zuweisung innerhalb eines Ausdrucks unterbringt. Der Grund ist, dass die grammatikalische Struktur von C die Verwendung von Ausdrücken als erstes Nichtterminal in Anweisungen vorsieht.

Somit kommt es bei einer einfachen Erweiterung der Zuweisung um die Variante *expression* **=** *statement* zum unlösbaren Konflikt bei der Tokenfolge *identifier* **=** *identifier*. Hierbei wird unentscheidbar, ob der Bezeichner auf der rechten Seite nur zu einem Ausdruck reduziert werden soll oder weiter zu einer Anweisung. Da auch das unmittelbar nachfolgende Token diese Unentscheidbarkeit nicht auflösen können, müsste diese Mehrdeutigkeit in der Phase der semantischen Analyse aufgelöst werden. Solche Ansätze sind zwar denkbar, führen aber zu einer Reihe von Problemen bei der Implementierung eines Compiler-Front-Ends. Außerdem lassen diese eine Implementierung wenig performant werden, und eine Verifikation der Sprachdefinition wird deutlich schwieriger.

Folglich kommt eine Integration in die bestehende Syntax für Ausdrücke und die Verwendung eines existierenden Operators nicht in Betracht. Als Alternative bleibt, eine Syntax zur Deklaration eines SSP und einen neuen Operator zu definieren, dessen Argument indirekt auf eine Anweisung verweist. In Hinblick auf die Zeichen der deutschen Tastatur finden die Symbole **$**, ° und **§** noch keine Verwendung. Das Dollarzeichen wird in vielen Implementierung als gültiges Zeichen in Bezeichnern unterstützt und kann somit nicht eingesetzt werden. Die Zeichen für Grad und Paragraph sind hingegen nicht im minimalen verlangten Buchstabensatz der ISO9899:1999 (Abschnitt 5.2.1). Unabhängig davon soll eine grammatikalische Konstruktion auch eingängig und in ihrer Verwendung klar sein. Diese Forderung wird von keinem der genannten Zeichen erfüllt.

In C gibt es neben den normalen Operatoren, die durch einen einzelnen Buchstaben repräsentiert werden, auch Operatoren, die aus zwei Buchstaben zusammengesetzt werden. Diese müssen explizit als gültige Token für die lexikalische Analyse ausgewiesen werden, damit die syntaktische Analyse konfliktfrei durchgeführt werden kann. In C sind die gültigen Interpunktionszeichen in Abschnitt 6.4.6 von [17] definiert. Dazu gehören sowohl einzelne Zeichen (z.B. **:** für Sprungmarken, **=** für Zuweisungen, **+** für Additionen), als

```
identifier :=  statement
```

Abbildung 6.4.: Der Parser *statement-structure-pattern*

auch Sequenzen aus zwei (z.B. **+=** für Akkumulationen, **«** für Linksschieben) oder drei Zeichen (z.B. **«=** für Linksschieben und Zuweisung, **...** für Ellipsen in Parameterlisten).

Die Token **+** und **=** werden beispielsweise in den Parsern *additive-expression* und *unary-expression* bzw. *assignment-expression* und *init-declarator* verwendet. Ihre Kombination zu dem Token **+=** wird als Akkumulationsoperator in *assignment-expression* aufgelöst. Das Token **:** wird im Parser *labeled-statement* genutzt, um Sprungmarken im Quelltext zu setzen und in *conditional-expression*, um die beiden alternativen Ausdrücke voneinander zu trennen.

Die in Pascal ([11], [13]) für Zuweisungen genutzte Sequenz **:=** findet in C keine Verwendung und kann dadurch in gleicher Weise wie der Akkumulationsoperator für eine zusätzliche Definition genutzt werden. Dazu ist nicht zwingendermassen erforderlich, diese Sequenz als neues Token zu definieren; es vereinfacht aber die Implementierung des Parsers.

Durch den Einsatz dieses neuen Tokens ist es möglich, Anweisungs-Struktur-Muster (SSP) konfliktfrei mit einem eigenen Parser als Spracherweiterung einzuführen (siehe Abb. 6.4). Dazu wird eine Syntax definiert, die eine ähnliche Sequenz von Nichtterminalen und Terminalen erwartet, wie in einem *labeled-statement*. Einzig das Terminal **:** wird durch das neue Token **:**= ersetzt, wodurch die erweiterte MetaC Syntax LALR(1) konform bleibt.

Dieser Parser ist dem Parser *statement* als neue Variante hinzugefügt worden. Auch *expression* kommt dafür prinzipiell in Frage – allerdings erhält der Quelltext in der Praxis dadurch seltsam anmutendes Aussehen (z.B. zwei aufeinanderfolgende Semikolons am Ende eines *expression-statement*, das ein *statement-structure-pattern* definiert). Durch die Möglichkeit *statement-pattern* direkt in Ausdrücken als Rvalue referenzieren zu können, werden sie in Zuweisungen verwendbar. Dadurch können SSP direkt dem Element eines Feldes zugewiesen werden.

Dies kann aber auch durch den Einsatz eines referenzierenden Bezeichners erreicht werden. Dafür ist das neue Schlüsselwort **codeof** vorgesehen, das *statement-pattern* in Ausdrücken verfügbar macht. Diese Form verbessert die Lesbarkeit deutlich gegenüber direkt in Ausdrücken eingebettete SSP, die üblicherweise mehrere Zeilen überspannen.

Durch die Markierung der Anweisungsmuster mit einem Bezeichner, der im Namensraum für Sprungmarken gespeichert wird, können *statement-structure-pattern* eindeutig referenziert werden. Dazu kann das Schlüsselwort **codeof** in Ausdrücken verwendet werden. Die Überführung der zugehörigen Bezeichner in einen alternativen Namensraum ermöglicht die Wiederverwendung des selben Namens für eine Variable vom Typ **stmt**, die diese Struktur referenziert. Außerdem wird dadurch erreicht, dass sich die Namen der SSP ansonsten wie Sprungmarken in C verhalten. Dadurch wird das Look&Feel der Sprache erhalten.

In Abbildung 6.5 befinden sich zwei Beispiele für die Deklaration von SSPs und eins zu ihrer Referenzierung innerhalb eines Ausdrucks. Das Muster **null_pattern** deklariert eine triviale, leere Anweisung. Das SSP **full_pattern** hingegen definiert eine komplexe Anweisungsstruktur mit Variablendeklaration und Sprungmarke. Dieses wird in der letzten Zeile mit dem **codeof** Operator der Variable **s** zugewiesen.

```
null_pattern:= ;

type T;
full_pattern:=
    {
        T d;
        start:
        if (d.x < abc) {
            d.x = f();
            goto start;
        }
    }

stmt s = codeof full_pattern;
```

Abbildung 6.5.: Zwei Beispiele für QSM von Anweisungen (SSP)

6.1.4. Implikationen für den Parsevorgang von QSM

Bezeichner werden während des Parsevorgangs nicht aufgelöst, da das Strukturmuster an einer zum Parsezeitpunkt unbekannten Stelle instanziiert oder zur Suche eingesetzt werden soll. Somit ist auch der Gültigkeitsbereich am Zielort nicht identisch mit dem Gültigkeitsbereich am Deklarationsort. Deswegen werden Symbole erst aufgelöst, wenn sie zur Anwendung kommen und dies notwendig ist.

Die Bezeichner ohne angegliedertes syntaktisches Schlüsselwort referenzieren neben Variablen und Funktionen auch Typdefinitionen. Diese Tatsache führt zu einem Dilemma während des Parsens: Wenn kein Gültigkeitsbereich verwendet werden darf, so können Typdefinitionen nicht aufgelöst werden und insbesondere Typnamen als solche nicht erkannt werden. Auf der anderen Seite ist es aber wünschenswert, Quelltextstrukturen für verschiedene Datentypen nutzen zu können.

Dieses Dilemma kann behoben werden, indem ein eingeschränkter Gültigkeitsbereich für die Symbole innerhalb von QSM verwendet wird. In diesem dürfen nur die Typdefinitionen des umschließenden Gültigkeitsbereiches sichtbar sein, damit der gewünschte Grad an Variabilität erhalten bleibt. Damit die Spezifikation von QSM variabel und flexibel gegenüber Datentypen wird, werden die Typdefinitionen noch um die Metavariablen erweitert, die den Metadatentyp **`type`** besitzen. Somit können bei einer Verwendung eines QSM im selben Gültigkeitsbereich die verwendeten Typnamen durch den Inhalt der entsprechenden Metavariablen ersetzt werden.

Alle anderen Bezeichner werden während dem Parsevorgang weder aufgelöst, noch auf Kollisionen hin überprüft. Ebenso wird die gesamte semantische Analyse übergangen, da für eine semantische Überprüfung von Ausdrücken und Anweisungen die Datentypen der Variablen bekannt sein müssen. Diese Eigenschaften sind aber zu diesem Zeitpunkt noch nicht bekannt und können deshalb nicht während des Parsens kontrolliert werden.

Letztendlich werden QSM erst zum Zeitpunkt ihrer Anwendung dem Gültigkeitsbereich des Zielortes zugeordnet. Insgesamt lassen sich die Implikationen wie folgt zusammenfassen:

- Bezeichner bleiben ungebunden; d.h. sie werden nicht mit Deklarationen im aktuellen Gültigkeitsbereich assoziiert und fehlende Deklarationen lösen keine Fehlersituation aus.

- Zur Parsezeit werden Deklarationen nur auf Syntax überprüft. Insbesondere werden die durchgeführten Definitionen nicht in die entsprechende Symboltabelle aufgenommen.

- Zur Auflösung von Typnamen die in Deklarationen oder Ausdrücken von QSM verwendet werden, werden die Typdefinitionen des umgebenden Gültigkeitsbereiches und Metavariablen vom Metatyp **`type`** herangezogen.

6.2. Suchen mit Hilfe von Quelltext-Strukturmuster

Ein zentrales Problem bei der Rekonfiguration und Modifikation von Quelltexten ist das Auffinden der entsprechenden Position, an der Veränderungen vorgenommen werden sollen. Dabei ist meist bekannt, welcher allgemeinen Struktur ein Codefragment folgt. Jedoch sind konkrete Eigenschaften, wie z.B. die verwendeten Namen einzelner Variablen oder der Typ einzelner Variablen unbekannt. Genauso kann es umgekehrt der Fall sein, dass der Name einer Funktion bekannt ist, jedoch ihre Parameter oder die umgebende Codestruktur unbekannt sind.

Für die Modifikation von Prosa eignet sich der Einsatz von regulären Ausdrücken ausgezeichnet. Jedoch gibt es keinen vergleichbaren Mechanismus für syntaktische Strukturen in C. Reguläre Ausdrücke lassen sich hierfür zwar auch anwenden, um einfache Modifikationen, wie das Austauschen von Parametern, zu realisieren. Sie scheitern jedoch an den rekursiven Strukturen, die durch die Syntax einer Programmiersprache wie

Metadatentyp	Bedeutung als Suchmuster
`zed`	Es wird eine *integer-constant* erwartet.
`expr`	Es wird ein Ausdruck entsprechend dem Wert erwartet.
`func`	Der Knoten muss eine Funktion referenzieren.
`ident`	Es wird ein passender Bezeichner erwartet.
`real`	Eine *floating-constant* wird akzeptiert.
`scope`	*Darf nicht verwendet werden,* *da es keine passende syntaktische Struktur gibt!*
`stmt`	Diese Metavariablen müssen als direktes Kind eines *expression-statement* benutzt werden und suchen nach passenden Anweisungen entsprechend ihrem Wert.
`strg`	Ein *string-literal*.
`symb`	Eine Variable wird erwartet.
`type`	Die Typgleichheit einer Deklaration wird geprüft.
keiner	Bezeichner ohne Metavariable werden literal verglichen.
anderer	*Abgeleitete Datentype dürfen nicht verwendet werden*

Tabelle 6.1.: Semantik eines Bezeichners abhängig von Metadatendeklarationen im gegenwärtigen Sichtbarkeitsbereich

C vorgegeben sind. Um solche Strukturen auffinden zu können, bedarf es eines Mechanismus, der es erlaubt, die möglichen Rekursionen als Freiheitsgrade zu spezifizieren. Andererseits muss es auch möglich sein, Einschränkungen an den Stellen vorzunehmen, an denen konrete Informationen über den Quelltext vorliegen. Dieser Abschnitt erläutert, wie QSM dazu eingesetzt werden können, um genau jene Aufgabe zu erfüllen.

6.2.1. Semantik von QSM bei der Suche

QSM eignen sich durch die mangelnde Bindung der verwendeten Bezeichner an Deklarationen sehr gut zum Suchen. Für den Einsatz als Suchmuster wird eine Assoziation der Bezeichner mit den Metadatendeklarationen im aktuellen Sichtbarkeitsbereich durchgeführt. Abhängig vom Metadatentyp der Deklaration bekommt der Bezeichner im Suchmuster dadurch eine unterschiedliche Bedeutung (siehe Tabelle 6.1).

Die Suche mit Hilfe eines Musters unterscheidet dabei noch den Wert der Metavariable zum Suchzeitpunkt. Ist dieser Wert das Nulläquivalent, so wird eine beliebige syntaktische Struktur des entsprechenden Metadatentyps akzeptiert. Dadurch wird das Problem der Rekursion mit beliebiger Struktur gelöst. Ist dieser Wert ungleich Null, so wird ein direkter Vergleich durchgeführt. Dabei wird für die Metadatentypen **`expr`** und **`stmt`** die Suche rekursiv fortgesetzt. Das bedeutet, dass die in der entsprechenden Metavariable referenzierte syntaktische Struktur wiederum als Muster für den Vergleich nach dem selben Schema herangezogen wird.

In Abbildung 6.6 ist ein einfaches Beispiel eines Suchmusters (linke Spalte) und seinen zugehörigen möglichen Treffern (mitte) in graphischer Parsebaumform dargestellt. In der rechten Spalte ist ein Beispiel für eines Codefragment gegeben, dass nicht zum Suchmuster passt. Hierbei ist entscheidend, dass der rechte Operand der Addition des Suchmusters eine Metavariable vom Typ **`s`**ymb referenziert. Dadurch wird der letzte Parsebaum nicht als Treffer für das Suchmuster akzeptiert, da hier der rechte Operand eine Multiplikation ist. Vertauscht man beide Operanden des letzten Beispiels, so bekommt man einen Treffer, da nun für die Metavariable **`s`** die Variable **`a`** als Treffer erkannt wird und die Multiplikation zur Metavariable **`e`** des Suchmusters passt.

In Tabelle 6.2 sind zwei weitere Beispiele zu Strukturmustern von Ausdrücken (ESP) dargestellt, mit jeweils einem Beispiel zu einem Möglichen Treffer beim Einsatz als Suchmuster, sowie einem Fehlschlag. Diese Tabelle dient der Verdeutlichung des Prinzips und

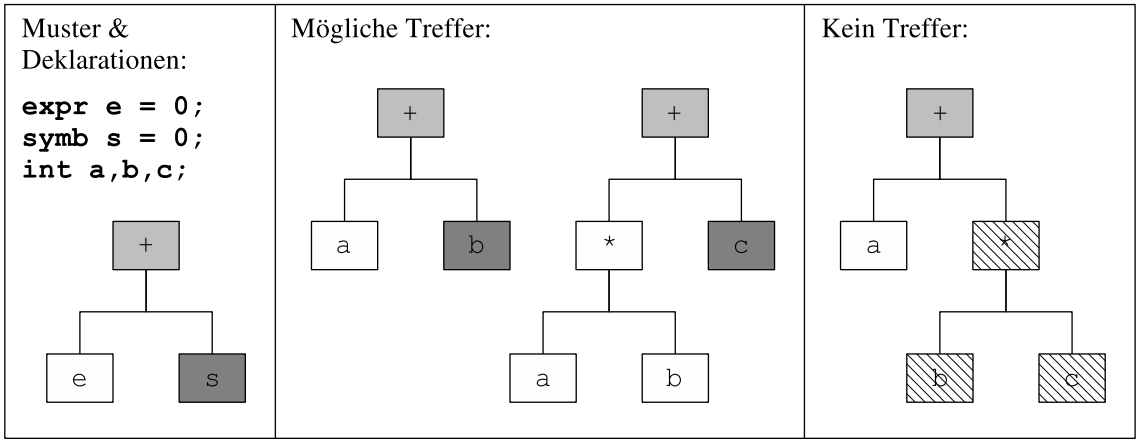


Abbildung 6.6.: Suchmuster und Treffer als abstrakte Syntaxbäume

ESP	Treffer	kein Treffer	Kommentar zum Fehlschlag
`(expr) f(e)`	`sin(x+y)`	`d->f(i)`	`f` erwartet eine Funktion, `d->f` ist Ausdruck
`(expr) e(s)`	`d->f(i)`	`sin(x+y)`	`s` erwartet ein Symbol, `x+y` ist ein Ausdruck
mit `expr e; func f; symb s;`			

Tabelle 6.2.: Weitere Suchbeispiele mit ESPs

der Darstellung der Funktionalität der Freiheitsgrade in den Suchmustern zur Einschränkung und Spezifikation der Suche.

6.2.2. Erweiterte Analysemöglichkeiten

Um das Suchergebnis weiter analysieren zu können, gibt es die Zusatzfunktion **`$match`**. Diese weist den zugehörigen Metavariablen den Ausdruck der Übereinstimmung zu. Damit dieser Vorgang eindeutig ist, darf jede Metavariable nur einmal in einem Suchmuster benutzt werden. Hingegen unterliegt ein QSM, das zur Instantiierung verwendet werden soll, keiner solchen Einschränkung. Verstößt der Benutzer gegen diese Einschränkung, so sollte eine Implementierung ihn darauf aufmerksam machen. Das Ergebnis einer solchen Operation ist in jedem Fall nicht definiert.

Die Verwendung von Metavariablen des Typs **`scope`** macht als Suchmuster und Quelle für Instantiierungen keinen Sinn, da Gültigkeitsbereiche keine direkte Repräsentanz in der Quelltextstruktur besitzen. Für **`stmt`** gilt letzteres zwar nicht, aber sie können trotzdem nicht uneingeschränkt verwendet werden. Da eine Anweisung niemals ein Element eines Ausdrucks sein kann, darf eine solche Metavariable in Ausdrücken keine Verwendung finden. Einzige Ausnahme hierbei ist, die Verwendung als einziges Element eines *expression-statement*. Durch diese Ausnahme können beliebige Anweisungsstrukturen über die entstehende Rekursion darstellt und gesucht werden.

Bei der Modifikation von Quelltext können durch den Einsatz der mit **`$match`** ermittelten Ausdrücke Codestrukturen erstellt werden, die zum Zeitpunkt der Erstellung des Metaprogramms nicht bekannt sind. Wichtig dabei ist aber, dass jeder Ausdruck *Nebenwirkungen* (Definition in Abschnitt 5.2.1) haben kann. Dadurch ist eine mehrfache Verwendung eines ermittelten Ausdrucks bei neu instantiiertem Quelltext mit der Gefahr verbunden, dass die Semantik auf ungewollte Art verändert wird.

Diese Situationen können durch eine detaillierte Analyse des entsprechenden Ausdrucks erkannt werden. Beispielsweise ist die Verwendung der Inkrement Operatoren problematisch, da sie den Wert des entsprechenden Objekts modifizieren. Wird diese Veränderung ungewollt mehrfach durchgeführt, so verändert sich die Semantik des Quelltextes. Eine einfache Verwendung, wie es bei Refaktorisierungen von Quelltext üblich ist, stellt hingegen kein Problem dar. Voraussetzung dabei ist, dass die alte Instanz des Ausdrucks durch die Neuinstantiierung überschrieben oder durch ein alternatives Verfahren gelöscht wird.

```
/* metavariablen */
expr i,e;
zed c;
type t;
func f;
symb v;
stmt x;

/* Muster von direkten Funktionsaufrufen */
expr call = (expr) f(e);
/* mögliche Treffer */
x(i++)
abc(f(x));

/* Muster eines allgemeinen Funktionsaufrufes */
expr ind_call = (expr) e(v);
/* mögliche Treffer */
p->x(a)
s.y(i)
f(j)

/* Muster einer for-Schleife */
pattern := for(t v = i; v != c; e) x;
/* möglicher Treffer */
for (int i = 0; i != 2; ++i) f();
```

Abbildung 6.7.: Beispiele von Suchmustern und ihre Treffer

6.2.3. Beispiele zur Suche von Quelltext

Der Einsatz von Suchmustern ist exemplarisch in Abbildung 6.7 dargestellt. Zu Beginn sind die Definitionen der Metavariablen, die unten in den Suchmusterdefinitionen benutzt werden.

Das erste Suchmuster **`call`** hat die Struktur eines Funktionsaufrufes mit einem einzelnen Parameter. Da **`f`** vom Typ **`func`** ist, werden dabei nur direkte Funktionsaufrufe berücksichtigt. Hat **`f`** außerdem einen konkreten Wert, so werden nur Funktionsaufrufe dieser Funktion berücksichtigt. Die Metavariable **`e`** hat keinen Einfluss, wenn ihr Wert Null ist. Ist ihr Wert ungleich Null, so werden nur Ausdrücke akzeptiert, die dem referenzierten Muster entsprechen.

