TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Praktische Informatik

Diplomarbeit

Same-Process Debugger für kompilierte domänenspezifische Sprachen
zur digitalen Signalverarbeitung in Echtzeit

Stefan Gränitz

Berlin, den 20. Dezember 2012

Prüfer: Prof. Dr. Gudula Rünger
Betreuer: Dr. Jörg Dümmler, Dr. Norbert Nemec

Gränitz, Stefan
Same-Process Debugger für kompilierte domänenspezifische
Sprachen zur digitalen Signalverarbeitung in Echtzeit
Diplomarbeit, Fakultät für Informatik
Technische Universität Chemnitz, Dezember 2012

Selbstständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der angegebenen Quellen angefertigt zu haben. Diese Diplomarbeit wurde noch keiner Prüfungsbehörde in dieser oder anderer Form vorgelegt.

Berlin, den 20. Dezember 2012

Stefan Gränitz

Inhaltsverzeichnis

1 Einleitung

Das englische Wort *bug* ist als Bezeichnung für einen Defekt oder ein Fehlverhalten einer Maschine in einigen Bereichen des Ingenieurwesens bereits seit dem 19. Jahrhundert umgangssprachlich gebräuchlich. Einer der frühesten schriftlichen Belege dafür findet sich in einem Brief des Erfinders Thomas Edison an seinen Kollegen Puskas vom 13. November 1878 [Hug04].

> It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise – this thing gives out and [it is] then that 'Bugs' – as such little faults and difficulties are called – show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached.

Ein früher Beleg für die Verwendung des Begriffs im Bereich der Informationstechnik ist der berühmte Notizbucheintrag der Computerpionierin Grace Hopper[1] vom 9. September 1947. Nachdem einer der Techniker eine in einem Relais eingeklemmte Motte als Ursache des Fehlverhaltens des Mark II Computers identifiziert hatte, fixierte sie diese mit Klebeband in ihrem Notizbuch und notierte dazu folgenden Eintrag:

> 15:45 Relay #70 Panel F (moth) in relay. First actual case of bug being found.

Die Begriffe *bug* und *debug* verwurzelten sich in der Folgezeit schnell im Wortschatz der frühen Pioniere der Informatik und sind heute allgemein bekannte Fachbegriffe. Das Notizbuch mitsamt der Motte ist heute im Smithsonian National Museum of American History ausgestellt. Der Begriff des *Debuggings*, also der Suche nach Fehlern in Computerprogrammen, wird von Christian Hermanns in [Her11] wie folgt definiert.

[1] Grace Brewster Murray Hopper, geboren am 9. Dezember 1906 in New York, war ebenfalls die Entwicklerin des ersten Compilers *A-0* im Jahre 1952. Darüber hinaus hat sie wesentliche Beträge zur weiteren Entwicklung der Programmiersprachen geleistet.

Definition 1.0.1 (Debugging)
In der Softwareentwicklung wird der Debugging-Prozess angestoßen, nachdem in der Testphase eine Fehlerwirkung entdeckt wurde. Das Debuggen von Software ist ein zweistufiger Prozess, der die genaue Bestimmung der Ursache und der Herkunft eines Programmfehlers sowie dessen Beseitigung umfasst.

Eine *Fehlerwirkung* bezeichnet hierbei eine erkennbare Abweichung der Programmausführung vom erwarteten Verhalten. Die Ursache einer Fehlerwirkung wird als *Defekt* bezeichnet. Häufig zeigt sich die Fehlerwirkung eines Defekts nicht sofort, sondern erst zu einem späteren Zeitpunkt der Ausführung. Für komplexe Programme bedeutet das oft, dass der für eine Fehlerwirkung verantwortliche Defekt nicht unmittelbar ausgemacht werden kann. Aus diesem Grund existieren spezielle Programmierwerkzeuge, sogenannte *Debugger*, zur Unterstützung des Debugging-Prozesses. Eine Begriffsdefinition wird in [Her11] wie folgt gegeben.

Definition 1.0.2 (Debugger)
Ein Debugger ist ein Softwaretool zur Untersuchung und Überwachung des Programmablaufs, das verwendet wird, um Defekte in Programmen zu finden. Mithilfe eines Debuggers kann ein zu untersuchendes Programm direkt gestartet und observiert werden. Darüber hinaus bieten Debugger in der Regel weitere Funktionalitäten, die die Suche nach Defekten zusätzlich erleichtern bzw. beschleunigen.

Das mit Hilfe eines Debuggers untersuchte Programm wird als *Prüfling* bezeichnet. Die Verfügbarkeit von Debuggern ist vor allem für kompilierte Programmiersprachen wichtig, da der Prüfling hier in Maschinencode vorliegt. Dieser wird direkt von der CPU ausgeführt und gestattet keine hinreichenden Rückschlüsse auf den ursprünglichen Quellcode, wenn zur Übersetzungszeit keine zusätzlichen Vorbereitungen getroffen werden. Im Gegensatz dazu kann das Anhalten der Ausführung und die Inspektion des Ausführungszustandes für interpretierte Sprachen vergleichsweise leicht vom jeweiligen Interpreter selbst bewerkstelligt werden.

Während heute für praktisch alle universellen Programmiersprachen Debugger existieren, die den Entwickler bei der Suche und der Korrektur von Defekten unterstützen, ist dies für domänenspezifische Programmiersprachen (*Domain-Specific Language*, kurz *DSL*) vielfach nicht der Fall. Damit vermindert sich der mit der Erstellung einer DSL angestrebte Effizienzgewinn für Entwicklungsaufgaben in der Zieldomäne. Die Gründe für

das Fehlen geeigneter Debugger-Mechanismen liegen einerseits in der hohen und oft von der Ausführung entkoppelten Abstraktionsebene von DSLs und andererseits in den teils sehr unterschiedlichen Anforderungen verschiedener Domänen. Voraussetzung für eine brauchbare Untersuchung zur Realisierbarkeit von Debugging-Mechanismen für DSLs ist daher die Eingrenzung der Betrachtung auf eine konkrete Domäne. Im Rahmen dieser Arbeit werden DSLs zur Programmierung digitaler Signalverarbeitungsprozesse (*Digital Signal Processing*, kurz *DSP*) betrachtet. Aufgabe von DSP-Programmen ist die Verarbeitung digitaler, zeitdiskreter Signale in Echtzeit. Ein Teilgebiet der digitalen Signalverarbeitung ist die digitale Klangsynthese, die sich mit der Erzeugung und Transformation von Audiosignalen beschäftigt.

Der Sprachumfang von DSLs der DSP-Programmierung ist oft auf Konstrukte zur Erzeugung und Transformation digitaler Signale begrenzt. Aufgaben wie die Ein- und Ausgabe von Signaldaten, die grafische Darstellung von Benutzeroberflächen oder die Ansteuerung von Hardware werden von einer Ausführungsumgebung übernommen. Für die Durchführung der Signalverarbeitungsvorgänge erfolgt dann ein Aufruf der erstellten DSP-Programme durch die Ausführungsumgebung.

Die Erstellung von Softwareanwendungen erfolgt i. A. im Rahmen von Entwicklungsumgebungen, die alle für die Entwicklung benötigten Programmierwerkzeuge zusammenfassen. Die Entwicklung allgemeiner Softwareanwendungen basiert üblicherweise auf einem Edit-Compile-Test Zyklus. In der DSP-Programmierung kommt hingegen oft der Ansatz der Continuous-Execution zum Einsatz, bei dem Kompilierung und Ausführung für den Benutzer transparent gehalten werden. Jede Änderung am Signalverarbeitungsprozess wirkt sich damit unmittelbar auf das Ergebnis aus. Dadurch wird auch während der Konstruktion von DSP-Programmen der Eindruck der kontinuierlichen Verarbeitung in Echtzeit erhalten.

In der DSP-Programmierung ergibt sich damit oft eine sehr enge Verflechtung zwischen Entwicklungsumgebung, Ausführungsumgebung und erstelltem Programm. Eine Ausführung in verschiedenen Prozessen des Betriebssystems hätte ein hohes Kommunikationsaufkommen und häufige Prozesswechsel zur Folge. Derartige Operationen sind in den verbreiteten Betriebssystemen jedoch generell aufwändig und damit für die Zwecke der Echtzeitverarbeitung ungeeignet. Im Rückschluss werden DSP-Programme oft gemeinsam mit der Entwicklungsumgebung und der Ausführungsumgebung in demselben Prozess des Betriebssystems ausgeführt[2].

[2] Neben Reaktor [Rea12] sind auch LabVIEW [Lab12] und Max/MSP [Max12] bekannte Beispiele hierfür.

Dies hat besondere Auswirkungen auf die Möglichkeiten zur Realisierung von Debugging-Mechanismen. In der allgemeinen Anwendungsentwicklung sind Debugger meist Teil der Entwicklungsumgebung, während der Prüfling jedoch stets in einem separaten Prozess ausgeführt wird. Zur Manipulation und Inspektion des Prüflings stehen spezielle Hilfsmittel des Betriebssystems zur Verfügung. Als Voraussetzung für deren Verwendung hat sich die prozesstechnische Trennung von Debugger und Prüfling etabliert. Debugger, die dieser Trennung folgen, werden in einem technischen Bericht von Paxson und Anderson [PA90] als *Separate-Process* Debugger bezeichnet. Sie bilden die überwiegende Mehrheit der heute verwendeten Debugger. Debugger, die direkt im Prozess des Prüflings ausgeführt werden, heißen *Same-Process Debugger*. In den verbreiteten Betriebssystemen existiert keine Unterstützung für diese Form von Debuggern.

Ziel dieser Arbeit ist es, im Kontext domänenspezifischer Sprachen der digitalen Signalverarbeitung Realisierungsmöglichkeiten für die manuelle Nachbildung von Debugging-Mechanismen für Same-Process Debugger zu finden. Im Mittelpunkt steht die Realisierung von Ablaufkontrollmechanismen unter den speziellen Anforderungen der Echtzeitverarbeitung in DSP-Programmen.

Die Gliederung der Arbeit gestaltet sich wie folgt. Kapitel 2 gibt zunächst eine Einführung in die verschiedenen Themenbereiche, die den Kontext der Arbeit bilden. In Kapitel 3 wird ein allgemeines Konzept für die Realisierung eines integrierten, interaktiven Same-Process Debuggers erarbeitet. Kapitel 4 stellt die Implementierung eines solchen Debuggers für die domänenspezifische Sprache Reaktor Core vor. Eine Zusammenfassung der Ergebnisse wird in Kapitel 5 gegeben.

2 Technischer Kontext

In diesem Kapitel soll zunächst ein Überblick zu den verschiedenen Themenbereichen gegeben werden, die den technischen Kontext dieser Arbeit bilden.

2.1 Domänenspezifische Programmiersprachen

Formale Programmiersprachen, die speziell auf die Bearbeitung von Aufgabenstellungen in einem abgeschlossenen Anwendungsbereich (einer *Domäne*) zugeschnitten sind, heißen domänenspezifische Sprachen (*Domain-Specific Language*, kurz *DSL*). Die Sprachmittel von DSLs können speziell auf die Lösung von Problemstellungen der Domäne zugeschnitten werden. Gegenüber den Anwendungsfällen in der Zieldomäne erreichen DSLs dadurch ein höheres Abstraktionsniveau als universelle Programmiersprachen. Bei der Verwendung von DSLs verringert sich damit die Modellierungslücke [KOM+10], also die Distanz zwischen der abstrakten Problembeschreibung als Vorstellung des Entwicklers und der konkreten Darstellung in der Programmiersprache. In der Folge sinkt der Entwicklungsaufwand für Programmieraufgaben in der Zieldomäne. Um dies zu gewährleisten, sollten DSLs einerseits alle Probleme der Domäne möglichst treffend darstellen können, während andererseits keine Probleme darstellbar sein sollten, die außerhalb der Domäne liegen [vDKV00].

Die folgenden Abschnitte widmen sich den Eigenschaften, der Werkzeugunterstützung und den Nutzergruppen von DSLs. Die Vor- und Nachteile der Verwendung von DSLs sind in der Literatur bereits vielfältig untersucht worden und sollen hier nicht erneut behandelt werden. Eine strukturierte Zusammenfassung findet sich in [Spi01].

2.1.1 Differenzierung

Die Differenzierung domänenspezifischer Sprachen erfolgt heute in erster Linie anhand ihrer Realisierungsform, dem Grad der Ausführbarkeit und der Notationsform. Die häufigste Unterteilung von DSLs nach Realisierungsformen wird u. a. von Fowler in [Fow05a] beschrieben. Fowler unterscheidet zwischen internen und externen DSLs[1].

[1] Eine andere Unterteilung wird z. B. von Czarnecki und Eisenecker in [CE00] vorgenommen. Sie unterscheiden die drei Kategorien „fixed separate“, „embedded“ und „modularly composable“. Neben rein terminologischen Unterschieden ist oft auch der ungenau abgegrenzte Begriff einer DSL selbst Ursache für verschiedene Einteilungen.

Interne DSLs werden als Untermenge einer Wirtssprache definiert und oft als „Languages inside languages“ bezeichnet. Die Bereitstellung interner DSLs gestaltet sich dadurch vergleichsweise leicht. Es besteht keine Notwendigkeit der Erstellung von Parsern, Interpretern oder Compilern. Die Syntax interner DSLs ist im Gegenzug auf die syntaktischen Möglichkeiten der Wirtssprache begrenzt [MHS05]. Als Wirtssprachen haben sich daher im Wesentlichen Sprachen mit sehr flexibler Syntax etabliert. Dabei handelt es sich häufig um dynamisch typisierte Sprachen wie Ruby oder Lisp. Eine wesentliche Problematik interner DSLs ist die mangelnde Begrenzung ihrer Mächtigkeit auf eine Problemdomäne, da prinzipiell alle Konstrukte der Wirtssprache innerhalb der DSL gültig sind. Dies erschwert die Einarbeitung vor allem für unerfahrene Benutzer. Fowler verweist hier auf das von Simonyi beschriebene Problem der „Degrees of Freedom“ [Sim05]. Interne DSLs entwickeln sich oft als Weiterentwicklung einfacher Programmbibliotheken. Die begriffliche Abgrenzung ist unscharf [MHS05]. Ein bekanntes Beispiel einer internen DSL ist das auf Ruby aufsetzende Build-Management-Werkzeug Rake [Fow05b].

Externe DSLs stellen eigenständige Sprachen dar. Damit können domänenspezifische Anforderungen, wie die in der Zieldomäne etablierten Notationen, meist wesentlich besser in der Sprache abgebildet werden [MHS05]. Der anfängliche Aufwand für die Bereitstellung externer DSLs ist jedoch vergleichsweise hoch, denn die Erstellung von Übersetzern oder Interpretern stellt i. A. wesentlich höhere Anforderungen an Entwurf und Implementierung als die allgemeine Anwendungsentwicklung in der jeweiligen Domäne. Die wohl bekannteste externe DSL ist die Datenbankanfragesprache SQL [DD87]. Eine exakte Abgrenzung externer DSLs gestaltet sich vor allem gegenüber Skriptsprachen (wie z. B. ActionScript oder Unix shell scripts) und Auszeichnungssprachen (wie z. B. TeX oder CSS) schwierig.

Im Gegensatz zu universellen Programmiersprachen müssen DSLs nicht notwendigerweise ausführbar sein[2]. Eine Klasseneinteilung nach dem Grad der Ausführbarkeit einer DSL wird von Mernik in [MHS05] gegeben. Die erste Klasse bilden direkt ausführbare DSLs. Hierzu zählen beispielweise Macro-Sprachen in Tabellenkalkulationsprogrammen und anwendungsspezifische Skriptsprachen wie VBA. Daneben existieren drei weitere Klassen für alle nicht unmittelbar ausführbaren DSLs. Für derartige DSLs kann der Programmbegriff irreführend sein. Mernik empfiehlt daher die Verwendung von Begriffen wie *Spezifikation* oder *Beschreibung*. Eingabesprachen für Anwendungs- oder Quellcodegeneratoren bilden eine Klasse nicht unmittelbar ausführbarer DSLs. Zwar wird aus ihnen

[2] Da interne DSLs an das Ausführungsmodell ihrer Wirtssprache gebunden sind und es sich dabei i. A. um eine universelle Programmiersprache handelt, gehören diese meist der Kategorie ausführbarer DSLs an.

ausführbarer Code generiert, doch für gewöhnlich haben sie einen eher deklarativen Charakter. Als Beispiel wird die Modellierungssprache ATMOL [vE01] genannt. Eine weitere Klasse umfasst vollständig deklarative DSLs, die nur eine unterstützende Rolle bei der Anwendungsgenerierung spielen. Dazu zählt z. B. die Backus-Naur-Form zur Darstellung kontextfreier Grammatiken. Die letzte Klasse umfasst schließlich alle nicht ausführbaren DSLs, wie formale Datenstrukturen sowie Analyse- und Visualisierungssprachen.

Die Notationsform einer Sprache kann als weiteres Unterscheidungsmerkmal dienen. Unterschieden werden kann zwischen textueller und grafischer Notation. Da alle verbreiteten universellen Programmiersprachen eine textuelle Notation verwenden, ist diese Eigenschaft ebenfalls vorrangig für DSLs relevant. Die bekannteste grafische Notationsform ist das Prinzip der „Boxes and Arrows“. Sie eignet sich vor allem für die Verwendung durch Endbenutzer und wird oft für Modellierungsaufgaben eingesetzt. Speziell in der datenflussorientierten Programmierung wird die Verwendung von Kästen zur Darstellung von Objekten und Verbindungen zur Darstellung von Daten- und Kontrollflüssen als natürliche Notation empfunden. Klassische Beispiele hierfür sind LabVIEW [Lab12] und Simulink [Sim12]. Daneben existieren experimentelle Ansätze zur Integration der grafischen Notation in der beispielorientierten Programmierung [Edw05].

Innerhalb dieser Kategorien kann eine weitere Differenzierung anhand allgemeiner Eigenschaften von Programmiersprachen vorgenommen werden. So können direkt ausführbare DSLs nach ihrem Programmierparadigma in imperative und deklarative DSLs und nach ihrer Ausführungsform in kompilierte und interpretierte DSLs unterteilt werden. Nicht unmittelbar ausführbare DSLs sind naturgemäß deklarativ und werden weder im klassischen Sinne kompiliert noch interpretiert. Stattdessen werden sogenannte „source-to-source“ Transformationen angewendet. Das Ergebnis ist entweder eine benutzerverständliche visuelle Darstellung oder ein maschinenverständlicher äquivalenter Quellcode in einer anderen Programmiersprache.

2.1.2 Werkzeugunterstützung

Erste Untersuchungen zu Verfügbarkeit und Qualität von Entwicklungswerkzeugen für DSLs [Don10][JG04] zeigen einen deutlichen Mangel gegenüber der Angebotssituation von Entwicklungswerkzeugen für universelle Programmiersprachen. Dieser Mangel wird häufig als zentraler Nachteil bei der Verwendung domänenspezifisher Sprachen genannt [Spi01][Fow05a]. Denn für einen dauerhaften produktiven Einsatz von DSLs ist eine adäquate Werkzeugunterstützung zwingend erforderlich [KPKP06].

Im Zuge der zunehmenden Popularität des Language-Oriented Programming, einem

von Ward bereits 1994 beschriebenen Programmierparadigma [War94], haben auch DSLs innerhalb der letzten Jahre stark an Bedeutung gewonnen. In diesem Zusammenhang sind erste umfangreichere Entwicklungswerkzeuge (sogenannte Language Workbenches [Fow05a] oder Language Development Systems [MHS05]) entstanden, deren Ziel es ist, die Entwicklung von und die Arbeit mit DSLs zu vereinfachen. Beispiele hierfür sind JetBrains' Meta-Programming System [VS10] sowie Intentional Software [SCC06]. Ein zentraler Aspekt von Language Workbenches ist die automatische Generierung zeitgemäßer Programmierwerkzeuge aus einer Sprachspezifikation[3]. Diese Entwicklung macht die Bedeutung adäquater Programmierwerkzeuge deutlich, bietet jedoch keine Perspektive für bestehende DSLs. Zudem können in bestimmten Domänen Anforderungen an Benutzerfreundlichkeit, Integrierbarkeit und Performanz existieren, die durch Language Workbenches (noch) nicht erfüllt werden können.

2.1.3 Nutzergruppen

Die Entwicklung von DSLs kann sehr unterschiedlich motiviert sein. Einerseits kann die Zieldomäne einer DSL ein Problemfeld im Entwicklungsprozess von Softwareanwendungen sein. Dies trifft beispielsweise in der Produktlinienentwicklung[4] zu. Die Benutzergruppe der DSL umfasst demzufolge Personen, von denen ein hoher Grad an Erfahrung bei der Verwendung von Werkzeugen zur Softwareentwicklung vorausgesetzt werden kann. Der Entwurf der Werkzeuge zur Arbeit mit einer solchen DSL kann sich in diesem Fall also an den Standards universeller Programmiersprachen und CASE Werkzeugen orientieren.

Andererseits kann die Zieldomäne jedoch auch ein Problemfeld sein, dessen Benutzer keinerlei Erfahrung mit der Entwicklung von Software haben. Ein klassisches Beispiel sind Tabellenkalkulationsprogramme. In diesem Zusammenhang spricht man auch vom sogenannten *End-User-Development*. Mit der stetig zunehmenden Präsenz von Computerprogrammen in Alltagssituationen steigt auch der Wunsch von Endbenutzern bestehende Software an ihre jeweiligen Bedürfnisse anzupassen und Programme in bestimmten Grenzen selbst zu erstellen [CFMP04]. DSLs spielen hierbei eine wichtige Rolle. Sprachen und Umgebungen die es Endbenutzern gestatten derartige Programmieraufgaben zu bewältigen, stellen jedoch vollkommen neue Anforderungen. Diese werden in [CFMP04]

[3] Während dies zunächst nur einer Erweiterung der Anwendungsbereiche traditioneller Programmierwerkzeuge, wie Parsergeneratoren und Compiler-Compilern entspricht, gibt es auf diesem Gebiet auch Bemühungen die Arbeit mit Programmiersprachen gänzlich zu revolutionieren [SCC06].

[4] Der Ansatz der Produktlinienentwicklung verfolgt das Ziel einer dauerhaften Reduktion des Entwicklungsaufwandes für wiederkehrende Aufgaben bei der häufigen Erstellung ähnlicher Produkte. Dies wird durch eine gezielte Priorisierung von Entwicklungsplattformen erreicht, die eine stetige Steigerung des Wiederverwendungsgrades von Programmfragmenten ermöglicht.

ausführlich diskutiert. Eine Teilmenge von Endbenutzern bilden sogenannte Domänenexperten. Die folgende Definition findet sich ebenfalls in [CFMP04].

> **Definition 2.1.1** (Domänenexperte)
> [Domain-expert users] are experts in a specific domain, not necessarily experts in computer science, who use computer environments to perform their daily tasks. They have also the responsibility for induced errors and mistakes.

Während es im allgemeinen End-User Development also um die Allgemeinverständlichkeit von Programmiersprachen und -werkzeugen geht, steht bei End-User-Development-Systemen für Domänenexperten die *eigenverantwortliche* domänenspezifische Programmierung im Vordergrund. Hier genügt es, dem Benutzer eine Programmierumgebung zur Verfügung zu stellen, die die in der Domäne üblichen Notationen und Arbeitsabläufe möglichst natürlich abbildet. DSLs in Entwicklungsumgebungen zur Programmierung digitaler Signalverarbeitungsprozesse sind solche Systeme.

2.2 Debugging Techniken

Debugger sind heute unverzichtbare Werkzeuge im Softwareentwicklungsprozess. Sie dienen der Identifikation der kausalen Zusammenhänge zwischen Ursache und Wirkung im untersuchten Programm auf Quellcodeebene. Über das Finden, Eingrenzen und Entfernen von Programmdefekten hinaus helfen sie Entwicklern damit auch ganz allgemein die dynamische Natur von Software zu verstehen. Debugger sind daher nicht nur zur Unterstützung bei der Fehlersuche, sondern auch zum Erlernen von Programmiersprachen und für die Einarbeitung in bestehende Softwareprojekte nützlich [Ros96].

2.2.1 Grundlagen

Rosenberg definiert drei für die Funktionsweise von Debuggern grundlegende Prinzipien [Ros96]. Das *Heisenberg Prinzip*[5] sagt aus, dass die Ausführung des Prüflings durch die Existenz eines Debuggers nicht beeinflusst werden darf. Fehlerwirkungen, die scheinbar verschwinden sobald versucht wird ihre Ursache mit Hilfe eines Debuggers zu finden, werden auch als Heisenbugs bezeichnet. Dem Prinzip des *Truthful Debugging* zufolge muss die Korrektheit aller von einem Debugger gelieferten Informationen zu jeder Zeit garantiert sein. Nur so ist es möglich das Vertrauen des Benutzers zu erhalten. Ein Fehler im Debugging-Mechanismus selbst hätte unabsehbare Konsequenzen. Schließlich muss ein Debugger Antworten auf die wichtigsten Fragen des Benutzers geben können. Nach Rosenberg sind das die Fragen danach, an welcher Stelle im Quellcode die Fehlerwirkung sichtbar wurde und wie dieser Ausführungspunkt erreicht worden ist. Zudem sollten z. B. Variablenwerte, Threads und CPU-Register inspiziert werden können.

Streng genommen kann die Einhaltung dieser Prinzipien durch einen Debugger jedoch nicht gewährleistet werden. So wird beispielsweise das Scheduling des zugrundeliegenden Betriebssystems durch die Existenz eines Debugger-Prozesses in gewisser Weise beeinflusst. Dies wirkt sich auch auf die Ausführung des untersuchten Prozesses aus. Es hat sich im praktischen Einsatz als ausreichend erwiesen, der Einhaltung der Prinzipien möglichst nahe zu kommen.

2.2.2 Verfahren

Bevor näher auf die konkrete Realisierung von Debugging-Mechanismen eingegangen wird, sollen in diesem Abschnitt zunächst verschiedene Debugging-Verfahren vorgestellt werden.

[5] Die Bezeichnung des Heisenberg Prinzips geht auf den deutschen Physiker Werner Heisenberg und dessen Unschärferelation zurück.

2.2.2.1 Trace-Debugging

Das Trace-Debugging ist die älteste und bis heute am weitesten verbreitete Debugging-Technik. Debugger für populäre objektorientierte Sprachen, wie Java, C# oder C++, basieren nahezu ausschließlich auf dieser Technik [Her11]. Das Trace-Debugging beruht auf der Unterbrechung der Programmausführung an einem im Vorfeld definierten Haltepunkt (*Breakpoint*) und der schrittweisen Ausführung von Folgeanweisungen (*Single-Stepping*). Kontextinformationen werden nur während einer Unterbrechung und nur für den aktuellen Ausführungszustand bereitgestellt. Zu den wichtigsten Elementen des Programmzustandes zählen die Variablen, die im Kontext des gegenwärtigen Ausführungspunktes gültig sind und der Aufrufstack, also die Abfolge von Funktionsaufrufen, durch die der gegenwärtige Ausführungspunkt erreicht wurde.

Trace-Debugger folgen der befehlsorientierten Natur der heute verbreitetsten Programmiersprachen. Sie sind daher sowohl vergleichsweise leicht realisierbar als auch von Programmierern intuitiv bedienbar. Für den grundlegenden Verwendungszweck von Debuggern, der Identifikation der Ursache für eine Fehlerwirkung, sind sie jedoch denkbar ungeeignet. Denn Trace-Debugger können naturgemäß nur der Ausführungsrichtung von Programmen *folgen*. Anweisungen, die einmal ausgeführt wurden, können nicht mehr rückgängig gemacht werden. Die Ursache eines Defekts liegt zeitlich jedoch stets *vor* dessen erkennbarer Wirkung. Hat der Entwickler Zeitpunkt und Ort der Wirkung im Quellcode identifiziert, müssen alle Ausführungspfade, die zur Quellcodeposition der Wirkung führen, manuell zurückverfolgt werden. Oft wird das durch eine sukzessive Wiederholung der Ausführung und das Setzen von Breakpoints an jeweils früheren Punkten im Programm erreicht. Die Fehlersuche gestaltet sich damit sehr zeitaufwändig. Die Einschränkung der Inspektionsrichtung bildet den zentralen Nachteil der Verwendung von Trace-Debuggern[6] [Her11].

2.2.2.2 Andere Debugging-Ansätze

Neben dem Trace-Debugging existieren einige experimentelle Debugging-Verfahren. Diese Verfahren haben den Einzug in die praktische Verwendung noch nicht geschafft und werden vorrangig zu Forschungszwecken untersucht. Die bekannteste und vielversprechendste Alternative zum Trace-Debugging ist das Omniscient-Debugging, welches von Lewis erstmals 2003 als „Debugging Backwards in Time“ umgesetzt wurde [Lew03]. Durch die

[6] Zwar wurde der Versuch unternommen, erweiterte Mechanismen zur Ablaufkontrolle in Trace-Debugger zu integrieren, diese haben sich im praktischen Einsatz jedoch nicht durchgesetzt. Beispiele hierfür sind *Reversible Execution* [TA95] und *Replay* [ANCS00]

Aufzeichnung sämtlicher Zustandsänderungen während der Ausführung eines Prüflings ist der Omniscient-Debugger in der Lage, dem Benutzer ein globales Bild der Programmausführung zu liefern. Der Programmzustand kann für beliebige Zeitpunkte rekonstruiert werden. Der Rückschluss von einer Fehlerwirkung auf den verantwortlichen Defekt wird damit deutlich erleichtert, da der Programmablauf von der Fehlerwirkung über die infizierten Zustände bis hin zum Defekt zurückverfolgt werden kann. Die Problematik des Ansatzes stellen die mit der Speicherung jeder einzelnen Zustandsänderung verbundenen, unverhältnismäßigen Datenmengen (etwa 100MB pro Sekunde) dar sowie die dadurch entstehenden Performanzeinbußen (Faktor 10 bis 300).