Das zweite Suchmuster **`ind_call`** hat eine sehr ähnliche Struktur. Allerdings ist die Variable **`e`** vom Type **`expr`**, wodurch hier wieder ein rekursives Matching stattfindet, wenn **`e`** ungleich Null ist. Verweist **`e`** beispielsweise auf die Metavariable **`f`**, so ergibt sich wieder eine ähnliche Situation, wie im ersten Fall. Ist **`e`** Null, so werden alle möglichen Funktionsaufrufe akzeptiert, wozu eben auch indirekte Funktionsaufrufe, wie der erste mögliche Treffer zählen, die bei **`call`** nicht berücksichtigt werden. Die Metavariable **`v`** beeinflusst das Suchmuster erheblich; es werden nur direkt übergebene Variablen berücksichtigt. Hat **`v`** dazu noch einen Wert ungleich Null, so wird nur diese Variable akzeptiert.

Bei Anweisungsmustern verhält sich das Ganze äquivalent. Auch hier wird die entsprechende syntaktische Struktur berücksichtigt und die Metavariable in Abhängigkeit von

ihrem Typ ausgewertet. Dabei kommt in diesem Beispiel der Variable `t` eine besondere Bedeutung zu. Hier gilt wie immer, dass ein beliebiger Typ in einer Deklaration akzeptiert wird, wenn `t` gleich Null ist. Ansonsten werden nur Deklarationen mit passendem Typ akzeptiert. Die besondere Bedeutung hierbei liegt darin, dass sich auf diese Art und Weise Schleifen auf ungewöhnlicher Typbasis erkennen lassen. Die MISRA Regeln verbieten die Benutzung von Gleitkommatypen als Schleifenzähler. Durch entsprechendes Suchen und Vergleichen lassen sich so alle Verletzungen dieser Regel erkennen.

Der Metadatentyp `stmt` wird in diesem letzten Beispiel exemplarisch genutzt. Ihm zugehörige Metavariablen müssen immer als einziges Element eines *expression-statement* genutzt werden, da *statements* nicht Element eines Ausdrucks sein können. Weiterhin gilt ebenso, wie bei *expressions*, dass der Vergleich rekursiv, d.h. anhand des Inhalts der Metavariable, fortgesetzt wird.

6.3. Instantiierungen auf Basis von QSM

Der Prozess, der QSM in bereits existierenden Quelltext integriert, wird als Instantiierung bezeichnet. Um neuen Quelltext instantiieren zu können, müssen eine Reihe von Voraussetzungen erfüllt sein. Für die Instantiierung von Quelltext gibt es spezielle Metafunktionen, von denen ein Auszug in Abbildung A.4 zu sehen ist.

Dieser Abschnitt erläutert zunächst die verschieden Varianten von Quelltextinstantiierung und geht dann auf die Voraussetzungen ein, die erfüllt sein müssen, damit eine Instantiierung durchgeführt werden kann. Danach werden die einzelnen Teilschritte mit ihrer Sequenz beschrieben, die für die Instantiierung notwendig sind. Dies geschieht unabhängig von der Implementierung eines Compilers, indem nur die prinzipiell dafür notwendigen Mechanismen erklärt werden. Am Ende dieses Abschnitts wird das Auflösen von Symbolen erläutert und daraus sich ergebende mögliche Fehlersituationen besprochen.

6.3.1. Voraussetzungen zur Instantiierung

Damit Quelltext instantiiert werden kann, muss zunächst einmal feststehen, an welcher Position er platziert werden soll. Die Zielposition kann dabei nicht nach Belieben gewählt werden, sondern unterliegt aufgrund der syntaktischen Struktur der Sprache C gewissen Einschränkungen. Somit kann beispielsweise ein Ausdruck nur an einer Stelle eingefügt werden, an der ein Ausdruck stehen darf. Das Gleiche gilt für Anweisungen.

Weiterhin muss unterschieden werden zwischen Instantiierungen, die an einer Stelle erfolgen, wo zuvor kein Quelltext gestanden war und Instantiierungen, die existierenden Quelltext ersetzen. Stellt der einzufügende Quelltext eine Erweiterung dar, so ist es unter Umständen notwendig Änderungen im Parsebaum vorzunehmen, bevor die Instantiierung erfolgen kann. Ein Beispiel hierfür ist eine Anweisung die in einem Ast einer If-Clause angehängt werden soll. Besteht dieser Ast nicht aus einem *Compound-Statement*, so muss ein solches zunächst erzeugt werden und der alte Ast darin untergebracht werden. Diese Refaktorisierungsarbeit ist von einem MetaC Compiler automatisch vorzunehmen. Denn erst danach kann die neue Anweisung angehängt werden. Textuell gesprochen entspricht die Refaktorisierung dem Hinzufügen von geschweiften Klammern um den entsprechenden Teilast der Klausel.

Der einzufügende Quelltext kann aus einem Strukturmuster abgeleitet werden. Ein direktes Einfügen des Quelltextes durch Kopieren des Musters an den Zielort ist nicht mög-

lich, da sichergestellt werden muss, dass alle Symbole korrekt aufgelöst werden können. Außerdem darf es durch die Veränderung des Parsebaums nicht zu einer Verletzung der Semantik kommen, wie sie beispielsweise durch mehrfache Verwendung des selben Namens zustandekommen würde.

6.3.2. Symbolauflösung während der Instanziierung

Die Instantiierung eines Quelltext-Strukturmusters wird in folgenden Phasen unterteilt:

1. **Typisierung/typing:** Symbole der Metaobjekte mit Typ **`type`** werden im gegenwärtigen Kontext aufgelöst.

2. **Benennung/naming:** Bezeichner werden, sofern möglich, im Gültigkeitsbereich der aufrufenden Funktion gegen Metaobjekte vom Typ **`ident`** aufgelöst und ihr Name wird auf den Wert des Metaobjekts gesetzt.

3. **Definition/defining:** Deklarationen im neu instantiierten Code werden in die passenden Symboltabellen eingetragen und auf Kollisionsfreiheit überprüft. Außerdem werden die zugehörigen Initialisierer auf semantische Konsistenz geprüft.

4. **Auflösung/resolving:** Noch ungebundene Bezeichner werden gegen Metaobjekte der verbleibenden Metadatentypen aufgelöst. Diese sind: **`zed`**, **`expr`**, **`func`**, **`real`**, **`strg`**, **`stmt`** und **`symb`**.

5. **Bindung/binding:** Verbleibende ungebundene Bezeichner in Ausdrücken werden im Gütligkeitsbereich der Instantiierung an entsprechende Deklarationen gebunden.

6. **Überprüfung/verification:** Abschließende semantische Überprüfung des instantiierten Parsebaums inklusive der Vaterknoten der Instantiierung von Ausdrücken.

In der ersten Phase werden alle Bezeichner, die im Kontext eines QSM als Metaobjekte vom Metadatentyp **`type`** deklariert wurden, durch den Inhalt der Metaobjekte ersetzt. Dabei kann eine Implementierung die Symbole der Metaobjekte entweder zu den entsprechenden Basistypen von C auflösen oder durch geeignete Symbole der Typalias Definitionen. Im Falle, dass Typdefinitionen Verwendung finden, muss sichergestellt werden, dass die zugehörigen Deklarationen im aktuellen Sichtbarkeitsbereich liegen. Das bedeutet, dass die Definition des Typalias ihrer Verwendung vorangehen muss und dass sie durch Deklarationen in einem inneren Namensraum mit einem gleichnamigen Bezeichner nicht verdeckt werden darf.

Beim Verwendung des vollständig aufgelösten Datentyps stellen sich diese Probleme nicht. Sogar eine Verwendung von unvollständigen Strukturdeklarationen ist möglich, da es explizit im Standard erlaubt ist, die zugehörigen Deklarationen zur Vervollständigung des Datentyps der Verwendung nachzustellen. Dadurch ist dieses Verfahren deutlich einfacher zu implementieren. Allerdings bietet die Variante, passende Typdefinitionen in der Deklaration zu verwenden, den Vorteil einer besseren Lesbarkeit des resultierenden Quelltextes.

In der zweiten Phase werden alle verbleibenden Bezeichner zunächst gegen Metadatenobjekte vom Typ **`ident`** im aktuellen Gültigkeitsbereich verglichen. Finden sich Metaobjekte mit gleichem Namen, so wird der Name des Bezeichners durch den Inhalt der Metavariable ersetzt. Dadurch wird es möglich, Quelltext-Strukturmuster entsprechend

```
meta void inst_failure(void)
{
        expr e0 = (expr) ( (*(a[i]++))() );
}
```

Abbildung 6.8.: Beispiel eines TSP mit erweiterter Typanforderung bei der Instantiierung

ihrem Verwendungskontext mit unterschiedlichen Bezeichnern für Variablendeklarationen zu instantiieren und somit Kollisionsfreiheit von Deklarationen sicherzustellen oder die Bezeichner an konkrete Deklarationen zu binden.

Die dritte Phase führt eine semantische Überprüfung lokaler Variablendeklarationen durch. Dazu werden alle entsprechenden Einträge in den jeweiligen Symboltabellen vorgenommen und dadurch sichergestellt, dass keine zwei Deklaratoren einen gleichnamigen Bezeichner enthalten. Andere Einschränkungen oder Voraussetzungen, wie die erforderliche Typkompatibilität der Initialisierer werden erst später überprüft.

Die vierte Phase bindet Bezeichner in Ausdrücken an passende Metaobjekte der Metadatentypen **`expr`**, **`func`** und **`symb`**. Wird ein Bezeichner zu einer Variable vom Typ **`expr`** aufgelöst, so wird das ESP, das sich dahinter verbirgt, instantiiert und eine Rekursion findet statt. Dies ist für Metavariablen der Typen **`func`** und **`symb`** nicht notwendig, da diese direkt gebunden werden können. Hierbei ist lediglich sicherzustellen, dass sich die zugehörigen Objekte im Sichtbarkeitsbereich des Instantiierungsortes befinden und nicht Null sind.

Alle verbleibenden Bezeichner werden in der fünften Phase an Deklarationen im Gültigkeitsbereich des Instantiierungsortes gebunden. Danach müssen alle Bezeichner an Deklarationen gebunden sein, ansonsten ist die Instantiierung unvollständig und fehlgeschlagen.

In der letzten Phase wird eine semantische Überprüfung des neuen Parsebaums durchgeführt. Als Startpunkt kann bei instantiierten Anweisungen, die Anweisung selbst verwendet werden. Die Überprüfung wird durch abwärtstraversieren des Parsebaums fortgesetzt. Bei der Instantiierung von Ausdrücken reicht eine Abwärtstraversierende Überprüfung nicht aus, da der Typ des Ausdrucks auch semantische Verletzungen im Vaterknoten verursachen kann.

Als Beispiel betrachte man hierfür das Codefragment in Abbildung 6.8. Bei der Instantiierung des Struktur-Musters, auf das die Metavariable **`e0`** verweist, müssen die Symbole **`a`** und **`i`** aufgelöst werden. Der Bezeichner **`i`** wird in dem ESP zum indizierten Zugriff auf ein Feld verwendet und muss deshalb nur zu einem skalaren Datentyp aufgelöst werden.

Anders verhält es sich jedoch mit dem Bezeichner **`a`**. Bei diesem zeigt sich nach dem ersten Schritt aufwärts im Parsebaum, dass er einen Feld- oder einen Zeigertyp haben muss. Der nächste Schritt führt zu dem Postinkrementoperator (**`++`**): Dieser erfordert, dass die Elemente von **`a`** einen skalaren Typ haben. Es folgt der der unäre **`*`** Operator, der auf den Inhalt einer Zeigervariable zugreift. Danach folgt die leere Argumentliste, die einen Funktionsaufruf durchführt. Die Elemente des Arrays müssen also Zeiger auf Funktionen beinhalten. Allerdings ist aus dem Strukturmuster nicht erkennbar, welchen Rückgabewert die Funktion haben muss. Am Instantiierungsort ist somit eine Typprüfung durchzuführen, die entsprechend der hier exemplarisch gezeigten Analyse abläuft. Dabei muss der Baum so lange nach oben traversiert werden, bis entweder ein Fehler auftritt oder nachgewiesen werden kann, dass die Instantiierung fehlerfrei ist.

6.3.3. Fehlersituationen während der Instantiierung

Die Auflösung eines Symbols schlägt während der Instantiierung eines QSM fehl, wenn sich eine der folgenden Situationen ergibt:

1. Das Auflösen eines Bezeichners zu einem Symbol ist in keinem der passenden Namensräume und zuständigen Gültigkeitsbereiche möglich.
2. Das Auflösen eines Typalias kann durch Instantiierung in einem Gülitigkeitsbereich Fehlschlagen, der eine Typalias Deklaration verdeckt.
3. Nach der Auflösung eines Bezeichners kommt es zu einem Typfehler bei der Symboleinsetzung durch:
   a) inkompatible Typen innerhalb eines Ausdrucks oder Initialisierers.
   b) ein nicht definiertes Element in einer Datenstruktur.

Alle diese Fehler sind auf menschliches Versagen zurückzuführen, deren Ursache in einem Programmierfehler liegt. Solche Fehler können nicht automatisch behoben werden, da die unterschiedliche Auflösung von Symbolnamen zu veränderter Semantik und somit abweichendem Programmablauf führt. Mögliche Fehlerbehandlungsstrategien, die über einen einfachen Abbruch mit Fehlermeldung hinausgehen, sind somit auf die Interaktion mit dem Menschen angewiesen. Denkbare Ansätze für eine erweiterte Fehlerbehandlung sind:

- Der Versuch, gegen ähnlich geschriebene Symbolnamen aufzulösen.
- Die Suche nach geeigneten Symbolen mit passendem Datentyp, die die Anforderungen für die Instantiierung erfüllen.

Mögliche Treffer eines solchen Alternativverfahrens müssten dann als Lösungsvorschläge für das Problem dem Nutzer offeriert werden. Der Einsatz solcher Algorithmen wird in kommerziellen Werkzeugen zum Teil unterstützt, wurde aber für die prototypische Testimplementierung, die im Rahmen dieser Arbeit erstellt wurde, aufgrund des Aufwandes nicht berücksichtigt.

6.3.4. Instantiierungsmechanismen ohne QSM

Für die Instantiierung von Variablendeklarationen sind QSM nicht unbedingt notwendig. Hierbei muss bekannt sein, mit welchem Sichtbarkeitsbereich (*scope*), Datentyp (*type*) und Namen (*ident*) die Deklaration erfolgen soll. Darüber hinaus kann die Variable noch durch einen oder mehrere Ausdrücke initialisiert werden. Sollten Ausdrücke zur Initialisierung verwendet werden, so müssen diese selbstverständlich als Strukturmuster dem Instantiierungsaufruf mit übergeben werden.

Um eine neue Variable mit Initialisierer zu definieren, gibt es eine Metafunktion, die im MetaC Compiler implementiert sein muss und die oben genannten Informationen als Parameter entgegen nimmt. Sie muss die korrekte Position für das Einfügen der Deklaration ermitteln und die entsprechenden Modifikationen im Quelltext durchführen. Dabei kann ein Initialisierer als eine Reihe von Ausdrucksstrukturen übergeben werden. Da diese Parameter als Ellipse (`...`) in der Funktion deklariert sind, kann eine beliebige Anzahl Argumente übergeben werden.

Wird kein QSM eines Ausdrucks übergeben, so erhält die Deklaration keinen Initialisierer. Wird ein Ausdruck als Argument für die Intialisierung übergeben, der nicht ein QSM referenziert, sondern einen Ausdruck an einer beliebigen Stelle im Programm, so wird dieser trotzdem wie ein Strukturmuster behandelt. Das bedeutet, dass die darin verwendeten Symbole losgelöst von ihrem Gültigkeitsbereich betrachtet werden. Während der Instantiierung werden sie dann an passende Deklarationen entsprechend ihrer Bezeichner und der zuvor erläuterten Semantik wie ein normales QSM gebunden.

Aus dem verwendeten Initialisierer ergeben sich auch Restriktionen für den Ort der Instantiierung. So dürfen, für Deklarationen mit statischer Speicherallokation, nur Initialisierer verwendet werden, die einen konstanten Wert haben oder Stringliterale (siehe 6.7.8§4 in [17]). Funktionsaufrufe sind also beispielsweise verboten. Weiterhin gilt, dass sich alle referenzierten Variablen sich im Sichtbarkeitsbereich der aktuellen Deklaration befinden müssen. Die Tatsache, dass für die Instantiierung von Deklarationen keine konkrete Position angegeben werden kann, sondern nur der Sichtbarkeitsbereich angegeben wird, in dem die Deklaration durchgeführt werden soll, erweist sich hierbei als Vorteil. Dadurch ist es möglich, dass der MetaC Compiler selbständig eine geeignete Position bestimmt und Abhängigkeiten automatisch auflöst bzw. bei nicht auflösbaren zyklischen Abhängigkeiten eine Fehlermeldung generiert.

7. Implementierung des MetaC Compilers

Die Implementierung eines MetaC Compilers kann grundsätzlich auf zwei verschiedene Arten erfolgen. Zum einen als voll integrierte Variante, die Maschinencode für eine bestimmte Zielarchitektur generiert. Zum anderen als ein Programm, das C Quelltext ausgibt, der dann durch einen normalen C Compiler für eine beliebige Zielarchitektur verarbeitet werden kann.

Beide Varianten haben ihre individuellen Vorteile. Die vollständige Integration vereinfacht die Anwendung und bietet mehr Optimierungspotential. Dadurch lassen sich sowohl die Kompilierzeit, als auch die Ausführungszeit eines Programms reduzieren. Auf der anderen Seite bietet eine Implementierung ohne feste Bindung an eine Zielarchitektur den Vorteil für beliebige Zielarchitekturen verwendbar zu sein. Der Integrationsaufwand in den Buildprozess ist dabei etwas höher. Dafür ergeben sich aber insbesondere bei Crossentwicklungsumgebungen mit zusätzlichem, hostbasiertem Simulationstarget, konsistente und einfacher zu wartende Entwicklungspfade als bei einem vollintegrierten Compiler.

7.1. Konzept

Die Implementierung, die im Rahmen dieser Arbeit durchgeführt wurde, basiert auf dem Konzept der losen Kopplung mit einem normalen C Compiler. Das bedeutet, dass der Compiler MetaC Quelltext einliest und C Quelltext ausgibt. Dies ermöglichte die Durchführung von Tests mit verschiedenen Architekturen und Betriebssystemen und demonstriert gleichzeitig die höhere Flexibilität dieser Implementierungsvariante. Diese Software kann über [56] bezogen werden.

Eingabeseitig wird dabei Quelltext erwartet, der den Präprozessor bereits durchlaufen hat. Dazu wird ein voreingestellter Präprozessor automatisch aufgerufen, der die Eingabedatei verarbeitet und das Ergebnis in eine temporäre Datei umleitet, die dann als Eingabequelle genutzt wird. Dabei ist es möglich, den Präprozessor zu wechseln oder ihm zusätzliche Makros mitzugeben oder voreingestellte Definitionen abzuschalten, wie es ein normaler Compiler ermöglicht. Diese Funktionalität ist insbesondere bei der Verwendung von betriebssystemspezifischen Headerdateien notwendig, da diese häufig Makros nutzen, um plattform- und compilerspezifische Varianten zu implementieren.

7.2. Evolution von Methodik und Implementierung

Der Lösungsansatz für das Problem der Rekonfiguration und Refaktorisierung von Quelltext war Anfangs nicht der Entwurf einer Programmiersprache oder gar einer Metaprogrammiersprache. Die ersten pragmatischen Versuche basierten auf Ansätzen, die entweder einen Makroprozessor benutzten oder dessen Funktionalität mit Spezialerweiterungen

nachbildeten. Die zahlreichen Probleme dieser Verfahren sollen hier nicht im Detail erläutert werden. Bei diesen Experimenten stellte sich allerdings schnell heraus, dass es ein fundamentales Problem mit einer Methodik gibt, die nur auf einer lexikalischen Analyse basiert.

Insbesondere bei C Quelltext werden viele Schlüsselwörter und Interpunktionszeichen in mehreren grammatikalischen Strukturen verwendet. Daraus folgt, dass aus einzelnen lexikalischen Symbolen nicht erkennbar ist, zu welcher grammatikalischen Struktur eine Tokensequenz gehört. Ist das ausgesprochene Ziel, Modifikationen nicht mit der Genauigkeit einer Schrotflinte, sondern eher mit der eines Skalpells durchzuführen, so muss dieser Ansatz verworfen werden. Entsprechend ist eine Methodik gefragt, die die Grammatik von C berücksichtigt.

Da der Sprachumfang von C recht groß ist, würde es Sinn machen nur ein relevantes Subset der Grammatik zu betrachten. Ein entsprechender Ansatz würde außerdem einem ingenieursmässigen Vorgehen entsprechen, bei dem der Aufwand nach Möglichkeit auf ein Minimum reduziert wird. Folglich basierten die ersten Implementierungen auf einer einfachen syntaktischen Analyse mit Hilfe eines *infinite look ahead* Parsers. Bei der Anwendung eines solchen Parsers ist kein detailliertes Hintergrundwissen und Verständnis der Sprachentheorie notwendig.

Allerdings haben entsprechende Grammatikdefinitionen den Nachteil, dass Probleme wie Konflikte und Widersprüche erst zur Laufzeit erkennbar werden. Außerdem ist eine Behebung solcher Situationen extrem schwierig und wird immer komplexer, je umfangreicher die Grammatik ist. Entsprechend war es nur eine Frage der Zeit, bis mit fortschreitender Implementierung eine Abkehr von diesem Konzept notwendig wurde.

Der Umstieg auf den LALR(1) Parser Yacc brachte dann den Durchbruch bzgl. der Erweiterbarkeit und Wartbarkeit der Grammatik. Allerdings war dafür zunächst einmal eine saubere Analyse der Grammatik von C und Definition der durchzuführenden Erweiterungen notwendig. Dabei zeigte sich, dass in dem Vorläuferkonzept noch viele Schwachstellen versteckt waren, die früher oder später eine Unmenge an Aufmerksamkeit verlangt hätten.

Ein anderes wesentliches Problem lag in der internen Verarbeitungsstruktur begraben. Wenn nicht klar ist, welche Informationen und Assoziationen notwendig sind, ist ein generischer Ansatz am besten zielführend. Deshalb wurde zunächst einmal möglichst viel Information an die Knoten der internen Datenstruktur gebunden und viele Assoziationen geknüpft. Für die Analysen zu Beginn des Projekts war dies der denkbar beste Ansatz, weil einfach weitere Informationen hinzugefügt werden konnten und alle Zugriffsverfahren einfach wiederverwendet werden konnten.

Allerdings wurde dieses Konzept mit fortschreitender Implementierung zunehemend zu einem Hindernis. Modifikationen im Baum wurden immer komplexer und es wurde zunehmend schwieriger sicherzustellen, dass bei einer solchen Veränderung im Baum keine Fehler entstehen. Folglich war eine Abkehr von diesem generischen Konzept unvermeidbar. Das neue Konzept basierte dann auf den in Abschnitt 7.3.1 beschriebenen Verfahren, dass zu wesentlichen Teilen den Standardverfahren des Compilerbaus entspricht.

Der letzte wesentliche evolutionäre Schritt, war die Abkehr vom Konzept der offenen Compilerarchitektur hin zu einem echten Metaprogramm Ansatz. Dadurch wurden häufig notwendige Codestrukturen gekapselt und einfach wiederverwendbar. Außerdem hatte dies zur Konsequenz, dass das Gesamtkonzept klarer und einfacher zu verstehen ist. Insbesondere wurden die tiefen Abhängigkeiten zur Semantik von C auf eine abstraktere Ebene gehoben. Durch entsprechende Sprachmittel und Implementierungen im Compi-

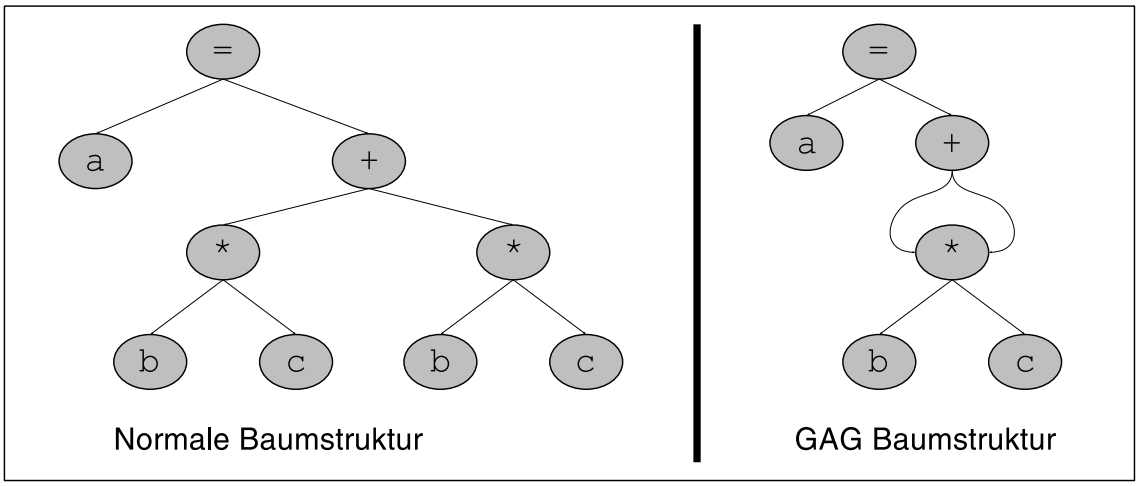


Abbildung 7.1.: Gerichteter Azyklischer Graph (GAG)

ler kann so ein großer Teil der Komplexität vor dem Anwender verborgen werden. Dies reduziert das notwendige Grundwissen zur Anwendung dieser Methodik.

Zusammenfassend sei festgehalten, dass im Rahmen der Konzeptionierung und Implementierung eine Reihe von wesentlichen Entscheidungen zu fällen waren. Schließlich war klar, dass eine Spracherweiterung von C zu MetaC unumgänglich ist, um die avisierten Ziele zu erreichen. Eine pragmatische Methode ist leider ungenügend, um Quelltext modifizieren zu können, wenn die Forderung besteht, dass der Anwender eine Rückkopplung zu den durchgeführten Modifikationen benötigt.

7.3. Besondere Eigenschaften der Implementierung

Die Details der Konzepte und des Designs der prototypischen Implementierung gehen über den Rahmen dieser Arbeit hinaus. Trotzdem sollen einige Besonderheiten nicht unerwähnt bleiben. Neben den hier erwähnten Verfahren, Ideen und Konzepten sind viele Standardverfahren zum Einsatz gekommen, die bereits in vielen Veröffentlichungen und Büchern dokumentiert und eingehend erläutert wurden.

7.3.1. GAG Struktur

Die Verwendung von *Gerichteten azyklischen Graphen* (GAG) ist eine Standardtechnik des Compilerbaus, die auch in [19] empfohlen und erläutert wird. Bei diesem Verfahren werden Teiläste des Parsebaums, die identisch sind, durch eine gemeinsame Datenstruktur repräsentiert (siehe Abbildung 7.1. Dies hat eine Reihe von Vorteilen:

- geringer Speicherbedarf und somit größere Cachelokalität
- Die Optimierung und Codegenerierung muss für gemeinsame Teiläste nur einmal ausgeführt werden.
- einfachere Ermittlung von wiederverwendbaren Zwischenergebnissen und dadurch verbesserte Optimierbarkeit des Codes

Bei der Implementierung eines MetaC Compilers kommt es bei der Verwendung von GAG als interne Darstellung aber zu einem erheblichen Problem: die Baumstrukturen müssen modifizierbar sein, damit Erweiterungen und Änderungen im Quelltext vorgenommen werden können, und der Baum sollte einfach traversierbar sein mit klaren Assoziationen zwischen den einzelnen Knoten. Das Durchsuchen eines GAG wirft aber beim aufwärts traversieren von einem Knoten zu seinem Vaterknoten eine Frage auf: da er von mehreren Knoten referenziert werden könnte, ist es nicht klar, mit welchem Knoten er im gegenwärtigen Kontext assoziiert ist.

Dieses Problem lässt sich aber durch entsprechende Referenzen in den GAG lösen: Die Referenz zu einem Ast im GAG muss dazu den Pfad zum referenzierten Knoten mit speichern. Der Pfad im GAG lässt sich durch alle Elternknoten von Knoten, die mehrere Eltern haben können, eindeutig darstellen. Können alle Knotentypen im GAG mehrfach benutzt werden, so bedeutet das für die Bildung von Referenzen einen enormen Overhead, da in jedem Fall der gesamte Pfad gespeichert werden muss. Durch Modifikationen in dem GAG kann es nach der Referenzbilidung noch zusätzlich zur Aliasbildung kommen, die eine Implementierung noch komplizierter gestaltet.

Dieser Aufwand ist im Verhältnis zum Nutzen nicht vertretbar. Deshalb baut das Frontend der Implementierung nur einen eingeschränkten GAG auf. Dabei werden nur Bezeichner und Konstanten zur Aliasbildung herangezogen. Dies hat zur Konsequenz, dass die Pfadbildung nur einen einzigen Knoten benötigt, der direkt über dem mehrfach verwendeten Knoten liegt. Dadurch kommen alle anderen Knotentypen vollkommen ohne Pfadbildung aus.

Dies ermöglicht es, bei der Referenzbildung mit einer konstanten maximalen Pfadlänge von einer zusätzlichen Referenz auszukommen. Würden Ausdrucksknotentypen mit eigenen Kindern zur Vereinfachung der Baumstruktur herangezogen werden, so würde sich die maximale Pfadlänge von eins auf unendlich verlängern, da durch die Rekursion jeder beliebiger Knotentyp das Kind eines jeden Knotentyps sein kann, der wiederum Kinder hat. Dies hätte zur Konsequenz, dass die Komplexität der Algorithmen zur Referenzbildung und ihr Speicherbedarf von O(1) auf O(n) steigt.

Auf der anderen Seite vereinfacht die einfache Referenzbildung der Bezeichner und Konstanten diverse Algorithmen, da dadurch sämtliche Matadaten einer Deklaration direkt mit dem Bezeichner gespeichert werden können. Somit wird ein Lookup über eine Symboltabelle und das Kopieren und Konsistenthalten der Metadaten überflüssig. Dies erhöht die Ausführungsgeschwindigkeit und vereinfacht die Implementierung deutlich.

7.3.2. One-Way dispatching Visitor

Das Besuchermuster [34] ist gut geeignet für die Implementierung von Interpretern und Codegeneratoren. Darüber hinaus gibt es noch eine spezialisierte Variante: das Interpretermuster. Das Besuchermuster definiert einen abstrakten Besucher, der abstrakte Knotentypen besucht und so eine Knotenhierarchie von unterschiedlichen Knotentypen traversiert und dabei seine Arbeit verrichtet. Die Spezialform, das *Interpretermuster*, nimmt einen Teil der Verallgemeinerung zurück und konkretisiert den Besucher auf genau eine Klasse.

Dies hat den Vorteil, dass beim Traversieren von einem Knoten zum nächsten nicht zwei, sondern nur ein virtueller Methodenaufruf durchgeführt werden muss. Dies reduziert die Ausführungszeit, macht die Implementierung aber weniger flexibel, da nun vom Interpretierer keine neue Klasse mehr abgeleitet werden kann, um beispielsweise eine Variante zu realisieren.

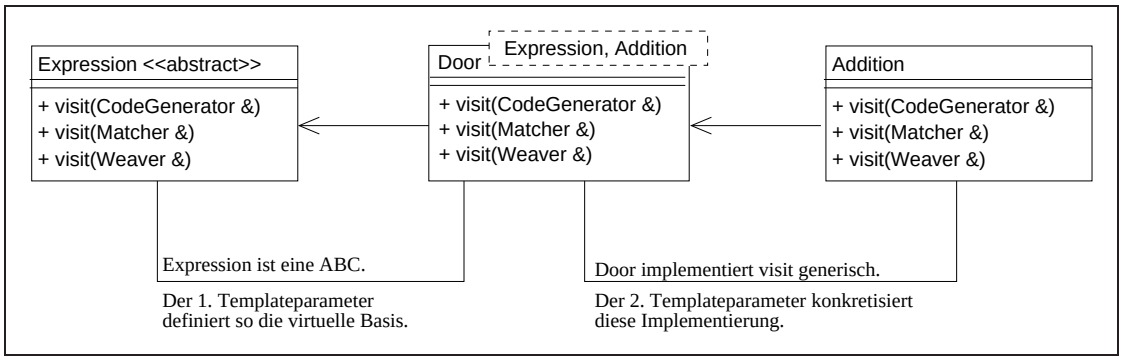


Abbildung 7.2.: Muster für einfach-virtuelle Besucher

Für die Implementierung des MetaC Compilers wurde eine spezielle Variante entworfen, die auf einem Metaprogramm basiert, das das Besuchermuster zur Compilezeit in ein Interpretermuster überführt. Dabei wird eine Seite (die des Besuchers) durch statischen Polymorphismus beschrieben und wird somit schon zur Kompilierzeit gebunden. Dadurch entfällt ein virtueller Methodenaufruf zur Laufzeit und damit verkürzt sich die Ausführungszeit.

Dreh- und Angelpunkt dafür ist die Templateklasse *Door*, die als Zwischenebene in der Ableitungshierarchie eingezogen wird (siehe Abbildung 7.2). Die Klasse hat zwei Templateparameter: die abstrakte Basisklasse (ABC), von der sie abgeleitet werden soll und die Klasse, die von ihr ableiten wird. Sie entspricht also dem Konzept des *Couriously Recurring Template Pattern* (CRTP, siehe [76]). Durch die beiden Parameter ist es möglich, für die verschiedenen Besucher Methoden zu implementieren, die jene der zugehörigen Basisklassen überschreiben. Durch Überschreiben der Methoden der Basisklasse werden die Methoden der Besucher nicht-virtuell gebunden und können dadurch schneller aufgerufen werden.

Neben dem positiven Effekt, dass dieses Konzept die Ausführungsgeschwindigkeit verbessert, führt es außerdem zu einer enormen Einsparung an Codezeilen. Dies vereinfacht nicht nur die Implementierung, sondern gestaltet sie auch übersichtlicher. Gleichzeitig bleiben spezifische Erweiterungen mit Hilfe von Ableitungen und Spezialisierungen möglich, ohne den Quelltext von *Door* dafür ändern zu müssen. Die Templateklasse hat einen Umfang von ca. 75 Zeilen Code, die in 80 Klassen verwendet wird. Eine direkte Implementierung ohne CRTP würde über 5000 Zeilen objektorientierten Quelltext, ohne Kommentare ausmachen. Eine äquivalente Implementierung in C is noch erheblich umfangreicher, da hier das Verzweigen bei den virtuellen Methodenaufrufen nachzubilden ist. Dafür sind aufwändige **`switch-case`** Strukturen notwendig, die alle möglichen Ableitungen der Vererbungshierarchie berücksichtigen.

7.3.3. Implementierung von `#include` als Metafunktion

Die Implementierung einer **`include`** Anweisung als eine MetaC Compiler Funktion wirft einige Fragen auf und hat mehrere Hürden bei der Realisierung. Der Grund hierfür liegt in der Art und Weise, wie der Präprozessor von C Makros verarbeitet. Dabei können Abhängigkeiten entstehen, die von außen nicht sichtbar sind. Ebenso gibt es keine offiziellen Einschränkungen, was in einer Headerdatei definiert sein darf und in welcher Form `#include` Anweisungen ineinander verschachtelt werden dürfen.

Trotzdem haben sich einige Regeln etabliert, die als bewährte und empfohlene Verfahrensweisen gesehen werden und für die meisten sauber geschriebenen Implementierungen gelten. Dazu gehören beispielsweise:

- Idempotenz[1)] der Headerdateien
- nur Deklarationen, die als extern qualifiziert sind, und Typdefinitionen (also keine Funktionen und keine Deklarationen die Objekte anlegen)
- kein Benutzersichtbaren Abhängigkeiten zwischen den Headerdateien
- nur syntaktisch abgeschlossen Präprozessorstrukturen (z.B. muss auf jedes **`#if`** oder **`#ifdef`** ein abschließendes **`#endif`** folgen)

Unter diesen Voraussetzungen kann ein nachträgliches Einfügen einer Headerdatei in ein Modul gelingen, ohne dass eine tiefe Integration des Präprozessors mit dem MetaC Compiler notwendig ist. Eine solche Integration würde zwar diese Voraussetzungen nicht erzwingen, doch ergeben sich daraus in der Anwendung ansonsten keine Vorteile gegenüber einem reinen MetaC basierten Ansatz.

Weiterhin würde ein Auflösen dieser Voraussetzungen Freiheitsgrade in das Konzept bringen, die nicht mehr ohne weiteres aufzulösen sind, da dann einzelne Deklarationen in C Dateigrenzen überschreiten können. Diese freizügige Gestaltungsweise von Quelltext lässt sich nicht durch eine Grammatik ausdrücken, die analysierbar ist und dabei noch mit der C Semantik in Einklang gebracht werden kann. Der wesentliche Hinderungsgrund ist, dass die grammatikalischen Strukturen von C keinerlei Assoziation mit den Dateigrenzen aufweisen. Diese werden normalerweise wie Weißzeichen behandelt, also ignoriert. Deshalb wird dieser Ansatz hier nicht weiter betrachtet.

Trotz der zuvor genannten Voraussetzungen können noch Problemsituationen entstehen die aufzulösen sind. Zu der Klasse der lösbaren Situationen zählen:

- Mehrfache Definition eines Typalias
- Mehrfache Deklartion einer Variable oder Funktion

Beide Situationen können aufgelöst werden und durch eine einzelne Definition bzw. Deklaration ersetzt werden, wenn der zugehörige Datentyp identisch ist. Ist dies nicht der Fall, so liegt ein Benutzerfehler im Modul, im Metaprogramm oder in der Headerdatei vor. In diesem Fall muss der konkrete Konflikt an den Benutzer gemeldet werden, damit dieser diesen auflösen kann.

Bei der nachträglichen Inklusion einer Headerdatei in ein *translation-unit* stellt sich die Frage, wo die hinzuzufügenden Deklarationen zu platzieren sind. Ein einfaches Anhängen an das Quelltextende ist die einfachste Lösung. Allerdings hat dies zur Folge, dass die neu hinzugekommenen Typdefinitionen und Variablen so nicht in den Funktionen verwendet werden können, da diese dann zur Kompilierzeit des erzeugten Quelltextes nicht im Sichtbarkeitsbereich liegen.

Ein besseres Verfahren ist, die neuen Quelltextstrukturen an einem Ort einzufügen, der möglichst weit am Anfang des Moduls liegt. Dabei sind folgende Einschränkungen zu berücksichtigen:

[1)] Als Idempotenz wird das den Inhalt kapselnde Makro einer Headerdatei bezeichnet, das sicherstellt, dass die Datei nur einmal inkludiert wird.

- Die verwendeten Typaliase müssen sichtbar sein oder aufgelöst werden.
- Die verwendeten Initialisierer müssen sichtbar sein.

Werden alle verwendeten Typaliase beim Einfügen in den Quelltext aufgelöst, so können sämtliche Deklarationen und Definitionen ohne Initialisierer ganz am Anfang eingefügt werden. Dies hat lediglich den Nachteil, dass die Abhängigkeiten zu den ursprünglich verwendeten Typdefinitionen nicht mehr erkennbar sind. Dies lässt sich durch eine entsprechende Kommentierung der Quelltextzeilen dokumentieren und auflösen, die allerding in der Implementierung noch nicht realisiert ist.

Deklarationen mit Initialisierer müssen, bei der Verwendung von Variablen oder Funktionen im Initialisierer, an einer Position instantiiert werden, an der sämtliche Deklarationen sichtbar sind. Werden alle diese Voraussetzungen erfüllt, so steht einer erfolgreichen, nachträglichen Inklusion von Headerdateien durch Metafunktionen nichts mehr im Weg.

Teil III.

Applikationen von MetaC

8. Metaprogramme für eingebettete Software

Die bisher durchgeführten Definitionen, Erläuterungen und Beschreibungen liefern lediglich die Infrastruktur für die Lösung von komplizierten Problemen bei der Entwicklung von eingebetteter Echtzeitsoftware. In diesem Kapitel werden in MetaC geschriebene Metaprogramme besprochen, die die zuvor eingeführte Infrastruktur nutzen und dadurch den Entwurf von Software für eingebettete Systeme mit Echtzeitanforderungen unterstützen oder vereinfachen.

8.1. Überprüfung von Randbedingungen und Regeln

Für den Entwurf und die Implementierung von Software für eingebettete Systeme mit besonderen Anforderungen, wie beispielsweise in der Luftfahrt- und Automobilindustrie, gibt es applikationsspezifische Richtlinien die eingehalten werden müssen. Die Einhaltung dieser Regeln ist erforderlich, um eine Verifikation und Validierung der Systeme durchführen zu können. Darüber hinaus sollen sie die Wartbarkeit von Quelltexten verbessern, häufig gemachte Fehler vermeiden und somit insgesamt die Kosten senken. Ein Beispiel eines solchen Regelsatzes sind die MISRA Regeln für den Entwurf von Software mit der Programmiersprache C der Automobilindustrie (siehe [15]). Im Folgenden wird untersucht, welche dieser Regeln mit Hilfe von MetaC automatisch überprüfen werden können.