Einen weiteren vollkommen anderen Ansatz verfolgen deklarative (oder auch algorithmische) Debugger. Diese Methode wurde ursprünglich für die logische Programmiersprache Prolog entwickelt [Sha82]. Die grundlegende Idee ist die Verwendung des Berechnungsbaums, der die Ausführung des Prüflings repräsentiert. Beim deklarativen Debugging werden keine einzelnen Anweisungen, sondern semantisch zusammenhängende Berechnungseinheiten auf ihre Validität überprüft. Für den Debugging-Prozess bedeutet dies, dass der Benutzer sich von der konkreten Implementierung lösen kann. Um einen Defekt zu lokalisieren, müssen die Berechnungen nicht mehr schrittweise nachvollzogen werden, sondern die Ergebnisse von Teilberechnungen bewertet werden. Weiterführende Informationen hierzu finden sich u. a. in [Her11].

2.2.3 Weitere Klassifizierung

Neben dem realisierten Debugging-Verfahren, können Debugger anhand einiger weiterer Eigenschaften wie folgt unterschieden werden:

- *Post-mortem Debugger* zeichnen Ausführungsdaten zur späteren Analyse auf (z. B. Kernel-Dumps oder Omniscient-Debugger). Im Gegensatz dazu ermöglichen *interaktive Debugger* eine Untersuchung während der Laufzeit des Prüflings.
- *Stand-alone Debugger* sind eigenständige Anwendungen, die unabhängig vom verwendeten Compiler arbeiten. Aufgrund zunehmend komfortablerer Entwicklungsumgebungen machen sie gegenüber *integrierten Debuggern* nur einen geringen Anteil aus.
- *Symbolische Debugger* verfügen über Mechanismen zur Ablaufkontrolle und Darstellung des Programmkontextes auf dem Abstraktionsniveau einer höheren Programmiersprache. Demgegenüber werden Debugger, die nur auf Maschinen- oder Assemblerebene arbeiten, als *Low-Level Debugger* bezeichnet.

2.2.4 Funktionsweise herkömmlicher Debugger

Als herkömmliche Debugger werden im Folgenden interaktive symbolische Trace-Debugger für kompilierte imperative Programmiersprachen bezeichnet, die zusammen mit anderen Entwicklungswerkzeugen innerhalb einer Entwicklungsumgebung verwendet werden. Zudem wird die Kompilierung in Maschinencode einer x86 Befehlssatzarchitektur angenommen. Die grundlegenden Verfahren sind jedoch für andere Befehlssatzarchitekturen oder auch virtuelle Maschinen sehr ähnlich. Die im folgenden vorgestellten Verfahrensweisen werden von Rosenberg in [Ros96] vollständig beschrieben.

2.2.4.1 Vorbetrachtung

Die zentrale Problematik bei der Bereitstellung von symbolischen Debuggern ist die Überwindung der semantischen Barriere zwischen den Ausdrücken auf Sprachebene (*Source-Level*) und den Operationen auf Maschinenebene (*Instruction-Level*). Aufgabe eines Compilers ist die Übersetzung des Programmcodes auf Sprachebene in Binärcode auf Maschinenebene. Um einem Debugger die Zuordnung in der Rückrichtung zu ermöglichen, muss der Compiler zusätzliche Informationen, sogenannte *Programmdaten*, speichern. Neben der zeilenweisen Zuordnung zwischen Sprach- und Maschinenebene, werden hier auch die Speicheradressen von Variablen, die Struktur abstrakter Datentypen, Dateinamen und andere Informationen hinterlegt. Je nach Beschaffenheit der verwendeten Hochsprache kann es zwischen Programm- und Maschinencode auch zu Uneindeutigkeiten kommen[7]. Im Allgemeinen wird eine exakte Zuordnung umso schwieriger, je höher das Abstraktionsniveau der Programmiersprache ist und je weiter sich ihr Programmierparadigma von der imperativen Funktionsweise des Maschinenbefehlssatzes loslöst. Zusätzliche Probleme können Compileroptimierungen verursachen, die zur Verringerung des Ausführungsaufwandes z. B. die Abfolge von Anweisungen ändern, implizit Parallelisierungen vornehmen oder Variablen eliminieren, die aus technischer Sicht überflüssig sind. Für die Zwecke des Debuggings werden derartige Optimierungen in heutigen Compilern für gewöhnlich deaktiviert.

Sobald der Übersetzungsvorgang vom Compiler abgeschlossen wurde, liegen die ausführbare Datei und die entsprechenden Programmdaten des Prüflings vor. Da der Debugger Zugriff auf die Instanz des Prüflingsprozesses benötigt, registriert er sich über spezielle API-Funktionen beim Betriebssystem[8]. Debugger und Prüfling müssen nicht

[7] Bekannte Probleme stellen z. B. Klassen-Templates in C++ dar (*One-to-Many Problems*) [Ros96].

[8] Je nach Betriebssystem sind dafür besondere Privilegien, wie eine Signierung des Debuggers, eine Authentifizierung als Administrator o. ä. erforderlich.

zwangsläufig auf demselben Computer ausgeführt werden. Wird der Prüfling auf einem entfernten Rechner ausgeführt, spricht man auch von *Remote-Debugging*. Ebenfalls muss der Debugger den Prüflingsprozess nicht zwangsläufig selbst starten. Die meisten Betriebsysteme gestatten es einem Debugger auch sich mit einem bereits laufenden Prozess zu verbinden.

2.2.4.2 Breakpoints

Soll nun ein Breakpoint an einer Anweisung im Programmcode gesetzt werden, ermittelt der Debugger zuerst die Speicheradresse des entsprechenden Maschinenbefehls aus den Programmdaten. Über eine Funktion des Betriebssystems kann ein Breakpoint an der jeweiligen Speicheradresse des Prüflingsprozesses gesetzt werden. Da ein schreibender Zugriff in dessen Speicherbereich nicht möglich ist, wenn der Prozess gerade ausgeführt wird, ist der genaue Zeitpunkt des Einfügens vom Scheduling des Betriebssystems abhängig.

Bevor der Breakpoint eingefügt werden kann, wird der Byte-Wert an der angegebenen Speicherstelle des Prüflings zuerst zwischengespeichert. Die Adresse der zu verwendenden Speicherzelle wird vom Debugger als Parameter übergeben. Im Anschluss wird die Speicherstelle mit einer INT3 Operation überschrieben[9]. Der INT3 Befehl der x86 Befehlssatzarchitektur stellt einen 1-Byte Opcode dar, der speziell für Debugging-Zwecke verwendet wird [Int12b]. Da ein Byte hier die kleinste Befehlseinheit ist, kann der ursprüngliche Befehlscode ebenfalls nur ein Byte oder länger gewesen sein. Beim INT3 besteht also nicht die Gefahr, versehentlich Folgebefehle zu verfälschen[10]. Nach dem Ersetzungsvorgang kehrt der Systemaufruf zum Debugger zurück. Dieser muss nun die Adresse der Speicherstelle zusammen mit dem ursprünglichen Bytewert speichern.

Wird bei der Fortsetzung der Ausführung des Prüflings ein Breakpoint erreicht, löst der Prozessor einen hochpriorisierten Interrupt aus. Das Betriebssystem unterbricht die Ausführung des verantwortlichen Prozesses daraufhin umgehend und benachrichtigt den für den Prozess registrierten Debuggerprozess. Dieser kann wieder mit Hilfe von Betriebssystemfunktionen Informationen zum Ausführungszustand des Prüflingsprozesses erhalten und damit u. a. ermitteln um welchen Breakpoint es sich handelt. Zu Beginn einer Unterbrechung stellt der Debugger meist zuerst den Originalcode des Prüflings wieder her. Das ist in Hinblick auf das Prinzip des Truthful-Debugging (2.2.1) wich-

[9] Einen Breakpoint an oder auf einen Befehl zu setzen, bewirkt daher stets eine Unterbrechung *vor* der Ausführung des Befehls.

[10] Diese Eigenschaft ist sehr wichtig, da der Folgebefehl im Speicher möglicherweise vor dem Breakpoint ausgeführt wird. Das ist genau dann der Fall, wenn es sich bei dem Folgebefehl um ein Sprungziel handelt.

tig. Da der Breakpoint einen Teil des ursprünglichen Befehls überschreibt, könnte es an der Stelle des Breakpoints andernfalls zu Fehlinterpretationen diverser Folgebefehle beim Betrachten des Maschinencodes in einer Assemblerdarstellung kommen. Das könnte zur Verwirrung des Benutzers führen und Skepsis gegenüber der Korrektheit des Debugging-Mechanismus' schüren. Das Löschen von Breakpoints erfolgt analog zum Einfügen. Da der INT3 Befehl wie jede andere Maschinenoperation zur Inkrementierung des Program-Counter Registers des Prozessors führt, ist dessen Wert zum Zeitpunkt der Unterbrechung um ein Byte zu groß. Es sollte ebenfalls bereits zu Beginn einer Unterbrechung korrigiert werden. Während einer Unterbrechung kann dem Benutzer schließlich der unveränderte Originalzustand des Prüflings zur Inspektion bereitgestellt werden.

Bevor die Ausführung des Prüflings nach einer Unterbrechung normal fortgesetzt wird, veranlasst der Debugger zuerst die Ausführung eines Einzelschrittes auf Maschinenebene (siehe Abschnitt 2.2.4.3). Dies erweist sich in vielen Fällen als günstig. Soll ein Breakpoint beispielsweise permanent sein, also im nächsten Durchlauf wieder eine Unterbrechung auslösen, muss der entsprechende Befehl unmittelbar nach seiner Ausführung erneut durch einen INT3 Befehl ersetzt werden. Dies kann nach Ausführung des Einzelschrittes erfolgen.

Breakpoints die nach dem soeben beschriebenen Verfahren realisiert werden, werden auch als *Software Breakpoints* bezeichnet. Alternativ dazu verfügen verschiedene Prozessoren über eine begrenzte Zahl von *Hardware Breakpoints*. Die Verwendung von Hardware Breakpoints gestaltet sich sehr einfach. In einem speziellen Breakpoint-Address Register [Int12b] wird die Speicheradresse des Maschinenbefehls hinterlegt, an dem eine Unterbrechung erfolgen soll. Nach der Ausführung von Befehlen durch den Prozessor wird das Program-Counter Register inkrementiert und der neue Wert mit den Werten der Breakpoint-Address Register verglichen. Im Falle einer Übereinstimmung wird ein Interrupt ausgelöst und der Kontrollfluss an den Debugger übergeben. Hardware Breakpoints werden aufgrund ihrer einfachen Handhabung und ihrer Nicht-Invasivität oft bevorzugt verwendet. Software Breakpoints kommen dann erst zum Einsatz, wenn alle verfügbaren Breakpoint-Address Register belegt sind.

2.2.4.3 Einzelschrittausführung

Neben einem Breakpoint-Mechanismus gehört die Einzelschrittausführung zu den Kernfunktionalitäten von Trace-Debuggern. Sie kommt vor allem dann zum Einsatz, wenn die Auswirkung einzelner Anweisungen auf den Ausführungszustand des Prüflings beobachtet werden soll. Die Grundlage für die Realisierung der Einzelschrittausführung bildet das

Trap-Flag, ein Bit im Prozesskontext des Prozessors[11]. Nach jeder Befehlsverarbeitung durch den Prozessor wird der Wert des Trap-Flags überprüft. Ist es gesetzt, wird die Ausführung des Prozesses wie im Falle eines Breakpoints unterbrochen. Das Betriebssystem sendet eine Benachrichtigung an den registrierten Debugger-Prozess. Der methodische Unterschied dieses Ansatzes gegenüber dem Breakpoint-Mechanismus ist seine Asynchronität. Das Setzen des Trap-Flags bewirkt eine Unterbrechung des Prüflingsprozesses nach der Verarbeitung des nächsten Maschinenbefehls – unabhängig davon welcher Befehl es ist. Auf diese Weise lassen sich auch beliebige asynchrone Unterbrechungen des Programmablaufs erreichen.

Die Ausführung eines einzelnen Maschinenbefehls, eines sogenannten *Instruction-Level Step*s, ist für den Benutzer einer höheren Programmiersprache i. A. unzureichend. Als Einzelschritt wird hier die Ausführung einer Anweisung auf Sprachebene verstanden, die auch als *Source-Level Step* bezeichnet wird. Um das zu erreichen, können im einfachsten Fall mehrere Instruction-Level Steps ausgeführt werden. Dies ist für einfache Anweisungen akzeptabel. Im Falle eines Funktionsaufrufes kann es jedoch zu längeren Verzögerungen kommen. Daher wird zwischen den folgenden Varianten von Source-Level Steps unterschieden:

- *Step Into* überspringt nicht den gesamten Funktionsaufruf, sondern nur den aufrufenden Befehl. Die nächste Unterbrechung erfolgt also an der ersten Anweisung der aufgerufenen Funktion. Ein ggf. wiederholtes Instruction-Level Stepping ist hier angemessen.

- *Step Over* überspringt alle Anweisungen des Funktionsaufrufs einschließlich aller geschachtelten Funktionsaufrufe. Ein wiederholtes Instruction-Level Stepping könnte hier unnötige Verzögerungen hervorrufen. Step Over wird daher meist über das Setzen eines Breakpoints auf die Rücksprungadresse, also die Adresse des ersten Maschinenbefehls nach der Verarbeitung der Funktion, realisiert.

Darüber hinaus verfügen viele Debugger über einen *Step Out* Mechanismus. Dieser kann als „nachträgliches Step Over“ verstanden werden: Die Ausführung wird bis zum ersten Befehl nach Abschluss der aktuellen Funktion fortgesetzt. Die Speicheradresse dieses Befehls entspricht der Rücksprungadresse im aktuellen Stackframe.

[11] Dieser Teil des Prozesskontextes ist i. A. auch threadspezifisch. Die schrittweise Ausführung eines Threads beeinflusst andere Prozesse oder Threads also nicht.

2.2.5 Same-Process Debugger

Ein Debugger wird genau dann als *Same-Process Debugger* bezeichnet, wenn er in demselben Prozess des Betriebssystems ausgeführt wird wie der Prüfling [PA90]. Damit hat ein Same-Process Debugger direkten Zugriff auf den Speicherbereich des Prüflings. Herkömmliche Debugger werden in einem eigenen Prozess ausgeführt und in diesem Zusammenhang auch als *Separate-Process Debugger* bezeichnet. Hier besteht keine Möglichkeit des direkten Speicherzugriffs. Stattdessen stehen ihnen spezielle Systemfunktionen für die Manipulation und Inspektion des Prüflings zur Verfügung. Historisch betrachtet waren alle Debugger vor der Entwicklung multitaskingfähiger Betriebssysteme Same-Process Debugger, so z. B. auch der Debugger CodeView für MS DOS Programme [Dun88].

Für die Realisierung von Debuggern in Multitaskingsystemen erwies sich der Separate-Process Ansatz als vorteilhaft. Die Gefahr der Verfälschung des Prüflingsverhaltens nach dem Heisenberg Prinzip (2.2.1) wird durch die strikte Trennung auf Prozessebene deutlich verringert. Die notwendigen Voraussetzungen, wie die Bereitstellung adäquater API Funktionen, wurden spätestens seit den 1990er Jahren von allen verbreiteten Betriebssystemen erfüllt. Die Trennung von Debugger und Prüfling auf Prozessebene gilt heute als Voraussetzung für die Verwendung von Debugging APIs. Im Hinblick auf die Inspektion des Prüflings entstehen für Same-Process Debugger daraus keine Nachteile. Jedoch müssen alle in Abschnitt 2.2.4 erwähnten Funktionalitäten zur Ablaufkontrolle des Prüflings ohne Betriebssystem- und Hardware-Unterstützung auskommen.

Die Grundlage für das in dieser Arbeit entwickelte Verfahren bildet die Forschungsarbeit von Peter Kessler [Kes90]. Anlass der Untersuchung Kesslers war der hohe Laufzeitoverhead, der bei der Verwendung herkömmlicher Breakpoints anfällt. Die Übergabe des Kontrollflusses an den Debugger und zurück zum Prüfling erfordert stets zwei Prozesswechsel im Betriebssystem. Prozesswechsel gelten generell als aufwändige Operationen. Auch wenn es beispielsweise im Falle von Conditional Breakpoints in vielen Fällen nicht zur Unterbrechung der Ausführung kommt, sind für die Auswertung der Unterbrechungsbedingung nach dem herkömmlichen Verfahren zur Realisierung von Breakpoints (siehe Abschnitt 2.2.4.2) trotzdem zwei Prozesswechsel erforderlich. Das Ziel Kesslers ist es, diesen Aufwand zu reduzieren.

Statt einen Breakpoint mit Hilfe eines Interrupt-Befehls zu realisieren, soll ein einfacher Verzweigungsbefehl zum Einsatz kommen. Damit wird der Ausführungsfluss in einen im Vorfeld vorbereiteten Behelfscode (*Breakpoint-Code*) im Speicherbereich des Prüflingsprozesses umgeleitet. Der Breakpoint-Code enthält einerseits die ursprüngliche Anweisung an der Stelle des Breakpoints sowie einen Verzweigungsbefehl zurück zum

Programmcode des Prüflings. Andererseits können hier beliebige Anweisungen, z. B. für die Auswertung einer Unterbrechungsbedingung, für die Überprüfung von Assertions oder für den Aufruf eines interaktiven Debuggers zur Inspektion des Prüflings ergänzt werden. Um das Ausführungsverhalten des Prüflings durch die zusätzlichen Anweisungen nicht zu verfälschen, muss dessen Ausführungszustand (z. B. die Werte von Registern) zuvor gesichert und im Nachhinein wiederhergestellt werden. Der Vorschlag Kesslers zur allgemeinen Form eines solchen Breakpoints ist in Abbildung 2.1 dargestellt.

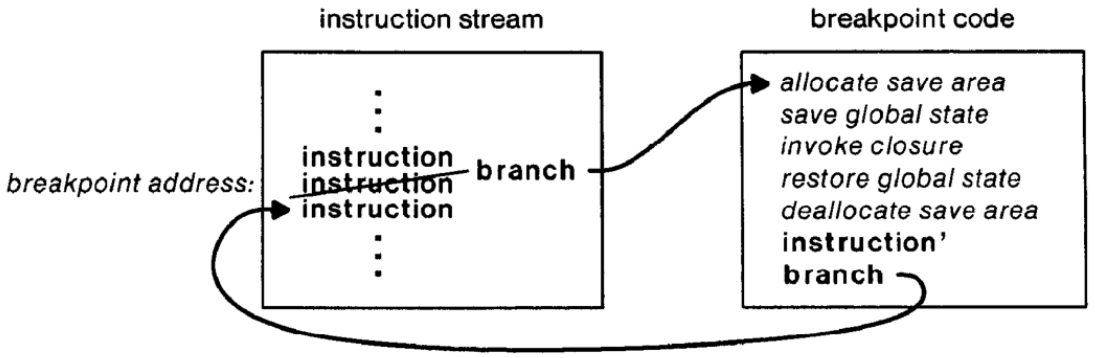

Abbildung 2.1 Darstellung der allgemeinen Form eines schnellen Breakpoints nach [Kes90]. Die ursprüngliche Operation an der Stelle der Breakpoint-Adresse wurde durch eine Verzweigungsoperation ersetzt, die eine Umleitung des Kontrollflusses initiiert. Innerhalb des Breakpoint-Codes können beliebige Operationen im Ausführungskontext des Prüflings ausgeführt werden (invoke closure). Dessen Ausführungszustand muss im Vorfeld gesichert und im Nachhinein wiederhergestellt werden. Schließlich kommt die ursprüngliche Operation zur Ausführung, bevor der Kontrollfluss zum Prüflingscode zurückkehrt.

Der Vorschlag Kesslers stellt einige Voraussetzungen an die Speicherschutzmechanismen des Betriebssystems sowie die Möglichkeiten der zugrundeliegenden Befehlssatzarchitektur. Diese sollen im Folgenden zusammengefasst werden. Zunächst ist ein schreibender Zugriff auf den Programmcode des Prüflings erforderlich. Darin unterscheidet sich der Ansatz Kesslers nicht von der gewöhnlichen Verfahrensweise für Software Breakpoints. Jedoch sind die üblichen Funktionen der Debugging API von Betriebssystemen oft auf das Einfügen von Interrupt-Befehlen beschränkt. Hierfür müssen Alternativen existieren. Der zusätzlich erforderliche Breakpoint-Code muss ebenfalls zur Laufzeit in einen ausführbaren Speicherbereich des Prüflingsprozesses geschrieben werden können. Der dafür nötige Speicherplatz muss verfügbar sein. Für die Sicherung des Ausführungszustandes muss weiterhin eine dynamische Allokation von Speicherplatz durch den Breakpoint-Code möglich sein. Eine weitere Problematik betrifft Befehlssatzarchitekturen mit Befehlen unterschiedlicher Länge. Nimmt eine Verzweigungsoperation mehr Speicherplatz ein als der zu ersetzende Befehl, wird der im Programmcode nachfolgende Befehl dadurch ganz oder

teilweise überschrieben. Das ist genau dann problematisch, wenn es sich bei dem Folgebefehl um ein Sprungziel handelt. Dies würde die Semantik des Prüflings unwiderruflich verfälschen.

Für die Implementierung seines Ansatzes verwendete Kessler die Programmierumgebung Cedar und das Betriebssystem UNIX auf Basis einer SPARC Befehlssatzarchitektur. Diese Konfiguration erfüllt alle oben beschriebenen Voraussetzungen. Insbesondere stellt die Ersetzung beliebiger Befehle durch eine Verzweigungsoperation hier kein Problem dar, da alle Operationen der SPARC Architektur Wortlänge haben [SPA92]. Das Ergebnis ist ein um den Faktor 1000 beschleunigter Breakpoint-Mechanismus (im Vergleich zu Standard-Breakpoints unter UNIX). Das Verfahren ist damit nicht nur für Conditional Breakpoints geeignet, sondern auch für Profiling-Zwecke oder zur Realisierung von Watchpoints [Kep93].

2.3 Digitale Signalverarbeitung

Die digitale Signalverarbeitung (Digital Signal Processing, DSP) beschäftigt sich mit der kontinuierlichen Verarbeitung digitaler Signale in Echtzeit. Ein digitales Signal ist dabei diskret in Zeit- und Wertebereich und wird während der Verarbeitung als Folge numerischer Werte gespeichert. Alle Betrachtungen in dieser Arbeit beziehen sich auf softwarebasierte DSP-Systeme. Die Verarbeitung erfolgt stets durch ein Softwareprogramm, welches auf einem der verbreiteten Betriebssysteme und einer nicht genauer definierten Hardware mit x86 Befehlssatzarchitektur zur Ausführung kommt. Ein Teilgebiet der digitalen Signalverarbeitung ist die digitale Klangsynthese, die sich mit der Erzeugung und Transformation von Audiosignalen befasst.

2.3.1 Echtzeitverarbeitung

In der Informatik spricht man von Echtzeitverarbeitung, wenn für den verarbeitenden Prozess[12] eine zeitliche Vorgabe für die Bereitstellung von Ergebnissen existiert. Der Zeitpunkt an dem ein Ergebnis vorliegen muss, wird als *Zeitschranke* bezeichnet.

Im Hinblick auf die Bedingungen unter denen ein System eine Verarbeitung in Echtzeit gewährleisten kann, wird oft versucht zwischen harter und weicher Echtzeit zu differenzieren. Diese Einteilung ist für softwarebasierte DSP-Systeme jedoch nicht praktikabel. Sie führt zudem häufig zu Kontroversen. Im Detail ist nicht klar, ob die Unterscheidung zwischen harter und weicher Echtzeit anhand der technischen Realisierung eines Systems oder anhand der Folgen zu treffen ist, die mit dem Verletzen einer Zeitschranke verbundenen sind. Wird die Unterscheidung anhand der Folgen eines Ausfalls getroffen, ist keine allgemeine Aussage für DSP-Systeme möglich, da sie von der Verwendungsweise des Systems abhängen. Wird die Unterscheidung anhand der technischen Realisierung getroffen, sind nur solche Systeme Echtzeitsysteme, deren Ausführungszeit im Vorfeld berechnet werden kann. Im einfachsten Fall könnte dafür die Summe aller Takte des Prozessors gebildet werden, die nötig sind, um alle Befehle eines Programms abzuarbeiten. Bereits hier besteht eine Abhängigkeit von der verwendeten Hardware. Für softwarebasierte DSP-Systeme kann also auch in diesem Fall keine Aussage zur Zugehörigkeit gegeben werden.

Die Unterteilung in harte und weiche Echtzeit trifft nicht den Kern der Problemstellung in einem DSP-System. Hier besteht das Ziel darin, neue Ergebnisse kontinuierlich in der Zeit bereitstellen zu können, die es braucht die vorherigen Ergebnisse auszuwerten

[12] Mit dem Begriff *Prozess* ist hier kein Prozess des Betriebssystems gemeint. Im Kontext der Echtzeitverarbeitung wird der Prozessbegriff im Sinne eines allgemeinen Datenverarbeitungsprozesses verwendet.

oder weiterzuverarbeiten. Weiterführende Differenzierungen erweisen sich als unnötig. Ein einfaches Beispiel kann für die digitale Klangsynthese wie folgt gegeben werden. Wenn ein Verarbeitungsprozess 2.01 Sekunden benötigt um 2 Sekunden Audioausgabe zu produzieren, handelt es sich nicht um einen Echtzeitprozess. Benötigt er hingegen nur 1.99 Sekunden ist es ein Echtzeitprozess. Diesen Ansatz greift auch die folgende Definition von Robert Bristow-Johnson [BJ98] auf. Sie bildet die Grundlage des Echtzeitbegriffs in dieser Arbeit.

> **Definition 2.3.1** (Echtzeitbedingung DSP)
> In a real-time DSP process, the analyzed input and/or generated output samples [...] can be processed (or generated) continuously in the time it takes to input and/or output the same set of samples independent of the processing delay.

In der digitalen Klangsynthese spricht man demzufolge immer dann von Echtzeitverarbeitung, wenn die Zeit zur Berechnung einer Audioausgabe die Zeit des Abspielens nie überschreitet. Die Größe der Verzögerung zwischen Eingabe und Ausgabe (*Latenz*) ist dabei zunächst unwichtig.

Durch die zunehmende Verfügbarkeit leistungsfähiger Mikroprozessoren wurde die rein softwarebasierte Signalverarbeitung gerade im Bereich von Endbenutzerprodukten immer attraktiver. Die Gestaltung der Verarbeitungsvorgänge ist hier nur von der Rechenleistung der zugrundeliegenden Hardware begrenzt. Die zunehmende Integration datenparalleler Operationen in moderene Befehlssatzarchitekturen (wie z. B. die SIMD-Erweiterungen des x86 Befehlssatzes) begünstigt diese Entwicklung. Die verbreiteten Betriebssysteme sind keine Echtzeitbetriebssysteme, stellen aber Mechanismen zur Verfügung um dies zu kompensieren. Dazu gehören beispielsweise speziell priorisierte *Echtzeit-Threads*. Unter bestimmten Voraussetzungen kann damit die Echtzeitbedingung aus Definition 2.3.1 im Mittel auch hier erfüllt werden. Diese Voraussetzungen werden im folgenden Abschnitt für das Teilgebiet der digitalen Klangsynthese erläutert.

2.3.2 Softwarebasierte digitale Klangsynthese

Softwarebasierte digitale Klangsyntheseverfahren kommen heute vor allem in digitalen Musikinstrumenten zum Einsatz. Für die Implementierung werden entweder gewöhnliche Programmiersprachen oder domänenspezifische Entwicklungswerkzeuge verwendet. Zu den bekanntesten Vertretern gehören die quelloffenen Projekte Pure Data [Ste06] und SuperCollider [McC02] sowie die proprietären Produkte Max/MSP [Max12] und Reaktor

[Rea12]. Ähnlich den in der allgemeinen Anwendungsentwicklung verwendeten Werkzeugen, werden die damit erstellten Programme entweder in Maschinencode übersetzt oder von einer Ausführungsumgebung interpretiert. Aufgrund der hohen Perfomanzanforderungen der Echtzeitverarbeitung sind interpretative Ansätze für den professionellen Einsatz jedoch i. A. ungeeignet.

Der grundlegende Verarbeitungsablauf erfolgt stets nach dem EVA-Prinzip [CS03]. Für die Speicherung von Ein- und Ausgabe stehen sogenannte *Sample-Puffer* zur Verfügung, die einen Ausschnitt des digitalen Signals durch eine Folge diskreter Werte abbilden. Die Konvertierung zwischen analoger Ein- und Ausgabe und digitaler Repräsentation in den Sample-Puffern wird durch den Hardware-Abstraction-Layer des Betriebssystems abstrahiert. Der zeitliche Abstand zwischen dem Befüllen des Eingabepuffers und dem Auslesen des Ausgabepuffers bestimmt die *Verarbeitungslatenz* sowie, nach Definition 2.3.1, die Zeitschranke für die Berechung aller Ausgabesamples[13]. Wird diese Zeitschranke nicht eingehalten, kommt es i. A. zu einem zwischenzeitlichen Aussetzen der Audioausgabe, einem sogenannten *Drop-Out*.

Die mangelnde Echtzeitunterstützung der verbreiteten Betriebssysteme stellt gewisse Bedingungen an die Implementierung des Verarbeitungsprozesses. Diese Bedingungen werden von Ross Bencina in [Ben11] detailliert beschrieben. Im Kern seiner Aussage steht, dass die Verwendung von Operationen, deren Ausführungszeit im Vorfeld nicht sicher vorhergesagt werden kann, für die Zwecke der Audioverarbeitung in Echtzeit nicht akzeptabel ist. Daher sollten blockierende Anweisungen, Locking-Mechanismen, Speicherallokationen sowie Festplatten- und Netzwerkzugriffe im Code des Verarbeitungsprozesses vermieden werden. Zudem sollten keine externen Funktionen aufgerufen werden, die ihrerseits derartige Operationen verwenden. Unter Einhaltung dieser Bedingungen kann die Wahrscheinlichkeit von Drop-Outs minimiert werden.

[13] Die akzeptablen Werte für Verarbeitungslatenzen liegen heute bei etwa 5 ms [Ben11]. Bei einer Samplerate von 44100 Hz wird dafür eine Puffergröße von 256 Samples gewählt. Neben der Verarbeitungslatenz kommt es auch zu Verzögerungen bei der Ein- und Ausgabe der Signale aufgrund der DA- und AD-Konvertierung.

2.4 Reaktor Core

Reaktor ist eine Entwicklungs- und Ausführungsumgebung für modulare, digitale Musikinstrumente. Reaktor Core bezeichnet eine Progammiersprache zur Implementierung eigener Grundschaltelemente für die digitale Klangsynthese in Reaktor. Es handelt sich um eine domänenspezifische Sprache, die für Domänenexperten der digitalen Signalverarbeitung zugeschnitten ist. Die Sprache lässt sich als grafische, direkt ausführbare, externe, deklarative, kompilierte DSL einordnen. Die mit Reaktor Core erstellten Programme folgen dem datenstromorientierten Programmieransatz. Die Sprache ist stark spezialisiert und in ihrer Mächtigkeit auf die Erzeugung und Transformation digitaler Signale begrenzt.

2.4.1 Programmierumgebung

Innerhalb von Reaktor stehen virtuell-analoge elektrotechnische Grundschaltelemente, wie z.B. Oszillatoren, Filter, Hüllkurven und logische Operatoren zur Verfügung, die beliebig kombiniert werden können. Mittels einer geeigneten Parametrisierung lassen sich so Signale generieren, deren Eigenschaften und Klang denen echter Instrumente sehr nahe kommen [Hag01]. Reaktor folgt dem datenstromorientierten Programmieransatz, der die grafische Notation nach dem Prinzip der „Boxes and Arrows“ begünstigt (siehe 2.1.1). Dieser Ansatz ist in der digitalen Signalverarbeitung weit verbreitet und findet sich in ähnlicher Form auch in anderen Entwicklungsumgebungen wie z. B. LabVIEW [Lab12], Pure Data [Ste06] und Max/MSP [Max12]. Der Verarbeitungsvorgang wird dabei als gerichteter Graph dargestellt, dessen Knoten einzelne Verarbeitungsschritte (*Module*) repräsentieren. Die Verbindungen zwischen den Knoten des Graphen (*Wires*) definieren die Abhängigkeiten zwischen den Modulen und legen damit den Datenfluss sowie die Reihenfolge der Verarbeitungsschritte fest. Module sind entweder Grundschaltelemente oder *Macros*, die selbst wieder eine innere Struktur enthalten. Durch diese hierarchische Organisation der Modulstruktur wird eine übersichtliche Darstellung des Signalverarbeitungsprozesses erreicht.

Die verfügbaren Grundschaltelemente wurden üblicherweise direkt in C oder Assembler implementiert und waren damit innerhalb von Reaktor unveränderlich. Reaktor Core ergänzt diese Funktionalität in Reaktor unter Beibehaltung des gewohnten Bedienkonzepts und ermöglicht damit die Implementierung eigener Grundschaltelemente. Diese Ebene der Low-Level Programmierung wird als *Core Level* bezeichnet. Demgegenüber findet die herkömmliche Arbeit mit Reaktor auf dem *Primary Level* statt. Als Schnittstellen zwischen diesen Ebenen kommen spezielle Module zum Einsatz, sogenannte *Core Cells*.

Sie dienen als Container für die mit Reaktor Core entwickelten Grundschaltelemente.

Reaktor dient gleichzeitig als Entwicklungs- und Ausführungsumgebung für Reaktor Core und übernimmt u. a. Aufgaben wie die Ein- und Ausgabe von Daten und die grafische Darstellung einer Benutzeroberfläche. Die Programmierung in Reaktor Core ist auf die Erzeugung und Transformation digitaler Signale innerhalb von Core Cells reduziert.

Abbildung 2.2 zeigt die Verwendung eines Low-Pass Filters in einer Testumgebung[14]. Die *Struktur-Ansicht* im oberen Teil stellt das Verarbeitungsmodell auf Primary Level dar. Das Signal einer Sägezahnkurve und der Wert einer Cutoff-Frequenz dienen als Eingabeparameter des Low-Pass Filters. Das transformierte Ausgabesignal wird nicht akkustisch ausgegeben, sondern durch ein Oszilloskop grafisch dargestellt. Das Ergebnis ist im unteren Teil der Abbildung, der sogenannten *Panel-Ansicht*, zu sehen. Mit Hilfe der Regler „Pitch“, „Ampl“ und „Cutoff“ kann die Parametrisierung in Echtzeit variiert werden.

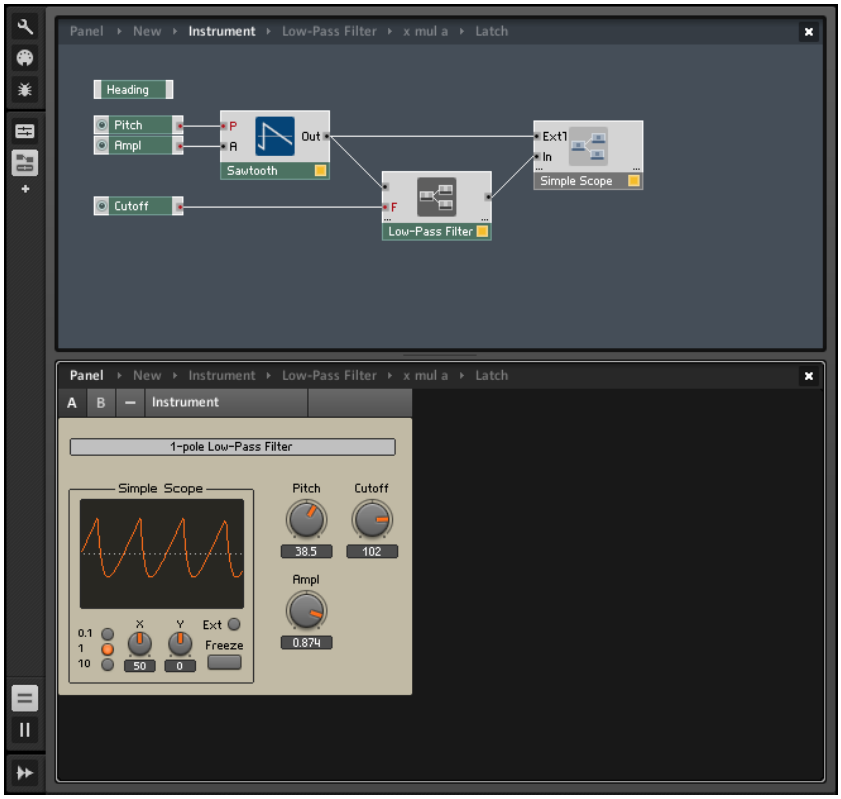

Abbildung 2.2 Testumgebung für einen Low-Pass Filter in Reaktor. Position und Darstellung der Bedienelemente werden in der Panel-Ansicht (unten) festgelegt. Ihre Aufgaben im Instrument gehen aus den Verknüpfungen in der Struktur-Ansicht (oben) hervor.

[14] Dieses Beispiel wurde der Reaktor 5 Core Reference entnommen [Nat10]

2.4.2 Einführendes Beispiel

Ausgehend von einer Core Cell erfolgt die Abbildung von Verarbeitungsvorgängen durch Module, Macros und Wires auf dem Core Level analog zum Primary Level. Die Primitive sind hier jedoch keine Grundschaltelemente sondern Maschinenoperationen. Dazu zählen arithmetische und logische Operationen sowie Lese- und Schreiboperationen.

Abbildung 2.3 zeigt die Implementierung des in Abbildung 2.2 verwendeten Low-Pass Filters in Reaktor Core. Die Struktur der Core Cell ist im oberen Teil der Abbildung dargestellt. Das Eingangssignal erreicht die Core Cell am oberen unbenannten Eingangsmodul (*In*). Der Wert der Cutoff-Frequenz steht am unteren Eingangsmodul mit der Beschriftung *F* bereit. Das transformierte Signal wird durch den unbenannten Ausgang (*Out*) am rechten Bildrand an die nächsthöhere Verarbeitungsebene weitergegeben. Die von einem einfachen einpoligen Low-Pass Filter ausgeführte Transformation bestimmt

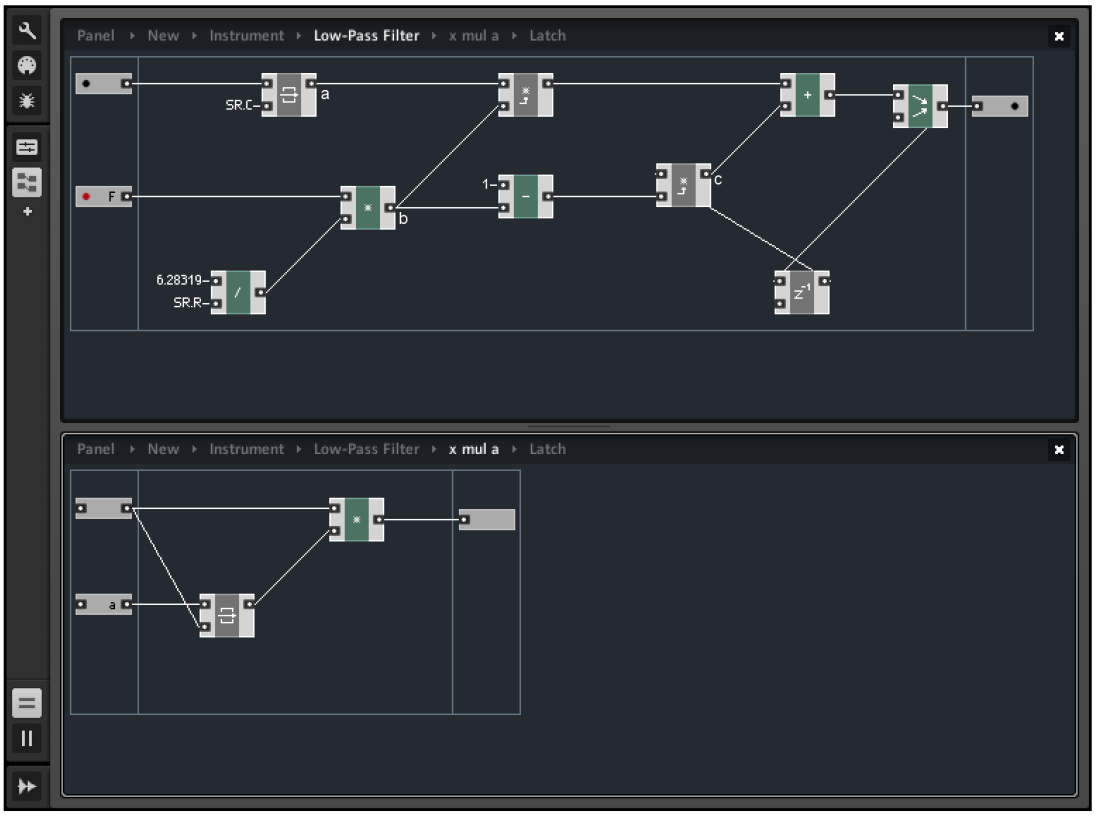

Abbildung 2.3 Implementierung eines einfachen Low-Pass Filters in Reaktor Core. Im oberen Teil der Abbildung ist die Struktur der Core Cell zu sehen. Module sind grün und Macros grau dargestellt. Die Struktur des Macros „x mul a“ ist in der zweiten Strukturansicht im unteren Teil dargestellt.

sich anhand der folgenden Rekursionsgleichung[15]:

$$\mathrm{y_i} \;=\; \mathrm{bx} + (1 - \mathrm{b})\mathrm{y_{i-1}}$$

Das Ausgabesample y_i ergibt sich als gewichtete Summe aus dem Eingabesample x und dem vorherigen Ausgabesample y_{i-1}. Als Gewichtung kommt die normalisierte zirkuläre Cutoff-Frequenz b zum Einsatz, die sich wie folgt aus der Eingabefrequenz F und der Samplerate f_{SR} ergibt:

$$b \;=\; \mathrm{F}\frac{2\pi}{\mathrm{f_{SR}}}$$

Eine Implementierung in Pseudo-Code könnte demnach wie folgt aussehen:

```
1 function LowPassFilter(In, F, y_{i-1}, f_SR)
2 {
3   a := In
4   b := F * (6,28319 / f_SR)
5   c := y_{i-1} * (1 - b)
6
7   Out := a * b + c
8   return Out
9 }
```

Die Berechnung kann in Form eines Schaltbildes direkt in der Notation von Reaktor Core dargestellt werden. Abbildung 2.3 zeigt die Referenzimplementierung im oberen Teil. Zum besseren Verständnis wurden die Variablennamen a, b und c an den Stellen ergänzt, an denen die jeweiligen Zwischenergebnisse zur Verfügung stehen.

Zur Erhöhung der Ausführungsgeschwindigkeit enthält die Implementierung einige Optimierungen. Anstatt einfacher Multiplikationen wird das sogenannte Modulations-Macro "x mul a" für die Berechnung von c und *Out* verwendet. Die Struktur dieses Macros ist im unteren Teil von Abbildung 2.3 dargestellt. Der Modulationswert a wird hier in einem *Latch* zwischengespeichert. An dieser Stelle werden die verschiedenen Berechnungsstränge des Low-Pass Filters getrennt, da sie unterschiedlich häufig ausgeführt werden müssen. Am oberen Eingangsmodul liegt ein Audiosignal an (schwarz), das mit jedem Takt der Samplerate verarbeitet wird[16]. Die Modulationswerte der Macros „x mul a" hängen jedoch ausschließlich vom Parameterwert F ab. Da sich dieser Wert nur vergleichsweise selten ändert, ist der Eingang als Event-Eingang (rot) markiert. Wurden die Modulationswerte einmal berechnet und von den Latches zwischengespeichert, müssen sie erst

[15] Dieses Beispiel wurde der Reaktor 5 Core Reference entnommen [Nat10]. Darüber hinaus finden sich Details zur Herleitung der angegebenen Gleichungen und zur Konstruktion virtuell-analoger Filter im Allgemeinen in [Zav12].

[16] Bei Verwendung der Standard-Samplerate etwa sind 44100 Neuberechnungen pro Sekunde nötig.

dann erneut berechnet werden, wenn sich der Wert des Parameters F ändert. In der überwiedenden Mehrheit der Fälle ist dies jedoch nicht nötig. Dann reduziert sich der Aufwand zur Berechnung der Transformation auf die Auswertung des Terms:

$$\mathrm{In} * \mathrm{L}_1 + \mathrm{y}_{\mathrm{i}-1} * \mathrm{L}_2 \tag{2.1}$$

Dabei bezeichnen L_1 und L_2 die durch die Latches zwischengespeicherten Werte. Dieses als *Latching* bekannte Verfahren kommt in der digitalen Signalverarbeitung sehr häufig und nicht nur für Optimierungen zum Einsatz.

2.4.3 Grundlegende Arbeitsweise

Ausgehend von ihrer hierarchischen Modulstruktur wird eine Core Cell vom Reaktor Core Compiler in direkt ausführbaren, optimierten Maschinencode der jeweiligen Befehlssatzarchitektur[17] kompiliert. Das Ergebnis liegt in Form mehrerer sogenannter *Handler* vor. Ein Handler ist eine einfache Funktionen die von der Ausführungsumgebung direkt aufgerufen werden kann. Es wird zwischen den folgenden Arten von Handlern unterschieden:

- Der *Init Handler* wird einmalig direkt im Anschluss an den Übersetzungsvorgang aufgerufen und dient der Initialisierung von Speicherfeldern.
- Der *Audio Handler* wird fortlaufend für jedes Sample im Sample-Puffer ausgeführt. Hier findet der eigentliche Signalverarbeitungsvorgang statt. Im Falle des Low-Pass Filters in Abbildung 2.3 wird vom Audio Handler nur der reduzierte Term (2.1) ausgewertet.
- *Event Handler* dienen der Modifikation von Parametern. Zwar kann die Kompilierung hier variieren, i. A. wird jedoch für jeden separat bedienbaren Parameter ein Event-Handler erzeugt.

Die Kompilierung wird bei jeder Änderungen an der Struktur einer Core Cell automatisch im Hintergrund ausgeführt. Nach Abschluss des Übersetzungsvorgangs kommt der kompilierte Programmcode sofort zur Ausführung. Damit bleiben Kompilierung und Ausführung für den Benutzer transparent. Der Eindruck einer kontinuierlichen Verarbeitung in Echtzeit kann dadurch auch während der Konstruktion von DSP-Programmen erhalten werden.

[17] Heute betriff dies im Wesentlichen die x86 Architektur von Intel. Variationen existieren jedoch beim Grad der Unterstützung von SSE-Optimierungen sowie der Adresslänge des verwendeten Systems.

3 Konzeption

In diesem Kapitel wird das allgemeine Konzept eines integrierten, interaktiven Same-Process Debuggers für externe, kompilierte, domänenspezifische Sprachen der digitalen Signalverarbeitung erarbeitet.

3.1 Vorbetrachtung

Unter domänenspezifischen Sprachen der digitalen Signalverarbeitung (Digital Signal Processing, *DSP*) ist eine starke Verflechtung zwischen Entwicklungsumgebung, Ausführungsumgebung und erstelltem Programm sehr verbreitet. Im Rückschluss erfolgt deren Ausführung oft in einem gemeinsamen Prozess des Betriebssystems. Aufwändige Prozesswechsel können damit vermieden werden. Ein integrierter Debugger kann in dieser Situation nur nach dem Same-Process Ansatz realisiert werden. In den verbreiteten Betriebssystemen existiert keine Unterstützung für diese Form von Debuggern. Ziel des Konzeptes ist es, möglichst weitreichend einsetzbare Realisierungsmöglichkeiten für die manuelle Nachbildung von Debugging-Mechanismen in Same-Process Debuggern zu finden. Im Mittelpunkt steht die Realisierung von Ablaufkontrollmechanismen unter den speziellen Anforderungen der Echtzeitverarbeitung in DSP-Programmen.

3.1.1 Ausgangssituation

Betrachtet wird eine Situation, in der Programme oder Programmfragmente zur digitalen Signalverarbeitung innerhalb einer Entwicklungsumgebung erstellt werden können. Diese Entwicklungsumgebung soll um einen integrierten, interaktiven Debugger erweitert werden, der über Ablaufkontrollmechanismen verfügt, die eine Inspektion erstellter Programme ermöglicht. Ein zu inspizierendes Programm wird als *Prüfling* bezeichnet. Die Entwicklungsumgebung enthält einen Compiler, der den Quellcode des Prüflings vor dessen Ausführung vollständig in Maschinencode der jeweiligen Plattform übersetzt. Als Plattform wird eines der verbreiteten Betriebssysteme, wie z. B. Mac OS X oder Windows, auf Basis einer x86 Befehlssatzarchitektur verwendet. Die Verwendung des Debuggers erfordert eine spezielle Kompilierung des Prüflings. Dafür wird im Compiler ein Debug-Modus eingerichtet.

Der Aufgabenbereich der betrachteten Programme begrenzt sich auf die Erzeugung und Transformation digitaler Signale. Alle weiteren Aufgaben, wie die Ein- und Ausgabe von Signaldaten oder die Darstellung grafischer Oberflächen übernimmt eine Ausführungsumgebung. Für die Zwecke der Signalverarbeitung wird der kompilierte Programmcode der erstellen Programme von der Ausführungsumgebung aufgerufen. Dies erfolgt in einem separaten, speziell priorisierten *Echtzeit-Thread*. Für die Ausführung im Echtzeit-Thread existieren fest vorgegebene Zeitschranken (2.3.1).

3.1.2 Grundlegende Problematik und Zielstellung

Die herkömmlichen Ansätze zur Realisierung von Debuggern (2.2.4) sind in einer solchen Situation nicht anwendbar, da sie stets nur die Unterbrechung eines ganzen Prozesses des Betriebssystems ermöglichen. Dadurch wäre auch die Entwicklungsumgebung selbst während einer Unterbrechung nicht bedienbar. Für die Konzeption eines integrierten Debuggers (2.2.3) kommt damit nur ein Ansatz in Frage, der die Inspektion einzelner Threads innerhalb des eigenen Prozesses erlaubt. Die Implementierung des zugrunde liegenden Verfahrens wurde von Kessler in [Kes90] bereits für eine SPARC Befehlssatzarchitektur beschrieben (2.2.5).

Im Zuge des vorliegenden Konzepts soll in Abschnitt 3.2 eine Übertragung dieses Ansatzes auf die heute verbreitete x86 Befehlssatzarchitektur vorgenommen werden. Im Konzept sollen die grundlegenden Konstrukture zur Ablaufsteuerung in strukturierten Programmiersprachen berücksichtigt werden. Konkret handelt es sich dabei um die sequenzielle Befehlsausführung, bedingte Verzweigungen des Kontrollflusses und Schleifen. Die Unterbrechung und Inspektion des Prüflings soll ausgehend von Interaktionen des Benutzers mit der Entwicklungsumgebung ermöglicht werden. Die Problematiken, die sich in diesem Zusammenhang aus der Echtzeitfähigkeit des Verarbeitungsprozesses ergeben, werden in Abschnitt 3.3 behandelt.

3.1.3 Wahl des Debugging-Verfahrens

Die betrachteten domänenspezifischen Sprachen begünstigen die Realisierung eines Omniscient-Debuggers (2.2.2.2). Ihr hohes Abstraktionsniveau verringert die Anzahl der im Quellcode verfügbaren Inspektionspunkte. Im Gegensatz zu universellen Programmiersprachen müsste ein Omniscient-Debugger den Ausführungszustand des Prüflings hier also seltener zwischenspeichern. Ihre begrenzte Mächtigkeit verringert die Vielfalt und Kombinierbarkeit verfügbarer Konstrukte zur Ablaufkontrolle. Der Maschinencode erstellter Programme weist damit häufiger ähnliche Strukturen auf als im Falle universeller

Programmiersprachen. Für die Aufzeichnung des Kontrollflusses genügt dann i. A. eine deutlich geringere Menge von Laufzeitdaten. Trotz dieser Vereinfachungen und den Vorteilen des Omniscient-Debugging, kann es im Kontext dieser Arbeit nicht angewendet werden. Dies hat folgende Ursachen. Die zusätzlichen Speicherzugriffe zur Aufzeichnung der Laufzeitdaten sind durchgängig – und insbesondere unabhängig von expliziten Unterbrechungen – nötig. Das könnte die Ausführungsgeschwindigkeit des Prüflings erheblich verringern und die Echtzeitfähigkeit des Verarbeitungsvorgangs gefährden. Im Falles eines Trace-Debuggers (2.2.2.1) hingegen ist die Interaktion mit dem Prüfling minimal, solange es nicht zu einer expliziten Unterbrechung kommt. Des Weiteren könnte ein Omniscient-Debugger die Weiterentwicklung der Sprache begrenzen. Beispielsweise könnte eine Ergänzung weiterer Konstrukte zur Ablaufkontrolle zu einer Vervielfachung der aufzuzeichnenden Laufzeitdaten führen. Die Auswirkungen auf die Echtzeitfähigkeit würden die Ergänzung der neuen Konstrukte möglicherweise verhindern. Eine solche Einschränkung ist trotz der Vorteile von Omniscient-Debuggern nicht akzeptabel. Die Konzeption zielt daher auf die Realisierung eines gewöhnlichen Trace-Debuggers.

3.2 Same-Process Debugger

Einzelschrittausführung und Breakpoints bilden die zentralen Ablaufkontrollmechanismen in Trace-Debuggern (2.2.2.1). Es folgt eine allgemeine Betrachtung zur Realisierbarkeit dieser Mechanismen für Same-Process Debugger unter den speziellen Anforderungen der Echtzeitverarbeitung.

3.2.1 Breakpoints

Die Befehle der x86 Befehlssatzarchiektur haben keine einheitliche Länge. Insbesondere ist ein *Call* Befehl, der zur Übertragung des Kontrollflusses an den Debugger verwendet werden soll (Breakpoint), länger als verschiedene andere Befehle. Damit kann es zum partiellen oder vollständigen Überschreiben von Folgebefehlen durch Breakpoints kommen. Handelt es sich bei einem solchen Folgebefehl um ein Sprungziel, ändert das die Semantik des Programms. Abbildung 3.1 stellt diese Problematik grafisch dar.

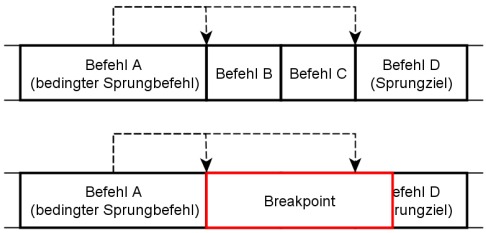

Abbildung 3.1 Beispiel für das Überschreiben von Folgebefehlen durch Breakpoints. Die Befehle A, B, C und D liegen im Speicher unmittelbar hintereinander. Nach A wird entweder B oder D ausgeführt (oben). Beim Setzen eines Breakpoints auf B wird ein Teil des Befehls D überschrieben (unten). Dies ändert die Programmsemantik, da D erreicht werden kann, bevor dessen ursprünglicher Befehlscode wiederhergestellt wurde.

Der obere Teil der Abbildung zeigt die vier im Speicher unmittelbar aufeinanderfolgenden Befehle A, B, C und D. Beim Befehl A handelt es sich um einen bedingten Sprungbefehl. Ist die Sprungbedingung erfüllt, wird die Ausführung im Anschluss bei Befehl D fortgesetzt. Damit ist D ein Sprungziel im dargestellten Programm. Ist die Sprungbedingung nicht erfüllt, wird der im Speicher nachfolgende Befehl B ausgeführt. Die Befehle B, C und D sollen einfache sequenzielle Operationen ausführen. Der untere Teil der Abbildung zeigt eine Situation, in der auf Befehl B ein Breakpoint gesetzt wurde, der länger ist als der Befehl selbst. Der Breakpoint überschreibt C vollständig und D partiell. Das Überschreiben von C ist unproblematisch, da vor C stets B ausgeführt wird. Erreicht die Ausführung den Breakpoint, wird der ursprüngliche Maschinencode von C

wiederhergestellt. Dies ist bei Befehl D nicht der Fall. Da D ein Sprungziel ist, besteht die Möglichkeit, dass die Ausführung D erreicht, während ein Teil des Befehls von dem Breakpoint auf dem Befehl B überlappt wird. Der durch diese Überlappung verfälschte Befehlscode kommt im Beispiel genau dann zur Ausführung, wenn die Sprungbedingung von A erfüllt ist. Der Breakpoint ändert damit die Semantik des Programms. Für diese Problematik sollen im Folgenden zwei Lösungsansätze vorgestellt werden.

3.2.1.1 Lösung durch Ausrichtung der Maschinenbefehle

Eine einfache Lösung wäre das Nachahmen der einheitlichen Befehlslänge der SPARC Befehlssatzarchitektur. Der Compiler könnte alle Maschinenbefehle auf ganzzahlige Vielfache der Länge eines Calls ausrichten. Jede entstehende Lücke könnte durch einen NOP[1] bzw. Multi-Byte-NOP[2] Befehl gefüllt werden. Beide werden im Folgenden unter der Bezeichnung *NOP* zusammengefasst. Breakpoints könnten weiterhin nur auf die ursprünglichen Befehle gesetzt werden. Sollte der zu ersetzende Befehl kürzer sein als der einzufügende Call, würde im Zweifel nur der darauffolgende NOP überschrieben. Dabei handelt es sich nur um Behelfscode, der kein Sprungziel sein kann. Das Problem wäre damit behoben. Für den Einsatz in der digitalen Signalverarbeitung ist diese Lösung ist jedoch ungeeignet. Die Gründe dafür liegen in dem damit verbundenen, vergleichsweise hohen Laufzeitoverhead. Die Ausrichtung aller Maschinenbefehle führt zur annähernden Verdopplung der Gesamtbefehlsmenge. Der überwiegende Teil der eingefügten NOPs folgt auf sequenzielle Befehle und kommt daher in jedem Verarbeitungsdurchlauf zur Ausführung. Zwar können NOP Befehle vom Prozessor i. A. in wenigen Takten verarbeitet werden, doch führt dies bei komplexen Verarbeitungsaufgaben zu einem erkennbaren Overhead. Dieser erhöht sich noch durch die Effekte, die durch die zusätzliche Streuung des Maschinencodes im Speicher entstehen. Da gerade häufig verwendete Operationen eine besonders geringe Befehlslänge aufweisen, kommt es zu einer Vervielfachung des für den Programmcode erforderlichen Speicherplatzes. Dies hat Konsequenzen wie größere Distanzen für Sprünge, eine verringerte Cache-Effizienz und nicht zuletzt die Gefahr von Page-Faults.

[1] Bei der Verarbeitung eines NOP Befehls („No-OPeration“) wird vom Prozessor keine Operation ausgeführt. In der x86 Befehlssatzarchitektur ist der NOP Befehl genau ein Byte lang [Int12a].

[2] Multi-Byte-NOPs werden durch normale Befehle simuliert, die so parametrisiert sind, dass sie nicht zur Veränderung des Ausführungszustandes führen. Vorschläge für entsprechende Befehle werden in [Int12a] für verschiedene Befehlslängen gegeben.