8.1.1. Analyse anhand der syntaktischen Struktur

Ein Großteil des MISRA C Regelsatzes bezieht sich auf die syntaktische Struktur von Quelltexten. Darüber hinaus werden auch Anforderungen an die Programmierumgebung, die verwendeten Bibliotheken und die Art und Weise der Kommentierung von Quelltext gestellt. Die in den vorangegangenen Kapiteln vorgestellte Spracherweiterung MetaC erlaubt eine syntaxbasierte Analyse von Quelltexten. Die Regeln, die hier untersucht werden, lassen sich auf eine Analyse der grammatikalischen Struktur zurückführen. Des Weiteren ist eine Regelüberprüfung mit Hilfe von MetaC auf der Basis von Datenfluss- und Kontrollflussanalyse denkbar. Darüber hinaus gibt es Regeln die Systemumgebung betreffend. Einschränkungen und Randbedingungen, die aus solchen Vorschriften entstehen können mit Metaprogrammen nur eingeschränkt oder auch gar nicht verifiziert werden. Es lässt sich also zusammenfassen: Mit MetaC ist nur C Quelltext überprüfbar; alle anderen Randbedingungen liegen nicht im Einflussbereich von MetaC.

Regeln, die festlegen, wie Quelltext formatiert werden soll, werden auch als Coding Convention oder Style bezeichnet. Zu ihrer Definition kann die grammatikalische Struktur der Sprache referenziert werden oder ein Bezug zur lexikalischen Formatierung her-

```
 1  #include "internal.mh"
    meta void misra59_verify_stmt(stmt s)
    {
       stmt ch[], *c;
 5     zed verify = 0;
       if (is_if(s) || is_for(s) || is_do(s) || is_while(s))
          verify = 1;
       get_children(s,ch);
       c = ch;
10     while (*c) {
          if (verify && !is_compound(*c))
             warn("misra rule 59 violated at line "
                  + (strg)line_of(*c));
          misra59_verify_stmt(*c);
          ++c;
15     }
    }

    meta void misra59(void)
    {
20     zed i;
       func funcs[], *f;
       get_all_functions(funcs);
       f = funcs;
       while (*f) {
25        stmt b;
          b = get_body(*f);
          info("checking function " + name((symb)*f));
          misra59_verify_stmt(b);
          ++f;
30     }
    }
```

Abbildung 8.1.: Metaprogramm zur Überprüfung der MISRA Regel 59

gestellt werden. Grundsätzlich kann man alle diese Richtlinien automatisch überprüfen. Die Infrastruktur von MetaC stellt allerdings nur reflexive Konstrukte zur Verfügung, die eine Analyse der grammatikalischen Struktur erlauben. Eine Erweiterung um zusätzliche Methoden, die Aufschluss über die lexikalische Struktur geben, ist aber denkbar. Neben der Möglichkeit diese speziellen Richtlinien zu überprüfen, würde eine solche Erweiterung keine zusätzlichen Vorteile für die Metaprogrammierung bringen. Deshalb wurde von einer solchen Erweiterung abgesehen.

Die MISRA Regel 59 ist ein Beispiel für eine Vorschrift in der die Formatierung von C-Code reglementiert wird und die sich dafür auf die grammatikalische Struktur des Quelltextes bezieht. Ihre Kurzfassung lautet: „The statements forming the body of an *if, else if, while, do ... while* or *for* statement shall always be enclosed in braces.“

Braces, also die Tokens „**{**“und „**}**“, begrenzen in der Sprache C so genannte *compound-statements*. Dies sind Anweisungen, die eine Liste von Anweisungen und/oder Deklarationen aufnehmen. Somit bedeutet diese MISRA Regel syntaktisch betrachtet,

dass nur *compound-statements* als Kinder von den oben genannten Anweisungen im Parsebaum erscheinen dürfen. Eine Überprüfung dieser Forderung ist mit Hilfe des in Abbildung 8.1 dargestellten Metaprogramms realisierbar. Der Vorteil gegenüber einem speziellen Programm zur Überprüfung der MISRA Regeln besteht darin, dass sich in Metaprogrammen ohne Weiteres Ausnahmen formulieren lassen und nachträgliche Erweiterungen durchgeführt werden können.

Das Metaprogramm zur Verifikation der MISRA Regel 59 wird durch Aufrufen der Metafunktion **`misra59`** gestartet. Diese fragt zunächst einmal ab, welche Funktionen im Quelltext definiert sind (Zeile 22). Die nachfolgende **`while`**-Schleife überprüft den Funktionskörper jeder Funktion mit Hilfe der Metafunktion **`misra59_verify_stmt`**. Diese wiederum verarbeitet die übergebene Anweisung rekursiv und überprüft somit alle anderen Anweisungen die im Funktionskörper enthalten sind. Dabei wird die Variable **`verify`** abhängig von der untersuchten Anweisung gesetzt (Zeile 6, 7), um in der nachfolgenden Schleife die geforderte Bedingung der MISRA Regel überprüfen zu können. Dazu werden alle Kindanweisungen der momentan untersuchten Anweisung erfragt und ebenfalls durch Rekursion mit der selben Metafunktion überprüft (Zeilen 11-14).

Als wesentliche Eigenschaft dieser Implementierung lässt sich die Rekursion herausstellen, welche durch die rekursive Syntax von C bedingt ist. Somit müssen alle Regelprüfalgorithmen, die sich auf rekursive grammatikalische Strukturen von C beziehen, mit Hilfe einer Rekursion implementiert werden. Bei Unabhängigkeit der Daten zwischen den einzelnen Rekursionsebenen, kann die Rekursion auch zu einer Schleife aufgelöst werden.

8.1.2. Analyse anhand semantischer Implikationen

Regeln die den Quelltext betreffen, haben meistens zum Ziel typische Fehlerquellen zu vermeiden. So ist die Verwendung von Gleitkommavariablen zwar häufig notwendig, aber sie bringt einige Risiken mit sich [39]. Beispielsweise ist der direkte Vergleich einer Gleitkommazahl mit einer anderen Zahl eine Operation, die möglich und erlaubt ist, aber zu unerwartetem Laufzeitverhalten führen kann. Hierbei spielen die verwendete Genauigkeit, die Implementierung des Rundungsverhaltens und andere Faktoren eine Rolle. Einer dieser Faktoren ist die optionale Unterstützung der negativen Null in der Systemumgebung. Ein direkter Vergleich der negativen Null mit einer fest eincodierten positiven Null kann so zu einem überraschenderweise negativ ausfallenden Vergleich führen.

Aus diesem Grund verlangt eine Regel der MISRA Richtlinien, dass Gleitkommavariablen nicht als Zähler für Schleifen eingesetzt werden. Diese Regel kann in MetaC durch ein entsprechendes Metaprogramm automatisch überprüft werden, denn die Infrastruktur stellt Methoden bereit, um den Typ eines Ausdrucks zu erfragen. Somit ist es möglich den Ausdruck der Wiederholungsbedingung von Anweisungen, die eine Schleife realisieren, zu erfragen und die verwendeten Typen von vergleichenden Operationen zu überprüfen. Eine detaillierte Diskussion des entsprechenden Metaprogramms wird an dieser Stelle nicht durchgeführt, da das vorangegangene Metaprogramm sehr ähnliche Techniken verwendet.

Der wesentliche Unterschied zwischen beiden Metaprogrammen ist, dass das vorangegangene Bezug auf die syntaktische Struktur nimmt und das andere die Datentypen der Ausdrücke betrachtet. Das zeigt, dass alle Informationen die sich im Quelltext befinden, zu einer solchen Analyse herangezogen werden können. Dafür erfüllt MetaC alle Voraus-

setzungen, denn es bietet eine entsprechende Infrastruktur, die Mechanismen zur Analyse dieser Informationen bereitstellt.

8.1.3. Alternative Verfahren

Bevor die zuvor genannten Konzepte und Verfahren der Sprache MetaC weit genug entwickelt waren, um sie einsetzen zu können, wurde ein alternatives Verfahren evaluiert. Dazu wurde im Rahmen eines studentischen Projekts (eine interdisziplinäre Arbeit mit Studenten der Fakultät Informatik) mit zwei Arbeiten eine Methodik untersucht, um einzelne Regeln aus den Empfehlungen der MISRA automatisch zu überprüfen (siehe [55], [36]). Das Konzept das hierbei eingesetzt wurde, basiert auf der Idee einer offenen Compiler Architektur [52].

Bei diesem Ansatz erzeugt der Parser des Compilers einen abstrakten Syntaxbaum (abstract syntax tree, AST), auf dem Transformationen und Analysen durchgeführt werden können. Sind alle Operationen abgeschlossen, wird aus dem AST wieder Code generiert. Der ausgegebene Quelltext gehorcht dabei der selben Syntax, die eingangs verwendet wurde.

Dieses Konzept funktioniert ebenso gut wie jenes das auf Metafunktionen in MetaC basiert. Allerdings ist hierbei eine Offenlegung der Compilerschnittstellen notwendig. Eine Öffnung dieser Programmierschnittstelle ist dabei nur sinnvoll, wenn gewährleistet werden kann, dass diese sich nicht durch Weiterentwicklung des Compilers ändert. Nur so kann sichergestellt werden, dass darauf basierende Analysemodule zumindest quellcodekompatibel oder noch besser binärkompatibel zu zukünftigen Versionen bleiben.

Da diese Forderung nicht ohne weiteres gewährleistet werden kann und dieses Problem auch ein Grund für den mangelnden Erfolg solcher Architekturen ist, hat eine Lösung auf MetaC basierten Metafunktionen deutliche Vorteile. Denn sie erzielt zwar nicht ganz die Verarbeitungsgeschwindigkeit des konkurrierenden Ansatzes, doch funktionieren Metaprogramme unabhängig von der Implementierung des Compilers und sind dadurch nicht nur Versionsunabhängig, sondern auch Herstellerunabhängig.

8.1.4. Grenzen der Analysemöglichkeiten

Wie zuvor bereits dargelegt, kann eine automatisierte Überprüfung von Regeln mit Hilfe von MetaC nur auf der Basis der Informationen erfolgen, die unmittelbar im Quelltext vorhanden sind und deren Semantik durch die Sprachdefinition von C geben ist. Leider hat nicht jedes mögliche Konstrukt in C ein wohldefiniertes Verhalten. Neben Quellcode mit wohl definiertem Verhalten, gibt es in C auch Konstrukte, deren Semantik *implementation-defined* (von der Implementierungsvariante definiert), *unspecified* (nicht spezifiziert) oder *undefined* (nicht definiert) ist.

Für Konstrukte mit *implementation-defined behaviour* muss die durch die Implementierung gegebene Semantik berücksichtigt werden. *Unspecified behaviour* liegt vor, wenn der Standard Implementierungsvarianten vorsieht, von denen eine ausgewählt werden muss. Entsprechend der Wahl der Implementierung gibt es somit eine Rückkopplung auf den Quelltext und Programmen, die mit diesem interagieren. Außerdem gibt es noch Konstruktionen mit nicht definiertem Verhalten (*undefined-behaviour*). Nicht definiertes Verhalten entsteht beispielsweise beim Zugriff auf ein Element eines Feldes außerhalb der Feldgrenzen. Die Ursache von undefiniertem Verhalten ist immer ein Fehler im Programm. Solche Konstruktionen können zum Teil durch eine statische Analyse erkannt

werden und somit auch mit Hilfe von Metaprogrammen aufgedeckt werden. Zum Teil ist dafür eine sehr aufwändige Datenflussanalyse notwendig oder es ist sogar unmöglich, nachzuweisen, dass kein undefiniertes Verhalten auftritt. Dadurch ergeben sich Grenzen für die Überprüfung dieser Eigenschaften, die sich ebenso auf die Implementierung von Programmen zur Regelprüfung auswirken.

In MISRA gibt es Regeln, deren Inhalt die Dokumentation oder Kommentierung betreffen. Ein Beispiel hierfür ist die 10. Regel der Richtlinien von MISRA, die das auskommentieren von Quelltext verbietet. Kommentare werden normalerweise schon während der lexikalischen Analyse verworfen. Es ist zwar denkbar die Kommentare als Zusatzinformationen im Parsebaum mit aufzunehmen; dies reicht aber nicht aus um entscheiden zu können, ob der in einem Kommentar enthaltene Text ein echter Quelltext ist oder ein dokumentierender Pseudocode. Ein nachträgliches Parsen könnte zwar durchgeführt werden; jedoch kann dieses ohne weiteres fehlschlagen, wenn der darin auskommentierte Quelltext undefinierte Bezeichner referenziert, beziehungsweise grammatikalisch unvollständige Strukturen im Kommentar enthalten sind. Entsprechend würde die Regelüberprüfung den auskommentierten Quelltext für Pseudocode oder normalen Text menschlicher Sprache halten und ihn daraufhin fehlerhafterweise akzeptieren.

Somit lässt sich festhalten, dass mit Hilfe von MetaC eine Automatisierung von Regelüberprüfungen möglich ist. Diese ist nur beschränkt durch die vorhandenen Informationen im Parsebaum und durch die im C Standard definierten Freiheitsgrade der Semantik. Grundsätzlich sind automatisierte Regelüberprüfungen im Vergleich zur Einzelverifikation durch Menschen schneller. Außerdem ist ein automatisiertes Verfahren auf große Datenmengen anwendbar, ohne dabei die statistische Fehlerhäufigkeit einer menschlichen Analyse zu zeigen.

8.2. Automatische Quelltextinstrumentierung

Software zur Regelung oder Steuerung von technischen Prozessen unterliegt zeitlichen Restriktionen. Die zeitlichen Anforderungen an die Software werden von den physikalischen Randbedingungen verursacht und müssen unbedingt eingehalten werden. Diese Anforderungen werden als Echtzeitbedingungen bezeichnet. Sie legen fest wie viel Zeit zur Verfügung steht, um eine bestimmte Berechnung durchzuführen.

In der Regel muss die Software in Echtzeitsystemen eine Vielzahl an Aufgaben bewältigen. Deshalb gibt es spezielle Verfahren, um mehrere Programme auf ein und demselben Prozessor parallel ausführen zu können. Die verschiedenen Programme werden dazu in so genannten Tasks gekapselt. Das System muss gewährleisten können, dass alle Tasks die an sie gestellten Echtzeitanforderungen einhalten können.

Dabei wird unterschieden zwischen harten und weichen Echtzeitbedingungen. Bei Verletzung von harten Echtzeitanforderungen kann die korrekte Funktion des Systems nicht mehr gewährleistet werden und es besteht unter Umständen sogar Gefahr für Mensch oder Gerät (z.B. bei zeitlich inkorrekter Ansteuerung der Bremsen durch das ESP im Auto). Bei weichen Echtzeitbedingungen degradiert lediglich die Qualität der erbrachten Funktion (z.B. Knackser oder Aussetzer bei der Tonwiedergabe eines MP3 Spielers). Deshalb ist für die erfolgreiche Realisierung von Echtzeitsystemen mit harten Zeitanforderungen, eine sorgfältige Analyse von Ausführungszeiten und der Einhaltung der zugehörigen maximal erlaubten Reaktionszeiten notwendig.

Zur Bestimmung der Ausführungszeiten im schlimmsten Fall werden die längsten potentiell auftretenden Ausführungszeiten der verschiedenen Pfade in der Software benö-

tigt. Die als *Worst Case Execution Time* (WCET) bezeichnete längste Ausführungsdauer eines Pfades hängt vom verwendeten Algorithmus, den Eingangsdaten, den Zuständen in denen sich Software und Hardware befinden und der darunter liegenden ausführenden Architektur ab. Diese Vielzahl an Randbedingungen erschwert die a-priori Bestimmung der WCETs.

Im Folgenden wird zunächst erläutert, welche prinzipiellen Schwierigkeiten bei bestimmten Architekturen eine exakte Modellierung unmöglich machen. Danach wird beschrieben, wie mit Hilfe von Messungen die Echtzeitanalyse mit genaueren Werten unterstützt werden kann. Die für Messungen erforderlich Quelltextinstrumentierung kann mit Hilfe eines Metaprogramms automatisiert werden. Dieses Metaprogramm wird abschließend erläutert und die Vollständigkeit seiner Funktion diskutiert. Außerdem werden die Grenzen dieser Methodik erläutert, die zuvor schon in [57] vorgestellt wurde.

8.2.1. Architekturbedingte Unvorhersagbarkeiten

Moderne Prozessoren verfügen über eine Vielzahl von Techniken um die Ausführung zu beschleunigen. Für die meisten Applikationen spielen dabei Schwankungen in der Ausführungszeit keine große Rolle, stattdessen sind maximale und durchschnittliche Rechenleistung weit wichtigere Kriterien. Folglich werden neue Techniken zur Beschleunigung der Verarbeitung anhand von statistischen Analysen entworfen. Beispiele für solche Verfahren sind: Pipelining, Caches, Sprungvorhersage und spekulative Ausführung.

In der Regel können die Details der Implementierung dieser Techniken in den einzelnen Prozessoren nicht ohne weiteres eingesehen werden. Folglich ist es auch nicht möglich ein Prozessormodell zu erstellen, das eine Zyklen genaue oder zumindest hinreichend genaue Berechnung der Ausführungszeit liefert. Auch müsste solch ein Modell Schritt halten mit der stetigen Entwicklung, also den regelmäßig einfließenden Verbesserungen, Erweiterungen und Varianten in den Prozessorfamilien.

Über die Ursachen und Wirkungen sowie Ansätze zur Problemlösung und Analyseverfahren, die Bestimmung und Vorhersage von Ausführungszeiten betreffend, gibt es eine ganze Reihe von Arbeiten. In [63] werden von Puschner und Burns die wesentlichen Ansätze zusammengefasst. Insbesondere zur Bestimmung der Zeiten im schlechtesten Fall (WCET Analysis) gibt es eine Reihe von Arbeiten (z.B. [33], [62], [69], [70]), die unterschiedliche Ansätze verfolgen und in verschiedenen Szenarios unterschiedlich gute Ergebnisse liefern.

8.2.2. Methodik zur Instrumentierung von Quelltext

Für einen Echtzeitnachweis ist die Kenntnis der *Worst Case Exection Time* der verschiedenen Tasks eines Systems erforderlich. Da der Pfad der längsten Ausführungszeit von vielerlei Faktoren abhängt, wird der ausgeführte Code in einzelne Blöcke, mit bedingungsfrei ausgeführtem Code, zerlegt. Diese werden auch als *Basic-Blocks* bezeichnet. Die WCET der einzelnen *Basic-Blocks* lässt sich dann für die unterschiedlichen Ausführungspfade, die sich abhängig von den Eingangsdaten und dem Zustand des Systems ergeben, addieren, um zu einer Abschätzung der WCET eines Pfades zu kommen.

Dies setzt voraus, dass sich der Quelltext in einer Art und Weise instrumentieren lässt, die ein Ausmessen genau dieser Codefragmente erlaubt. *Basic-Blocks* sind durch folgende Strukturen im Quelltext begrenzt:

- bedingte Sprünge

- Ausdrücke:
  - *logical-AND-expression* (siehe Abschnitt 6.5.13 in [17])
  - *logical-OR-expression* (siehe Abschnitt 6.5.14 in [17])
  - *conditional-expression* (siehe Abschnitt 6.5.15 in [17])
- Anweisungen:
  - *expression-statement* (bei Verwendung eines entsprechenden Ausdrucks, siehe Abschnitt 6.8.3 in [17])
  - *selection-statement* (siehe Abschnitt 6.8.4 in [17])
  - *iteration-statement* (siehe Abschnitt 6.8.5 in [17])
  - *jump-statement* (siehe Abschnitt 6.8.6 in [17])

Abbildung 8.2.: Ausdrücke und Anweisungen die instrumentiert werden müssen

- Sprünge mit unbekanntem Ziel
- Sprünge an eine Sprungmarke, an die von mehreren Stellen aus verzweigt wird

Die Darstellung in Abbildung 8.2 zeigt, welche grammatikalische Strukturen der Sprache C zu Instruktionen führen können, die instrumentiert werden müssen. Da von der Zielarchitektur und den Optimierungsverfahren des eingesetzten Compilers abhängt, ob tatsächlich Instruktionen verwendet werden, die zu instrumentierende Sprünge ausführen, müssen grundsätzlich alle diese Strukturen instrumentiert werden. Wird von einem Compiler für eine dieser Konsturktionen kein bedingter Sprung erzeugt, so hat die Sprungbedingung normalerweise trotzdem eine Abhängigkeit mit der Ausführungszeit. Konstrukte, deren Daten konstant sind und deshalb immer das gleiche Ergebnis liefern, müssen hingegen nicht instrumentiert werden.

Für alle anderen Ausdrücke und Anweisungen gilt, dass sie aufgrund ihrer rekursiven Definition ebenfalls eine der oben genannten Strukturen beinhalten könnten. Deshalb muss die Analyse des Quelltextes ebenfalls rekursiv erfolgen. Nach der Detektion der entsprechenden Knoten im Parsebaum muss der Quelltext, der eine Messung realisiert, an dafür geeigneten Stellen eingefügt werden. Diese Positionen sind so zu wählen, dass sie möglichst dicht vor den jeweiligen Blockgrenzen liegen, um eine hohe Messgenauigkeit zu erreichen.