3.2.1.2 Lösung durch Einteilung in sequenzielle Befehlsblöcke

Optimierungen in DSP-Programmen zielen oft auf die Minimierung der Anzahl von Sprüngen im Maschinencode[3]. Es ergeben sich oft lange sequenzielle Befehlsfolgen. Vor diesem Hintergrund erscheint es sinnvoll, den Maschinencode des Prüflings als Abfolge *sequenzieller Befehlsblöcke* zu betrachten. Sequenzielle Befehlsblöcke bilden eine Erweiterung des Standardkonzepts von *Grundblöcken* im Compilerbau [ASL+08]. Ein sequenzieller Befehlsblock beginnt stets an einem Sprungziel und umfasst alle Maschinenbefehle bis zum nächsten Sprungziel. Beginn und Ende des Maschinencodes begrenzen jeweils den ersten und letzten sequenziellen Befehlsblock. Jeder Maschinenbefehl des Prüflingsprogramms ist damit genau einem sequenziellen Befehlsblock zugeordnet. Die Verarbeitung eines sequenziellen Befehlsblocks beginnt stets mit der ersten Anweisung (dem Sprungziel) und setzt sich unter Einhaltung der sequenziellen Befehlsreihenfolge im Speicher fort. Bis zu dieser Stelle gleichen sequenzielle Befehlsblöcke den Grundblöcken im Compilerbau. Der Unterschied besteht darin, dass Sprunganweisungen in sequenziellen Befehlsblöcken an beliebigen Stellen enthalten sein können. Damit muss die Ausführung nicht notwendigerweise das Ende eines sequenziellen Befehlsblocks erreichen.

Die sequenzielle Ausführungsreihenfolge innerhalb der Befehlsblöcke ermöglicht das koordinierte Einfügen und Entfernen von Breakpoints, da im Vorfeld bekannt ist, in welcher Reihenfolge sie getroffen werden. Ein Breakpoint wird damit nicht notwendigerweise sofort in den Maschinencode des Prüflings eingefügt. Im Hinblick auf die Terminologie soll in dieser Arbeit das *Setzen* die Registrierung des Breakpoints gegenüber dem Debugger bezeichnen. Das *Implementieren* eines Breakpoints entspricht dem Einfügen des entsprechenden Call Befehls im Maschinencode.

Es genügt, zunächst nur den jeweils ersten Breakpoint eines sequenziellen Befehlsblocks zu implementieren. Dies stellt die Ausgangssituation dar. Sie muss immer dann vorliegen, wenn die Ausführung den Befehlsblock erreicht. Im Zuge der Unterbrechung, die von einem Breakpoint ausgelöst wird, wird zunächst der ursprüngliche Maschinencode des gesamten sequenziellen Befehlsblocks wiederhergestellt und der darauffolgende Breakpoint implementiert. Die Ausführung wird an der Stelle des ursprünglichen Maschinenbefehls des getroffenen Breakpoints fortgesetzt.

Da die Ausführung den sequenziellen Befehlsblock jedoch an beliebigen Stellen verlassen kann, ist nicht sichergestellt, dass die nachfolgenden Breakpoints auch getroffen werden. Somit ist nicht klar, wann der erste Breakpoint erneut implementiert werden

[3] Für den Signalverarbeitungsvorgang haben Sprünge keinen direkten Nutzen. Zudem führen bedingte Sprünge oft zu Pipeline-Stalls im Prozessor und drosseln damit die Ausführungsgeschwindigkeit von Programmen.

muss, um die Ausgangssituation wiederherzustellen. Der erste Breakpoint eines sequenziellen Befehlsblocks nimmt daher eine Sonderstellung ein. Es muss sichergestellt sein, dass er erneut implementiert wird, bevor die Ausführung den sequenziellen Befehlsblock verlässt. Dies kann analog zur Vorgehensweise herkömmlicher Debugger (siehe 2.2.4.2) nach der Ausführung eines impliziten Einzelschritts erfolgen. Implizite Einzelschritte stellen einen Behelfsmechanismus dar, um den Kontrollfluss erneut an den Debugger zu übertragen. Sie führen nicht zu einer sichtbaren Unterbrechung der Ausführung[4]. Hier besteht jedoch die Möglichkeit, dass der Breakpoint nach dem Einzelschritt noch nicht erneut implementiert werden kann. Dies ist der Fall, wenn der durch den Einzelschritt erreichte Folgebefehl noch innerhalb des vom ersten Breakpoint überlappten Speicherbereichs liegt. Demzufolge müssen solange implizite Einzelschritte ausgeführt werden, bis ein Befehl erreicht wird, der nicht mehr vom ersten Breakpoint des Befehlsblocks überlappt wird. Zu diesem Zeitpunkt kann der erste Breakpoint erneut implementiert werden. Alle nachfolgenden Breakpoints des sequenziellen Befehlsblocks werden dann, wie eingangs beschrieben, sukzessive implementiert. Für diese Breakpoints sind keine impliziten Einzelschritte erforderlich, da vor ihnen stets der erste Breakpoint getroffen wird, wenn die Ausführung den sequenziellen Befehlsblock erneut erreicht.

Um eine Überlappung über die Grenzen eines sequenziellen Befehlsblocks hinaus zu verhindern, wird vom Compiler zwischen den sequenziellen Befehlsblöcken (und damit vor jedem Sprungziel) jeweils ein Puffer eingefügt. Die Größe des Puffers bemisst sich an der Länge eines Call Befehls. Als Befehlscode in diesem Puffer kommt ein NOP Befehl zum Einsatz. Er kommt nur beim direkten Übergang zweier aufeinanderfolgender sequenzieller Befehlsblöcke zur Ausführung. Die Streuung des Maschinencodes nimmt im Vergleich zu 3.2.1.1 nur in geringem Umfang zu. Der resultierende Laufzeitoverhead ist im Hinblick auf die oft langen sequenziellen Befehlsblöcke in DSP-Programmen vertretbar.

3.2.1.3 Nachbetrachtung

Bei der Verwendung von Calls zur Realisierung von Breakpoints in Same-Process Debuggern wird eine spezielle Kompilierung des Prüflings nach 3.2.1.1 oder 3.2.1.2 erforderlich. Programmcode, der nicht in dieser Weise präpariert wurde, kann nicht unter Gewährleistung der grundlegenden Prinzipien von Debuggern (2.2.1) inspiziert werden. Die Einteilung des Prüflingscodes in sequenzielle Befehlsblöcke erweist sich für DSP-Programme als vorteilhaft und wird nachfolgend verwendet. Für das Einfügen von Puffern im Ma-

[4]Sollte durch einen impliziten Einzelschritt eine Stelle im Maschinencode erreicht werden, für die ein Breakpoint gesetzt ist, muss dies vom Debugger erkannt werden. Dann wird zusätzlich zur Behelfsfunktion des impliziten Einzelschritts eine sichtbare Unterbrechung erforderlich.

schinencode des Prüflings muss der Compiler sämtliche Sprungziele im Programmablauf identifizieren. Diese Daten können auch für das in 3.2.2 beschriebene Verfahren zur Einzelschrittausführung verwendet werden.

Im Falle herkömmlicher Debugger wird der Ausführungszustand des Prüflings vor bzw. nach einer Unterbrechung im Zuge des Prozesswechsels vom Betriebssystem automatisch gesichert bzw. wiederhergestellt. Da die Prozesswechsel bei Same-Process Debuggern entfallen, muss die Sicherung und Wiederherstellung des Ausführungszustandes hier manuell erfolgen. Das Setzen von Breakpoints auf Sprachebene ist unabhängig vom zugrundeliegenden Realisierungsansatz und kann nach der Vorgehensweise herkömmlicher Debugger erfolgen (2.2.4.2).

Im Gegensatz zum Konzept Kesslers wird der durch einen Breakpoint überschriebene Maschinenbefehl hier nicht innerhalb der Behandlungsroutine ausgeführt. Denn dafür wäre ein erheblicher Mehraufwand für die Umcodierung relativer Sprungdistanzen nötig. Stattdessen wird ein Breakpoint im Zuge der Unterbrechung wieder durch den ursprünglichen Maschinencode ersetzt (im Falle des ersten Breakpoints nur vorübergehend). Nach der Unterbrechung wird die Ausführung nicht am Befehl *nach* dem Breakpoint fortgesetzt, sondern an der Stelle des wiederhergestellten Maschinencodes. Dieses Vorgehen orientiert sich an der Verfahrensweise herkömmlicher Debugger.

3.2.2 Einzelschrittausführung

Die folgende Betrachtung konzentriert sich auf die Realisierung von Instruction-Level-Steps. Source-Level-Steps können dann analog zur Verfahrensweise herkömmlicher Debugger durch wiederholte Instruction-Level-Steps oder Breakpoints realisiert werden (siehe 2.2.4.3). Seperate-Process Debugger verwenden das Trap-Flag des Prozessors für Instruction-Level-Steps. Same-Process Debugger können das Trap-Flag des Prozessors nicht verwenden. Die Einzelschrittausführung kann jedoch mit Hilfe *temporärer Breakpoints* nachgebildet werden. Im Gegensatz zu herkömmlichen, sogenannten *permanenten Breakpoints* werden temporäre Breakpoints nach erfolgter Unterbrechung der Ausführung nicht erneut implementiert.

3.2.2.1 Voraussetzungen

Vorausgesetzt wird ein funktionierender Breakpoint-Mechanismus nach 3.2.1. Zudem werden die Speicheradressen von Sprungbefehlen und deren Sprungziele im Maschinencode des Prüflings benötigt. Diese können entweder vom Debugger durch Befehlsdekodierung an der Stelle einer Unterbrechung gewonnen oder vom Compiler bereitgestellt werden.

3.2.2.2 Ablauf

Wurde die Ausführung des Prüflings an einem Breakpoint unterbrochen, wird zunächst der unveränderte Originalzustand des Prüflings wiederhergestellt (2.2.4.2). Der nächste auszuführende Maschinenbefehl ist der Befehl an der Speicherstelle der Unterbrechung. Dabei kann es sich entweder um einen Sprungbefehl[5] oder um eine beliebige andere Operation handeln. Im Falle eines beliebigen anderen Befehls ist die Ausführung eines Instruction-Level-Steps trivial. Das Setzen eines temporären Breakpoints auf den im Speicher direkt folgenden Maschinenbefehl genügt. Handelt es sich um einen Sprungbefehl, gibt es je nach Auswertung der Sprungbedingung zwei mögliche Folgebefehle: den im Speicher direkt folgenden Maschinenbefehl und den Maschinenbefehl an der Stelle des Sprungziels. Liegen die Daten aller Sprungbefehle und -ziele bereits vor, stellt das Setzen temporärer Breakpoints auf beide mögliche Folgebefehle eine einfache Lösung dar. Andernfalls ist die Dekodierung des Befehlscodes ohnehin erforderlich. Dabei könnte auch das für die Sprungbedingung zuständige Flag gefunden und ausgewertet werden, um den tatsächlichen Folgebefehl vorherzusagen.

Nach erfolgtem Einzelschritt kommt es erneut zu einer Unterbrechung der Ausführung. Da dem Benutzer die Inspektion des ursprünglichen Prüflingscodes möglich sein soll, müssen wie üblich alle Breakpoints für die Dauer der Unterbrechung entfernt werden. Temporäre Breakpoints kommen im hier vorgestellten Konzept nur zum Zweck der Einzelschrittausführung zum Einsatz und müssen nach einer Unterbrechung nicht erneut implementiert werden. Nach einem Einzelschritt können alle temporären Breakpoints daher auch aus der Registrierung im Debugger entfernt werden. Die Vorgehensweise ist hier also unabhängig davon, wie viele temporäre Breakpoints für die Realisierung des Einzelschrittes verwendetet worden sind.

3.2.2.3 Überlappungsproblem bei Schleifen

Bei der Verwendung des in 3.2.1.2 beschriebenen Verfahrens zur Realisierung von Breakpoints kann sich ein solches Einzelschrittverfahren als problematisch erweisen. Die erneute Implementierung des ersten Breakpoints eines sequenziellen Befehlsblocks erfolgt erst nach der Ausführung des ursprünglichen Maschinenbefehls in einem Einzelschritt. Das hier beschriebene Einzelschrittverfahren wird jedoch ebenfalls mit Breakpoints realisiert. Dies ist unproblematisch, solange die möglichen Folgebefehle im Speicher hinter dem

[5] Bei Sprungbefehlen kann zudem zwischen bedingten und unbedingten Sprüngen unterschieden werden. Sprungbefehle kommen auch für die Realisierung von Schleifen zum Einsatz. Eine Unterscheidung der verschiedenen Sprungbefehle würde die Betrachtung jedoch unnötig verkomplizieren. Es wird stets von bedingten Sprungbefehlen ausgegangen.

auszuführenden Maschinenbefehl liegen. Denn dann kann der auszuführende Maschinenbefehl selbst nicht durch die nötigen temporären Breakpoints überschrieben werden. Im Falle einer Schleife liegt das Sprungziel im Speicher jedoch vor der Sprunganweisung. Die Implementierung eines temporären Breakpoints an der Stelle des Sprungziels könnte damit die auszuführende Sprunganweisung überschreiben. Eine solche Überlappung würde die Programmsemantik ändern.

Abbildung 3.2 zeigt in Teil A ein Beispiel einer solchen Situation[6]. Den Ausgangspunkt bildet der in A.0 dargestellte sequenzielle Befehlsblock, der mit einer Schleife beginnt (Speicherstelle s_5). Die dafür verwendete Loop Anweisung dekrementiert den Wert des Zählregisters ECX in jedem Durchlauf [Int12a]. Die Schleife wird verlassen, wenn ECX den Wert Null enthält. Dann wird die Ausführung beim Folgebefehl an der Speicherstelle s_7 fortgesetzt. Das Sprungziel im Wiederholungsfall ist die Schleifenanweisung selbst. Nach 3.2.1.2 wurde vom Compiler vor jedem Sprungziel ein Puffer vorbereitet, der die Länge eines Call Befehls hat (1 Byte Opcode und 4 Byte relative Sprungadresse). In A.1 wurde ein Breakpoint auf den Loop Befehl bei s_5 gesetzt. Der ursprüngliche Maschinencode wurde vom Debugger zwischengespeichert und mit einem Call Befehl überschrieben. Wenn die Ausführung den sequenziellen Befehlsblock erreicht (A.2), kommt es zu einer Unterbrechung. Nach der Unterbrechung soll der ursprüngliche Befehl an der Speicherstelle s_5 zunächst in einem Einzelschritt ausgeführt werden, um den Breakpoint im Anschluss erneut implementieren zu können[7]. Wird dieser Einzelschritt wie zuvor beschrieben mit Hilfe temporärer Breakpoints realisiert, müssen Breakpoints auf alle in Frage kommenden Folgebefehle gesetzt werden. Da es sich hier um eine bedingte Sprunganweisung handelt, gibt es zwei mögliche Folgebefehle: den im Speicher nachfolgenden Befehl (an der Stelle s_7) und das Sprungziel (an der Stelle s_5). Dabei kommt es zu der in A.3 dargestellten Überlappung[8]. Der ursprüngliche Maschinenbefehl an der Stelle des Breakpoints wurde überschrieben.

Dieser Sonderfall muss separat behandelt werden. Konkret kann die betroffene Sprunganweisung so korrigiert werden, dass sie einen Sprung auf den Maschinenbefehl im Puffer vor dem ursprünglichen Sprungziel ausführt. Eine Überlappung ist dann ausgeschlossen. Dies hat keine Auswirkung auf die Programmsemantik des Prüflings, da es sich bei dem Befehl im Puffer nach 3.2.1.2 ohnehin um einen NOP handelt. Eine solche Korrektur ist stets möglich, da vor jedem Sprungziel immer ein Puffer existiert.

[6] Angenommen wird ein x86 Befehlssatz und eine Adressbreite von 32 Bit.
[7] Der Loop Befehl ist der erste Maschinenbefehl des sequenziellen Segments.
[8] Je nach Reihenfolge der Implementierung könnte der Breakpoint bei s_5 auch die ersten drei Byte des Breakpoints bei s_7 überschreiben. Dann kommt es zu einer Verfälschung des Befehlscodes beim Verlassen der Schleife.

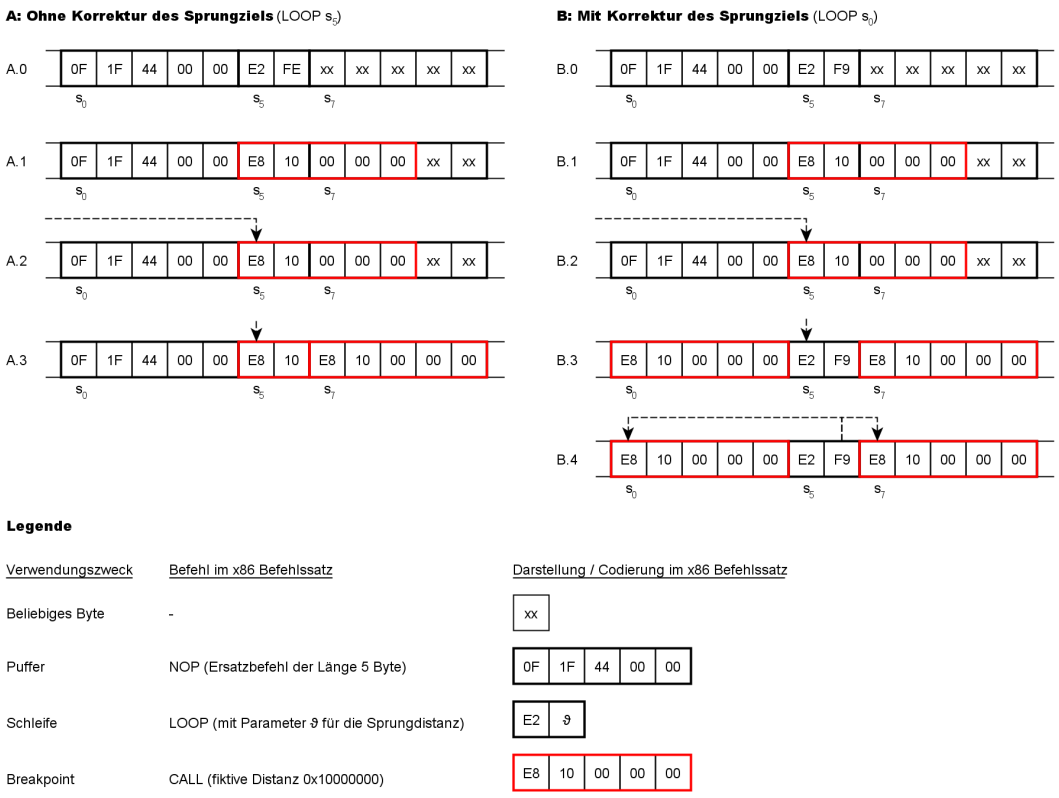

Abbildung 3.2 Beispiel des Überlappungsproblems bei Schleifen. Die Ausgangspunkte bilden die in A.0 und B.0 dargestellten sequenziellen Befehlsblöcke. Beide beginnen mit einer Schleifenanweisung bei s_5. Dargestellt sind ebenfalls die vorgelagerten Puffer bei s_0. Das Sprungziel der Schleifenanweisung in A.0 ist die Speicherstelle s_5 (die Schleifenanweisung selbst). B.0 zeigt die korrigierte Variante. Hier erfolgt der Sprung im Wiederholungsfall zum Maschinenbefehl im Puffer bei s_0. Soll im Anschluss an eine Unterbrechung bei s_5 (A.2 bzw. B.2) die Ausführung des ursprünglichen Maschinenbefehls in einem Einzelschritt erfolgen, müssen temporäre Breakpoints an den Stellen der möglichen Folgebefehle gesetzt werden. Im Falle des unkorrigierten Sprungbefehls in A wird der ursprüngliche Maschinenbefehl damit überschrieben (A.3). Durch die Korrektur in B kann dies vermieden werden (B.3).

Der Ablauf unter Verwendung einer solchen Korrektur ist in Teil B von Abbildung 3.2 dargestellt. Der Loop Befehl an der Speicherstelle s_5 hat hier s_0 als Sprungziel im Wiederholungsfall. Wie in B.3 zu sehen ist, überlappen sich die für den Einzelschritt nötigen Breakpoints hier nicht. Nach erfolgtem Einzelschritt (B.4), wird zunächst der ursprüngliche Maschinencode an den Stellen der temporären Breakpoints wiederhergestellt. Dies führt zur Ausgangssituation in B.0. Im Anschluss kann der permanente Breakpoint bei s_5 erneut implementiert werden. Der resultierende Laufzeitoverhead von einem Maschinenbefehl pro Schleifendurchlauf ist im Hinblick auf die Seltenheit einer solchen Situation vertretbar. Die Korrektur der Sprungadresse könnte zudem bereits zur Übersetzungszeit vom Compiler vorgenommen werden.

3.3 Echtzeitfähigkeit

In der digitalen Signalverarbeitung bildet die Echtzeitverarbeitung die Grundlage des beobachtbaren Verhaltens des Prüflings und ist damit Voraussetzung für das Erkennen einer Fehlerwirkung[9]. Der verantwortliche Defekt wiederum kann nur anhand seiner Fehlerwirkung identifiziert werden. Eine zentrale Anforderung stellt daher die Erhaltung der Echtzeitverarbeitung bei der Verwendung des Debuggers dar. Ausnahmen bilden explizite Unterbrechungen zu Inspektionszwecken. Die Echtzeitverarbeitung muss möglichst unmittelbar nach einer solchen Unterbrechung wieder aufgenommen werden können.

3.3.1 Ablauf der Echtzeitverarbeitung

Die Ausführung von Echtzeitprogrammen erfolgt in speziell priorisierten *Echtzeit-Threads* des Betriebssystems. Für diese Threads wird in machen Betriebssystemen ein kooperatives Scheduling-Verfahren angewendet. Echtzeit-Threads müssen dafür bestimmte Bedingungen erfüllen, wie z. B. die Einhaltung einer vorgegebenen *Zeitschranke* (2.3.1). Ein Verstoß gegen diese Bedingungen wird i. A. mit einer Herabstufung der Prozesspriorität geahndet [App12]. Eine derartige Herabstufung muss in Programmen der digitale Signalverarbeitung in jedem Fall vermieden werden. Eine Überschreitung der Zeitschranke könnte zudem von der Ausführungsumgebung als Fehler gewertet werden und in nachgeschalteten Verarbeitungsschritten oder angeschlossenen Hardware-Geräten kritische Folgefehler nach sich ziehen.

Die Ausführung der Echtzeitverarbeitung wird in vorgegebenen Zeitabständen vom Betriebssystem oder einem speziellen Gerätetreiber gestartet. Dies erfolgt durch eine Benachrichtigung der Ausführungsumgebung, eine vorgegebene Anzahl digitaler Signalwerte zu berechnen. Die Ausführungsumgebung übernimmt alle nötigen Verwaltungsaufgaben und ruft das DSP-Programm für die Berechnung der einzelnen Signalwerte auf. Die Berechnung eines einzelnen Signalwertes durch ein DSP-Programm wird im Folgenden als *Verarbeitungsdurchlauf* bezeichnet. Die Berechnung aller Signalwerte muss vor Ablauf der Zeitschranke abgeschlossen sein. Der Verarbeitungsvorgang gilt als abgeschlossen, sobald die Ausführung im Echtzeit-Thread zum Betriebssystem zurückkehrt.

Im Falle der digitalen Klangsynthese werden einzelne Signalwerte als *Samples* bezeichnet. Bei jedem Aufruf der Ausführungsumgebung muss eine bestimmte Anzahl von Samples berechnet werden. Die Datenstruktur zur Speicherung dieser Samples heißt *Sample-Puffer*.

[9] Im Falle der digitalen Klangsynthese kann eine Fehlerwirkung z. B. ein Knacksen in der Audioausgabe sein.

3.3.2 Einfügen von Breakpoints zur Laufzeit

Das Einfügen von Software Breakpoints erfordert einen Schreibzugriff im Speicherbereich des Prüflingscodes (2.2.4.2). In herkömmlichen Debuggern können Breakpoints im Maschinencode des Prüflings implementiert werden, wenn der Prüflingsprozess das nächste Mal vom Dispatcher des Betriebssystems in den Wartezustand versetzt wird. Same-Process Debugger können dieser Verfahrensweise nicht folgen, da sie im Prozess des Prüflings ausgeführt werden (2.2.5). Damit stellt sich die Frage, wie Breakpoints zur Laufzeit des Prüflings implementiert werden können. Im folgenden Abschnitt soll dafür zunächst eine weitere Problematik betrachtet werden, die sich aus der Echtzeitfähigkeit des Verarbeitungsprozesses im Prüfling ergibt.

3.3.2.1 Problematik blockierender Anweisungen

Eine Lösung könnte die Verwendung blockierender Anweisungen darstellen. Der Prüflingscode für die Echtzeitverarbeitung würde dann innerhalb eines kritischen Abschnittes ausgeführt. Eine atomare Anweisung zu Beginn des Verarbeitungsvorgangs würde beispielsweise auf die Freigabe eines Mutex' warten. Wird dieser Mutex vom Thread des Debuggers gehalten, wäre sichergestellt, dass der Prüflingscode zu diesem Zeitpunkt nicht ausgeführt wird und ein Schreibzugriff erfolgen kann. Nach Abschluss der Ersetzungsvorgänge zum Einfügen von Breakpoints, würde der Mutex vom Debugger freigegeben und die Ausführung des Prüflingscodes fortgesetzt.

In den verbeiteten Betriebssystemen kann die Dauer der Verzögerung, die durch die Ausführung blockierender Anweisungen entsteht, jedoch im Vorfeld nicht sicher bestimmt werden. Gemäß den in Abschnitt 2.3.2 vorgestellten Bedingungen für die zuverlässige Durchführung der Echtzeitverarbeitung können blockierenden Anweisungen daher nicht verwendet werden.

3.3.2.2 Lösung durch Callback-Points

Unter der Voraussetzung, dass es innerhalb des Verarbeitungsprozesses nur einen Echtzeit-Thread gibt, der den zu inspizierenden Programmcode verarbeitet, stellt das Einfügen von Breakpoints durch den Echtzeit-Thread selbst eine adäquate Lösung dar. Zu Beginn eines Verarbeitungsvorgangs könnte vom Prüfling selbst ermittelt werden, ob das Einfügen von Breakpoints erforderlich ist. Der dafür nötige Programmcode wird im Folgenden als *Callback-Point* bezeichnet. Er muss zur Übersetzungszeit vom Compiler erzeugt werden. Da Debugger und Prüfling im Same-Process Fall über einen gemeinsamen Speicherbereich verfügen, kann für die Koordinierung ein einfaches Flag verwendet werden. Solange dieses

Callback-Flag nicht gesetzt ist, wird die Echtzeitverarbeitung wie gewohnt fortgesetzt. In diesem Fall ist der entstehende Laufzeitoverhead moderat. Er begrenzt sich im Wesentlichen auf die Ausführung eines Speicherzugriffs und eines bedingten Sprungs. Andernfalls wird eine Routine des Debuggers aufgerufen, die das Callback-Flag zurücksetzt und die nötigen Ersetzungsvorgänge im Programmcode des Prüflings vornimmt. Alle nötigen Daten sollten bereits im Vorfeld vom Debugger vorbereitet worden sein. Im Optimalfall fallen für jeden einzufügenden Breakpoint nur wenige Speicheroperationen an. Diese begrenzen sich auf das Zwischenspeichern des ursprünglichen Maschinencodes und das Einfügen eines Call Befehls. Die Sicherung und Wiederherstellung des Ausführungszustandes des Prüflings kann entfallen, da sich der Callback-Point vor der ersten Anweisung des Verarbeitungscodes befindet. Zudem kann von Seiten des Debuggers sichergestellt werden, dass das Callback-Flag nur zu bestimmten Zeitpunkten oder in bestimmten Abständen gesetzt wird. In der digitalen Klangsynthese könnte das Callback-Flag beispielsweise nur nach Abschluss der Berechnung aller Samples des Sample-Puffers gesetzt werden. Die Ausführung von Ersetzungsvorgängen würde dann pro Sample-Puffer maximal einmalig erfolgen. Dies begrenzt den erforderlichen Mehraufwand im Worst-Case.

3.3.3 Unterbrechung der Echtzeitverarbeitung

Eine wesentliche Problematik stellt die Unterbrechung der Echtzeitverarbeitung zur Inspektion des Prüflings dar. Unterbrechungen erfolgen innerhalb eines Verarbeitungsdurchlaufs. Der Ausführungszustand des Prüflings zum Zeitpunkt der Unterbrechung muss dabei erhalten bleiben. Im Anschluss muss die Ausführung an der Stelle der Unterbrechung fortgesetzt werden können.

3.3.3.1 Problembeschreibung

Die Ausführung im Echtzeit-Thread muss vor Ablauf der vorgegebenen Zeitschranke zurückkehren. Die für die nötigen Berechnungsaufgaben verfügbare Zeitspanne liegt i. A. im Millisekundenbereich. Würden Unterbrechungen zur Inspektion des Prüflings mit Hilfe blockierender Anweisungen realisiert, würde sich die Rückkehr des Echtzeit-Threads um die Dauer der Unterbrechung verzögern. Die vorgegebene Zeitschranke würde damit in jedem Fall überschritten. Ein Blockieren der Echtzeitverarbeitung ist daher als Lösung ungeeignet.

3.3.3.2 Lösungsansatz

Eine akzeptable Lösung muss also zunächst gewährleisten, dass die Rückkehr der Ausführung im Echtzeit-Thread durch eine Unterbrechung nicht wesentlich verzögert wird. Für die Zwecke der späteren Inspektion könnte der Ausführungszustand des Prüflings an der Stelle der Unterbrechung gesichert und der begonnene Verarbeitungsdurchlauf regulär abgeschlossen werden. Nachfolgend getroffene Breakpoints könnten entweder ignoriert oder ebenso behandelt werden. Die Sicherung des Ausführungszustandes würde eine Post-Mortem-Inspektion (2.2.3) eines Verarbeitungsdurchlaufs des Prüflings ermöglichen. Die Unterbrechung und anschließende Fortsetzung der Ausführung ist jedoch nicht möglich. Der Lösungsansatz eignet sich in dieser Form daher noch nicht für die Realisierung eines interaktiven Debuggers.

Die Fortsetzung eines zuvor unterbrochenen Verarbeitungsdurchlaufs wird erst durch eine Manipulation des Rücksprungverhaltens der Ausführung an der Stelle der Unterbrechung möglich. Abbildung 3.3 soll die Idee hinter dem in Abschnitt 3.3.3.3 vorgestellten Verfahren illustrieren.

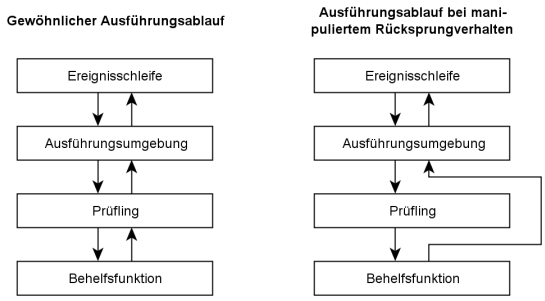

Abbildung 3.3 Manipulation des Rücksprungverhaltens von Funktionen. Im linken Teil ist der gewöhnliche Ausführungsablauf bei der Verwendung von Funktionen dargestellt. Er folgt der Symmetrie von Funktionsaufruf und -rücksprung. Im rechten Teil ist ein Ausführungsablauf zu sehen, der in 3.3.3.3 durch die Manipulation des Rücksprungverhaltens von Funktionen erreicht werden soll.

In der linken Hälfte der Abbildung ist eine gewöhnliche Ausführungsabfolge von Funktionen dargestellt[10]. Ausgehend von einer Funktion `Ereignisschleife` werden sukzessive Funktionen aufgerufen, die selbst weitere Funktionen aufrufen. Damit erreicht die Ausführung nacheinander die Funktionen `Ausführungsumgebung`, `Prüfling` und `Behelfsfunktion`. Innerhalb der Behelfsfunktion erfolgt kein weiterer Funktionsaufruf. Es folgt eine Rückkehr zur Funktion Prüfling. Dort wird die Ausführung an dem Befehl fortge-

[10] Die Benennung der Funktionen erfolgt in Anlehnung an die betrachtete Situation. Die vergebenen Namen repräsentieren Funktionen, werden aber im Folgenden wie die Elemente des Signalverarbeitungssystems verwendet.

setzt, der auf den Befehl zum Aufruf der Behelfsfunktion folgt. Nach der Abarbeitung von Prüfling kommt es auf dieselbe Weise zur Rückkehr zur Ausführungsumgebung und schließlich zur Ereignisschleife. Klar zu erkennen ist hier die *Symmetrie von Funktionsaufruf und -rücksprung*. Dabei handelt es sich um eine grundlegende Eigenschaft des Ablaufkontrollkonstruktes der *Funktion* in der Informatik. In der rechten Hälfte der Abbildung sind die gleichen Funktionen dargestellt. Sie werden ebenfalls in der gleichen Reihenfolge erreicht wie im vorherigen Fall. Der entscheidende Unterschied zur Ausführungsabfolge im linken Teil besteht im Rücksprungverhalten der Behelfsfunktion. Statt zum Prüfling zurückzukehren erfolgt hier ein direkter Rücksprung zur Ausführungsumgebung. Damit wird die Symmetrie der Funktionsaufrufe durchbrochen. Alle Befehle die im Prüfling hinter der Stelle des Aufrufs der Behelfsfunktion stehen, wurden nicht verarbeitet.

Diese Idee bildet die Grundlage des im folgenden Abschnitt vorgestellten Konzepts zur Unterbrechung der Echtzeitverarbeitung im Prüfling. Der Rücksprungbefehl ist der letzte Befehl im Programmcode des Prüflings. Wird dieser nicht ausgeführt, kommt es auch nicht zum Abbau des Stackframes der Funktion im Aufrufstack. Durch den direkten Rücksprung zur Ausführungsumgebung wird der Zustand der Stackframes aller danach aufgerufenen Funktionen eingefroren. Darauf aufbauend kann die Ausführung zu einem späteren Zeitpunkt fortgesetzt werden. Die Ausführung im Echtzeit-Thread hingegen kann unter Einhaltung der vorgegeben Zeitschranke zurückkehren. Damit sind alle Voraussetzungen für die Realisierung von Unterbrechungen der Echtzeitverarbeitung erfüllt.

3.3.3.3 Konkretes Lösungskonzept

Zunächst wird vor jedem Verarbeitungsdurchlauf des Prüflings der Aufrufstack des Echtzeit-Threads in einen separaten Speicherbereich umgeleitet. Dieser Speicherbereich heißt *separater Stackspeicher*. Demgegenüber wird der Speicherbereich, der normalerweise für die Aufnahme der Stackframes eines Threads vorgesehen ist, als *regulärer Stackspeicherbereich* bezeichnet. Diejenige Funktion in der Ausführungsabfolge des Echtzeit-Threads, an der die Umleitung des Aufrufstacks stattfindet, heißt *Entkopplungspunkt*. Im Beispiel aus Abbildung 3.3 handelt es sich bei der Funktion `Ausführungsumgebung` um den Entkopplungspunkt. Die Stackframes aller Funktionsaufrufe bis einschließlich dem Entkopplungspunkt liegen im regulären Stackspeicherbereich des Echtzeit-Threads. Für die Durchführung der Umleitung genügt es im Wesentlichen, die Werte der Stackregister[11]

[11] Unter dem Begriff *Stackregister* sollen hier die Register für Stack-Pointer und Base-Pointer des Prozessors zusammengefasst werden.

des Prozessors am Entkopplungspunkt mit geeigneten Speicheradressen des separaten Stackspeichers zu überschreiben. Die Stackframes aller nachfolgenden Funktionsaufrufe werden dann im separaten Stackspeicher aufgebaut. Um den Aufrufstack später wieder auf den regulären Stackspeicherbereich zurückführen zu können, werden die ursprünglichen Werte der Stackregister zuvor gesichert.

Wird während der Verarbeitung des Prüflingsprogramms kein Breakpoint getroffen, muss die Verarbeitung des Prüflings nicht unterbrochen werden. In diesem Fall kehrt die Ausführung im Echtzeit-Thread regulär zum Entkopplungspunkt zurück. An dieser Stelle wird wieder der Beginn des separaten Stackspeichers erreicht. Die Werte der Stackregister müssen nun aus den am Entkopplungspunkt gesicherten Werten wiederhergestellt werden. Die Ausführung im Echtzeit-Thread wird regulär abgeschlossen.

Wenn andernfalls während der Verarbeitung des Prüflingscodes ein Breakpoint getroffen wird, erfolgt ein Aufruf der Behandlungsroutine für Breakpoints im Debugger. Diese Funktion entspricht der Behelfsfunktion in Abbildung 3.3. Hier werden die am Entkopplungspunkt gesicherten Werte der Stackregister unmittelbar übernommen. Bei der Ausführung des Rücksprungbefehls der Behandlungsroutine wird nun nicht die Rücksprungadresse aus deren Stackframe im separaten Stackspeicher verwendet, sondern die Rücksprungadresse des Stackframes der Funktion am Entkopplungspunkt. Der Abbau der im separaten Stackspeicher aufgebauten Stackframes wird damit übersprungen. Diese Manipulation des Rücksprungverhaltens wird als *Entkopplung* bezeichnet. Infolge der Entkopplung wird die Ausführung im Echtzeit-Thread in der Funktion vor dem Entkopplungspunkt fortgesetzt. Die Ausführung wird von hier aus regulär abgeschlossen. Die Ausführung des Prüflings ist zu diesem Zeitpunkt unterbrochen. Sein Ausführungszustand kann inspiziert werden.

Ab dem Zeitpunkt der Entkopplung ist die Echtzeitverarbeitung im Prüfling unterbrochen. Die Verarbeitung eingehender Signale ist in dieser Zeit nicht möglich. Die Ausführung im Echtzeit-Thread wird von der Ausführungsumgebung umgeleitet. Im Falle der digitalen Klangsynthese könnten beispielsweise die Werte aller Samples auf Null gesetzt werden. Bei der Ausgabe dieser Samples wird dann kein Ton erzeugt.

Die Fortsetzung der Ausführung kann in einem Behelfs-Thread erfolgen. Zu diesem Zweck müssen die Werte der Stackregister ebenfalls in der Behandlungsroutine für Breakpoints gesichert werden. Werden diese Werte innerhalb einer Funktion im Behelfs-Thread wiederhergestellt, bewirkt dies eine Umschaltung des Stackframes der Funktion auf den letzen, bis zu diesem Zeitpunkt eingefrorenen Stackframe im separaten Stackspeicher. Bei der Ausführung des Rücksprungbefehls der Funktion erfolgt ein Rücksprung zum ersten Befehl hinter dem Funktionsaufruf im Programmcode des Prüflings. Auf diese Weise wird

die Ausführung des Prüflings fortgesetzt. Weitere Unterbrechungen können analog realisiert werden. Eine Entkopplung des Aufrufstacks ist dann nicht mehr erforderlich. Eine vollständige Beschreibung zur technischen Realisierung dieses Verfahrens findet sich im Kapitel zur Implementierung des Same-Process Debuggers in Abschnitt 4.4.3.

3.3.3.4 Anmerkungen

Die Änderung der Ausführungsabfolge von Funktionen durch die Manipulation der Stackregister des Prozessors gehört nicht zu den konventionellen Programmiertechniken, da sie die Symmetrie von Funktionsaufruf und -rückkehr durchbricht. Die Funktionen, die diese Manipulationen durchführen, kehren nicht zum Aufrufenden zurück. In der Folge kommt es schnell zu einem schwer verständlichen Ausführungsverhalten und einer hohen Fehleranfälligkeit. Für die Unterbrechung der Echtzeitverarbeitung stellt das Verfahren jedoch eine adäquate Lösung dar.

Der damit einhergehende permanente Laufzeitoverhead begrenzt sich auf die Umleitung des Aufrufstacks vor und nach jedem Verarbeitungsdurchlauf. Insgesamt werden dafür vier Austauschoperationen zwischen Stackregistern und Speicher erforderlich. Die resultierende Verzögerung ist akzeptabel und gestattet die Beibehaltung der Verarbeitung in Echtzeit. Der separate Stackspeicher muss nur einmal angelegt werden. Er wird für jeden Umschaltvorgang wiederverwendet.

Im Unterbrechungsfall erfolgt die Entkopplung des Aufrufstacks. Damit wird die Verarbeitung des Prüflings unterbrochen und kann zu einem späteren Zeitpunkt von einem Behelfs-Thread fortgesetzt werden. Die Rückkehr des Echtzeit-Threads wird nur durch die Sicherung des Ausführungszustandes des Prüflings und die Wiederherstellung des ursprünglichen Maschinencodes verzögert (siehe Abschnitt 3.2.1.3 zur Realisierung von Breakpoints). Dies ist in jedem Fall vor Ablauf der Zeitschranke des Echtzeit-Threads realisierbar.

3.4 Zusammenfassung

Zusammenfassend soll im Folgenden die Verbindung der in diesem Kapitel erarbeiteten Konzepte vorgestellt werden. Die Instanziierung des Debuggers sollte spätestens zu Beginn der Kompilierung des Prüflings im Debug-Modus erfolgen. Bei der Instanziierung wird der Speicher des Callback-Flags (3.3.2.2) alloziert. Auch die Allokation des Speicherbereichs für den separaten Aufrufstack (3.3.3.3) sollte bereits zur Übersetzungszeit geschehen, damit die spätere Umleitung nicht durch die dafür nötigen Systemaufrufe verzögert wird. Vom Compiler wird zuerst der Programmcode des Callback-Points (3.3.2.2) erzeugt. Dabei ist die Angabe der Speicherstelle des Callback-Flags und der behandelnden Routine des Debuggers erforderlich. Bei der Kompilierung des Verarbeitungscodes des Prüflings müssen die sequenziellen Befehlsblöcke im Maschinencode durch die Identifizierung sämtlicher Sprungziele ermittelt werden (3.2.1.2). Vor jedem Sprungziel wird vom Compiler ein Puffer in Form eines NOPs eingefügt. Sind im Maschinencode des Prüflings Schleifen enthalten, müssen deren Sprungziele für den Wiederholungsfall vom Compiler gemäß 3.2.2.3 u. U. korrigiert werden.

Zur Laufzeit wird der Aufrufstack des Prüflings bei jedem Verarbeitungsdurchlauf am Entkopplungspunkt in den separaten Stackspeicher umgeleitet (3.3.3.3). Während der Ausführung des Prüflings kann der Benutzer in der Entwicklungsumgebung Breakpoints auf Source-Level-Anweisungen setzen. Dies bewirkt das Setzen des Callback-Flags. Im nächsten Verarbeitungsdurchlauf des Prüflings wird damit am Callback-Point eine Behandlungsroutine des Debuggers aufgerufen. Sie setzt zunächst das Callback-Flag zurück und implementiert im Anschluss die gesetzten Breakpoints im Programmcode des Prüflings nach dem in 3.2.1.2 beschriebenen Verfahren. Danach wird die Echtzeitverarbeitung regulär fortgesetzt. Wird in einem der folgenden Verarbeitungsdurchläufe ein Breakpoint getroffen, wird zunächst der Ausführungszustand des Prüflings gesichert und der ursprüngliche Programmcode wiederhergestellt. Danach erfolgt die Entkopplung des Aufrufstacks, sodass der Funktionsaufruf im Echtzeit-Thread zurückkehren kann und die Stackframes der übersprungenen Funktionen im separaten Stackspeicher verbleiben (3.3.3.3). Die Ausführung des Prüflings ist zu diesem Zeitpunkt unterbrochen. Sein Ausführungszustand kann inspiziert werden.

Wie in 3.3.3.3 beschrieben, wird die Fortsetzung der Ausführung von einem Behelfs-Thread übernommen. Dabei wird zunächst der Ausführungszustand des Prüflings wiederhergestellt. Gemäß 3.2.1.2 müssen die ursprünglichen Befehle an der Stelle des Breakpoints nun u. U. in einem oder mehreren impliziten Einzelschritten (3.2.2) ausgeführt werden. Nach der erneuten Implementierung der gesetzten Breakpoints im Programmcode

des Prüflings wird die Verarbeitung fortgesetzt. Die Ausführung expliziter Einzelschritte erfolgt unter Verwendung temporärer Breakpoints nach dem in 3.2.2 beschriebenen Verfahren. Der weitere Ablauf erfolgt analog zur Vorgehensweise bei der Verwendung permanenter Breakpoints.

Erreicht die Ausführung das Ende des Prüflingscodes, werden die Stackframes im separaten Stackspeicher vollständig abgearbeitet. Die Entkopplung ist damit aufgehoben. Die Echtzeitverarbeitung des Prüflings kann wieder aufgenommen werden.

4 Implementierung in Reaktor Core

In diesem Kapitel wird die Implementierung des Prototypen eines interaktiven, integrierten Same-Process Debuggers für eine externe, kompilierte, domänenspezifische Sprache der digitalen Klangsynthese beschrieben. Die Grundlage bildet das in Kapitel 3 erarbeitete Konzept. Als Beispiel dient die Programmiersprache Reaktor Core (2.4). Es werden Spezialisierungen vorgenommen, technische Fragen geklärt und konkrete Implementierungsansätze vorgestellt.

4.1 Funktionsumfang des Prototypen

Ziel der Implementierung des Prototypen ist es, einen Machbarkeitsnachweis für die in Kapitel 3 erarbeiteten Grundfunktionalitäten zu erbringen. Dafür wurde die Realisierung des in 3.2.2 beschriebenen Verfahrens zur Ausführung des Prüflings in Einzelschritten gewählt. Die Anbindung der Benutzeroberfläche gestaltet sich damit einfach. Die Ausführung eines Source-Level-Steps kann vom Benutzer durch das Betätigen einer Funktionstaste der Tastatur erfolgen. Die Implementierung kann sich damit auf die Umsetzung der Kernfunktionalitäten konzentrieren. Da die Einzelschrittausführung auf der Verwendung von Breakpoints basiert, demonstriert sie die Funktionsweise aller grundlegenden Mechanismen zur Ablaufkontrolle eines Trace-Debuggers. Die Einzelschrittausführung soll zunächst nur für den Audio-Handler (2.4.3) des Prüflings verfügbar sein. Im Audio-Handler findet die Signalverarbeitung in Echtzeit statt. Auch im Hinblick auf die Problematik der Echtzeitverarbeitung werden damit alle Schwerpunkte abgedeckt.

Mit einem als *Wire-Debugging* bezeichneten Mechanismus steht in Reaktor Core zudem bereits eine einfache Möglichkeit zur Inspektion des Prüflings zur Verfügung. Er soll zu den zu erstellenden Ablaufkontrollmechanismen kompatibel sein. Die Implementierung des Prototypen soll zukünftige Ergänzungen ermöglichen. Dazu zählen explizite Breakpoints auf beliebigen Modulen, Step-Out und Step-Over Funktionalitäten für Macros, das Debugging mehrerer Core Cells zugleich sowie die Modifikation von Laufzeitwerten.

4.2 Technische Realisierung von Reaktor Core

Bei der Implementierung des Debuggers muss der Aufbau und die Arbeitsweise des bestehenden Projekts berücksichtigt werden. Aufbauend auf der Einführung in Reaktor Core in Abschnitt 2.4 soll hier auf die Aspekte zur technischen Realisierung eingegangen werden.

4.2.1 Eigenschaften von Reaktor Core Programmen

Aufgrund der eingeschränkten Mächtigkeit von Reaktor Core eignet sich die Programmiersprache nur für die Implementierung von Anwendungsteilen, die von einer Ausführungsumgebung verwendet werden. Reaktor Core Programme sind also keine eigenständigen Anwendungen. Diese Eigenschaft findet sich unter domänenspezifischen Sprachen häufig. Ein Reaktor Core Programm liegt in Form mehrerer *Handler* vor. Ein Handler ist eine einfache Funktion, die von der Ausführungsumgebung aufgerufen werden kann. Innerhalb eines Handlers erfolgen keine weiteren Funktionsaufrufe.

In Reaktor Core existiert kein Schleifenkonstrukt. Ein Handler wird damit auf eine Abfolge sequenzieller Befehle und verschachtelter Verzweigungen reduziert. Alle Sprünge im Maschinencode der Programme erfolgen stets in Richtung der Ausführung. Abbildung 4.5 auf Seite 68 zeigt die typische Struktur eines Reaktor Core Programms in Pseudocode.

4.2.2 Architektur

Reaktor Core ist in C++ geschrieben und liegt als Programmbibliothek vor (*Reaktor Core Library*), die von der Ausführungs- und Entwicklungsumgebung Reaktor verwendet wird. Reaktor ist ein Softwareprogramm, das mit der Zeit gewachsen ist und keine durchgängige Quellcodestruktur aufweißt. Der Quellcode von Reaktor Core hingegen ist stark strukturiert. Er unterteilt sich in mehrere Module, die über eine einheitliche Schnittstelle mit Reaktor kommunizieren. Der Programmcode des Debuggers soll hier in Form eines zusätzlichen Moduls ergänzt werden. Abbildung 4.1 zeigt die Organisation der Module und deren Abhängigkeiten.

Das Herzstück der Reaktor Core Library bildet der Reaktor Core Compiler. Der Quellcode des Compilers ist im Modul `SEC` (Synth Engine Compiler) gekapselt. Neukompilierungen werden implizit durch Änderungen der Datenbasis im Modul `SD` (Synth Data) initiiert. Kommt es zur Übersetzungszeit zu weiteren Änderungen, wird die Kompilierung abgebrochen und neu gestartet. Nach Abschluss des Übersetzungsvorgangs erfolgt eine Benachrichtigung der Ausführungsumgebung über das Modul `SDI` (Synth Data Inter-

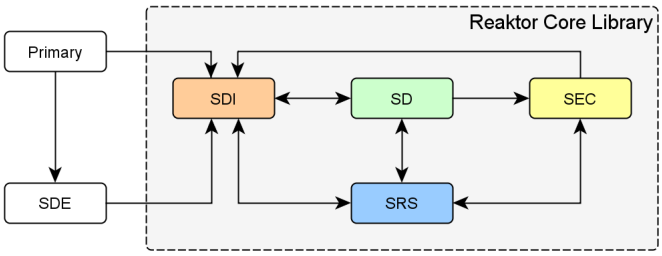

Abbildung 4.1 Integration des Moduls SRS in die bestehende Projektarchitektur. Die Kommunikation mit der grafischen Benutzeroberfläche erfolgt über das Schnittstellenmodul *SDI*.

face). `SDI` bildet die einheitliche Schnittstelle der Reaktor Core Library. Sie reglementiert den gesamten Kommunikationsfluss. Damit sind nicht alle Zugriffsmöglichkeiten, die sich die Module der Reaktor Core Library untereinander gewähren, auch nach außen verfügbar. Für den Quellcode des Debuggers ist das Modul `SRS` (Synth Runtime Services) hinzugekommen. Außerhalb der Reaktor Core Library steht die Entwicklungsumgebung im Modul `SDE` (Synth Data Editor) und die Ausführungsumgebung Reaktor (Primary).

4.2.3 Arbeitsweise von Reaktor Core

Die Ausführungsumgebung Reaktor arbeitet mit Ensembles. Ein Ensemble kann beliebig viele Instrumente enthalten. Innerhalb eines solchen Reaktor-Instruments können wiederum beliebig viele Core Cells verwendet werden. Wie in Abschnitt 2.4.1 beschrieben wurde, dienen Core Cells als Container für die mit Reaktor Core entwickelten Grundschaltelemente. Als Vereinfachung wird für die Entwicklung des Prototypen davon ausgegangen, dass zunächst nur eine einzige Core Cell existiert. Die Core Cell ist im Quellcode als Instanz der Klasse `SD::Instrument` repräsentiert. Neben einer Reihe von Eigenschaften enthält `SD::Instrument` Listen ihrer Ein- und Ausgabeports sowie direkt enthaltener Module (Instanzen der Klasse `SD::Module`). `SD::Module` dient als Basisklasse für alle Arten von Modulen. Macros sind spezielle Module, die ihrerseits wieder Module enthalten können. Damit ist die hierarchische Struktur der Core Cell in der Datenbasis repräsentiert.

Änderungen der Datenbasis, wie das Laden eines Ensembles in Reaktor oder die Modifikation der Core Cell durch den Benutzer, führen zur Neukompilierung der Core Cell. Der Ablauf der Kompilierung wird in Abschnitt 4.2.4 genauer betrachtet. Nach der Kompilierung liegt der ausführbare Maschinencode der Core Cell im Speicher bereit. Ist zu diesem Zeitpunkt noch ein Sample-Puffer in Bearbeitung, wird dieser Vorgang zunächst abgeschlossen. Für die Freischaltung des neuen Programmcodes genügt die Anpassung ei-

niger Funktionszeiger in Reaktor, sodass die Ausführung bei der Verarbeitung des nächsten Sample-Puffers den neuen Programmcode verwendet. Der Speicherbereich des alten Programmcodes kann dann freigegeben werden.

4.2.4 Ablauf der Kompilierung

Da es sich bei Reaktor Core um eine datenstromorientierte, grafische Programmiersprache handelt, ist kein Parsing im klassischen Sinne nötig. Die in Reaktor Core erstellten Datenflussgraphen entsprechen einem abstrakten Syntaxbaum bereits in vielerlei Hinsicht. Die Übersetzung einer Core Cell in Maschinencode erfolgt in drei Schritten. Der gesamte Übersetzungsvorgang wird von der Klasse `SEC::Instrument` kontrolliert. Nachfolgend wird auf jeden der Schritte im Einzelnen eingegangen.

4.2.4.1 Vorbereitungsphase

Jede im Folgenden betrachtete Core Cell hat mindestens einen Port für die Ausgabe der transformierten Signalwerte[1]. Davon ausgehend werden über eine Tiefensuche nach und nach alle Module bestimmt, die an der Berechnung der transformierten Signalwerte direkt teilhaben. Wird eine Core Cell als Datenflussgraph betrachtet, beginnt die Tiefensuche also an dessen Senke und erfolgt gegen die Flussrichtung. Die Grenzen von Macros werden dabei nicht beachtet, denn in Reaktor Core werden Macros nicht als Funktionen realisiert. Stattdessen wird der Code eines Macros für jede Instanz separat erzeugt. In gewisser Weise ähnelt dies der Funktionsweise von Präprozessormacros in universellen Programmiersprachen wie C oder C++. Die Tiefensuche endet u. a. an Leseoperationen von Speicherzellen, welche z. B. in Latches (siehe 2.4.2) verwendet werden. Die Module die aus Sicht der Tiefensuche hinter diesen Speicherzellen liegen, müssen – sofern die Speicherzelle die einzige Verbindung darstellt – nicht Teil der Echtzeitverarbeitung sein. Sie werden zu einem späteren Zeitpunkt in einen Event-Handler (2.4.3) aufgenommen. Auf diese Weise wird die Zuordnung der Module zum späteren Audio-Handler bestimmt. Der Audio-Handler enthält den Teil des Maschinencodes, der die Audioausgabe in Echtzeit berechnet.

Router sind spezielle Module. In Abhängigkeit von einem booleschen Wert, wie dem Ergebnis einer Vergleichsoperation, wird die Ausführung entweder am oberen oder am unteren Ausgang des Moduls fortgesetzt. Sie stellen die einzige Möglichkeit zur Verzweigung des Kontrollflusses dar und spielen daher eine zentrale Rolle. Das Pedant des Routers

[1] Es existieren auch spezielle Event Core Cells, die nicht über einen solchen Port verfügen. Auf sie soll im Folgenden jedoch nicht gesondert eingegangen werden, da sie für die Echtzeitverarbeitung keine Rolle spielen.

ist das *Merge* Modul, welches den Kontrollfluss aus verschiedenen Verzweigungen zusammenfasst. Eine Verzweigung entspricht stets einer Laufzeitbedingung (*Condition*). Die Module die im Verarbeitungsfluss hinter einem Router liegen, werden in Abhängigkeit zu dessen Laufzeitbedingung ausgeführt. Die Menge der Laufzeitbedingungen, unter denen ein Modul zur Ausführung kommt, wird als *Condition-Stack* bezeichnet. Für jedes Modul der Core Cell wird ein solcher Condition-Stack aufgebaut. Im Tiefensuchbaum aufeinanderfolgende Module mit äquivalentem Condition-Stack werden zu *Terminal-Instances* zusammengefasst (`SEC::ConditionTerminalInstance`). Als Stellvertreter der Module der Datenbasis kommen hier Instanzen der Klasse `SEC::Module` zum Einsatz. Ein `SEC::Module` enthält übersetzungsspezifische Daten sowie eine Referenz auf das zugehörige `SD:Module`. Router gehören zu keiner Terminal-Instance. Für jeden Router wird eine Node-Instance (`SEC::ConditionNodeInstance`) erzeugt, die zwei Nachfolger in Form von Node- oder Terminal-Instances haben kann. Damit wird die Verzweigungsstruktur des Kontrollflusses zur Übersetzungszeit abgebildet.

Innerhalb der Terminal-Instances existiert bereits eine feste Reihenfolge der Module. Hier besteht aber noch Optimierungspotenzial. Für ein gegebenes Modul im Audio-Handler einer Core Cell sind einerseits ab einem bestimmten Ausführungszeitpunkt des Handlers alle nötigen Eingabewerte verfügbar. Andererseits gibt es ab einem bestimmten Ausführungszeitpunkt des Handlers keine anderen Module mehr, die unabhängig vom Ergebnis der Operation des Moduls ausgeführt werden können. Für die Positionierung des Moduls bleibt dem Compiler dieser Spielraum für Optimierungen.

Die Zuordnung der Module zu Handlern und die Unterteilung der Handler in Node- und Terminal-Instances bildet die erste Phase der Kompilierung. Die Koordination der dafür nötigen Schritte wird im Quellcode durch die Klasse `SEC::MRSCompiler` abgebildet. Für die Integration des Debuggers spielt diese Phase nur eine untergeordnete Rolle.

4.2.4.2 Compilezeit

Während der zweiten, auch als *Compilezeit* bezeichneten Phase der Kompilierung findet die tatsächliche Übersetzung der Module in Maschinenbefehle statt. Die Klasse `SEC::BCBCompiler` ist die abstrakte Basisklasse des Compiler-Backends und stellt u. a. Hilfsmittel für die Registerallokation und das sequenzielle Schreiben von Maschinencode bereit. Die konkrete Übersetzung von Befehlen wird über rein virtuelle Funktionen an abgeleitete Klassen delegiert. `SEC::BCBCompilerSSE` ist eine solche abgeleitete Klasse für Systeme die die SSE-Erweiterungen des x86 Befehlssatzes unterstützen. Die Auswahl des konkreten Compiler-Backends erfolgt bereits beim Start von Reaktor auf Grundlage verschiedener Systemeigenschaften.

Zu Beginn der zweiten Phase liegen die relevanten Module der Core Cell als Instanzen der Basisklasse `SEC::Module` und gruppiert in Terminal-Instances vor. Alle Terminal-Instances werden nun durchlaufen und deren Module in Maschinencode übersetzt. Für jeden Modultyp existiert eine eigene Ableitung von `SEC::Module`. Die Basisklasse delegiert den Funktionsaufruf zur Übersetzung eines Moduls durch eine virtuelle Funktion an die abgeleiteten Klassen. In den abgeleiteten Klassen ist bekannt, welchem Maschinenbefehl bzw. welchen Maschinenbefehlen der jeweilige Modultyp entspricht. Für die strukturierte Übersetzung werden die entsprechenden rein virtuellen Funktionen des Compiler-Backends aufgerufen. Die tatsächliche Verarbeitung erfolgt dann in der jeweiligen befehlssatzspezifischen Instanz (z. B. `SEC::BCBCompilerSSE`).