8.2.3. Metaprogramm zur automatisierten Instrumentierung

Für die Realzeitanalyse ist eine Kenntnis der maximalen Ausführungszeit vom Beginn eines *Basic-Blocks* bis zu dessen Ende mit möglichst hoher Genauigkeit erforderlich. Das bedeutet, dass die Messpunkte möglichst nahe vor den Maschineninstruktionen platziert werden müssen, die eine Veränderung des Kontrollflusses nach sich ziehen können. Das entscheidende Problem bei der Positionierung von Messpunkten in C Quelltext ist, dass vor einigen dieser bedingten Sprunganweisungen ein Ausdruck berechnet wird, der entscheidet, ob der Sprung vollzogen wird oder nicht. Wann die im Folgenden vorgestellte Form der Instrumentierung zu Ungenauigkeiten führen könnte und wie dieses Problem umgangen werden kann, wird in Abschnitt 8.2.5 erläutert.

Im vorangegangenen Abschnitt wurden bereits die grammatikalischen Strukturen aufgelistet, die für die Realzeitanalyse den Quelltext in mehrere *Basic-Blocks* aufteilen. Das Konzept zur Instrumentierung wird am Beispiel des *selection-statements* erläutert. Dazu ist in Abbildung 8.3 die Syntax dieser Anweisung noch einmal dargestellt. Die genaue Definition der Semantik ist in Abschnitt 6.8.4 des ISO Standards beschrieben.

selection-statement:
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
switch (*expression*) *statement*

Abbildung 8.3.: Syntax von *selection-statement*

Die **if-else** Anweisungen führen einen bedingten Sprung nach der Auswertung des als *expression* angegebenen Ausdrucks durch. Das potentielle Ziel dieses bedingten Sprunges ist das *statement* nach dem **else**. Ein Compiler könnte auch die beiden Anweisungen vertauschen, wenn die Datenflussanalyse den Hinweis gibt, dass der **else** Teil häufiger genommen wird. Je nachdem ist das Sprungziel die Adresse der ersten Instruktion der Anweisung vor dem **else** oder jene nach der Sprungbedingung.

```
 1  /* count > 0 assumed */
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0:  do {  *to = *from++;
 5  case 7:        *to = *from++;
    case 6:        *to = *from++;
    case 5:        *to = *from++;
    case 4:        *to = *from++;
    case 3:        *to = *from++;
10  case 2:        *to = *from++;
    case 1:        *to = *from++;
                } while (--n > 0);
    }
```

Abbildung 8.4.: Duff's Device

Bei der **switch** Anweisung sind die Sprungziele die unterschiedlichen **case** Labels beziehungsweise das **default** Label irgendwo innerhalb der Anweisung, die Teil des **switch** Konstrukts ist (für ein Beispiel hierfür siehe Abbildung 8.4). Die Sprungziele der **switch** Anweisung können somit auch indirekte Kinder sein, wie es bei *Duff's Device* [31] für die **case** Labels eins bis sieben der Fall ist. Dies erschwert das Auffinden des Verzweigungspunktes ausgehend von einem Teilast im Parsebaum. Deshalb gibt es die Metafunktion **branch_creator**, die im MetaC Compiler direkt implementiert ist und die diese Aufgabe effizient löst und vereinfacht.

Für alle drei Formen des *selection-statement* gilt aber, dass zunächst ein Ausdruck ausgewertet wird, aus dessen Ergebnis hinterher ein entsprechender Sprung abgeleitet wird. Das bedeutet, dass vor Auswertung der Sprungbedingung die Messung erfolgen muss. Da das Ergebnis des Ausdrucks für den Sprung verwendet wird, ist es auch nicht möglich mit einem Komma (siehe Abschnitt 6.5.17 der Sprachdefinition von C) einen weiteren

Ausdruck anzuhängen, der diese Messung durchführt. Denn beim Kommaoperator ist das Ergebnis des letzten Teilausdrucks, das Ergebnis des Kommaausdrucks. Die Rückgabewerte aller anderen Teilausdrücke werden verworfen. Deshalb muss die Messung als eigene Anweisung vor der **`if-else`** Sequenz eingefügt werden.

```
1  /* Eingabe Code */
   irgendwas();
   if (f() < x) {
       a = 0;
5      return;
   } else
       a = a - x;
```

```
 1  /* Ergebnis Code */
    irgendwas();
    messung(1);
    if (f() < x) {
 5      a = 0;
        messung(2);
        return;
    } else {
        a = a - x;
10      messung(3);
    }
```

Abbildung 8.5.: Beispiel einer Quelltextinstrumentierung

Für beide Teiläste des **`if-else`** Konstrukts wird der Kontrollfluss mit der nächsten Anweisung nach dem Block fortgesetzt. Der nachfolgende Code kann demnach von mehreren Positionen aus angesprungen werden. Deshalb muss er als eigenständiger Block ausgemessen werden, was zur Folge hat, dass die vorhergehenden Blöcke einen weiteren Messpunkt an ihrem Ende benötigen. Jeder dieser Messpunkte stellt somit die Grenze zwischen zwei *basic-blocks* dar.

Wenn die bedingt ausgeführte Anweisung, also der jeweilige Ast des *selection-statements* im Parsebaum, aus einer Liste von Anweisungen in geschweiften Klammern (*compound-statement*) besteht, so kann die Messroutine einfach am Ende dieser Liste angehängt werden. Hierbei muss lediglich beachtet werden, welche Anweisung am Ende steht. Führt diese einen Sprung aus (mögliche Kandidaten hierfür sind **`break`**, **`continue`**, **`goto`** und **`return`**), so muss die Messanweisung vor dieser Anweisung platziert werden (siehe Abb. 8.5, Zeile 5 in der Eingabe und 6,7 im Ergebnis). Ist der Teilast kein *compound-statement*, so kann das selbe Verfahren angewendet werden. Allerdings muss dafür ein neues *compound-statement* erzeugt werden und die Anweisung, die zuvor den Teilast dargestellt hat, darin als erstes Element einfügt werden.

Auf eine Berücksichtigung der Funktion **`longjmp`** wurde verzichtet, da die MISRA Regeln ihre Verwendung verbieten. Diese Funktion verändert den Kontrollfluss auf eine ähnliche Art, wie **`return`**. Um sie berücksichtigen zu können muss ein passendes Quelltext-Strukturmuster erstellt werden, das geeignet ist um die Stellen zu finden an denen die Funktion verwendet wird. Für die Instrumentierung der relevanten Positionen kann die selbe Technik angewendet werden, die für normale Sprunganweisungen benutzt wird.

8.2.4. Minimierung der Instrumentierung

Bisher wurde erläutert, an welchen Stellen eine Instrumentierung erforderlich ist. Grundsätzlich bringt aber jeder Messpunkt einen gewissen Overhead mit sich. Außerdem steigt der Aufwand für die Realzeitanalyse in Abhängigkeit der Komplexität des verwendeten

```
meta zed instrument(func f, zed i) {
   stmt branches[], *branch=branches, body,parents[],
        *xp, bc, p;
   measure_code:= measurement(i);

   get_all_branches(f,branches);
   while (*branch) {
      bc = branch_creator(*branch);
      xp = parents;
      /* search bc in parents */
      while (*xp && (*xp != bc))
         ++xp;
      /* is bc already instrumented? */
      if (*xp == 0) {
         *xp = bc;       /* add bc */
         p = parent(bc);
         if ((predecessor(bc))
              || (!is_branch(p) && parent(p))) {
            before_branch(codeof measure_code,*branch);
            ++i;
         }
      }
      end_branch(codeof measure_code,*branch);
      ++i;
      ++branch;
   }
   body = get_body(f);   /* instrument function body */
   begin_branch(codeof measure_code,body);     ++i;
   end_branch(codeof measure_code,body);       ++i;
   return i;
}
```

Abbildung 8.6.: Metaprogramm zur Instrumentierung von Quelltext

Verifikationsalgorithmus mit der Anzahl der Messpunkte. Auf der anderen Seite können die Ausführungszeiten relativ kurzer Instruktionssequenzen hinreichend genau auch ohne Messung berechnet werden. Diese beiden Randbedingungen machen es wünschenswert, Messpunkte zu eliminieren, die redundant sind. Ein Messpunkt kann als redundant bezeichnet werden, wenn ihm direkt ein anderer Messpunkt vorangeht oder nur sehr wenige Instruktionen dazwischen liegen. Außerdem sollte der Instrumentierungsalgorithmus auch so gestaltet sein, dass er keine redundanten Messpunkte einfügt.

Redundante Messpunkte können durch eine entsprechende Systematik im Instrumentierungsalgorithmus verhindert werden. Im vorgestellten Metaprogramm (Abbildung 8.6) werden dafür alle Äste im Kontrollfluss erfragt und ihre Enden mit einem Messpunkt versehen. Weiterhin wird vor jedem Verzweigungspunkt ein Messpunkt eingefügt. Dabei muss insbesondere beachtet werden, dass an manchen Verzweigungspunkten (bei **`switch`** und **`if-else`** Konstrukten) mehrere Teiläste entstehen können, die hinterher wieder auf den gleichen Kontrollfluss zurückführen. Damit diese Stellen nicht mehrfach instrumentiert werden, wird zwischengespeichert, welche Verzweigungen im Kontroll-

```
 1  void xyz(int a, int b) {
       messung(7);
       /* Doppelinstrumentierung unterbunden */
       do {
 5        /* Verwerfbare Messung für if */
          if (b) {
             null();
             messung(5);
             break;
10        } else {
             messung(6);
             return;
          }
          messung(4);
15     } while (a);
       messung(8);
    }
```

Abbildung 8.7.: Überflüssige Messpunkte

fluss bereits instrumentiert worden sind. Dafür wird die Variable **`parents`** benutzt – diese speichert, welche Verzweigungspunkte bereits instrumentiert worden sind.

Weiterhin werden bei jeder Funktion der Eintrittspunkt und der Austrittspunkt mit einem Messpunkt instrumentiert. Dadurch kann man die exklusiv genutzte Prozessorzeit einzelner Funktionen berechnen, da die Zeit der Funktionsaufrufe somit abgezogen werden kann. Der Messpunkt am Anfang kann auch dazu führen, dass zwei Messpunkte direkt aufeinander folgen. Diese Situation entsteht, wenn die erste Anweisung der Funktion ein Verzweigungspunkt im Kontrollfluss ist (siehe Abbildung 8.7, Zeile 4). Dies muss durch eine entsprechende Analyse erkannt werden, damit eine überflüssige Instrumentierung verhindert werden kann. Im abgebildeten Metaprogramm wird das durch die **`if`** Anweisung in Zeile 14 realisiert.

Darüber hinaus lässt sich die Anzahl der Messpunkte weiter reduzieren, wenn man eine Prämisse hinzufügt. Ausgehend davon, dass die Ausdrücke an den Entscheidungspunkten im Kontrollfluss von relativ kurzer Ausführungszeit sind und deshalb hinreichend genau abgeschätzt werden können, so kann auf einzelne Messpunkte direkt nach einer vorangegangenen Verzweigung im Kontrollfluss verzichtet werden (siehe Abbildung 8.7, Zeile 6). Im Falle der Abbildung ist die vorangegangene Verzweigung im Kontrollfluss ohnehin nur ein Sprungziel, an dessen Stelle kein Ausdruck berechnet wird. Ein möglicher Ansatz wäre, das instrumentierende Metaprogramm so zu realisieren, dass nur an solchen Stellen die Messpunkte verworfen werden, an denen überhaupt kein Assembler zwischen den beiden Messaufrufen generiert wird.

Eine weitere Minimierung der Anzahl der Messpunkte kann durch Datenflussanalyse erreicht werden: Lässt sich durch statische Analyse feststellen, dass gewisse Teiläste im Kontrollfluss niemals betreten werden, so kann man diese von der Instrumentierung ausschließen. Die hier vorgestellte Implementierung unterstützt dies nicht direkt. Der MetaC-Compiler führt aber zur Parsezeit eine Analyse durch, die detektiert, welche Ausdrücke einen konstanten Wert ergeben. In Fällen, in denen ein konstanter Wert die Schaltbedin-

```
 1  void abc(double a, double b) {
        /* Messpunkt */
        f1();                   // Block 1
        /* Messpunkt */
 5      if (sqrt(a) > b) {   // ÜBERLAPPUNG
            f2();               // Block 2
            /* Messpunkt */
        } else {
            f3();               // Block 3
10          /* Messpunkt */
        }
    /* ... */
```

Abbildung 8.8.: Überlappung von *basic-blocks*

gung für den Kontrollfluss ergibt, wird der entsprechende Ast aus dem Parsebaum entfernt und somit eine überflüssige Instrumentierung unmöglich gemacht.

8.2.5. Präzision der Instrumentierung

Wie zuvor bereits betont kann in C Quelltext kein Code an eine Position eingefügt werden, die direkt vor dem Sprung aber nach der Auswertung der zugehörigen Sprungbedingung liegt. Folglich wird die Verarbeitungszeit zur Berechnung der Sprungbedingung nicht in die Messung mit aufgenommen, zu der sie eigentlich noch gehört (siehe Abbildung 8.8). Der Fehler der dabei entsteht ist um so größer, je länger diese Berechnung dauert (in diesem Fall der Vergleich der beiden Gleitkommavariablen **a** und **b** in Zeile 5). Inwiefern das zu einem Problem für einen nachfolgenden Realzeitnachweis wird, kann nicht pauschal beantwortet werden. Da diese Zeit den im Kontrollfluss nachfolgenden Blöcken (hier die Blöcke 2 und 3 in Abbildung 8.8) zugeordnet wird, führt dies erst zu einer Ungenauigkeit, wenn statistische Ausreißer in den Ausführungszeiten der Bedingung existieren.

Ein statistischer Ausreißer ist für die Realzeitanalyse nur von Relevanz, wenn er den Worst-Case darstellt und nur in einem der beiden Blöcke gemessen wurde. Dieser Ausreißer müsste aber für beide Blöcke berücksichtigt werden. Dieses Problem entsteht nicht, wenn die Triggerbedingung und somit der statistische Ausreißer, Block 1 zugeordnet wird.

Ähnlich verhält es sich mit dem Ausdruck der den Rückgabewert einer Funktion berechnet. Da auch sein Ergebnis für die nachfolgende Verarbeitung benötigt wird, kann auch ihm nicht einfach mit Hilfe des Kommaoperators eine Messung angehängt werden. Somit wird auch die Ausführungszeit der Berechnung des Ausdrucks der aufrufenden Funktion zugeordnet und nicht der ausführenden. Diese potentielle Ungenauigkeit lässt sich aber durch eine kleine Anpassung des Codes durch ein Metaprogramm beseitigen, wie im folgenden Abschnitt erläutert wird.

8.2.6. Erhöhung der Messgenauigkeit

Das Messergebnis von Ausführungszeiten von Ausdrücken, deren Rechenergebnisse das Ziel von bedingten Sprüngen beeinflusst, ist besonders kritisch für die Echtzeitanalyse. Bei normaler Instrumentierung des Quelltextes kann ein einzelner statistischer Ausreißer

```
 1  meta void adjust_returns(func f) {
       stmt rets[], *r = rets;
       expr re,ne;
       symb tmp;
 5     type rt = get_return_type(f);
       if (rt == typeof(void))
          return;
       get_returns(f,rets);
       tmp = define_local(f,rt,
10        (ident)"return_temporary");
       ne = (expr) tmp;
       assign_tmp: tmp = re;
       while (*r) {
          re = get_return_expr(*r);
15        insert_before(codeof assign_tmp,*r);
          set_return_expr(*r,ne);
          ++r;
       }
    }
```

Abbildung 8.9.: Anpassung von Rücksprung Anweisungen

durch seine Zuordnung zu einem bestimmten Block das Ergebnis signifikant beeinflussen, wie im vorangegangenen Abschnitt erläutert wurde.

Eine andere Situation, die zu ungenauer Zeitmessung führt, ergibt sich, wenn die Rücksprunganweisung einen signifikanten Anteil an der Ausführungszeit der Gesamtfunktion hat. Diese Zeit wird fehlerhafterweise der aufrufenden Funktion zugeordnet, da Messpunkte nur unmittelbar vor der Rücksprunganweisung platziert werden können. Der Ausdruck einer solchen *return-statement*s kann allerdings eine aufwändige Berechnung beinhalten, wie die Gleitkommaoperation in der Funktion **`my_fabs`** in Abbildung 8.10 zeigt.

In beiden Fällen ist durch den Einsatz von MetaC eine Erhöhung der Messgenauigkeit erreichbar. Dazu kann das exemplarisch dargestellte Metaprogramm aus Abbildung 8.9 eingesetzt werden, um diese Problem zu lösen. Dieses ist nur für die Anpassung von **`return`** Anweisungen geeignet und muss für **`if-else`** oder andere Anweisungen entsprechend angepasst werden.

Um den Messpunkt so zu platzieren, dass die Ausführungszeit des relevanten Ausdruckes dem richtigen Block zugeordnet wird, muss der Quelltext refaktorisiert werden. Dazu wird zunächst einmal eine Variable erzeugt, die im Falle einer **`return`** Anweisung den Typ des Rückgabewerts der Funktion hat. Für die **`if-else`** Anweisungen ist es immer ein skalarer Typ (also **`int`**). Für alle anderen Anweisungen müssen entsprechende Alternativen zum Einsatz kommen. Der Typ muss dann aus der semantischen Definition der Anweisung abgeleitet werden.

Dann wird diese neue Variable genutzt, um das Ergebnis des Ausdrucks mit Hilfe einer Zuweisung zu speichern. Diese Zuweisung wird als *expression-statement* vor der betreffenden Anweisung platziert (im Beispiel: Zeile 12), gefolgt von einem Messpunkt. Der den Kontrollfluss verändernde Ausdruck wird daraufhin durch die Variable ersetzt, die das Ergebnis des Ausdrucks gespeichert hat. In dem Beispiel in Abbildung 8.10 ist diese Modifikation durch ein Vergleich der Zeile 5 mit den Zeilen 10-14 erkennbar.

```
 1   /* unmodifizierte Version */
     double my_fabs(double x) {
        /* messung 1 */
        /* messung 2 */
 5      return sqrt(x * x);
     }

     /* angepasste Version */
     double my_fabs(double x) {
10      double return_temporary;
        /* messung 1 */
        return_temporary = sqrt(x * x);
        /* messung 2 */
        return return_temporary;
15   }
```

Abbildung 8.10.: Refaktorisierter Rücksprung zur Verbesserung der Messgenauigkeit

Semantisch gesehen ist die Funktionalität des Quelltextes unverändert. Der einzige Unterschied ist, dass eine Variable mehr definiert wurde. Für den generierten Maschinencode zieht dies aber keine Änderung nach sich, denn für jeden berechneten Ausdruck, der nicht unmittelbar einer Variablen zugewiesen wird, wird ohnehin eine temporäre Variable erzeugt (siehe [19], Band 2, Kapitel Zwischencode Generierung).

Optimierende Compiler, wie sie heutzutage Standard sind, erkennen den Bereich in dem eine Variable gültig und notwendig ist und allozieren nur dafür ein Register oder einen Platz im Hauptspeicher. Dies hat zur Konsequenz, dass die durchgeführte Veränderung nicht nur semantisch unverändert ist, sondern unter den genannten Voraussetzungen auch zum selben Maschinencode führt.

8.2.7. Grenzen der Instrumentierung

Die bisher besprochenen Instrumentierungsverfahren zur automatisierten Instrumentierung mit Hilfe von Metaprogrammen haben nur Messpunkte an Stellen eingefügt, an denen Anweisungen *Basic-Blocks* begrenzen. Wie aber am Anfang des Kapitels festgestellt wurde, gibt es auch Ausdrücke die eine ähnliche Wirkung haben können. Die vorgestellten Konzepte lassen sich zwar in äquivalenter Art auf Ausdrücke anwenden, einzig der visuelle Eindruck dieser Modifikationen weicht derart stark vom Eingangsquelltext ab, dass von diesem Vorgehen abgesehen wurde.

Außerdem muss die Instrumentierung solcher Ausdrücke noch unter folgenden Gesichtspunkten in Bezug auf die Realzeitanalyse gesehen werden:

- Der daraus resultierende Maschinencode ist meistens sehr kurz und seine Ausführungszeit lässt sich deshalb hinreichend genau mit Hilfe eines Modells der Zielarchitektur berechnen.
- Diese Ausdrücke erzeugen meistens mehrere Grenzen von verschiedenen *Basic-Blocks* innerhalb einer Quelltextzeile und sind dadurch schwierig zu analysieren.