Im Anschluss liegt der Binärcode jeder Terminal-Instance des Handlers vor. Diese binären Codeblöcke werden als *lineare Segmente* bezeichnet. Sie bilden ein zentrales Konzept im weiteren Übersetzungsvorgang und spielen eine wichtige Rolle für die spätere Realisierung von Breakpoints. Lineare Segmente enthalten keine Sprunganweisungen. Sprungziel kann in einem linearen Segment darüber hinaus ausschließlich die erste Operation sein. Lineare Segmente bilden damit die Grundblöcke in Reaktor Core Programmen[2].

4.2.4.3 Linkzeit

In der dritten, auch als *Linkzeit* bezeichneten Phase der Kompilierung werden die zur Compilezeit erstellten linearen Segmente zwischen Sprunganweisungen eingebettet, die den Kontrollfluss regeln. Zu diesem Zeitpunkt ist die Gesamtgröße des Programmcodes bereits bekannt. Zu Beginn der Linkzeit wird ein entsprechender Speicherbereich alloziert. Die Größe des Speicherbereichs entspricht einem Vielfachen der Größe einer Speicherseite des Betriebssystems. Die Speicherseiten sind zunächst editierbar, können aber zu einem späteren Zeitpunkt über entsprechende Funktionen des Betriebssystems als ausführbar markiert werden[3]. Zur Linkzeit wird der Maschinencode der Core Cell in diesem Speicherbereich zusammengefügt. Dieser Vorgang wird von der Klasse `SEC::BCBLinker` koordiniert. Den Ausgangspunkt bildet die abstrakte Klasse `SEC::BCBElement`. Für sie existieren die beiden abgeleiteten Klassen `SEC::BCBRawCode` und `SEC::BCBFork`. Zu den Instanzen der Klasse `SEC::BCBRawCode` gehören die bereits übersetzten linearen Segmente. Daneben liegt zur Linkzeit auch der Maschinencode zur Auswertung der Laufzeitbe-

[2] Die vollständige Bedeutung von Grundblöcken erhalten lineare Segmente erst zur Linkzeit durch die Ergänzung von Sprungbefehlen an deren Ende.

[3] Unter Windows kann dies beispielsweise über die API Funktion VirtualProtect erreicht werden. Der betreffende Speicherbereich muss dafür zuvor über die API Funktion VirtualAlloc alloziert worden sein [Mic12b].

dingungen als Instanzen dieser Klasse vor. Durch die Instanzen der Klasse `SEC::BCBFork` werden die Verzweigungen des Kontrollflusses abgebildet. Ein solches Fork besteht aus einem Prolog, einem Then-Zweig, einem Else-Zweig und einem Epilog. Jeder dieser Teile kann wiederum eine beliebige Anzahl von Instanzen der Klasse `SEC::BCBElement` enthalten (also auch leer sein). Die Informationen für die Verschachtelung der Forks gehen aus den Node-Instances der Compilezeit hervor.

Ausgehend von einem ersten Fork, das eine stets erfüllte Verzweigungsbedingung darstellt, werden im Zuge des Link-Prozesses eines Handlers alle Elemente sequenziell in den Zielspeicher kopiert. Zwischen Prolog und Then-Zweig sowie zwischen Then- und Else-Zweig werden die nötigen bedingten Sprungbefehle eingefügt. Im Falle von verschachtelten Forks erfolgt ein rekursiver Aufruf des Link-Prozesses. Alle Sprungdistanzen sind zur Linkzeit bekannt. Sie ergeben sich als Summe der Länge der zu überspringenden linearen Segmente und der Länge der dazwischen nötigen Sprungbefehle.

Nach Abschluss der dritten Phase liegt der ausführbare Maschinencode der Core Cell im Zielspeicher bereit. Danach wird der verwendete Speicherbereich als ausführbar markiert. Die Echtzeitverarbeitung im Audio-Handler kann nach der einmaligen Ausführung des Init-Handlers beginnen.

4.3 Aufgaben zur Übersetzungszeit

Im folgenden Abschnitt werden die Aspekte der Implementierung eines Same-Process Debuggers betrachtet, die bei der Übersetzung des Prüflings erforderlich werden.

4.3.1 Überblick

Für die Verwendung eines Same-Process Debuggers nach dem in Kapitel 3 erarbeiteten Konzept sind verschiedene Informationen zum Prüflingsprogramm erforderlich. Sie werden zur Übersetzungszeit vom Compiler gesammelt und dem Debugger als *Programmdaten* (2.2.4) übergeben.

Alle nötigen Änderungen am Quellcode des Compilers sollen möglichst gering gehalten werden. Aus diesem Grund erfolgt die Aufzeichnung der Programmdaten in einem Modul des Debuggers. Dieses Modul wird im Folgenden als *Debug-Data-Factory* bezeichnet. Es ist im Quellcode durch die Klasse `SRS::DebugDataFactory` repräsentiert. Instanzen der Klasse gehören[4] dem Compiler und existieren nur zur Übersetzungszeit. Der Compiler selbst speichert keine Programmdaten. Stattdessen übergibt er sie der Debug-Data-Factory in den Parametern entsprechender Funktionsaufrufe. Diese Funktionsaufrufe erfolgen direkt an denjenigen Stellen des Übersetzungsvorgangs, an denen die jeweiligen Daten anfallen. Die Zuordnung und strukturierte Speicherung der Programmdaten wird von der Debug-Data-Factory vorgenommen.

Die Änderungen am Quellcode des Compilers können noch weiter beschränkt werden, wenn nicht bei jedem Funktionsaufruf zur Übergabe von Programmdaten überprüft werden muss, ob der Prüfling überhaupt im Debug-Modus übersetzt wird. Dies wäre notwendig, wenn die Debug-Data-Factory nur im Debug-Fall existieren würde und die vom Compiler gehaltene Referenz andernfalls ungültig wäre. Um sich eine solche Abfrage bei jedem Funktionsaufruf zu sparen, wurde die Debug-Data-Factory nach dem Abstract Factory Pattern implementiert [GHVJ09]. Die Klasse `SRS::DebugDataFactory` bildet demnach nur die abstrakte Schnittstelle zur Debug-Data-Factory. Die tatsächliche Implementierung ist durch `SRS::DebugDataFactoryImpl` gegeben. Diese Klasse wird im Debug-Fall instanziiert. Andernfalls dient eine Instanz der Klasse `SRS::DebugDataFactoryDummy` als Ersatz für einen ungültigen Zeiger. Sie zeichnet keinerlei Daten auf und verhält sich gegenüber dem Compiler vollkommen neutral. Sie folgt damit dem sogenannten Null Object Pattern [MRB98]. Diese Konstruktion hat einen weiteren interessanten Vorteil.

[4] Mit dem Begriff „gehören“ ist der Besitz eines Objekts gemeint (engl. Ownership). Der Besitzer eines Objekts ist dafür verantwortlich, das Objekt zu löschen und dessen Speicher freizugeben, wenn es nicht mehr benötigt wird.

Sie erlaubt die einfache Einbindung beliebiger zusätzlicher Klassen zur Aufzeichnung von Programmdaten. Als Hilfestellung bei der Implementierung des Debuggers kam beispielsweise die Klasse `SRS::DebugDataFactoryImplDump` zum Einsatz, die in einem Konsolenfenster Informationen zum Aufzeichnungsprozess ausgab[5].

Abbildung 4.2 gibt einen Überblick zum Ablauf des Aufzeichnungsprozesses. Der Übersetzungsvorgang wird von der Klasse `SEC::Instrument` koordiniert und erfolgt in drei Phasen (4.2.4). Die Debug-Data-Factory wird noch vor Beginn der Vorbereitungsphase von `SEC::Instrument` instanziiert. Während der Vorbereitungsphase fallen keine Programmdaten an. Zur Compilezeit werden die Bestandteile des Prüflings aufgezeichnet und für den strukturierten Zugriff vorgereitet. Dies wird in Abschnitt 4.3.2 beschrieben. Abschnitt 4.3.3 widmet sich dann den Aufgaben zur Linkzeit. Hier erfolgt im Wesentlichen die Analyse der Kontrollflussstruktur des Prüflings.

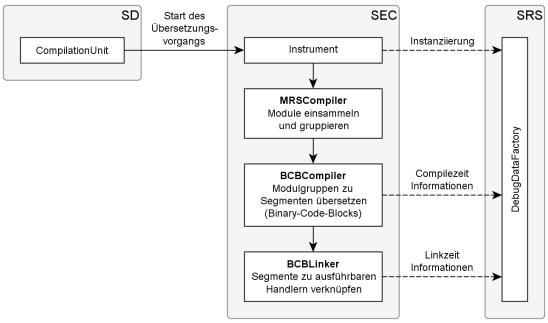

Abbildung 4.2 Übersicht zum Ablauf des Aufzeichnungsprozesses. Die Übersetzung einer Core Cell wird in Folge von Änderungen der Datenbasis initiiert. Der Vorgang wird von einer Instanz der Klasse *SEC::Instrument* koordiniert. Sie instanziiert ebenfalls die *Debug-Data-Factory*. Die Übersetzung erfolgt in drei Phasen. Programmdaten fallen dabei nur zur Compilezeit und zur Linkzeit nicht aber während der Vorbereitungsphase an.

4.3.2 Aufzeichnung der Programmelemente zur Compilezeit

Die charakteristische Eigenschaft eines symbolischen Debuggers bildet die Möglichkeit, mit dem Debugger auf dem Abstraktionsniveau der verwendeten Programmiersprache zu arbeiten (2.2.3). Im Hinblick auf die Realisierung von Ablaufkontrollmechanismen wird dafür eine zeilenweise Zuordnung zwischen den Befehlen auf Source-Level und den Befehlen auf Instruction-Level benötigt. Eine solche Zuordnung findet sich in den Programmdaten eines jeden symbolischen Debuggers. Da die Darstellung des Source-Levels in Reaktor Core grafisch erfolgt, kann es keine zeilenweise Zuordnung geben. Hier ist eine

[5] Tatsächlich bot sie in erster Linie Hilfestellung beim *Debugging* des Debuggers.

Zuordnung zwischen den Modulen der hierarchischen Struktur einer Core Cell und deren Operationen im Maschinencode erforderlich.

Zur Compilezeit wird die Methode `compileBCB` vom Compiler für alle Module der Core Cell aufgerufen. Der Aufruf erfolgt während der Übersetzung der entsprechenden Terminal-Condition in ein lineares Segment (4.2.4.2). Zu diesem Zeitpunkt kann die Zuordnung zwischen einem Modul der Datenbasis und der Position des dafür erzeugten Befehlscodes[6] *relativ* zum jeweiligen linearen Segment hergestellt werden. Diese Information wird der Debug-Data-Factory durch einen Aufruf der Methode `addModule` übergeben. Für den weiteren Verarbeitungsvorgang sind jedoch zusätzliche Informationen nötig. Dazu gehört das lineare Segment und der Handler, zu dem das Modul gehört sowie – für die künftige Weiterentwicklung des Prototypen – auch die zugehörige Core Cell.

All diese Daten könnten nun durch die Aufrufebenen des Compilers bis zur Funktion `compileBCB` weitergegeben werden. Dies hätte jedoch unverhältnismäßig viele Änderungen am Quellcode des Compilers zur Folge. Eine weitere Folge wäre ein stark zunehmendes Speicheraufkommen. Denn die Daten müssten in den Stackframes der Funktionen aller Aufrufebenen zwischengespeichert werden. Eine günstigere und flexiblere Lösung stellt die Realisierung der Debug-Data-Factory als Zustandsautomat dar. Beginn und Ende eines Übersetzungsabschnitts könnten durch den Aufruf bestimmter Funktionen markiert werden. Im Kontext der Debug-Data-Factory wäre dann klar, dass die dazwischenliegenden Funktionsaufrufe diesem Übersetzungsabschnitt zuzuordnen sind. Dieser Ansatz wird für die gesamte Kommunikation zwischen Compiler und Debug-Data-Factory verwendet. Eine schematische Darstellung des Kommunikationsverlaufs zur Compilezeit ist in Abbildung 4.3 dargestellt. Zur Identifizierung der beteiligten Konstrukte werden zudem verschiedene Parameter übergeben. Zur Compilezeit erfolgt je eine Benachrichtigung beim Betreten und Verlassen der in Tabelle 4.1 aufgeführten Übersetzungsabschnitte.

Abschnitt	**Funktion** (begin/end)	**Parameter**
Core Cell	CompileInstrument	Zeiger auf SD::Instrument
Handler	CompileHandler	Handler ID
Terminal-Instance	CompileSegment	Segment ID

Tabelle 4.1 Benachrichtigungen des Compilers an die Debug-Data-Factory beim Betreten und Verlassen von Übersetzungsabschnitten zur Compilezeit

Bei der Handler ID handelt es sich um eine nicht negative, ganze Zahl. In Reaktor Core existiert die Konvention, dass die ID des Init-Handler Null und die ID des Audio-Handlers Eins ist. IDs von Zwei aufwärts bezeichnen Event-Handler. Weder Terminal-Instances

[6] Der Begriff *Befehlscode* bezeichnet hier einen oder mehrere Maschinenbefehle.

noch lineare Segmente verfügen über einen expliziten Identifikationsmechanismus. Als Segment ID kann jedoch die Speicheradresse eines linearen Segments dienen. Da das lineare Segment beim Betreten der Terminal-Instance noch nicht existiert, wird die ID durch die Funktion `endCompileSegment` übergeben.

Die Methode `addModule` wird ausschließlich während der Verarbeitung von Terminal-Conditions aufgerufen. Damit kann jedes Modul eindeutig einem linearen Segment zugeordnet werden. Lineare Segmente können wiederum Handlern und diese schließlich Core Cells zugeordnet werden. Ausgehend von einer relativen Speicherposition innerhalb eines linearen Segments kann nun bereits das zugehörige Modul in der Datenbasis identifiziert werden. Sobald zur Linkzeit die endgültige Position des linearen Segments bekannt ist, ist die Zuordnung zwischen den Befehlen auf Source-Level und Instruction-Level vollständig.

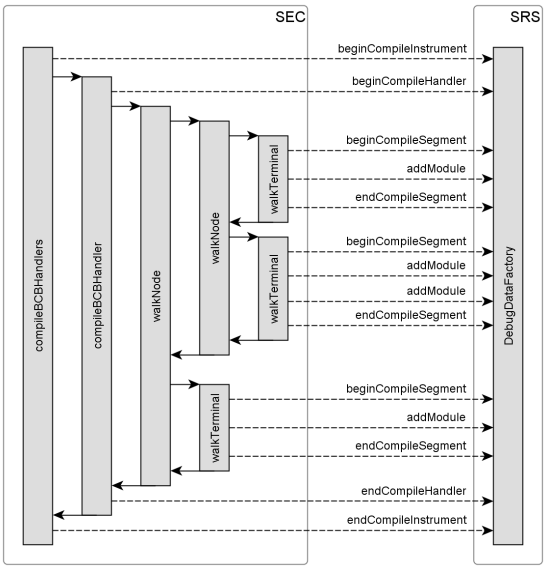

Abbildung 4.3 Schematische Darstellung der Kommunikation zwischen Compiler und Debug-Data-Factory zur Compilezeit. Benachrichtigungen zu Beginn und Ende von Übersetzungsabschnitten stellen eine flexible Möglichkeit dar, den Aufbau des Prüflings zu übermitteln.

4.3.3 Kontrollflussanalyse zur Linkzeit

Die Kontrollflussanalyse bezeichnet ein statisches Verfahren zur Analyse des Ausführungsverlaufs in einem Programm [ASL+08]. Es wird ermittelt, auf welchen Ausführungspfaden ein Programm oder ein Programmfragment durchlaufen werden kann. Das Ergebnis liegt in Form eines *Kontrollflussgraphen* vor.

4.3.3.1 Vorbetrachtung

Für die Realisierung eines Einzelschrittverfahrens nach 3.2.2 müssen die möglichen Folgebefehle eines Maschinenbefehls bekannt sein. Dafür könnte der Maschinenbefehl an der Stelle der Unterbrechung dekodiert werden[7]. Handelt es sich um einen Sprungbefehl, könnte die Sprungbedingung ausgewertet und das Sprungziel anhand der Sprungdistanz im Argument des Befehls ermittelt werden. Die Befehlsdekodierung erfordert jedoch einen erheblichen Aufwand bei der Implementierung und wird hier daher nicht verwendet.

Stattdessen erfolgt eine einfache Kontrollflussanalyse zur Linkzeit. Dem Verfahren kommt die einfache Struktur von Reaktor Core Programmen zugute. Da die Adressen der linearen Segmente für die Befehlszuordnung zwischen Source- und Instruction-Level ohnehin von der Debug-Data-Factory aufgezeichnet werden, genügt es zusätzlich die Verzweigungsstruktur der linearen Segmente zu ermitteln und in den Programmdaten zu hinterlegen. Innerhalb von linearen Segmenten existieren keine Sprungbefehle. Der einzig mögliche Folgebefehl eines Maschinenbefehls ist hier also stets der im Speicher direkt nachfolgende Befehl. Der Maschinencode aller Module, die am Verarbeitungsprozess direkt mitwirken, befindet sich innerhalb der linearen Segmente. Nur die Module *Router* und *Merge* lassen sich nicht eindeutig einem linearen Segment zuordnen. Ihre Folgemodule ergeben sich aus der Verzeigung der linearen Segmente.

Die Kontrollflussanalyse findet in der Debug-Data-Factory statt. Während des Link-Prozesses wird sie vom Compiler über die entscheidenden Übersetzungsabschnitte benachrichtigt. Dafür kommt das gleiche Verfahren zum Einsatz wie auch schon zur Compilezeit. Es erfolgt je eine Benachrichtigung beim Betreten und Verlassen der in Tabelle 4.2 aufgeführten Übersetzungsabschnitte.

Die Benachrichtigungen zu Then- und Else-Zweigen erfolgen während der Verarbeitung eines Forks (4.2.4.3). Wird während des Link-Prozesses der Quellcode eines linearen Segments in den Zielspeicherbereich kopiert, ist deren endgültige Position im Prüflingscode bekannt. Hier erfolgt ein Aufruf der Funktion `addRawCodeElement` unter Angabe

[7] Einzelschritte können nur in Folge einer Unterbrechung ausgeführt werden

Abschnitt	**Funktion** (begin/end)	**Parameter**
Core Cell	LinkInstrument	Zeiger auf SD::Instrument
Handler	LinkHandler	Handler ID
Then-Zweig	LinkThen	-
Else-Zweig	LinkElse	-

Tabelle 4.2 Benachrichtigungen des Compilers an die Debug-Data-Factory beim Betreten und Verlassen von Übersetzungsabschnitten zur Linkzeit

der Segment ID und der Zieladresse. Der Benachrichtigungsverlauf während des Link-Prozesses ist in Abbildung 4.4 beispielhaft dargestellt.

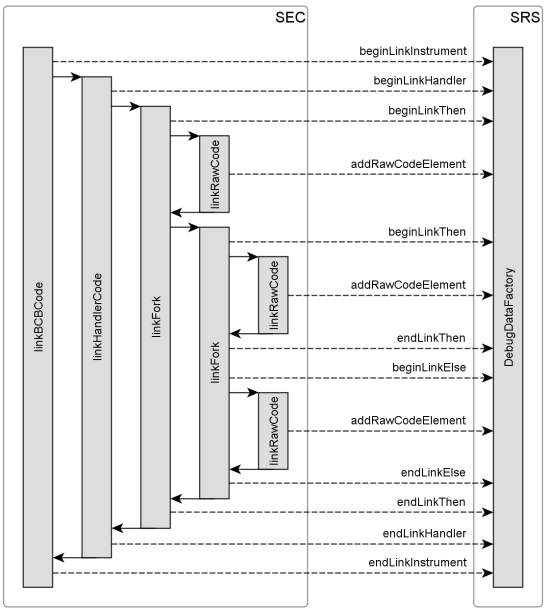

Abbildung 4.4 Schematische Darstellung der Kommunikation zwischen Compiler und Debug-Data-Factory zur Linkzeit. Es kommt derselbe Benachrichtigungsmechanismus zum Einsatz wie auch schon zur Compilezeit.

4.3.3.2 Algorithmus zur Rekonstruktion des Kontrollflusses

Aus diesen Informationen kann die Debug-Data-Factory den Kontrollfluss im Prüfling rekonstruieren. Damit die Daten nicht zwischengespeichert werden müssen, soll dies direkt während des Link-Prozesses erfolgen. Konkret müssen die möglichen Nachfolgesegmente für jedes lineare Segment des Audio-Handlers ermittelt werden. Die Segmente werden in der sequenziellen Reihenfolge ihrer Position im Maschinencode gelinkt. Die eigentliche Problematik besteht also darin, die Vorgänger für jedes ankommende Segment zu finden.

Ein Beispiel für eine mögliche Verzweigungsstruktur ist in Abbildung 4.5 dargestellt.

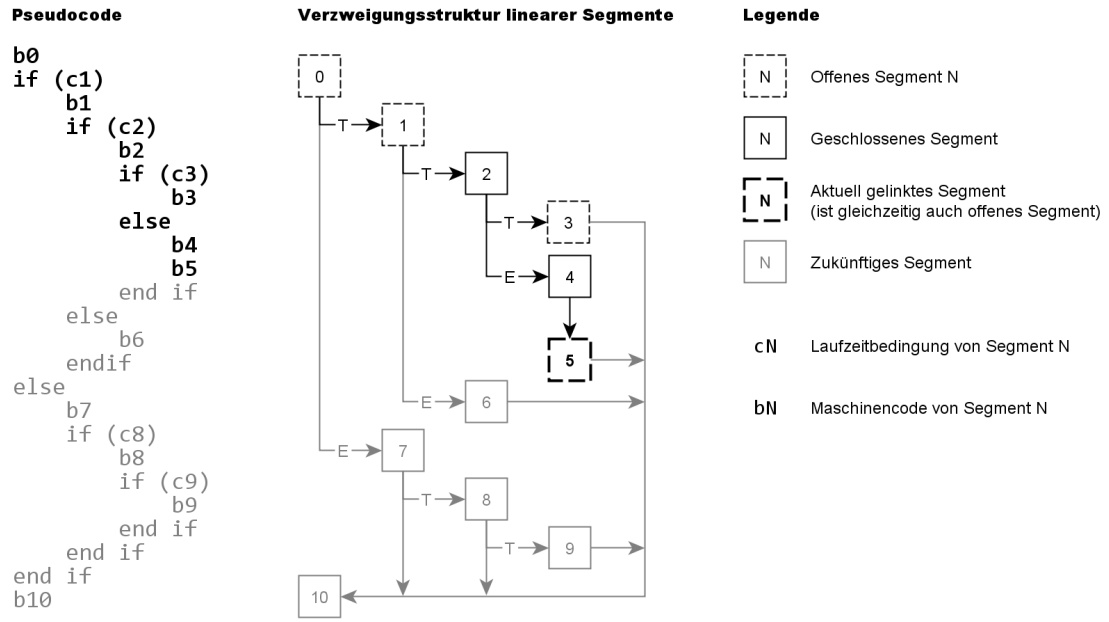

Abbildung 4.5 Beispiel einer zu rekonstruierenden Verzweigungsstruktur. Die linearen Segmente sind in der Reihenfolge nummeriert, in der sie vom Linker verarbeitet werden. Nachfolger im Kontrollfluss sind mit Pfeilen markiert. Für die Segmente 2 und 4 konnten bereits alle Nachfolger gefunden werden, während für die Segmente 0, 1, 3 und 5 noch Nachfolger erwartet werden. Alle ausgegrauten Elemente sind noch unbekannt.

Die Idee des Algorithmus' bildet die Zusammenfassung der ankommenden linearen Segmente zu *Branches*, die eine rekursive Struktur aufbauen. Branches sind Datenstrukturen, die über die folgenden Variablen verfügen:

- **type**: Der Typ eines Branches, der einen der Werte `TopLevel`, `Then` oder `Else` annehmen kann.
- **openSegment**: Das Segment in einem Branch, für welches noch ein Nachfolgesegment erwartet wird. Es wird im Folgenden als *offenes Segment* bezeichnet. Ein Branch kann stets nur maximal ein offenes Segment enthalten.
- **neverHadSegments**: Boolescher Wert der mit `false` initialisiert wird. Er wird auf `true` gesetzt, sobald ein Segment für den Branch registriert wird.
- **lastThen**: Untergeordneter *Kind-Branch* des letzten Then-Zweigs innerhalb des Branches. Die Variable wird mit `NULL` initialisiert. Dieser Wert sagt aus, dass es (noch) keinen untergeordneten Then-Zweig gab.

- **lastElse**: Untergeordneter *Kind-Branch* des letzten Else-Zweigs innerhalb des Branches analog zu *lastThen*.
- **parent**: Übergeordneter *Eltern-Branch*. Im Falle eines Top-Level-Branches ist der Wert dieser Variablen `NULL`. Alle anderen Branches besitzen einen Eltern-Branch. Der Eltern-Branch wird bei der Initialisierung gesetzt und ist von da an konstant.

Der Branch für den über einen Aufruf der Funktion `addRawCodeElement` das zuletzt gelinkte Segment übergeben wird, heißt *aktiver Branch*. Der jeweils aktive Branch wird in der globalen Variable `currentBranch` gespeichert. Die globale Variable `parentBranch` speichert den Eltern-Branch des jeweils aktiven Branches. Diese Variablen stellen nur Referenzen dar. Eine Änderung der Werte ändert nichts an der rekursiven Struktur der Branches. Den Ausgangspunkt bildet ein Top-Level-Branch, der zu Beginn der Kontrollflussanalyse beim Aufruf der Funktion `beginLinkHandler` erstellt wird. Er ist einzigartig und besitzt keinen übergeordneten Eltern-Branch. Er ist der aktive Branch zu Beginn des Verfahrens. `parentBranch` hat zunächst den Wert `NULL`.

Listing 4.1 stellt eine Implementierung der Funktionen zur Verwaltung der Branches in Pseudocode dar. Die Funktionen `beginLinkThen` und `beginLinkElse` werden beim Betreten von Then- und Else-Zweigen vom Linker aufgerufen. In Abbildung 4.5 erfolgt ein Aufruf der Funktion `beginLinkThen` vor dem Erreichen der Segmente 1, 2, 3, 8 und 9. `beginLinkElse` wird vor dem Erreichen der Segmente 4, 6 und 7 aufgerufen. Innerhalb der Funktionen wird zuerst ein neuer Branch als Kind-Branch des aktiven Branches erstellt (Zeile 2 bzw. 8). Als Parameter ist dabei der Typ des Branches (`Then` oder `Else`) sowie der Eltern-Branch anzugeben. Beim Eltern-Branch handelt es sich stets um den zu diesem Zeitpunkt aktiven Branch. Danach wird der neu erstellte Branch zum aktiven Branch (Zeile 4 bzw. 10). Als `parentBranch` wird der zuvor aktive Branch übernommen (Zeile 3 bzw. 9).

Die Funktionen `endLinkThen` und `endLinkElse` werden beim Verlassen von Then- und Else-Zweigen vom Linker aufgerufen. In Abbildung 4.5 erfolgt ein Aufruf der Funktion `endLinkThen` nach dem Erreichen der Segmente 3, 5, 6, 8 und 9. `endLinkElse` wird nach dem Erreichen der Segmente 5, 6 und 7 aufgerufen. Innerhalb der Funktionen wird zuerst überprüft, ob für den aktiven Branch Segmente registriert wurden (Zeile 14 bzw. 20). Ist dies *nicht* der Fall, ist der Branch leer und kann ignoriert werden. Im Anschluss werden die Werte der globalen Variablen `currentBranch` und `parentBranch` wieder auf die jeweils übergeordneten Branches gesetzt (Zeile 15 und 16 bzw. 22 und 23). Beim Verlasen eines Else-Zweigs muss zudem überprüft werden, ob es sowohl einen Then- als auch Else-Zweig gab. Ist dies der Fall, wurden für das offene Segment des aktiven Bran-

ches bereits alle Nachfolger gefunden. Das Segment kann dann *geschlossen* werden. In der Datenstruktur des aktiven Branches wird das durch das Zurücksetzen der Variablen `openSegment` auf `NULL` realisiert.

```
1  function beginLinkThen() {
2    currentBranch.lastThen = new tBranch(type = Then, parent = currentBranch)
3    parentBranch = currentBranch
4    currentBranch = currentBranch.lastThen
5  }
6
7  function beginLinkElse() {
8    currentBranch.lastElse = new tBranch(type = Else, parent = currentBranch)
9    parentBranch = currentBranch
10   currentBranch = currentBranch.lastElse
11 }
12
13 function endLinkThen() {
14   if (currentBranch.neverHadSegments) parentBranch.lastThen = NULL
15   parentBranch = parentBranch.parent
16   currentBranch = parentBranch
17 }
18
19 function endLinkElse() {
20   if (currentBranch.neverHadSegments) parentBranch.lastElse = NULL
21
22   parentBranch = parentBranch.parent
23   currentBranch = parentBranch
24
25   if (currentBranch.lastThen != NULL
26   and currentBranch.lastElse != NULL) currentBranch.openSegment = NULL
27 }
```

Listing 4.1 Implementierung der Funktionen zur Verwaltung von Branches in Pseudo-Code. Die Funktionen mit dem Präfix *begin* erstellen untergeordnete Branches. Die Funktionen mit dem Präfix *end* schließen diese wieder. Für die enthaltenen Segmente sind zu diesem Zeitpunkt noch nicht notwendigerweise alle Nachfolger gefunden worden. In allen Funktionen werden die Variablen *currentBranch* und *parentBranch* aktualisiert.

Bei der Verarbeitung eines linearen Segments durch den Linker wird die Funktion `addRawCodeBlock` aufgerufen. Für die Kontrollflussanalyse bedeutet das, dass das Segment dem aktuell aktiven Branch hinzugefügt werden muss. Zudem muss es als Nachfolger verschiedener zuvor gelinkter Segmente registriert werden. Die Registrierung eines Nachfolgers für ein Segment erfolgt durch den Aufruf der Methode `addSuccessor` eines Segments. Listing 4.2 zeigt die Implementierung der Funktion `addRawCodeBlock` in Pseudocode. In Zeile 3 wird hier zunächst abgefragt, ob für den aktiven Branch zu einem früheren Zeitpunkt bereits ein Segment registriert wurde. Ist dies nicht der Fall, wurde der Branch gerade neu erstellt. Damit ist das neue Segment ein möglicher Nachfolger des offenen Segments im Eltern-Branch (Zeile 6). In Beispiel 4.5 betrifft dies die Segmente 1 bis 4 und 6 bis 9. Im Eltern-Branch existiert ein offenes Segment genau dann, wenn es sich beim

aktiven Branch nicht um den Top-Level-Branch handelt (Zeile 5). Das neue Segment wird im Anschluss als offenes Segment des aktiven Branches registriert (Zeile 12).

```
1  function addRawCodeElement(tSegment Segment)
2  {
3    if (currentBranch.neverHadSegments) {
4      currentBranch.neverHadSegments = false
5      if (currentBranch.type != TopLevel) {
6        parentBranch.openSegment.addSuccessor(Segment)
7      }
8    }
9    else {
10     finalizePreviousNestedBranches(currentBranch, Segment)
11   }
12   currentBranch.openSegment = Segment
13 }
14
15 function finalizePreviousNestedBranches(tBranch branch, tSegment Segment)
16 {
17   if (branch.lastThen != NULL) {
18     finalizePreviousNestedBranches(branch.lastThen, Segment)
19     branch.lastThen = NULL
20   }
21   if (branch.lastElse != NULL) {
22     finalizePreviousNestedBranches(branch.lastElse, Segment)
23     branch.lastElse = NULL
24   }
25   if (branch.openSegment != NULL) {
26     branch.openSegment.addSuccessor(Segment)
27   }
28 }
```

Listing 4.2 Implementierung der Funktionen zur Registrierung eines Segments als Nachfolger vorheriger Segmente in Pseudo-Code. Neue Segmente werden vom Linker als Parameter der Funktion *addRawCodeBlock* übergeben. Die Registrierung des neuen Segments als Nachfolger der offenen Segmente in untergeordneten Kind-Branches erfolgt durch Rekursion in der Funktion *finalizePreviousNestedBranches*.

Wurde hingegen für den aktiven Branch zu einem früheren Zeitpunkt bereits ein Segment registriert, wird der Kontrollfluss das neue Segment immer dann erreichen, wenn das zuvor registrierte Segment auch erreicht wurde. In Beispiel 4.5 ist dies für die Segmente 5 und 10 der Fall. Das neue Segment ist damit der Nachfolger des offenen Segment des aktiven Branches und der offenen Segmente aller untergeordneter Branches. Dafür wird in Zeile 10 die Funktion `finalizePreviousNestedBranches` aufgerufen. Innerhalb dieser Funktion erfolgt je ein rekursiver Aufruf für untergeordnete Then- und Else-Branches, sofern diese existieren (Zeilen 18 und 22). Für alle auf diese Weise durchlaufenen Branches wird das neue Segment in Zeile 26 als Nachfolger des jeweils offenen Segments registriert. In Beispiel 4.5 werden für das Segment 10 der Reihe nach die Vorgängersegmente 3, 5, 6, 9, 8 und 7 registriert. Die Segmente 0, 1 und 2 hingegen sind keine Vorgänger von 10, da sie zu diesem Zeitpunkt bereits über je einen Nachfolger für Then- und Else-Fall verfügen. Beim Verlassen der entsprechenden Else-Zweige wurde `openSegment` für

deren Branches zu diesem Zweck auf `NULL` gesetzt (siehe Zeile 26 in Listing 4.1 zum Vergleich). Im Falle von Segment 5 kommt es hingegen zu keinem rekursiven Aufruf der Funktion `finalizePreviousNestedBranches`, da keine untergeordneten Branches existieren. Hier wird das Segment 5 in Zeile 26 nur als direkter Nachfolger des offenen Segments 4 des aktiven Branches registriert.

Der Abschluss des Link-Prozesses für einen Handler wird durch einen Aufruf der Funktion `endLinkHandler` markiert. In der Debug-Data-Factory liegt zu diesem Zeitpunkt der vollständige Kontrollflussgraph des Handlers vor.

4.3.4 Anpassungen des Prüflingsprogramms

Nach dem in Kapitel 3 erarbeiteten Konzept erfordert die Verwendung eines Same-Process Debuggers eine spezielle Kompilierung des Prüflings. In diesem Abschnitt werden die dafür nötigen Anpassungen am Reaktor Core Compiler beschrieben. Da alle Sprünge im Maschinencode von Reaktor Core Programmen in Richtung der Ausführung erfolgen (4.2.1), kann die Korrektur von Sprunganweisungen (3.2.2.3) bei der Kompilierung des Prüflings entfallen. Zudem sollen die Mechanismen zur Ablaufkontrolle zunächst nur für den Audio-Handler verfügbar sein. Die Anpassungen am Reaktor Core Compiler begrenzen sich damit auf das Einfügen von Puffern vor jedem Sprungziel im Audio-Handler und die Ergänzung eines Callback-Points an dessen Beginn.

4.3.4.1 Einfügen von Puffern vor jedem Sprungziel

Gemäß Abschnitt 3.2.1.2 der Konzeption ist das Einfügen von Puffern zwischen den dort definierten sequenziellen Befehlsblöcken nötig. Ein sequenzieller Befehlsblock beginnt an einem Sprungziel und umfasst alle Maschinenbefehle bis zum nächsten Sprungziel. In Reaktor Core Programmen können nur die jeweils ersten Maschinenbefehle von linearen Segmenten Sprungziele sein. Das ist jedoch nicht für jedes lineare Segment der Fall. Demnach können lineare Segmente nur Teilstücke sequenzieller Befehlsblöcke sein. Die Zuordnung gestaltet sich wie folgt. Ein lineares Segment bildet den Anfang eines sequenziellen Befehlsblocks, wenn es das erste lineare Segment im Maschinencode ist, oder wenn dessen erster Befehl ein Sprungziel ist. Andernfalls gehört das lineare Segment zum gleichen sequenziellen Befehlsblock wie sein direkter Vorgänger im Maschinencode. Anhand des Beispiels aus Abbildung 4.5 wird das ersichtlich. Da hier nur die Segmente 4, 6, 7 und 10 Sprungziele sind, bilden die folgenden Gruppen linearer Segmente jeweils einen sequenziellen Befehlsblock: $(0, 1, 2, 3)$, $(4, 5)$, (6), $(7, 8, 9)$ und (10). Anhand der bei der Kontrollflussanalyse gewonnen Daten könnte nun ermittelt werden, welche linearen Seg-

mente ein Sprungziel enthalten. Vor diesen Segmenten könnten dann Puffer eingefügt werden[8].

Aufgrund der einfachen Struktur von Reaktor Core Programmen lässt sich die Frage, an welchen Stellen Puffer notwenig werden, jedoch wie folgt vereinfachen. Sprungziele existieren nur an den Stellen des Prüflingscodes, an denen direkt zuvor ein Teil des Maschinencodes übersprungen werden kann. In Reaktor Core Programmen kann es sich dabei nur um Then- und Else-Zweige handeln. Das Einfügen eines Puffers *hinter* dem letzten linearen Segment von Then- und Else-Zweigen erfüllt im Rückschluss dieselbe Funktion, wie das Einfügen eines Puffers *vor* jedem Sprungziel. Im Falle des Beispielprogramms aus Abbildung 4.5 wird jeweils hinter den linearen Segmenten 3, 5, 6 und 9 ein Puffer nötig.

Der konkrete Einfügevorgang erfolgt bei der Verarbeitung der verschachtelten Struktur der Forks zur Linkzeit (4.2.4.3). Hier werden alle Then- und Else-Zweige entsprechend ihrer Reihenfolge im Prüflingscode durchlaufen. Hinter jedem letzten linearen Segment eines solchen Zweigs wird als Puffer ein Multi-Byte-NOP[9] der Länge eines Call Befehls in den Zielspeicherbereich geschrieben. Auf die gleiche Weise wird die Länge und die Position aller Puffer auch in die Berechnung von Sprungdistanzen und die Berechnung der Gesamtlänge des Prüflingscodes einbezogen.

4.3.4.2 Kompilierung des Callback-Points

Der Maschinencode des Callback-Points wird von einer Routine des *Memory-Backends* des Moduls `SRS` erzeugt. Das Memory-Backend ist durch die Klasse `SRS::MemoryBackend` repräsentiert. Die Implementierung folgt dem für das Compiler-Backend in der Klasse `SEC::BCBCompiler` gewählten Spezialisierungsansatz (4.2.4.2). `SRS::MemoryBackend` kann somit leicht für beliebige Architekturen überladen werden. Im Rahmen der Implementierung des Prototyps wurde zunächst nur eine Variante für x86 Architekturen mit 32 Bit Adressbreite und SSE-Unterstützung erstellt.

Die Struktur eines Callback-Points ist immer gleich. Listing 4.3 zeigt den Maschinencode. Für die Adresse des Callback-Flags (`pCallbackFlag`), den Wert für die Identifizierung des zugehörigen Handlers (`pHandler`), die Adresse zur Fortsetzung der Ausführung im Nicht-Callback-Fall (`continue`) und die Adresse der Behandlungsroutine (`pCallee`) wurden hier Platzhalter verwendet. Diese Werte sind bereits zur Übersetzungszeit be-

[8] Tatsächlich wären dafür zusätzliche Änderungen notwendig gewesen, da die Kontrollflussinformationen zu einem Segment erst bekannt sind, wenn das Segment bereits gelinkt *wurde*.

[9] Bei einer Adressbreite von 32 Bit ist ein Call Befehl fünf Byte lang. Dann kommt z. B. die Bytefolge *0F 1F 44 00* zum Einsatz [Int12a].

kannt und werden direkt in den Maschinencode geschrieben.

```
1    ...
2    mov ebx, pCallbackFlag
3    cmp byte ptr [ebx], 0x00
4    je continue
5    push ecx
6    push edx
7    push pHandler
8    mov ebx, pCallee
9    call ebx
10   pop edx
11   pop ecx
12
13 continue:
14   ...
```

Listing 4.3 Maschinencode eines Callback-Points. Die Werte der kursiv dargestellten Variablen sind zur Übersetzungszeit bekannt und werden direkt in den Maschinencode geschrieben. In den Zeilen 2 und 3 wird der Wert des globalen Callback-Flags ausgelesen. Ist es gesetzt, wird der Zeiger auf den jeweiligen Handler in Zeile 7 als Funktionsparameter auf den Stack gelegt. Der Aufruf der Behandlungsroutine erfolgt in Zeile 8. In der überwiegenden Mehrheit der Durchläufe ist kein Callback nötig. Der Mehraufwand ist dann auf drei Maschinenbefehle begrenzt.

Zunächst wird in Zeile 2 der Wert des Callback-Flags gelesen und in Zeile 3 mit Null verglichen. Im Gleichheitsfall ist kein Callback erforderlich. Der Sprungbefehl in Zeile 4 wird dann ausgeführt. Es folgt die reguläre Verarbeitung des Prüflingscodes. Andernfalls kommt es zur Ausführung des Callbacks. Die Register ECX und EDX enthalten zu diesem Zeitpunkt u. U. bereits wichtige Werte, wie z. B. Parameter für die Ausführung des Audio-Handlers. Bei der Ausführung der Behandlungsroutine des Callbacks werden diese Registerwerte jedoch nicht garantiert geschützt. In den Zeilen 5 und 6 werden die Werte von ECX und EDX daher auf dem Stack gesichert. Für die Identifikation des Handlers wird der Zeiger auf das in den Programmdaten für den Handler angelegte Objekt in Zeile 7 als Funktionsparameter auf den Stack gelegt. Der Aufruf der Behandlungsroutine erfolgt in Zeile 8. Nach der Rückkehr der Ausführung werden die Werte von ECX und EDX in den Zeilen 10 und 11 wiederhergestellt. In der überwiegenden Mehrheit der Durchläufe ist kein Callback nötig. Der Mehraufwand ist dann auf die Ausführung der drei Maschinenbefehle für Abfrage, Vergleich und bedingtem Sprung begrenzt. Der Maschinencode wird zu Beginn des Linking-Prozesses des Audio-Handlers angefordert und noch vor dem ersten linearen Segment in den Prüflingscode eingefügt.

4.4 Laufzeitprozess

Im folgenden Abschnitt werden die Aspekte der Implementierung eines Same-Process Debuggers betrachtet, die zur Laufzeit des Prüflings erforderlich werden. Dabei handelt es sich um die Realisierung von Breakpoints nach 3.2.1.2 und die Unterbrechung der Echtzeitverarbeitung nach 3.3.3.

4.4.1 Voraussetzungen

Alle zur Laufzeit des Prüflings anfallenden Arbeitsschritte werden von einer Instanz der Klasse `SRS::Debugger` realisiert. Sie hat Zugriff auf die Datenbasis der Core Cell des Prüflings und ist zudem über die Schnittstelle `SDI` der Reaktor Core Library mit der Entwicklungsumgebung `SDE` verbunden. Innerhalb der Entwicklungsumgebung existiert nur eine einzige solche Instanz. Sie wird im Folgenden als *Debugger* bezeichnet.

Bei der Übersetzung des Prüflingsprogramms wurden alle nötigen Anpassungen zur Verwendung eines Same-Process Debuggers vorgenommen. Zudem wurden die von der Debug-Data-Factory gesammelten Programmdaten nach Abschluss des Übersetzungsvorgangs dem Debugger übergeben.

4.4.2 Breakpoints

Das Einzelschrittverfahren des Debuggers basiert auf der Verwendung temporärer Breakpoints. In diesem Abschnitt wird die Realisierung des dafür nötigen Breakpoint-Mechanismus' im Prototyp des Debuggers für Reaktor Core Programme vorgestellt.

4.4.2.1 Vereinfachungen für die Einzelschrittausführung

Für die Implementierung des Prototypen kann das Verfahren aus Abschnitt 3.2.1.2 der Konzeption vereinfacht werden. Denn für das Einzelschrittverfahren werden zunächst nur temporäre Breakpoints benötigt. Diese müssen nach einer Unterbrechung nicht erneut implementiert werden. Da es in Reaktor Core Programmen zudem keine Sprünge entgegen der Ausführungsrichtung gibt, kann es nicht zur gegenseitigen Überlappung temporärer Breakpoints nach 3.2.2.3 kommen. Demzufolge können die jeweils nötigen Breakpoints ohne die Gefahr einer Änderung der Programmsemantik stets an den Stellen der möglichen Folgebefehlen implementiert werden. Im Hinblick auf die zukünftige Weiterentwicklung wurden die Mechanismen zur Verwaltung und sukzessiven Implementierung von Breakpoints aber bereits für den allgemeinen Fall implementiert.

4.4.2.2 Verwaltung von Breakpoints

Die Verwaltung von Breakpoints erfolgt durch die Klasse `SRS::RTDSegment`. Zur Laufzeit existiert je eine Instanz dieser Klasse für jedes lineare Segment des Prüflingscodes, für das mindestens ein Breakpoint gesetzt ist. Die Instanzen werden im Folgenden als *RTDSegment* bezeichnet[10]. Breakpoints werden hier zunächst nur registriert und gemäß ihrer Reihenfolge im linearen Segment sortiert. Die Implementierung von Breakpoints im Maschinencode des Prüflings wird in Abschnitt 4.4.2.3 erläutert. Da für jedes RDTSegment immer nur ein Breakpoint implementiert sein kann, erfolgt die Zwischenspeicherung des ursprünglichen Maschinencodes ebenfalls in der Datenstruktur des zugehörigen RTDSegments. Der zu diesem Zweck allozierte Speicherbereich wird im Folgenden als *Swap-Speicher* bezeichnet.

Wird einem RTDSegment ein Breakpoint hinzugefügt, wird zunächst überprüft ob er vor dem zu diesem Zeitpunkt implementierten Breakpoint liegt. Ist das der Fall, muss der aktuell implementierte Breakpoint entfernt und der neu hinzugefügte Breakpoint implementiert werden. Für diesen Zweck wird das Callback-Flag gesetzt, sodass die nötigen Änderungen am Maschinencode des Prüflings beim nächsten Durchlauf des Callback-Points vorgenommen werden. Liegt der neu hinzugefügte Breakpoint hinter dem aktuell implementierten Breakpoint, genügt die einfache Registrierung des Breakpoints. Das Entfernen von Breakpoints erfolgt analog dazu. Wurden alle Breakpoints eines linearen Segments entfernt, wird auch das zugehörige RTDSegment gelöscht.

4.4.2.3 Implementierung im Maschinencode

Für die Implementierung von Breakpoints im Maschinencode des Prüflings kommt, wie bereits für die Kompilierung von Callback-Points in 4.3.4.2, das Memory-Backend zum Einsatz. Die Aufrufabfolge der im Folgenden beschriebenen Routinen wird vom jeweils verantwortlichen RTDSegment vorgegeben und muss bei der Ausführung der Behandlungsroutine des Callback-Points eingehalten werden.

Die Implementierung eines Breakpoints wird von der Methode `insertCall` realisiert. Als Parameter wird ihr der Zeiger auf das zugehörige RTDSegment, die Speicherstelle des Breakpoints und die Adresse des zu verwendenden Swap-Speichers übergeben. Zuerst wird der ursprüngliche Maschinencode an der Stelle des Breakpoints in den Swap-Speicher kopiert. Dann wird die Distanz zwischen der Speicherstelle des Breakpoints und dem Einstiegspunkt der Behandlungsroutine berechnet. Innerhalb der Behandlungsroutine werden verschiedene architekturspezifische Low-Level-Operationen nötig. Sie liegt

[10] Das Präfix RTD steht für *Runtime-Data*.

daher ebenfalls im Memory-Backend (siehe 4.4.2.5). Der Maschinencode des Breakpoints besteht aus dem Opcode des Call-Befehls (1 Byte) und der zuvor berechneten relativen Sprungdistanz (4 Byte). Die Werte werden direkt hintereinander in den Speicher geschrieben. Abschließend wird eine Referenz auf das für den Breakpoint verantwortliche RTDSegment in einer Map-Struktur im Memory-Backend abgelegt[11]. Als Schlüssel dient die Adresse des Breakpoints. Die Zuordnung zwischen einer solchen Speicheradresse im Prüflingscode und dem dafür verantwortlichen RTDSegment ist stets eindeutig. Innerhalb der Behandlungsroutine für Breakpoints wird damit das Auffinden des zugehörigen RTDSegments beschleunigt. Bei n implementierten Breakpoints liegt die Laufzeitkomplexität für das Auffinden des korrekten RTDSegment im Worst-Case bei $\mathcal{O}(log(n))$.

Das Entfernen eines Breakpoints aus dem Maschinencode des Prüflings erfolgt in der Methode `restoreOriginalCode` des Memory-Backends. Als Parameter sind hier nur die Adressen von Breakpoint und Swap-Speicher anzugeben. Innerhalb der Methode wird der Inhalt des Swap-Speichers einfach zurück in den Maschinencode des Prüflings kopiert und der zugehörige Eintrag in der Map-Struktur zur Zuordnung der jeweils verantwortlichen RTDSegments gelöscht.

4.4.2.4 Die Funktion onHitBreakpoint

Zur Unterbrechungen der Ausführung kommt es beim Erreichen eines Breakpoints im Maschinencode des Prüflings. Dies kann zu einem beliebigen Zeitpunkt der Ausführung geschehen. Für einige Bestandteile des Ausführungszustandes des Prüflings besteht die Gefahr während einer Unterbrechung verändert zu werden. Für die Werte in den Registern des Prozessors ist dies sogar unausweichlich. Die betroffenen Bestandteile des Ausführungszustandes müssen daher im Vorfeld einer Unterbrechung gesichert und im Nachhinein wiederhergestellt werden. Bei der Implementierung in Reaktor Core erfolgt dies in der Behandlungsroutine für Breakpoints `onHitBreakpoint` im Memory-Backend.

Bei der Funktion `onHitBreakpoint` handelt es sich um eine statische Funktion in einer von `SRS::MemoryBackend` abgeleiteten Klasse. Die folgenden Betrachtungen beziehen sich auf die Implementierung in `SRS::MemoryBackendSSE`. Die Funktion soll stets nach der Aufrufkonvention *stdcall* kompiliert werden. Damit kann sichergestellt werden, dass die Funktion nach ihrer Ausführung die Bereinigung des Stacks selbst übernimmt. Andere Aufrufkonventionen wie z. B. *cdecl* überlassen dies dem Aufrufenden, was im Falle der Behandlungsroutine für Breakpoints ungeeignet wäre. Um eine Kompilierung nach stdcall

[11] Konkret kommt hier die Klasse *std::map* der C++ Standard Template Library zum Einsatz. Eine einzige solche Map genügt für alle Handler und Core Cells, da die Speicheradressen, die als Schlüssel verwendet werden, stets eindeutig sind.

zu erzwingen wurde bei der Definition von `onHitBreakpoint` das Attribut `__stdcall` angegeben.

Eine weitere Besonderheit ist hinsichtlich des Schutzes aller Prozessorregister beim Aufruf der Funktion zu beachten. Da unter der stdcall-Konvention der Rückgabewert einer Funktion im Register EAX übergeben wird, muss dieses beim Aufruf nicht garantiert geschützt werden. Im Falle von `onHitBreakpoint` ist der Schutz von EAX jedoch zwingend erforderlich, um den unveränderten Ausführungszustand des Prüflings sichern zu können. Konkret problematisch sind spezielle Validierungsbefehle die vom Compiler automatisch im Funktions-Header generiert werden können. Die Generierung derartiger Validierungsbefehle wird unterbunden, wenn es sich wie im Falle von `onHitBreakpoint` beim ersten Befehl im Funktionsrumpf um einen Inline-Assembler-Befehl handelt. Dies ist eine allgemeine Konvention in C++ Compilern.

4.4.2.5 Implementierung der Funktion onHitBreakpoint

Im Funktionsrumpf von `onHitBreakpoint` werden die folgenden Arbeitsschritte in der hier angegebenen Reihenfolge ausgeführt. Zuerst werden die General-Purpose-Register des Prozessors auf dem Stack gesichert. Im x86 Befehlssatz existiert hierfür der Befehl `pushad`. Damit werden die Werte der Register EAX, ECX, EDX, EBX, ESP, EBP, ESI und EDI in dieser Reihenfolge auf den Stack gelegt[12]. Das FLAGS Register gehört nicht zum Ausführungszustand von Reaktor Core Programmen und kann ignoriert werden.

Danach erfolgt die Sicherung des erweiterten Registersatzes des Prozessors. Hierbei handelt es sich um die Register für Gleitkommaarithmetik sowie die XMM-Register von SSE-Erweiterungen. Sie werden mit dem Assemblerbefehl `fxsave` in einem 512 Byte großen Speicherbereich des Hauptspeichers gesichert. Zu beachten ist, dass der verwendete Speicherbereich bereits vor dem Aufruf von `fxsave` alloziert und an einer 16-Byte-Grenze ausgerichtet sein muss [Int12a]. Für die Allokation derart ausgerichteter Speicherbereiche existieren in den verbreiteten Betriebssystemen spezielle Funktionen. Unter Windows handelt es sich dabei beispielsweise um die Funktion `_aligned_malloc` [Mic12a]. Derartige Funktionen können in diesem Fall jedoch nicht verwendet werden, da sie in einigen Fällen dazu führen, dass der erste vom Compiler registrierte Befehl im Funktionsrumpf kein Inline-Assembler-Befehl mehr ist. Damit könnten vom Compiler erneut Validierungsbefehle im Funktions-Header generiert werden, die den Wert des Registers EAX verfälschen. Stattdessen wird der Speicherbereich als statische Variable der Funktion deklariert und manuell ausgerichtet. Der Speicherbereich wird damit bereits

[12] Im Falle des Stack-Pointers ESP handelt es sich um den Wert *vor* der Ausführung des pushad Befehls.

beim Start von Reaktor im Datensegment der Anwendung angelegt.

Im nächsten Schritt wird die Rücksprungadresse der Funktion korrigiert. Das ist nötig, da nach der Rückkehr der Funktion der ursprüngliche Maschinenbefehl an der Stelle des Breakpoints ausgeführt werden soll. Die Rücksprungadresse wird dafür an der Adresse `EBP+4` aus dem Stackframe der Funktion gelesen, um die Länge eines Call-Befehls dekrementiert und wieder zurückgeschrieben.

Im Anschluss muss das für den Breakpoint verantwortliche RTDSegment gefunden werden. Dafür wird der bei der Implementierung des Breakpoints angelegte Eintrag in der Map-Struktur des Memory-Backends anhand der Adresse des Breakpoints ermittelt. Wurde das RTDSegment gefunden, wird abschließend noch der Speicherbereich des Prüflingscodes als editierbar markiert (siehe 4.2.4.3), bevor die weitere Behandlung der Unterbrechung durch eine Routine des RDTSegment erfolgt. Diese Routine initiiert die Unterbrechung der Echtzeitverarbeitung nach 4.4.3, die vollständige Wiederherstellung des ursprünglichen Prüflingscodes und die Darstellung des Ausführungszustandes in der Entwicklungsumgebung. In der Folge ist die Inspektion des Prüflings durch den Benutzer möglich.

Wird die Ausführung des Prüflings fortgesetzt, kehrt der Funktionsaufruf der Routine des RTDSegment zur Funktion `onHitBreakpoint` zurück. Zunächst werden nun alle nötigen Breakpoints erneut implementiert, bevor der Speicherbereich des Prüflingscode wieder als ausführbar markiert wird. Danach werden zuerst die Werte des erweiterten Registersatzes mit Hilfe der Assembleranweisung `fxrstor` und schließlich die Werte der General-Purpose-Register mit `popad` wiederhergestellt[13]. Der Rücksprungbefehl von `onHitBreakpoint` veranlasst die Bereinigung des Stacks und führt dazu, dass die Ausführung mit dem ursprünglichen Maschinenbefehl an der Stelle des Breakpoints fortgesetzt wird.

4.4.3 Realisierung von Unterbrechungen

In Abschnitt 4.4.2.5 wurde die Implementierung der Behandlungsroutine für Breakpoints beschrieben. Sie realisiert die Sicherung und Wiederherstellung des Ausführungszustands des Prüflings sowie die Korrektur der Rücksprungadresse. Für die weitere Behandlung von Unterbrechung wird eine Routine von `SRS::RTDSegment` aufgerufen. Sie initiiert u. a. die Unterbrechung der Ausführung. In diesem Abschnitt soll die Implementierung des dafür verwendeten Mechanismus' vorgestellt werden. Die Grundlage bildet das in Abschnitt 3.3.3 der Konzeption erarbeitete Verfahren.

[13] Die Wiederherstellung in der entgegengesetzten Reihenfolge der Speicherung erfolgt hier automatisch.

4.4.3.1 Realisierung des separaten Stackspeichers

Voraussetzung für die Unterbrechung der Echtzeitverarbeitung bildet die Umleitung des Aufrufstacks. Dafür ist ein zusätzlicher Speicherbereich erforderlich, der als *separater Stackspeicher* bezeichnet wurde (3.3.3.3). Als geeignet erweist sich ein Speicherbereich, dessen Position und Größe drei aufeinanderfolgenden Speicherseiten entspricht. Sowohl für die erste als auch für die letzte Speicherseite wird jeglicher Zugriff gesperrt[14]. Die dazwischenliegende Seite dient als separater Stackspeicher. Durch die Sperrung der umliegenden Speicherseiten können Fehler durch Über- und Unterläufe des Stacks erkannt werden. Die Grenzen der verwendeten Speicherseite werden in den Membervariablen `StackTop` und `StackBottom` der Klasse `StackForge` gespeichert. Der separate Stackspeicher wird bei seiner Verwendung wie üblich in Richtung absteigender Adresswerte gefüllt.

4.4.3.2 Wahl des Entkopplungspunkts

Die Stelle in der Aufrufhierarchie an der die Umleitung des Aufrufstacks stattfindet, wurde als *Entkopplungspunkt* bezeichnet (3.3.3.3). Die Ausführung des Echtzeit-Threads muss den Entkopplungspunkt vor dem Prüflingscode erreichen. Dementsprechend muss sich der Entkopplungspunkt für Reaktor Core Programme in der Ausführungsumgebung Reaktor befinden. Eine geeignete Position stellt die Methode `processAudio` der Klasse `TReaktor` dar. Für die Vereinfachung der späteren Entkopplung wird hier ein zusätzlicher Funktionsaufruf eingefügt. Bei der aufgerufenen Funktion handelt es sich um die Methode `centralDecouplingRoutine` der Klasse `StackForge`. Sie stellt den Entkopplungspunkt dar und wird im folgenden Abschnitt 4.4.3.3 detailliert betrachtet.

4.4.3.3 Umleitung des Aufrufstacks

Der gesamte Ablauf der im Folgenden beschriebenen Arbeitsschritte ist in Abbildung 4.6 schematisch dargestellt. Pfeile mit durchgezogener Linie repräsentieren den gewöhnlichen Kontrollfluss innerhalb der Aufrufhierarchie der beteiligten Threads. Die einzelnen Manipulationen des Kontrollflusses sind durch die Pfeile mit gestrichelter Linie und den Nummerierungen (1) bis (4) dargestellt.

Neben der Funktionalität zur Erstellung des separaten Stackspeichers enthält die Klasse `StackForge` auch eine Methode, die sowohl die Umleitung als auch die Entkopplung des Aufrufstacks durchführt. Dabei handelt es sich um die Funktion `centralDecoupling`

[14] Im Falle von Windows kommt dafür die API-Funktion *VirtualProtect* mit dem Parameter *PAGE_NOACCESS* zum Einsatz. Unter Mac OS X wird *sys_icache_invalidate* verwendet.