Der zweite Punkt verursacht eine uneindeutige Assoziation zwischen Codezeilen und *Basic-Blocks* und kann somit zu Irritationen führen. Aus diesem und dem zuvor genann-

ten Grund wurde eine Instrumentierung solcher Codestrukturen verworfen. Stattdessen wird überprüft ob sich solche Ausdrücke im Quelltext befinden, und der Benutzer wird aufgefordert, gegebenenfalls diese Strukturen durch äquivalente **`if-else`** Anweisungen zu ersetzen. Dafür wird die Metafunktion eingesetzt, die in Abbildung 8.11 dargestellt ist.

```
meta void verify_expressions(func f) {
   stmt stmts[], *st;

   st = stmts;
   get_all_stmts(st,f);
   while (*st) {
      expr e, i = 0, j = 0;
      stmt s = *st;
      ++st;
      e = get_expr(s);
      if (is_for(s)) {
         i = get_for_init(s);
         j = get_iterator(s);
      }
      if ( (e && has_cfbranch(e))
         ||(i && has_cfbranch(i))
         ||(j && has_cfbranch(j))) {
         warn("uninstrumented control flow branch " +
              "in expression at " + (strg)line_of(s));
      }
   }
}
```

Abbildung 8.11.: Metafunktion zum Auffinden von Ausdrücken mit Kontrollfluss

Grundsätzlich kann eine automatisierte Instrumentierung nur ein unterstützendes Werkzeug für die Realzeitanalyse sein. Denn bei den verschiedenen Architekturvarianten mit parallelen Einheiten und vielen weiteren Verfahren zur Beschleunigung der Berechnung von Instruktionen, ist es unmöglich allgemein gültige Kriterien für die Instrumentierung zu finden, die ein Erzeugen redundanter Messpunkte ausschließen. Unter den Verfahren zur Performancesteigerung gibt es verschiedene Varianten zur bedingten Ausführung von Instruktionen, die unter Umständen alternative Äste im Quelltext in einen einzigen Pfad von Maschinencode übersetzen (aktuelle Beispiele für solche Architekturen sind ARM, Itanium und SPARC).

Das Einfügen von Messpunkten kann unter Umständen die Ergebnisse des Optimierungsprozesses des architketurspezifischen Compilers negativ beeinflussen. Deshalb ist es durchaus denkbar, dass ein Kompilat von uninstrumentiertem Quelltext ohne bedingte Sprünge auskommt, eines mit Messpunkten jedoch nicht. Solche Effekte stehen außerdem in Abhängigkeit zu der Art der Implementierung des Messpunktes (z.B. inline Assembler oder Funktionsaufruf). Abschließend ist festzuhalten, dass dieses Verfahren den entscheidenden Vorteil bietet, dass es automatisch abläuft.

9. Bildung von Abstraktionsebenen mit MetaC

CASE-Tools vereinfachen den Umgang mit komplizierten Laufzeitumgebungen durch eine Abstraktion auf der semantischen Ebene. Hierzu werden dem Benutzer Sprachprimitive zur graphischen Modellierung angeboten die domänenspezifische Probleme abstrahieren. Ein Beispiel hierfür ist die asynchrone Kommunikation mit Nachrichten zwischen zwei voneinander unabhängigen Instanzen. Diese Funktionalität wird in der Sprache SDL [60] mit *Signalen* und in RoseRT mit Nachrichten angeboten.

Solche Abstraktionen haben für den Entwickler den Vorteil, dass häufig verwendete domänenspezifische Funktionen nicht auf der Basis der vorhandenen Laufzeitumgebung von Hand umgesetzt werden müssen. Stattdessen erfolgt die Umsetzung automatisch, wodurch Fehler vermieden werden können. Allerdings hat eine automatische Umsetzung normalerweise den Nachteil, dass sie weniger effizient ist, da kein Applikationswissen in die Implementierung mit eingebracht werden kann. Somit ist es dem Entwickler nicht ohne weiteres möglich, geeignete Verfahren anzuwenden, wie beispielsweise eine bestimmte Variante zur Realisierung des wechselseitigen Ausschluss, wie Spinlock, Mutex, Reader/Writer Lock oder das zusätzliche Priority Inheritance Protocol. Stattdessen muss er sich darauf verlassen, dass in der Phase der Codegenerierung dieses Optimierungspotential durch die Analyse des Designs erkannt wird.

Für die Übersetzung einer Applikation in ausführbaren Code die auf einer solchen Abstraktionsschicht basiert, stehen verschiedene Verfahren zur Verfügung. Dabei sind in den meisten Fällen ein hoher Grad an Portierbarkeit und eine effiziente Umsetzung konkurierende Ziele. Denn um eine Implementierung effizient umsetzen zu können, ist eine detaillierte Kenntnis der Zielplattform erforderlich. Auf der anderen Seite dürfen für eine flexible Abbildung auf verschiedene Ziele keine restriktiven Annahmen gemacht werden.

Die folgenden Abschnitte behandeln die Fragestellung, wie mit Hilfe von MetaC die Effizienz in der Umsetzung im Vergleich zu normalen Verfahren gesteigert werden kann. Die vorgestellten Konzepte konzentrieren sich dabei auf Verfahren, die den hohen Grad an Portierbarkeit beibehalten und ebenso eine starke Abstraktion unterstützen.

9.1. Konzepte zur Abstraktion

Grundsätzlich kann eine Abstraktion zur Implementierung einer Modellierungssprache auf zwei Arten erfolgen: von oben nach unten oder von unten nach oben. Bei der häufiger angewendeten Abstraktion von unten nach oben werden typische Konstrukte der Betriebsumgebung semantisch abstrahiert, um darauf eine generische Laufzeitumgebung zu realisieren. Der umgekehrte Weg von oben nach unten erfordert eine direkte Abstraktion der Modellierungskonstrukte. Beide Konzepte werden in den nachfolgenden Teilabschnitten vorgestellt und bzgl. ihrer Vor- und Nachteile analysiert.

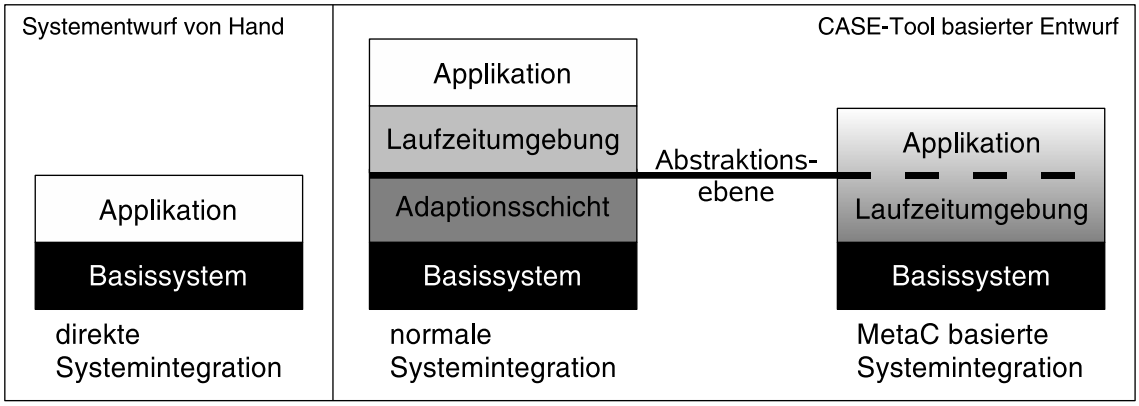


Abbildung 9.1.: Abstraktionsprinzipien zur Systemintegration

9.1.1. Abstraktion von RTOS APIs

Eine Möglichkeit, die Codegenerierung für verschiedene Zielsystem zu vereinfachen, ist die Abstraktion des verwendeten RTOS. Dazu werden gängige Primitive von Realzeit-Betriebssystemen durch abstrakte Funktionen und Datentypen ersetzt. Wichtig ist hierbei eine exakte Definition der Semantik der einzelnen Funktionen. Da sich viele Funktionen aus den grundlegenden Primitiven nachbilden lassen, ist es möglich, nur die notwendigen Basisprimitive zu abstrahieren. Unter Verwendung dieser Abstraktion können dann die erforderlichen Funktionen unabhängig von der Zielplattform in einer Adaptionsschicht nachgebildet werden. RoseRT geht beispielsweise diesen Weg, wobei die Kommunikation über Message-Queues mit Hilfe von Semaphoren und gemeinsame genutzten Speicher nachgebildet wird.

Anstatt eine spezielle Abstraktionsschicht einzusetzen, ist es auch möglich das POSIX Interface oder einen ähnlichen Standard zu benutzen. Dies setzt voraus, dass alle möglichen Zielplattformen inklusive dem Simulationstarget dieses API unterstützen. Dies kann zu einer wesentlichen Einschränkung der Flexibilität in der Praxis werden, da man dadurch auf den Einsatz von Betriebssystemen beschränkt ist, die die entsprechende Funktionalität unterstützen. Andererseits wird durch dieses Vorgehen ein Teil der Arbeit für die Abstraktion an den RTOS Anbieter ausgelagert. Allerdings ist dabei zu beachten, dass das POSIX API Implementierungsvarianten vorsieht, die aus einer leeren Schnittstellenimplementierung ohne jegliche Funktionalität besteht. Das bedeutet, dass bei dieser Vorgehensweise sichergestellt werden muss, dass die Zielplattform alle notwendige Funktionen bereitstellt und korrekt umsetzt.

Die Option der POSIX Spezifikation, nur einen Teil der Funktionalität zu implementieren, ermöglicht es Anbietern von RTOS mit relativ geringem Aufwand, zusätzlich zu ihrem nativen API, das POSIX API zu unterstützen. Dabei bleibt der Mehraufwand in einem überschaubaren Rahmen, da es möglich ist, nur ein kundenspezifisches Subset der Funktionalität zu unterstützen. Die restlichen Funktionen des API werden dann dem Programmierer ohne hinterlagerte Implementierung dargeboten. Das bedeutet, dass die Funktion die spezifizierte Semantik des APIs nicht realisisert und statt dessen nur den Fehlercode **`ENOTSUP`** zurück, der für „Operation not supported“ steht. Andererseits bringt diese Zwischenschicht unter Umständen auch einen gewissen Overhead mit sich. Der Grad des Overhead ist abhängig von der Ähnlichkeit des nativen APIs mit der Semantik des POSIX APIs.

9.1.2. Abstraktion von Modellierungskonstrukten

Alternativ kann eine Abstraktion der Laufzeitumgebung durchgeführt werden, indem abstrakte Datentypen und Funktionen der Modellierungskonstrukte eingeführt werden. Weiterhin werden Metafunktionen benötigt, die eine Quelltextrefaktorisierung der Applikation durchführen und dabei die abstrakten Datentypen und Funktionen zu konkreten Implementierungen auflösen. Ein solches Vorgehen ist auf Präprozessorbasis nicht realisierbar, da dieser keine Strukturanalyse und -rekonfiguration des Quelltextes erlaubt.

Bei dieser Form der Abstraktion wird die Applikation direkt an das darunterliegende Betriebssystem gekoppelt und die dedizierte Laufzeitumgebung stellt nur noch Spezialfunktionen zur Systemintegration bereit (siehe Abbildung 9.1). Dazu müssen die verwendeten Datentypen während der Codegenerierung an das darunterliegende Betriebsystem angepasst und die Semantik der Funktionen entsprechend nachgebildet werden. Durch diese spezielle Methodik verschmelzen Applikation und Laufzeitumgebung, die Abstraktionsschicht wird aufgelöst und eine besonders effiziente Abbildung entsteht. Eine Adaptionsschicht, wie sie bei einer Abstraktion auf der Ebene von Betriebssystemkonstrukten nötig ist, entfällt hier vollständig.

Die notwendigen Modifikationen am Quelltext der Applikation können mit Hilfe von MetaC realisiert werden. Insbesondere ist es mit Hilfe von MetaC möglich, die verwendeten Datenstrukturen auf das Notwendige zu reduzieren und somit den Bedarf an Speicher zu minimieren. Diese Minimierung ist prinzipiell nur durch eine semantische Analyse des zu realisierenden Systems möglich. Verfahren die lediglich Modifikationen auf der Basis lexikalischer Analyse durchführen, können deshalb hierfür nicht eingesetzt werden.

Darüber hinaus sind bei einem MetaC basierten Ansatz weitere Optimierungen denkbar. Beispielsweise kann auf eine resourcenintensive globale Benennung von Synchronisationsstrukturen (z.B. Semaphore, die Prozessgrenzen übergreifen) verzichtet werden, wenn es keine davon abhängigen dynamischen Systemstrukturen gibt. Die statische Systemstruktur beinhaltet ebenso einige Informationen, die für Optimierungen genutzt werden können. Beispielsweise kann dadurch der Zugriff auf die Kommunikationsstrukturen auf den Zeitpunkt der Initialisierung der Tasks vorgezogen werden, wenn statische (d.h. über die Laufzeit sich nicht verändernde) Kommunikationsmechanismen eingesetzt werden. Normalerweise muss jeder Kanal bei einer Sendeoperation einzeln geöffnet und wieder geschlossen werden, wenn nicht nachweisbar ist, ob die Kommunikation noch steht oder sich der Kommunikationspartner geändert haben könnte.

9.2. Modellabstraktion am Beispiel von SDL

Bevor es möglich, ist ein mit SDL modelliertes System für ein bestimmtes Zielsystem umzusetzen, müssen die Idealisierungen der Sprache an die Wirklichkeit angepasst werden. So berücksichtigt das Ausführungsmodell von SDL keine Zeit für die Verarbeitung einer Transaktion, die Längen der Warteschlangen für Signale ist unbegrenzt, alle Prozesse arbeiten wirklich gleichzeitig und Kommunikationskanäle arbeiten verzögerungs- und fehlerfrei.

Da aber sowohl der Arbeitsspeicher als auch die Geschwindigkeit von Prozessoren und Kommunikationsverbindungen begrenzte Resourcen auf realen Systemen sind, müssen diese Idealisierungen aufgelöst werden. Dies wird einerseits durch eine entsprechende Dimensionierung des Zielsystems erreicht, andererseits müssen Randbedingungen wie

```
meta void prepare_infrastruktur(void);
meta void static_process(func f, func ic,
                         type dt,zed ql);
meta symb sdl_timer_create(ident i);
meta type sdl_pid_t, sdl_timer_t, sdl_signal_t;
```

Abbildung 9.2.: Abstraktion grundlegenden SDL-Strukturen mit MetaC Definitionen

fehlerbehaftete Kommunikationsmechanismen bereits in der Applikation berücksichtigt werden.

Bei entsprechender Modellierung der Systemstrukturen und Generierung der selben mit Metafunktionen, können diese Systemparameter als Argumente für die Implementierung übergeben werden. Dies ermöglicht es, durch Systemanalyse die jeweiligen Parameter der verschiedenen Konstrukte so zu bestimmen, dass die Anforderungen an die darunterliegende Hardware minimal werden und die Implementierung trotzdem der Modellierung entspricht. Die nachfolgenden Abschnitte erläutern, wie eine solche Umsetzung von SDL Systembeschreibungen auf eingebetteten Systemen mit Hilfe von MetaC prinzpiell möglich ist. Hier wird somit nur gezeigt, dass sich ein solches Konzept verwirklichen lässt, die Details einer solchen Implementierung gehen über den Rahmen des vorgestellten Verfahrens hinaus.

9.2.1. Metamodellierung des SDL Laufzeitmodells

Bei der Abbildung von SDL Modellen muss unterschieden werden zwischen der statischen Struktur eines Modells und seiner dynamischen Semantik. Die statische Struktur kann durch entsprechende Metafunktionen zur Kompilierzeit in Quelltext umgesetzt werden. Der erzeugte Quelltext setzt diese Struktur zur Initialisierungszeit des Systems mit Hilfe der Betriebssystemfunktionen des Zielsystems um. Konstrukte, die diese Voraussetzung erfüllen und mit Metafunktionen als Initialisierungscode nachgebildet werden können, sind beispielsweise statische Prozesse, Timer und Signalrouten (siehe Abbildung 9.2 für eine Liste der entsprechenden Metafunktionen).

Die SDL-Konstrukte, die das dynamische Verhalten des Systems spezifizieren, können zum Teil direkt vom Codegenerator in C umgesetzt werden. Dazu zählen Zustandsautomaten in den Prozessen und die Operationen in den Transitionen. Einige Konstrukte in den SDL Prozessen sind hingegen auf eine Unterstützung durch das Laufzeitsystem angewiesen. Dazu gehört Funktionalität, wie das Versenden eines Signals, das Aufsetzen eines Timers oder das dynamische Erzeugen eines Prozesses. Dabei werden meistens zusätzlich Datenstrukturen benötigt, die neben den Parametern für die Realisierung der Semantik, Informationen über das System beinhalten.

Ein Beispiel hierfür ist ein SDL Timer, der aufgesetzt werden soll. Der zugehörigen Funktion muss als Parameter eine Referenz auf den entsprechenden Timer übergeben werden und weiterhin mitgeteilt werden, zu welchem Zeitpunkt der Timer ein Signal erzeugen soll, an wen dieses Signal geschickt wird und welche Parameter es trägt. Die Verarbeitung dieser Informationen ist implementierungsspezifisch und kann in abstrakten Datentypen versteckt werden. Diese müssen dann von Metafunktionen in konkrete Datentypen der Implementierung umgewandelt werden.

```
signal_t sdl_signal_receive(void);
void sdl_signal_send(sdl_pid_t pid,sdl_signal_t s);
sdl_time_t sdl_now(void);
void sdl_timer_set(sdl_timer_t t, sdl_time_t at,
                   sdl_signal_t s, sdl_pid_t pid);
int sdl_timer_active(sdl_timer_t timer);
```

Abbildung 9.3.: C Funktionen mit systemspezifischer Implementierung

9.2.2. Umsetzung als Metaprogramme

Eine Umsetzung in MetaC mit Hilfe von Metafunktionen funktioniert zweistufig. Die eine Ebene repräsentiert das SDL-Laufzeit und -Systemmodell, wie es im letzten Teilabschnitt erläutert wurde. Die zugehörigen Metafunktionen müssen für die verschiedenen Systemarchitekturen implementiert werden. Die andere Ebene besteht aus MetaC Quelltext, der diese Metafunktionen aufruft und vom Codegenerator entsprechend dem Systemmodell generiert wird.

Diese MetaC Funktionen werden zur Kompilierzeit ausgeführt. Wie oben bereits erwähnt wurde, ersetzen sie die abstrakten Datentypen durch konkrete und realisieren darüber hinaus auch Teile der Laufzeitinfrastruktur. Dabei können die Fähigkeiten von MetaC zur Rekonfiguration von Sourcecode eingesetzt werden, um zusätzliche Funktionalität in der Implementierung zu realisieren. Insbesondere kann dadurch die Qualität und Effizienz der Umsetzung verbessert werden, indem die speziellen Eigenschaften des Zielsystems berücksichtigt werden. Beispiele hierfür sind besondere Kommunikations- und Synchronisationsmechanismen des Zielsystems. Darüber hinaus ist es möglich Maßnahmen zu integrieren, die die Entwicklung beschleunigen und vereinfachen. Beispiele hierfür sind Mechanismen zur Protokollierung von Signalübertragungen oder dem Auftreten von Timerereignissen.

Die prototypische Evaluierung dieses Konzepts basierte unter anderem auf den Metafunktionen, die in Abbildung 9.2 dargestellt sind. Der SDL-Compiler, der im Rahmen des interdisziplinären Projekts von Herrn Dörfel [32] entstanden ist, benutzt diese Metafunktionen. Zur Kompilierzeit wird durch Ausführung dieser Metafunktionen C Quelltext erzeugt, der die Struktur des SDL Systems auf der Zielplattform nachbildet. Die in Abbildung 9.3 dargestellten Funktionen zur Anbindung der Laufzeitumgebung werden hingegen vom SDL Codegenerator direkt benutzt. Trotzdem können sie unter Umständen erst zur Kompilierzeit von MetaC Funktionen implementiert oder rekonfiguriert werden, wenn dadurch Optimierungen möglich sind.

9.3. Pfadoptimierte Implementierung

Bei der Auflösung einer Abstraktionsebene mit Hilfe von MetaC kann im Gegensatz zu herkömmlichen Verfahren eine Pfadoptimierung durchgeführt werden. Dazu wird die Tatsache genutzt, dass die bereitgestellten abstrakten Funktionen entsprechend der verwendeten Parameter unterschiedlich implementiert oder gebunden werden können. Dadurch ist es möglich, Entscheidungen, die sonst zur Laufzeit gefällt werden, auf den Zeitpunkt der Kompilierung vorzuziehen. Dies wird im Folgendem an einem Beispiel konzeptionell erläutert, das aus dem Bereich der RTOS Abstraktion abgeleitet ist. Auf eine konkrete Beschreibung der Implementierung wird an dieser Stelle verzichtet, da diese keine we-

Primitiv	Basisfunktion	Laufzeitoptionen
POSIX-Semaphore	atomare Zählvariable, blockiert den Aufrufer wenn Null	Interprozess Unterstützung, zeitbegrenztes Warten, Abbruchpunkt für Threads
POSIX-Mutex	atomare binäre Variable, blockiert den Aufrufer wenn Null	Fehlerüberprüfung, rekursives Locking, Interprozess Unterstützung, Fehlertoleranz
RTEMS Semaphore	atomare Zählvariable, blockiert wenn Null	zeitbegrenztes Warten
MicroC/OS-II Semaphore	atomare Zählvariable, blockiert wenn Null	keine

Tabelle 9.1.: Synchronisationsprimitive von verschiedenen APIs

sentliche Zusatzinformation liefert. Die selben Prinzipien und Konzepte sind äquivalent bei einer Abstraktion von domänenspezifischer Semantik anwendbar.

Zur Erläuterung wird die Verwendung von RTOS Primitiven zum Resourcenmanagement betrachtet. Die gängigsten Variante, um begrenzte Resourcen effizient zu verwalten, die von konkurierende Tasks benutzt werden sollen, ist der Einsatz von Semaphoren. Berücksichtigt man dabei, die verschiedenen Implementierungsvarianten und spezifischen Vor- und Nachteile der Verfahren, so ergibt sich ein Optimierungspotential, das zur Kompilierzeit genutzt werden kann. Dazu wird im speziellen die Semantik der entsprechenden POSIX Konstrukte betrachtet, die in Tabelle 9.1 zusammengefasst sind.

Aus der Liste der optionalen Funktionen ist erkennbar, dass zur korrekten Implementierung des POSIX APIs eine Reihe von Überprüfungen zur Laufzeit durchgeführt werden muss. Am extremsten wirkt sich hierbei aus, dass Operationen, die auf Semaphore wirken gleichzeitig als Abbruchpunkt in dem zugehörigen Thread implementiert werden müssen. Bei den leichtgewichtigeren Mutexes wirkt sich negativ aus, dass es eine Vielzahl von Optionen gibt, die zur Laufzeit aktiviert werden können. Somit muss die Implementierung diese Optionen bei jeder Operation neu prüfen. Dies wirkt sich nicht nur auf die Codegröße, sondern auch auf das Laufzeitverhalten negativ aus. Eine Nachimplementierung der Semaphore Funktionalität ist somit zwar mit Hilfe von Mutexes denkbar, bringt unter Umständen aber auch keinen deutlichen Vorteil.

Bei Implementierungen auf leistungsstarken Arbeitsplatzrechnern oder Applikationsservern spielt der genannte Overhead eine untergeordnete Rolle. Auf eingebetteten Systemen mit begrenztem Speicher und reduzierter Rechenleistung kann dieser Overhead hingegen die Kosten unangenehm in die Höhe treiben. Insbesondere wenn die Zusatzoptionen des POSIX APIs nicht genutzt werden, entsteht schnell der Wunsch, auf eine effizientere Implementierung umstellen zu können. Eine denkbare Variante wäre die native Implementierung eines RTOS.

MicroC/OS-II [51] bietet beispielsweise eine eigene Variante an, die auf den komplexen Code zur Realisierung eines asynchonen Taskabbruchs verzichtet. Auch auf die codeintensive Unterstützung zum zeitbegrenzten Warten wird verzichtet. Die Semaphore Implementierung von RTEMS hingegen unterstützt einen Timeout Mechanismus, verzichtet allerdings gleichermaßen auf den speziellen Abbruchcode. Ansonsten sind die Implementierungen semantisch kompatibel mit der Spezifikation der POSIX Semaphore. Darüber

hinaus ist es möglich, sie als Substitut für den POSIX Mutex einzusetzen. Dabei führt die native Variante zur Laufzeit keine Überprüfungen aus, ob die optionalen konfigurierbaren Tests durchgeführt werden sollen. Somit sind Implementierungen, die die nativen Verfahren benutzen, mit geringerem Laufzeitoverhead und Quellcode verbunden.

Eine Umstellung auf das effizientere native API ist bei einer CASE-Tool basierten Implementierung mit automatischer Codegenerierung nicht möglich, sofern es der Codegenerator nicht explizit unterstützt. Bei einer MetaC basierten Targetierung können solche Randbedingungen jedoch berücksichtigt werden. Wenn die Applikation beispielsweise die Funktionalität zum vorzeitigen Abbrechen einer Task nicht benötigt oder die Entwurfsmethodik diese Eigenschaft gar nicht nutzt, ist es sinnvoll, auf eine alternative Semaphore Implementierung auszuweichen. Diese müsste die zugehörigen Operationen nicht als Abbruchpunkt unterstützen und könnte dadurch deutlich effizienter realisiert werden.

Gleiches gilt auch für die Verwendung von Mutexes und alle anderen Synchronisations- und Kommunikationsprimitive. Durch den Einsatz von effizienten targetspezifischen Funktionen lassen sich die Ausführungspfade im Code optimieren und die Codegröße reduzieren. Dabei sinkt nicht nur die Anzahl der Zeilen an Applikationsquelltext, sondern das Betriebssytems auf dem Target kann ohne schwergewichtige Wrapperlibraries für das POSIX API auskommen. Letztendlich kann so durch den Einsatz von MetaC zur Systemabstraktion eine effizientere Implementierung realisiert werden. Ohne MetaC ist eine vergleichbare Realisierung nur durch kostenintesive und zeitaufwändige Handarbeit erreichbar.

10. Zusammenfassung

Das Ergebnis dieser Arbeit ist eine vollständige Spracherweiterung einer existierenden komplexen Programmiersprache mit breiter Marktakzeptanz, zu einer Metasprache. Die vorgenommenen Erweiterungen ermöglichen detaillierte Analysen und querschneidende Modifikationen von C Quelltexten. Entsprechend der ursprünglichen Motivation kann MetaC dadurch eingesetzt werden um den Quelltext der Software von eingebetteten Systemen zu rekonfigurieren, zu refaktorisieren und die Einhaltung von Regeln zu überprüfen. Dadurch kann die Wiederverwendung von existierendem Quelltext verbessert werden und der Software Entwicklungsprozess lässt sich durch die Möglichkeit der einfachen Rekonfiguration flexibler gestalten. Diese angedachten Problemstellungen aus der Domäne der eingebetteten Systeme wurden untersucht, und die Spracherweiterungen von MetaC haben sich in diesen Einsatzgebieten als nutzbringend erwiesen.

In diesem abschließenden Kapitel wird zunächst ein Ausblick auf eine mögliche logische Fortsetzung der Arbeit gegeben. Im zweiten Teil folgt eine finale Zusammenfassung der wichtigsten Ergebnisse dieser Arbeit.

10.1. Ausblick

Die angedachten und untersuchten Anwendungsszenarien sind nicht gleichbedeutend mit den Grenzen des Einsatzbereiches von MetaC. Der hohe Grad an Flexibilität und die fehlende Domänenbindung der Spracherweiterung ermöglichen es sie auch in anderen Bereichen zu verwenden, die in dieser Arbeit nicht explizit untersucht wurden. Dazu ist es unter Umständen sinnvoll den Sprachschatz von MetaC noch zu erweitern, um die Implementierung von bestimmten Metaprogrammen zu vereinfachen. Die nachfolgenden Teilabschnitte geben einen Überblick über denkbare Erweiterungen und Einsatzszenarien, die nicht explizit untersucht wurden, da sie im Umfeld der Softwareentwicklung für eingebettete Systeme keinen unmittelbaren Nutzen mitsichbringen.

10.1.1. Weitergehende Spracherweiterung

Der in dieser Arbeit beschriebene Stand der Spracherweiterung MetaC stellt einen umfangreichen Satz an Funktionalität zur Verfügung. Darüber hinaus sind noch zusätzliche Erweiterungen denkbar, die in bestimmten Situationen die Realisierung von Metaprogrammen zur Quelltextrekonfiguration vereinfachen können. Diese Zusatzfunktionalität ist keine zwingende Voraussetzung zur Lösung von spezifischen Problemen, sondern erlaubt den Aufwand der Implementierung bestimmter Metaprogramme zu verringern. Die folgenden Beispiele denkbarer Erweiterung sind lediglich Vorschläge zu einer möglichen Fortsetzung der Arbeit. Darüber hinaus könnte auch Funktionalität in die Sprache eingebracht werden, die hier nicht explizit aufgeführt ist.

Eine mögliche Erweiterung von MetaC besteht in einem Ausbau der Funktionalität, die durch den `typeof` Operator angeboten wird. Dieser erlaubt es Datentypen auf der Basis einer Metavariable vom Typ `type` zu konstruieren und zu modifizieren. Dabei ist es

bisher nicht möglich, Freiheitsgrade in strukturierte Datentypen einzubringen. Die dafür notwendigen Erweiterungen lassen sich direkt aus den existierenden ableiten. Die semantischen Implikationen sind mit den zuvor besprochenen Implikationen der Funktionalität des `typeof` Operators verwandt.

Um diese Erweiterung durchzuführen sind zusätzliche Parser notwendig, die die Grammatik für abgeleitete Typdefinitionen innerhalb von strukturierten Datentypen erkennen. Diese Anforderung entsteht aus der Tatsache, dass Variablen innerhalb von Datenstrukturen anderen Einschränkungen unterliegen und zusätzliche Eigenschaften gegenüber normalen Variablen mitbringen. So ist es beispielsweise nur bei Strukturvariablen möglich, die Anzahl der verwendeten Bits zur Speicherung des Objektes zu spezifizieren. Eine äquivalente Funktionalität für normale Variablen gibt es nicht, wodurch sich die unterschiedliche Grammatik bei ansonsten ähnlicher Struktur begründet.

Eine weitere Variante die Ausdrucksstärke von MetaC zu verbessern, ist durch eine Semantikerweiterung von globalen Metavariablen gegeben. Erlaubt man die Verwendung von Metavariablen in normalen C Quelltext, so kann dazu eine Semantik definiert werden, die äquivalent zu einem normalen Strukturmuster interpretierbar ist. Die Synthese des konkreten Quelltextes kann bei dieser Variante Metavariablen zu verwenden, vom letzten Wert der Metavariablen abgeleitet werden. Das bedeutet, dass eine implizite Synthese nach Abarbeitung des Metaprogramms durchgeführt wird.

Ein solcher Ansatz entspricht vom Programmierparadigma eher den Verfahren, die von der C++ Template Programmierung bekannt sind, da die generierte Codestruktur implizit von anderen Definitionen abhängen und keiner expliziten Beschreibung unterliegen. Deshalb wurde im Rahmen dieser Arbeit auf die Untersuchung einer solchen Methodik verzichtet, da sich diese nicht nahtlos in die Programmierparadigmen der eingebetteten Systeme integrieren lässt. Abgesehen davon erscheint diese Idee ein großes Potential für bestimmte Szenarien zu bieten, in denen ähnliche Varianz im Quelltext wiederkehrend genutzt werden soll und die Positionen vorab bekannt sind. Dadurch erübrigt sich die Notwendigkeit, die relevanten Stellen im Quelltext durch ein Metaprogramm zu ermitteln, und das Metaprogramm kann darauf reduziert werden, die notwendige Struktur des zu instantiierenden Quelltextes zu konstruieren.

10.1.2. Zukünftige Applikationen

Der Nutzen einer Sprache ist entscheidend von ihren Anwendungsbereichen abhängig. MetaC lässt sich in vielen Teilbereichen des Softwareentwurfs für eingebettete Systeme, aber auch für normale Applikationen einsetzen. Einige Beispiele wurden hierfür bereits genannt und die zugehörigen Konzepte analysiert. Beim Entwurf einer Programmiersprache besteht in der Regel eine klare Zielsetzung, die erfüllt werden muss. Das gilt für MetaC, wie für jede andere Computersprache. Allerdings können beim Sprachentwurf nicht alle Einsatzszenarien vorausgesehen werden können. Entsprechend sind die vorgestellten Applikationen als eine Untermenge der möglichen Anwendungsbereiche aus der Domäne der eingebetteten Systeme zu sehen und folglich auch als eine Teilmenge der Verwendungsmöglichkeiten in anderen Domänen. Außerdem lassen sich viele typische Anwendungen von Metaprogrammen ebenso mit MetaC basierten Metaprogrammen lösen.

Ein denkbares Einsatzgebiet von MetaC ist die Optimierungen von C-Programmen zur Kompilierzeit. Die notwendigen Refaktorisierungsmaßnahmen lassen sich durch eine detaillierte Analyse von Kontroll- und Datenfluss sowie dem zusätzlichen Applikationswis-

sen des Entwicklers erreichen. Die für diese Methoden notwendigen Algorithmen zur Ermittlung des Datenfluss im Quelltext eines Programmes sind im Compilerbau gängige Verfahren. Diese wurden jedoch aufgrund des enormen Aufwandes nicht im MetaC Compiler implementiert und entsprechend diesem Applikationsfeld keine besondere Aufmerksamkeit gewidmet.

Insbesondere lassen sich aus diesem Ansatz keine allgemein gültigen Verfahren ableiten, sondern lediglich applikationsspezifische Lösungen, die sich nicht ohne weiteres auf andere Szenarien übertragen lassen. Grund hierfür ist, dass ein Verfahren zur Nutzung von Optimierungspotential, das spezifische Eigenschaften einer Applikation nutzt, naturgemäß nicht zu einer wiederverwendbaren Methodik verallgemeinert werden kann, da die zur Optimierung verwendbaren Eigenschaften zwischen den Applikationen zu stark variieren. Ein mögliches Beispiel ist ein Metaprogramm zur Verkürzung von Datenflusspfaden in Protokollstapeln. Hierbei ist das Optimierungspotential von dem Protokoll und den Eintrittspunkten zur Verarbeitung abhängig.

Optimierungsverfahren die auf diesem Konzept basieren, sind in den Bibliotheken anderer Sprachen heute schon gängige Praxis. Dabei kommen Sprachkonstrukte zum Einsatz, die in C nicht existieren. Bei der Verwendung von MetaC stehen alternative Sprachmittel bereit, die ähnliches bewerkstelligen können. Dadurch erweitert sich der mögliche Einsatzbereich der erarbeiteten Methodik weit über das ursprüngliche Ziel hinaus.

Applikationsspezifisches Wissen kann ebenso bei der Verwendung von komplexen Mathematischen Funktionen Optimierungspotential bereithalten. Denn viele mathematische Operationen wie Tangens und Logarithmus können von einem C-Compiler nicht ohne weiteres bei mehrfachem Aufruf mit dem gleichen Wert wegoptimiert werden. Die Ursache dafür ist, dass das Ergebnis dieser Funktionen nicht nur ein Rückgabewert ist, sondern auch Fehlervariablen verändert werden. Durch eine gezielte Datenflussanalyse ist es jedoch möglich nachzuweisen, dass ein weiterer Funktionsaufruf mit dem selben Eingabewert überflüssig ist.

Ein solcher Nachweis ist mit Hilfe der Reflexiven Infrastruktur von MetaC denkbar. Dazu müssen zum einen die verwendete Variablen und zum anderen der Kontrollfluss des Programms betrachtet werden. Bei einem erfolgreichem Beweis eines überflüssigen Funktionsaufrufes könnte dieser dann durch den gespeicherten Wert eines vorangegangenen Aufrufes mit gleichen Eingabedaten ersetzt werden. Solche Optimierungen können durch die besondere Betrachtungsweise von MetaC auch Funktionsgrenzen übergreifen und somit Optimierungspotential freilegen, das beim Einsatz eines normalen C-Compilers ungenutzt bleibt. Insbesondere die Möglichkeit applikationsspezifisches Wissen in ein Metaprogramm einzubringen, das eine solche Optimierung durchführt könnte das Ergebnis gegenüber Standardverfahren verbessern.

Die Portierung einer Applikation von einem Echtzeitbetriebssystem zu einem anderen scheitert oftmals an dem verwendeten nativen API des entsprechenden RTOS, wenn dieses auf dem neuen RTOS nicht zur Verfügung steht. In solchen Fällen, in denen die Notwendigkeit der Portierbarkeit nicht apriori als notwendige Eigenschaft der Software erkannt wurde, kann es zu einer Kostenexplosion kommen, wenn die Portierung unumgänglich ist. Mit Hilfe von MetaC ist eine Portierung denkbar, die den Quelltext an ein anderes API automatisch anpasst, sofern beide Schnittstellen eine ähnliche Semantik bereitstellen oder diese nachgebildet werden kann.

Dazu müssen entsprechende Suchmuster definiert werden, die die Stellen im Quelltext ausfindig machen können, an denen das API des alten RTOS benutzt wird. Durch eine Analyse der übergebenen Parameter können dann entsprechende äquivalente Sys-

temaufrufe basierend auf dem neuen API eingefügt werden und so die ursprünglichen Systemaufrufe ersetzt werden. Insbesondere kann so auch automatisch ermittelt werden, ob ein Programm überhaupt mit den primitiven eines alternativen RTOS auskommt oder ob Zusatzaufwand zur Nachbildung von Funktionalität notwendig ist.

10.2. Ergebnisse

Besonders hervorhebenswert ist die Spracherweiterung MetaC, auf der alle anderen in den voran gegangenen Kapitel erläuterten Methoden basieren. Die notwendigen Konzepte um C Quelltext rekonfigurieren und refaktorisieren zu können, benötigen eine Reihe von sprachlichen Konstrukten, die neu eingeführt wurden. Diese werden im nächsten Teilabschnitt zusammengefasst. Die Applikation der Sprache MetaC, die dazu notwendigen Methoden und der daraus resultierende Nutzen werden daraufhin festgehalten.

10.2.1. Erweiterung von C zu MetaC

Die Integration von reflexiven Mechanismen ist eine grundlegende Voraussetzung zur applikationsspezifischen Modifikation von Quelltext. Die wichtigste Grundlage zur Realisierung entsprechender Methoden, ist die Definition von passenden Metadatentypen, die es erlauben unterschiedliche Bereiche im Quelltext zu referenzieren. Dies beinhaltet insbesondere die Berücksichtigung der spezifischen Eigenschaften und semantischen Bausteine der Programmiersprache C.

Die eingeführten Erweiterungen erlauben es einerseits mit Variablen der Metadatentypen Assoziation zu bestimmten Knoten im Parsebaum zu knüpfen. Für die Definition der entsprechenden Metadatentypen werden in MetaC die Rekursionspunkte der Grammatik von C verwendet, um die Anzahl der notwendigen Datentypen auf ein Minimum zu reduzieren. Andererseits müssen ebenso die nicht-rekursiven und die semantischen Strukturen ohne explizite Repräsentanz im Quelltext abstrahiert werden. Dies erfordert die Einführung von Metadatentypen für Gültigkeitsbereiche und Variablen. Nicht zuletzt werden auch gewöhnliche arithmetische Datentypen für Gleitkomma- und Ganzzahlberechnungen benötigt. Die Integerdatentypen sind für die Schleifenbildung in Metaprogrammen unersetzlich und können darüber hinaus zur Synthese von Konstanten im Quelltext herangezogen werden.

Mit den neuen Datentypen allein lassen sich zwar einige Analysen durchführen, doch eine Instantiierung von Quelltext wird dadurch noch nicht ermöglicht. Deswegen ist zusätzlich das Konzept des Quelltext-Strukturmusters notwendig, das in Verbindung mit seiner sechsphasigen Instantiierung unter Verwendung von Metadaten zu einer sehr flexiblen Synthese von Quelltext benutzt werden kann. Durch eine entsprechende nach Metadatentyp differenzierte Reinterpretation der Variablen können Strukturmuster außerdem zur Suche von bestimmten Abschnitten im Quelltext herangezogen werden. Bei beiden Einsatzgebieten verbessert die Möglichkeit der Rekursion die Flexibilität erheblich, und die Erzeugung von und die Suche nach hochspezialisiertem Quelltext aus sehr generischen Strukturen wird einfach realisierbar. Somit ergibt sich in der Definition der QSM eine Symmetrie, die das Konzept zu einer abgeschlossenen Methodik abrundet.

Die neu eingeführten Konzepte sind transparent in die existierende Syntax von C integriert worden. Dabei ist die uneingeschränkte Rückwärtskompatibilität von Syntax und statischer Semantik das wichtigste Ergebnis. Ebenso lehnt sich die neu eingeführte dynamische Semantik für Metaprogramme eng an das Laufzeitverhalten von normalen C

Programmen an. Die Kombination dieser Tatsachen bewirkt, dass MetaC alle Eigenschaften von C erbt, die sich dem Programmierer als Look and Feel darbieten. Dadurch kann gleichermaßen das Programmierparadigma beim Umstieg von C auf MetaC übernommen werden, was den Einstieg deutlich vereinfacht.

10.2.2. Methoden und Applikation

Die wichtigsten Instrumente für die Modifikation von Quelltext sind die zuvor zusammengefassten Mechanismen zur Reflexion und Codeinstantiierung. Sie ermöglichen es dem Metaprogrammierer sein Metaprogramm so zu gestalten, dass Veränderungen in Abhängigkeit zum Applikationsquelltext vorgenommen werden können. Dabei lässt sich durch durch die Möglichkeit, Expertenwissen in das Metaprogramm einzubringen, das Potential applikationsspezifische Verbesserungen zu erreichen noch weiter erhöhen. Durch diesen Ansatz entsteht eine Art Whitebox View des Quelltextes eines C Programmes, die dazu genutzt werden kann, konfigurationsspezifische Abhängigkeiten zu erkennen. Dadurch kann eine bedarfsgesteuerte Anbindung von Funktionen der Infrastruktur an den Applikationscode realisiert werden.

Entsprechend den Anforderungen der Applikation ist es möglich, Entscheidungen die zur Laufzeit unveränderlich sind, auf den Zeitpunkt der Kompilierung vorzuziehen. Dazu ist die Auswahl eines entsprechenden APIs erforderlich, das eine Konfiguration dieser Parameter zur Kompilierzeit der Applikation vorsieht. Im Gegensatz zu Applikationsschnittstellen wie POSIX, die eine flexible Laufzeitparametrierung erlauben, kann so eine effizientere, den Anforderungen angepasste Implementierung realisiert werden. Dadurch können sowohl die Anforderungen an die bereitgestellte Rechenleistung als auch an den verfügbaren Hauptspeicher gesenkt werden.

Dieses Anwendungsszenario ist speziell für Entwicklungsumgebungen mit automatischer Codegenerierung ausgelegt. MetaC kann aber ebenso in Umgebungen ohne automatische Codegenerierung eingesetzt werden. Dabei kommen insbesondere jene Verfahren in Frage, die zur Unterstützung von Debug und Test verwendet werden können. Besonders die Fähigkeit zur Integration von querschneidender Funktionalität in beliebigen Quelltext ist hierbei von großer Bedeutung, da dazu keine besonderen Vorkehrungen notwendig sind, die vorab getroffen werden müssen. Somit kann das Verfahren auch auf Applikationscode angewendet werden, in dem die Integration von querschneidender Funktionalität vom Design ausgehend nicht vorgesehen wurde. Diese den Entwicklungsprozess unterstützenden Methoden können unbeschränkt sowohl in CASE-Tool basierten Abläufen, als auch in Prozessen ohne automatische Codegenerierung eingesetzt werden.

Die in dieser Arbeit untersuchten Verfahren zur Instrumentierung für Echtzeitmessungen gehören zu dieser besonderen Kategorie von Applikationen. Insbesondere auf eingebetteten Systemen ohne spezielle integrierte Debughardware, sind zudem Softwareimplementierungen zur Überwachung des Kontrollflusses oder des Wertebereichs von Variablen, die als Kompilierzeitoption eingefügt werden können, von großem Wert. Durch die Flexibilität, die durch eine Softwarerealisierung gegeben ist, kann die unterstützende Analysefunktionalität anforderungsspezifisch angepasst werden. So kann dieses Verfahren mitunter extrem teure Spezialhardware ersetzen und eignet sich für unterschiedliche Zielsysteme gleichermaßen und erfüllt somit alle wesentlichen Anforderungen zum Einsatz in komplexen Szenarien zur Entwicklung von Software für eingebettete Systeme.

A. Anhang

A.1. Typumwandlungen der MetaC Metadatentypen

Eine wichtige Eigenschaft bei der Verwendung von Variablen in C ist die Tatsache, dass sie sich einfach von einem Datentyp in einen anderen umwandeln lassen. Dieser Mechanismus ist in MetaC für die Metadatentypen ebenso verfügbar. Dabei wird allerdings nicht das Bitmuster der Variablen einer neuen Interpretation unterzogen und auf einen anderen Speicherbereich abgebildet. Dies ist so in MetaC nicht möglich, da die Datentypen keine definierte binäre Darstellung im Speicher besitzen. Stattdessen werden die Zusatzinformationen zur Typwandlung genutzt, die implizit durch die Assoziationen der Metadatentypen untereinander gegeben sind. Durch eine entsprechende Wandlung werden damit die Metadaten sichtbar gemacht, die durch die Verknüpfungen enthalten sind.

So kann es eine Deklaration einer Funktion geben, die über eine Metavariable vom Type `symb` referenziert wird und deren Name `a` sich durch eine Typumwandlung zu `ident` lesen lässt und äquivalent der Datentyp `void (*)(int)` durch eine Wandlung mit `type`. Alle diese Informationen sind implizit durch Wandlungen des Metadatentyps `symb` erreichbar. Entsprechendes gilt ebenso für alle anderen Metadatentypen. Die möglichen Wandlungen sind in den nachfolgenden Tabellen der Vollständigkeit halber aufgeführt. Implizit sind diese Information auch aus Abbildung 5.2 im Abschnitt 5.4 ersichtlich.

Zieltyp	Semantik
zed	keine Umwandlung
expr	*Semantik von cast nach* `expr` *ist entsprechend* expression-template
func	-
ident	-
real	Typumwandlung entsprechend `int` nach `double` in C
scope	-
stmt	-
strg	eine textuelle Repräsentation der Zahl als String-Literal
symb	-
type	-

Tabelle A.1.: Typumwandlungen vom Metadatentyp `zed`

Zieltyp	Semantik
zed	Typumwandlung entsprechend **`float`** nach **`int`** in C
expr	*Semantik entsprechend* expression-template
func	-
ident	-
real	keine Umwandlung
scope	-
stmt	-
strg	eine textuelle Repräsentation der Zahl als String-Literal
symb	-
type	-

Tabelle A.2.: Typumwandlungen vom Metadatentyp **`real`**

Zieltyp	Semantik
zed	-
expr	*Semantik entsprechend* expression-template
func	-
ident	keine Umwandlung
real	-
scope	-
stmt	-
strg	Inhalt der Variable in Anführungszeichen
symb	-
type	-

Tabelle A.3.: Typumwandlungen vom Metadatentyp **`ident`**

Zieltyp	Semantik
zed	entspricht einer Umwandlung mit `sscanf`
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	-
ident	der Name der Funktion
real	entspricht einer Umwandlung mit `sscanf`
scope	-
stmt	-
strg	keine Umwandlung
symb	-
type	entspricht immer `typeof(const char const *)`

Tabelle A.4.: Typumwandlungen vom Metadatentyp **`strg`**

Zieltyp	Semantik
zed	-
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	die Funktion, die das Symbol referenziert oder null
ident	der Name des Symbols
real	-
scope	der Namensraum in dem das Symbol definiert ist
stmt	-
strg	die textuelle Repräsentation des Symbolnamen als String-Literal
symb	keine Umwandlung
type	der Datentyp des Symbols

Tabelle A.5.: Typumwandlungen vom Metadatentyp **`symb`**

Zieltyp	Semantik
zed	-
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	keine Umwandlung
ident	der Name der Funktion
real	-
scope	der Namensraum der Funktion
stmt	der Körper der Funktion, sofern die Funktionsdefinition verfügbar ist
strg	eine textuelle Repräsentation der Funktion als String-Literal
symb	das Symbol ohne die explizite Assoziation zur Funktionsdefiniton
type	der Datentyp der Funktion

Tabelle A.6.: Typumwandlungen vom Metadatentyp **`func`**

Zieltyp	Semantik[1)]
zed	-
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	-
ident	-
real	-
scope	-
stmt	-
strg	erzeugt eine textuelle Repräsentation des Ausdrucks
symb	das durch den Ausdruck direkt referenzierte Symbol, sonst Null
type	der Typ des Ausdrucks

Tabelle A.7.: Typumwandlungen vom Metadatentyp **`expr`**

Zieltyp	Semantik
zed	-
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	die Funktion zu der die Anweisung gehört oder Null für QSM
ident	-
real	-
scope	der Gültigkeitsbereich in der die Anweisung liegt oder Null für QSM
stmt	keine Umwandlung
strg	erzeugt eine textuelle Repräsentation der Anweisung
symb	-
type	-

Tabelle A.8.: Typumwandlungen vom Metadatentyp **`stmt`**

Zieltyp	Semantik
zed	-
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	-
ident	-
real	-
scope	-
stmt	-
strg	eine textuelle Repräsentation des Typs als String-Literal
symb	-
type	keine Umwandlung

Tabelle A.9.: Typumwandlungen vom Metadatentyp **`type`**

Zieltyp	Semantik
zed	-
expr	*Semantik von cast nach* **`expr`** *ist entsprechend* expression-template
func	gibt die Funktion zurück, in der der Namensraum liegt oder Null
ident	-
real	-
scope	keine Umwandlung
stmt	gibt das *statement* zurück, das den Namesraum definiert oder Null
strg	-
symb	-
type	-

Tabelle A.10.: Typumwandlungen vom Metadatentyp **`scope`**

A.2. Tabellen der internen Metafunktionen

```
meta void $debug(strg);
meta void $error(strg);
meta void $fatal(strg);
meta void $info(strg);
meta void $warn(strg);
```

Abbildung A.1.: Metafunktionen für Statusmeldungen

```
meta expr *$find_expr(expr p, expr w, expr *m);
meta stmt *$find_stmt(stmt p, stmt w, stmt *m);
meta zed $match_expr(expr p, expr w);
meta zed $match_stmt(stmt p, stmt w);
```

Abbildung A.2.: Metafunktionen zum Suchen von Quelltextstrukturen

```
meta zed $accesses(expr e, symb s);
meta stmt $branch_creator(stmt);
meta zed $has_cfbranch(expr);
meta zed $is_branch(stmt);
meta zed $is_compound(stmt);
meta zed $is_const(symb);
meta zed $is_do(stmt);
meta zed $is_else_body(stmt);
meta zed $is_for(stmt);
meta zed $is_if(stmt);
meta zed $is_if_body(stmt);
meta zed $is_volatile(symb);
meta zed $is_while(stmt);
```

Abbildung A.3.: Prädikative Metafunktionen

```
meta void $before_branch(stmt what, stmt where);
meta void $begin(stmt what, stmt where);
meta symb $define(scope, type, ident, ...);
meta symb $define_type(type, ident);
meta stmt $end(stmt what, stmt where);
meta func $implement(symb, stmt);
meta void $include(strg);
meta void $insert_after(stmt what, stmt where);
meta void $insert_before(stmt what, stmt where);
```

Abbildung A.4.: Metafunktionen zur Quelltextinstantiierung

```
meta stmt *$get_all_branches(func, stmt *);
meta func *$get_all_functions(func *f);
meta expr $get_argument(expr, zed);
meta stmt $get_body(func);
meta stmt *$get_children(stmt, stmt *);
meta expr $get_expr(stmt);
meta expr $get_for_init(stmt);
meta stmt *$get_all_stmts(stmt *, func);
meta expr $get_iterator(stmt);
meta void $get_returns(func, stmt *);
meta expr $get_return_expr(stmt);
meta type $get_return_type(func);
meta stmt *$get_stmts(stmt *, symb);
meta symb $get_symbol(ident);
meta void $get_usages(expr *, symb);
meta void $get_users(func *, symb);
meta void $get_variables(func, symb *);
meta scope $global(void);
meta zed $line_of(stmt);
meta stmt $parent(stmt);
meta stmt $predecessor(stmt);
```

Abbildung A.5.: Metafunktionen zur Quelltextanalyse

```
meta void $add_code(func f, stmt s);
meta void $add_epilogue(func f, stmt s);
meta void $add_prologue(func, stmt);
meta void $make_static(symb);
meta void $remove(stmt);
meta void $reset_type(symb, type);
meta void $resize_array(stmt, zed);
meta void $set_return_expr(stmt, expr);
```

Abbildung A.6.: Metafunktionen zur Quelltextmodifikation

A.3. Syntax von C nach ISO9899:1999

Die in Klammern gesetzten Ziffern referenzieren die Abschnitte der Definitionen der entsprechenden Parser im ISO Standard.

```
(6.5.1) primary-expression:
    identifier
    constant
    string-literal
    (expression )
(6.5.2) postfix-expression:
    primary-expression
    postfix-expression [expression ]
    postfix-expression (argument-expression-listopt )
    postfix-expression . identifier
    postfix-expression -> identifier
    postfix-expression ++
    postfix-expression --
    ( type-name ) { initializer-list }
    ( type-name ) { initializer-list , }
(6.5.2) argument-expression-list:
    assignment-expression
    argument-expression-list , assignment-expression
(6.5.3) unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-name )
(6.5.3) unary-operator: one of
    & * + - ~ !
(6.5.4) cast-expression:
    unary-expression
    ( type-name ) cast-expression
(6.5.5) multiplicative-expression:
    cast-expression
    multiplicative-expression * cast-expression
    multiplicative-expression / cast-expression
    multiplicative-expression % cast-expression
(6.5.6) additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression
(6.5.7) shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression
(6.5.8) relational-expression:
    shift-expression
```

```
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression
(6.5.9) equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression
(6.5.10) AND-expression:
    equality-expression
    AND-expression & equality-expression
(6.5.11) exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ AND-expression
(6.5.12) inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression
(6.5.13) logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && inclusive-OR-expression
(6.5.14) logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression
(6.5.15) conditional-expr[2]:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expr
(6.5.16) assignment-expr[3]:
    conditional-expr
    unary-expression assignment-operator assignment-expr
(6.5.16) assignment-operator: one of
    = *= /= %= += -= <<= >>= &= ^= |=
(6.5.17) expression:
    assignment-expr
    expression , assignment-expr
(6.6) constant-expression:
    conditional-expr
```

[2] Aus drucktechnischen Gründen wird **conditional-expression** zu **conditional-expr** abgekürzt.

[3] Aus drucktechnischen Gründen wird **assignment-exression** zu **assignment-expr** abgekürzt.

Literaturverzeichnis

[1] AspectC++. http://www.aspectc.org/.

[2] AspectJ Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

[3] Blitz++. http://www.oonumerics.org/blitz/.

[4] Boost C++ Libraries. http://www.boost.org.

[5] Freebsd. http://www.freebsd.org.

[6] Java. http://java.sun.com.

[7] Java Community Process. http://www.jcp.org.

[8] Matrix Template Library. http://www.osl.iu.edu/research/mtl/.

[9] OpenC++. http://www.csg.is.titech.ac.jp/ chiba/openc++.html.

[10] The Standard Template Library (STL). http://www.sgi.com/tech/stl/.

[11] *Information technology – Programming Languages – Pascal*. International Standards Organization, 1990.

[12] *X3.159-1989 (Rationale of the C Standard)*. Silicon Press, 1990.

[13] *Information technology – Programming Languages – Extended Pascal*. International Standards Organization, 1991.

[14] *Information technology – Programming Languages – Ada*. International Standards Organization, 1995.

[15] Guidelines For The Use Of The C Language In Vehicle Based Software. *MISRA*, 1998.

[16] Ada Semantic Interface Specification. In *ISO/IEC 15291*. International Standards Organization, 1999.

[17] *Information technology – Programming Languages – C*. International Standards Organization, 1999.

[18] *Programming Languages – C++*. International Standards Organization, 2003.

[19] A. Aho, R. Sethi, and J. Ullmann. *Compiler Bau*. Oldenbourg, 1999.

[20] Aonix. Software Through Pictures. http://www.aonix.com.

[21] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[22] J. Baker. Macros that play: Migrating from Java to Maya. http://www.cs.utah.edu/ jbaker/maya/thesis.html, 2001.

[23] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming, 2001.

[24] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281, New York, NY, USA, 2002. ACM Press.

[25] A. Burns and A. Wellings. Real-Time Systems and Programming Languages, 2000.

[26] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. In *Communications of the ACM Nr.6*, 1966.

[27] S. Chiba. A Metaobject Protocol for C++. In *SIGPLAN Notices 30*, 1995.

[28] W.-N. Chin, S.-C. Khoo, and D. N. Xu. Deriving pre-conditions for array bound check elimination. In *Proceedings of 2001 Symposium on Programs as Data Objects (PADO-II), Aarhus, Danmark*, 2001.

[29] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of ESEC/FSE, ACM*.

[30] K. Czarnecki and U. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison Wesley, 2000.

[31] T. Duff. Tom Duff on Duff's Device. http://www.lysator.liu.se/c/duffs-device.html.

[32] R. Dörfel. Der SDL MetaC Compiler - Automatische Generierung von MetaC Code aus einem SDL Diagramm. IDP-Arbeit am Lehrstuhl für Realzeit-Computersysteme.

[33] J. Engblom and B. Jonsson. Processor pipelines and their properties for static wcet analysis. In *Proceeding of the Second Embedded Software Conference (EMSOFT 02)*. Springer Verlag, 2002.

[34] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides. *Design Patterns*. Addison-Wesley-Longman, 1995.

[35] Esterel Technologies. Esterel. http://www.esterel-technologies.com.

[36] A. Flexeder. Implementierung eines Regelcheckers für Realzeit-Applikationen in C. IDP-Arbeit am Lehrstuhl für Realzeit-Computersysteme, 2005.

[37] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. Aspectc++: Language proposal and prototype implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[38] R. Glück and J. Jørgensen. An automatic program generator for multi-level specialization. In *LISP and Symbolic Computation*, volume 10, pages 113–158, 1997.

[39] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, 1991.

[40] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, Mass., 1996.

[41] H. Herold. *Linux, Unix Profitools*. Addison-Wesley, 1999.

[42] IBM. Rational rose suite. http://www.rational.com.

[43] IEEE and The Open Group. POSIX – IEEE Std 1003.1-2004. *http://www.opengroup.org*.

[44] M. Inc. Matlab/simulink. http://www.mathworks.com.

[45] ITU. Recommendation Z.100 (08/02), 2002.

[46] S. Johnson. Lint, a c program checker. *Computer Science Technical Report 65*, 1977.

[47] S. Johnson. Yacc: Yet another compiler-compiler, 1979.

[48] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[49] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[50] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP*. Springer Verlag, 1997.

[51] J. Labrosse. *MicroC/OS-II – The Real-Time Kernel*. R&D Books, 1999.

[52] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.

[53] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components, 1999.

[54] C. Lopes and G. Kiczales. Recent developments in AspectJ, 1998.

[55] L. Lutz. Implementierung eines regelcheckers für realzeit-applikationen in c. IDP-Arbeit am Lehrstuhl für Realzeit-Computersysteme, 2005.

[56] T. Maier-Komor. MetaC Compiler Implementation and Application examples. http://www.maier-komor.de/metac.

[57] T. Maier-Komor, A. von Bülow, and G. Färber. MetaC and its Use for Automated Source Code Instrumentation of C Programs for Real-Time Analysis. *Proceedings of the Work-in-Progress Session of the 17th Euromicro Conference on Real-Time Systems*, 2005.

[58] F. Marschall and P. Braun. Model transformations for the MDA with botl, 2003.

[59] A. Mitschele-Thiel. Systems Engineering with SDL, 2001.

[60] A. Olsen, O. Færgemand, B. Møller-Pdersen, R. Reed, and J. Smith. SDL-92, 1994.

[61] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[62] S. M. Petters and G. Färber. Making worst case execution time analysis for h ard real–time tasks on state of the art processors feasible. In *6th Int. Conf. on Real–Time Computing Systems and Applications (RTCSA'99)*, Hongkong, ROC, Dec. 13–15 1999. IEEE.

[63] P. Puschner and A. Burns. A review of worst case execution–time analysis (editorial). *Real–Time Systems, 18(2/3)*, pages 115–128, 2000.

[64] C. Raistrick, I. Wilkie, and J. Wright. Model driven architecture and executable uml. *Proceedings of Forum on Specification and Design Languages 2002*, 1, 2002.

[65] Rechenberg and Pomberger. *Informatik-Handbuch*. Hanser, 2002.

[66] P. Rogers and A. Wellings. Openada: Compile-time reflection for ada 95. In A. Llamosi and A. Strohmeier, editors, *Proccedings Software Technologies - Ada Europe*, volume LNCS 3063, pages 166–177. Springer, 2004.

[67] B. Smith. Definition of *reflection*. *Abbendum to the OOPSALA'90 Proceedings*, 1990.

[68] R. Stansifer. *Theorie und Entwicklung von Programmiersprachen*. Prentice Hall, 1995.

[69] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture, The EUROMICRO Journal*, 46:339–355, 2000.

[70] F. Stappert, A. Ermedahl, and J. Engblohm. Efficient longest executable path search for programs with complex flows and pipeline effects. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 132–140, Atlanta, Giorgia, USA, Nov. 16–17 2001.

[71] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. In *Lecture Notes in Computer Science 1826, Reflection and Software Engineering*, pages 117–133. Springer Verlag, 2000.

[72] Telelogic. Tau. http://www.telelogic.com.

[73] A. Tešanović, J. Hansson, D. Nyström, and C. Norström. Integrating symbolic worst-case execution time analysis into aspect-oriented software development. In *Proceedings of OOPSLA 2002 Workshop on Tools for Aspect-Oriented Software Development*, 2002.

[74] A. Tešanović, J. Hansson, D. Nyström, C. Norström, and P. Uhlin. Aspect-level WCET analyzer: a tool for automated wcet analysis of a real–time software composed using aspect and components. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, Polytechnic Institute of Porto, Portugal, July 2003.

[75] E. Unruh. Metaprogramm zur Primzahlberechnung in C++. *ANSI X3J16-94-0075/ISO WG21-462*, 1994.

[76] D. Vandevoorde and N. Josuttis. *C++ Templates*. Addison Wesley, 2003.

[77] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[78] T. L. Veldhuizen. C++ Templates are Turing Complete. http://osl.iu.edu/t̃veldhui/papers/2003/turing.pdf, 2003.

[79] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998.

[80] D. von Rönnebeck. Untersuchung der Implementierungsdetails einer Abbildung von SDL nach C-Quellcode für eingebettete Systeme. IDP-Arbeit am Lehrstuhl für Realzeit-Computersysteme, 2001.