`Routine`. Die Verwendung einer einzigen Funktion für alle Aufgaben im Zusammenhang mit der Entkopplung erweist sich als günstig. Zur Unterscheidung der verschiedenen Aufgaben erwartet die Funktion einen Parameter vom Aufzählungstyp `tCallingMode`. Er kann die Werte `INIT`, `PROCESSAUDIO`, `BREAK` und `CONTINUE` annehmen. Listing 4.4 zeigt die Implementierung der Funktion in Pseudocode. Im Anschluss an die Erstellung des separaten Stackspeichers wird die Funktion einmalig mit dem Parameterwert `INIT` aufgerufen. Vor und nach diesem Aufruf wird der Wert des Stack-Pointer-Registers des Prozessors mit dem Wert der Variablen `StackTop` vertauscht. Der Stackframe für diesen Funktionsaufruf wird demzufolge im separaten Stackspeicher aufgebaut. Innerhalb der Funktion werden nun in Zeile 5 die Werte der Register von Stack- und Base-Pointer in den Membervariablen `StoredSP` und `StoredBP` gesichert. Diese Werte werden später für die Entkopplung des Aufrufstacks verwendet. Abbildung 4.7 zeigt den Inhalt des separaten Stackspeichers während der Initialisierung.

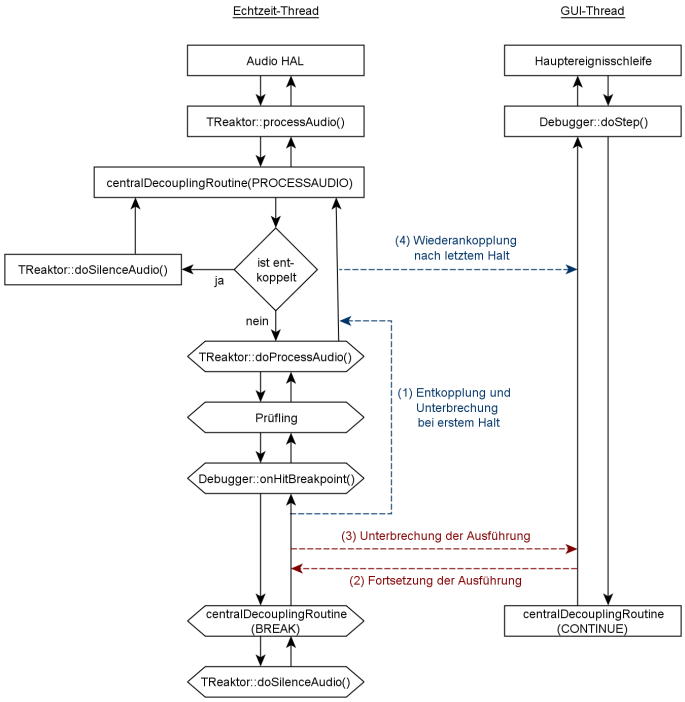

Abbildung 4.6 Schematische Darstellung der Manipulationen des Kontrollflusses zur Realisierung von Unterbrechungen. Funktionen deren Stackframes im separaten Stackspeicher aufgebaut werden, sind als Hexagons dargestellt. Das Erreichen eines Breakpoints im Prüflingscode führt zur Entkopplung des Aufrufstacks (1). Die Ausführung im Echtzeit-Thread kehrt daraufhin zurück und kann etwaige Zeitschranken einhalten. Nach der Unterbrechung übernimmt der GUI-Thread die Ausführung des Prüflings (2). Der Echtzeit-Thread bleibt entkoppelt, bis das Ende des Prüflings erreicht wird (4). Zuvor kann es zu beliebig vielen weiteren Unterbrechungen kommen (2 und 3).

```
1  function centralDecouplingRoutine(tCallingMode Mode)
2  {
3    switch (Mode) {
4      case INIT:
5          STORE(StoredBP, StoredSP)
6          return
7
8      case PROCESSAUDIO:
9          if(Status = DECOUPLED) {
10             doSilenceAudio()
11             return
12         }
13         else {
14             Status = NOT_DECOUPLED
15             XCHG(StoredBP, StoredSP)
16
17             doProcessAudio()
18
19             XCHG(StoredBP, StoredSP)
20             Status = NOT_DECOUPLED
21             return
22         }
23
24     case BREAK:
25         if(Status = NOT_DECOUPLED) doSilenceAudio()
26         XCHG(StoredBP, StoredSP)
27         Status = DECOUPLED
28         return
29
30     case CONTINUE:
31         XCHG(StoredBP, StoredSP)
32         return
33   }
34 }
```

Listing 4.4 Implementierung der Funktion zur Umleitung, Entkopplung und Umschaltung des Aufrufstacks in Pseudocode. Der Parameter *Mode* dient der Unterscheidung der verschiedenen Aufgaben. *STORE* und *XCHG* sind als Präprozessormacros realisiert. *STORE* speichert die Werte der Stackregister in den angegebenen Variablen. *XCHG* führt einen Austausch zwischen den Werten der Stackregister und den Werten der angegebenen Variablen durch.

Wie in 4.4.3.2 bereits erwähnt wurde, erfolgt in der Funktion `processAudio` der Klasse `TReaktor` ein Aufruf der Funktion `centralDecouplingRoutine`. Als Parameterwert wird dabei `PROCESSAUDIO` verwendet. In Zeile 9 wird daraufhin zunächst abgefragt, ob der Aufrufstack bereits entkoppelt wurde. Ist der Aufrufstack *nicht* entkoppelt, erfolgt die Umleitung in den separaten Stackspeicher. Dies wird durch das Vertauschen der aktuellen Werte der Stackregister mit den während der Initialisierung gespeicherten Werten in Zeile 15 realisiert. Der Stackframe der in Zeile 17 aufgerufenen Funktion `doProcessAudio`

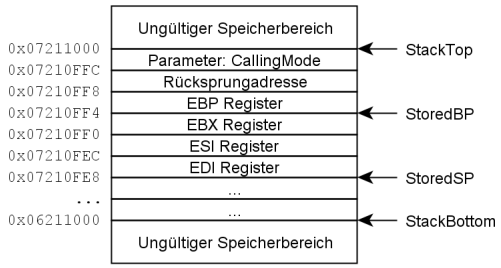

Abbildung 4.7 Separater Stackspeicher während der Initialisierung. Die Grenzen der verwendeten Speicherseite werden von den Zeigern *StackTop* und *StackBottom* markiert. Da alle Operationen für das Umschalten des Stacks in derselben Funktion implementiert sind, können die bei der Initialisierung gespeicherten Adressen von Base- und Stack-Pointer für jeden Umschaltvorgang verwendet werden.

wird daraufhin hinter dem bei der Initialisierung verwendeten Stackframe im separaten Stackspeicher erstellt. Die Ausführung erreicht den Prüflingscode während der Verarbeitung von `doProcessAudio`. Sollte dabei kein Breakpoint getroffen werden, wird die Funktion `doProcessAudio` regulär verlassen und deren Stackframe wieder abgebaut. Die Ausführung erreicht dann Zeile 19 von Listing 4.4. Die Werte der Stackregister des Prozessors entsprechen hier wieder den bei der Initialisierung gespeicherten Werten. Durch den Rücktausch in Zeile 19 wird also die Ausgangssituation wiederhergestellt. Die `return` Anweisung in Zeile 21 veranlasst den Abbau des Stackframes der Funktion `centralDecouplingRoutine` im regulären Stackspeicherbereich.

4.4.3.4 Entkopplung des Aufrufstacks

Wird bei der Verarbeitung des Prüflings ein Breakpoint getroffen, wird die Funktion `centralDecouplingRoutine` von der Behandlungsroutine der Klasse `SRS::RTDSegment` mit dem Parameterwert `BREAK` aufgerufen. Im Folgenden soll zunächst der Fall betrachtet werden, dass der Aufrufstack zu diesem Zeitpunkt *nicht* entkoppelt ist. Dann befindet sich die Ausführung im Echtzeit-Thread und im Prüflingscode wurde gerade zum ersten Mal ein Breakpoint getroffen. Der Wert des aktuellen Samples wird daraufhin in Zeile 25 durch den Aufruf der Funktion `doSilenceAudio` auf Null gesetzt. Viel wichtiger ist jedoch, dass die Variablen `StoredSP` und `StoredBP` in diesem Fall die Werte des Ausgangszustandes enthalten (die bei der Initialisierung gespeicherten Werte). Der Austausch der aktuellen Werte der Stackregister mit denen von `StoredSP` und `StoredBP` in Zeile 26 realisiert damit die Entkopplung des Aufrufstacks. Denn bei der Ausführung der `return` Anweisung in Zeile 28 wird nun nicht die Rücksprungadresse im Stackframe der Funktion `centralDecouplingRoutine` im *separaten* Stackspeicher verwendet, sondern die Rücksprungadresse im Stackframe der Funktion `centralDecouplingRoutine`

im *regulären* Stackspeicherbereich. Tatsächlich erfolgt hier also bereits der Rücksprung, der im Normalfall erst nach der Rückkehr aus der Funktion `doProcessAudio` in Zeile 21 erfolgt wäre (Pfeil (1) in Abbildung 4.6). Die Stackframes der dazwischen erfolgten Funktionsaufrufe verbleiben damit im separaten Stackspeicher. Die Ausführung im Echtzeit-Thread wird damit nur unwesentlich verzögert. Sie kehrt unter Einhaltung der vorgegeben Zeitschranke zurück.

In allen weiteren Verarbeitungsdurchläufen wird nun in Zeile 9 erkannt, dass der Aufrufstack entkoppelt wurde. Durch den Aufruf der Funktion `doSilenceAudio` in Zeile 10, wird dann der Wert des aktuellen Samples im Sample-Puffer auf Null gesetzt. Bei der Ausgabe des Samples wird später kein Ton erzeugt. Die Ausführung kehrt zurück ohne den Prüfling zu erreichen. Damit ist die Ausführung des Prüflings unterbrochen. Sein Ausführungszustand kann vom Benutzer in der Entwicklungsumgebung inspiziert werden.

4.4.3.5 Fortsetzung der Ausführung

Der Austausch der Werte der Stackregister mit den Werten der Variablen `StoredSP` und `StoredBP` in 4.4.3.4 diente nicht nur zur Entkopplung des Aufrufstacks, sondern ermöglicht ebenfalls die Fortsetzung der Ausführung an der Stelle der Unterbrechung. Denn die nun in den Variablen `StoredSP` und `StoredBP` gespeicherten Werte sind genau die Werte der Stackregister des letzten im separaten Stackspeicher aufgebauten Stackframes. Auch hier handelt es sich um einen Stackframe der Funktion `centralDecouplingRoutine`.

Wie in Abschnitt 3.3.3.3 des Konzeptionsteils vorgeschlagen wird, erfolgt die Fortsetzung der Ausführung des Prüflings durch einen Behelfs-Thread. Da die Fortsetzung der Ausführung durch eine Benutzerinteraktion initiiert wird, die im GUI-Thread von Reaktor behandelt wird, bietet sich die Verwendung des GUI-Threads als Behelfs-Thread an. Wie in allen ereignisbasierten Softwareanwendungen wartet der GUI-Thread in der Hauptereignisschleife der Anwendung auf das Eintreffen von Nachrichten.

Initiiert der Benutzer die Fortsetzung der Ausführung in der Entwicklungsumgebung, wird eine entsprechende Nachricht an die Hauptereignisschleife der Anwendung gesendet. Im GUI-Thread wird daraufhin die zugehörige Behandlungsroutine des Debuggers aufgerufen. Soll beispielsweise ein Einzelschritt ausgeführt werden, handelt es sich dabei um die Methode `doStep`. Innerhalb der Methode wird die Funktion `centralDecouplingRoutine` mit dem Parameterwert `CONTINUE` aufgerufen. Die Ausführung erreicht dann den Befehl zum erneuten Austausch der Stackregister in Zeile 31 von Listing 4.4. Damit wird der Stackframe der Funktion auf den letzten im separaten Stackspeicher aufgebauten Stack-

frame umgeschalten. Im Anschluss liegen die Werte der Stackregister, die den ursprünglichen Stackframe der Funktion im regulären Stackspeicherbereich des GUI-Threads markieren, in den Variablen `StoredSP` und `StoredBP`. Die Übernahme des Stackframes im separaten Stackspeicher ist unproblematisch, da er dieselbe Struktur aufweist wie der ursprüngliche Stackframe – es handelt sich in beiden Fällen um einen Stackframe der Funktion `centralDecouplingRoutine`. Die `return` Anweisung in Zeile 32 verwendet daraufhin die Rücksprungadresse des letzten Stackframes im separaten Stackspeicher. Tatsächlich erfolgt hier also der Rücksprung, der im Normalfall in Zeile 28 den Aufruf der Funktion mit dem Parameterwert `BREAK` beendet hätte (Pfeil (2) in Abbildung 4.6). Dieser Aufruf erfolgte im Anschluss an die Sicherung des Ausführungszustandes des Prüflings und der Wiederherstellung des ursprünglichen Maschinencodes durch die Behandlungsroutine des RTDSegment. Die Verarbeitung wird an dieser Stelle fortgesetzt und erreicht schließlich wieder die Funktion `onHitBreakpoint` im Memory-Backend (4.4.2.5).

4.4.3.6 Nachfolgende Unterbrechungen

Sollten bei der Verarbeitung des Prüflings durch den GUI-Thread weitere Breakpoints getroffen werden, kommt es erneut zur Vertauschung der Werte der Stackregister mit den Werten der Variablen `StoredSP` und `StoredBP`. Im Unterbrechungsfall (Pfeil (3) in Abbildung 4.6) enthalten die Variablen `StoredSP` und `StoredBP` stets die Werte, die sie bei der letzten Fortsetzung der Ausführung in 4.4.3.5 erhalten haben. Sie markieren den letzten Stackframe im regulären Stackspeicherbereich des GUI-Threads. Bei der Unterbrechung der Ausführung kehrt der GUI-Thread zur Hauptereignisschleife zurück und wartet dort auf das Eintreffen von Nachrichten. Im Fortsetzungsfall (Pfeil (2) in Abbildung 4.6) enthalten die Variablen `StoredSP` und `StoredBP` stets die Werte, die sie bei der letzten Unterbrechung der Ausführung in 4.4.3.4 erhalten haben. Die Ausführung des Prüflings kann also beliebig häufig unterbrochen und wieder fortgesetzt werden. Im Gegensatz zur ersten Unterbrechung der Ausführung ist der Aufrufstack bei nachfolgenden Unterbrechungen bereits entkoppelt. In Listing 4.4 wird dies durch die Abfrage der Statusvariablen `Status` ermittelt.

4.4.3.7 Ende der Verarbeitung des Prüflings

Wenn der Programmcode des Prüflings vollständig abgearbeitet wurde (Pfeil (4) in Abbildung 4.6), kehrt die Ausführung zum Aufruf der Funktion `doProcessAudio` in Zeile 17 der Funktion `centralDecouplingRoutine` zurück. Die Werte der Stackregister des Prozessors entsprechen nun wieder den bei der Initialisierung gespeicherten Werten.

Sie markieren also den ersten Stackframe des separaten Stackspeichers. Die Variablen `StoredSP` und `StoredBP` enthalten zu diesem Zeitpunkt dieselben Werte wie im Unterbrechungsfall in 4.4.3.6. In Zeile 19 erfolgt eine abschließende Umschaltoperation. Damit wird für die Variablen `StoredSP` und `StoredBP` der Ausgangszustand wiederhergestellt. Die `return` Anweisung in Zeile 21 verwendet die Rücksprungadresse des Stackframes der Funktion `centralDecouplingRoutine` im regulären Stackspeicherbereich des GUI-Threads. Die Ausführung kehrt damit zur jeweiligen Behandlungsroutine des Debuggers[15] und schließlich zur Hauptereignisschleife der Anwendung zurück.

Beim Abschluss der Verarbeitung des Prüflings ist die Statusvariable `Status` in Zeile 20 von Listing 4.4 auf `NOT_DECOUPLED` zurückgesetzt worden. Beim Berechnungsdurchlauf für das nächste Sample im Sample-Puffer erreicht die Ausführung im Echtzeit-Thread daraufhin wieder den Aufruf der Funktion `doProcessAudio` in Zeile 17. Damit wird die reguläre Echtzeitverarbeitung im Prüfling fortgesetzt.

4.4.3.8 Fazit

Die in diesem Abschnitt beschriebene Implementierung des Verfahrens zur Manipulation des Kontrollflusses ermöglicht die Unterbrechung der Echtzeitverarbeitung des Prüflings ohne eine Verletzung bestehender Zeitschranken. Es ermöglicht zudem die Fortsetzung der Ausführung durch den GUI-Thread von Reaktor. Die Implementierung ist weitgehend unabhängig von den technischen Details zum Aufbau von Stackframes und der Durchführung von Funktionsaufrufen. Diese Details werden weiterhin vom verwendeten C++ Compiler bestimmt. Allein die in Listing 4.4 verwendeten Mechanismen für die Sicherung und den Austausch der Stackregister müssen architekturspezifisch in Assembler implementiert werden. Die Implementierung ermöglicht außerdem die Kapselung der Funktionalität aller Manipulationsoperationen an einer zentralen Stelle im Quellcode.

Die Echtzeitverarbeitung wird regulär ausgeführt, bis im Prüflingscode zum ersten Mal ein Breakpoint getroffen wird. Die Werte aller nachfolgenden Samples werden auf Null gesetzt. Bei der Ausgabe dieser Samples wird kein Ton erzeugt. Dies entspricht dem vom Benutzer erwarteten Verhalten, wenn die Verarbeitung des Prüflings unterbrochen ist. Nach Abschluss des durch einen Breakpoint unterbrochenen Verarbeitungsdurchlaufs wird die Echtzeitverarbeitung automatisch wieder aufgenommen. Die Audioausgabe wird ab diesem Zeitpunkt regulär fortgesetzt.

[15] Im Falle eines Einzelschrittes, handelt es sich dabei wieder um die Methode *doStep*.

4.5 Ablauf des Single-Stepping

Für die Ausführung eines Reaktor Core Programms in Einzelschritten muss der Benutzer in der Entwicklungsumgebung zunächst in den Debug-Modus wechseln (siehe Abbildung 4.8). Daraufhin erfolgt die spezielle Kompilierung des Prüflings wie in Abschnitt 4.3.4 beschrieben automatisch. Die Echtzeitverarbeitung wird regulär fortgesetzt.

Die Ausführung des Prüflings in Einzelschritten wird vom Benutzer durch das Betätigen der Funktionstaste F12 initiiert. Der Debugger ermittelt daraufhin den oder die ersten auszuführenden Module im Audio-Handler der Core Cell wie folgt. Ausgehend vom ersten linearen Segment im Maschinencode des Prüflings wird eine Breitensuche über den in 4.3.3.2 aufgebauten Kontrollflussgraph des Prüflings gestartet[16]. Ziel der Breitensuche ist es, den oder die ersten auszuführenden Module der Core Cell zu finden. Es werden also sukzessiv solange alle Nachfolgesegmente durchlaufen, bis für jeden möglichen Ausführungspfad ein erstes Modul gefunden wurde. Besuchte lineare Segmente werden entsprechend markiert und später nicht erneut durchlaufen. Im Falle der Programmstruktur aus Beispiel 4.5 werden die linearen Segmente 0 bis 10 also in der folgenden Reihenfolge durchsucht: 0, 1, 7, 2, 6, 8, 10, 3, 4, 5, 9. Endet die Suche, ohne dass auf jedem möglichen Ausführungspfad ein Modul gefunden wird, werden im Folgenden nur die jeweils gefunden Module verwendet[17]. In diesem Fall kommt es möglicherweise erst nach einer Modifikation der Parametrisierung zu einer Unterbrechung der Ausführung (da andernfalls evtl. stets leere Ausführungspfade eingeschlagen werden).

Für alle im Rahmen der Breitensuche gefundenen Module wird im Anschluss ein temporärer Breakpoint im jeweils zuständigen RTDSegment registriert. Dafür wird die Zuordnung zwischen den Modulen der Core Cell und den Speicheradressen ihrer Maschinenbefehle im Prüflingscode verwendet, die zur Übersetzungszeit in den Programmdaten hinterlegt worden sind. Schließlich wird das Callback-Flag vom Debugger gesetzt. Bei der Verarbeitung des nächsten Samples wird dies am Callback-Point des Prüflings registriert. Die entsprechende Behandlungsroutine des Debuggers markiert den Speicherbereich des Prüflingscodes als editierbar, implementiert die gesetzten Breakpoints, markiert den Speicherbereich wieder als ausführbar und kehrt zum Prüfling zurück.

Zu diesem Zeitpunkt erfolgt die Ausführung des Prüflings noch im Echtzeit-Thread von Reaktor. Wird in einem der folgenden Verarbeitungsdurchläufe des Prüflings ein Breakpoint getroffen, wird der Ausführungszustand des Prüflings in der Funktion `onHitBreak`

[16] Beim ersten linearen Segment handelt es sich stets um den Prolog eines Handlers. Es ist zwar eindeutig, enthält aber keinen Maschinencode von Modulen der Core Cell.

[17] Es kann auch vorkommen, dass bei der Suche überhaupt kein Modul gefunden wird. Dies ist z. B. bei einer leeren Core Cell der Fall. Die Einzelschrittausführung wird dann bereits hier abgebrochen.

`point` gesichert. Dies wurde in Abschnitt 4.4.2.5 detailliert betrachtet. Außerdem kommt es zur Entkopplung des Aufrufstacks und zur Unterbrechung der Echtzeitverarbeitung. Hierfür kommt das Verfahren aus Abschnitt 4.4.3.4 zum Einsatz. Ab diesem Zeitpunkt erfolgt keine Audioausgabe mehr. Die Parametrisierung des Prüflings kann ebenfalls nicht mehr verändert werden. Nun wird der GUI-Thread benachrichtigt, die im folgenden Absatz beschriebenen Anpassungen der Benutzeroberfläche vorzunehmen. Dabei werden auch die für den Einzelschritt verwendeten temporären Breakpoints aus dem Maschinencode des Prüflings entfernt und erneut die zu Beginn beschriebene Breitensuche ausgeführt. In diesem Fall startet die Breitensuche am Punkt der Unterbrechung. Die Breakpoints für die gefundenen Module werden bereits jetzt registriert.

Der Benutzer kann nun den Wire-Debugging-Mechanismus der Entwicklungsumgebung benutzen, um den Ausführungszustand des Prüflings zu inspizieren. Dafür wird der Mauszeiger über einen beliebigen Verbindungsdraht der Core Cell bewegt. Es erscheint eine Anzeige, die den Wert darstellt, welcher zum Zeitpunkt der Unterbrechung an dieser Verbindung anliegt. Zudem werden die Module der Core Cell ihrem Ausführungszustand entsprechend eingefärbt. Module deren Maschinencode innerhalb des aktuellen Verarbeitungsdurchlaufs bis zum Punkt der Unterbrechung nicht ausgeführt wurde, werden ausgegraut dargestellt. Die weitere Farbwahl orientiert sich an den gängigen Farbkonventionen der verbreiteten Entwicklungsumgebungen. Das Modul an dem die Ausführung unterbrochen wurde, erscheint gelb eingefärbt (aktueller Ausführungspunkt). Die möglichen Folgemodule erscheinen rot eingefärbt (Breakpoints). Der oder die Maschinenbefehle des gelb eingefärbten Moduls sind die im folgenden Einzelschritt auszuführenden Befehle. Abbildung 4.8 zeigt die Core Cell eines Low-Pass Filters nach der Ausführung einiger Einzelschritte.

Betätigt der Benutzer die Taste F12 erneut, führt das zur Ausführung des nächsten Einzelschrittes. Die dafür zu verwendenden Breakpoints wurden bereits im Vorfeld der Unterbrechung ermittelt und müssen nun nur noch auf die in Abschnitt 4.4.2.3 beschriebene Weise implementiert werden. Im Anschluss wird die Ausführung des Prüflings vom GUI-Thread von Reaktor nach dem in 4.4.3.5 beschriebenen Verfahren fortgesetzt. Mit Ausnahme der Entkopplung des Aufrufstacks werden alle weiteren Einzelschritte analog hierzu realisiert.

Wird bei der Breitensuche kein Nachfolgemodul mehr gefunden, hat die Einzelschrittausführung das Ende der Core Cell erreicht. Es werden keine weiteren Breakpoints implementiert. Die Ausführung wird jedoch wie üblich mit F12 fortgesetzt. Das Verfahren zur Realisierung von Unterbrechungen wird dann, wie in Abschnitt 4.4.3.7 beschrieben, beendet und ist für den erneuten Einsatz bereit. Ab der Verarbeitung des nächsten Samples

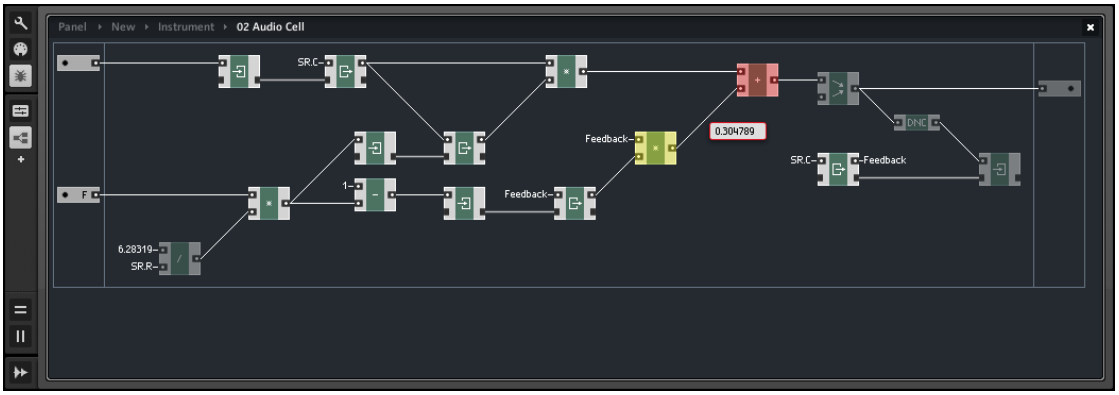

Abbildung 4.8 Ausführung eines Low-Pass Filters in Einzelschritten. Die Behandlung von Macros wurde bei der Implementierung des Prototypen ausgespart. Die Abbildung zeigt daher eine zu Abbildung 2.3 äquivalente Version eines Low-Pass Filters in der alle Macros in ihre Bestandteile zerlegt wurden. Der Debug-Modus wurde durch einen Klick auf den Knopf mit dem Symbol eines Käfers am linken Bildrand aktiviert.

wird die Echtzeitverarbeitung und damit auch die Audioausgabe wieder aufgenommen.

5 Ergebnisse

Das in Kapitel 3 erarbeitete Konzept eines integrierten, interaktiven Same-Process Debuggers wurde für den Spezialfall der domänenspezifischen Sprache Reaktor Core in Form eines Prototypen erfolgreich realisiert. Im Mittelpunkt stand die Realisierung der Ablaufkontrollmechanismen von Trace-Debuggern unter den speziellen Anforderungen der Echtzeitverarbeitung.

Zunächst wurden einige Arbeitsschritte zur Übersetzungszeit erforderlich. Neben der Aufzeichnung verschiedener Programmdaten wurde eine Kontrollflussanalyse sowie eine spezielle Kompilierung des Prüflingsprogramms nötig. Durch die Einteilung des Prüflingscodes in sequenzielle Befehlsblöcke wurde die Realisierung eines Breakpoint-Verfahrens bei minimaler Streuung des Maschinencodes im Speicher möglich. Damit kann die Ausführbarkeit des Prüflings in Echtzeit gewährleistet werden. Für die Aufzeichnung der Programmdaten kam ein Benachrichtigungsmechanismus zum Einsatz, der es ermöglicht die Verfahren für Analyse und strukturierte Speicherung der Programmdaten außerhalb des Compilers zu implementieren. Durch die Realisierung des Aufzeichnungsmoduls als Zustandsautomat konnten die nötigen Anpassungen am Quellcode des Compilers minimiert werden. Der für die Kontrollflussanalyse entworfene Algorithmus arbeitet parallel zum Übersetzungsvorgang. Damit ist weder eine zusätzliche Analysephase noch die Speicherung größerer Datenmengen erforderlich. Der Übersetzungsvorgang wird daher nur in geringem Maße verzögert.

Zur Laufzeit lag das Hauptaugenmerk auf der Manipulation des Prüflingscodes und der Realisierung von Unterbrechungen unter den speziellen Anforderungen der Echtzeitverarbeitung. Die Idee des Callback-Points im Prüfling ermöglicht das Einfügen von Breakpoints zur Laufzeit ohne die Verwendung blockierender Anweisungen. Die Unterbrechung der Ausführung des Prüflings wurde durch die Entkopplung des Aufrufstacks erreicht. Der Mechanismus ermöglicht eine Rückkehr der Ausführung im Echtzeit-Thread innerhalb der gebräuchlichen Zeitschranken. Eine wichtige Voraussetzung für Debugger – die Möglichkeit Fehlerwirkungen zu erkennen – wird dadurch gewährleistet. Denn der Prüfling kann auch im Debug-Modus wie gewohnt in Echtzeit ausgeführt werden, ohne dass bei einer Unterbrechung der Ausführung die Gefahr einer Verletzung von Echtzeitbedingungen besteht. Die Ausführung kann zudem durch einen Behelfs-Thread beliebig

häufig fortgesetzt und erneut unterbrochen werden. Die Implementierung konnte weitgehend unabhängig von den technischen Details der zugrundeliegenden Plattform erfolgen.

Die in Kapitel 4 beschriebenen Implementierungen wurden in Version 5.7 von Reaktor vorgenommen. Des Weiteren wurden verschiedene Funktionalitäten wie die Anbindung der Benutzeroberfläche implementiert, die im Rahmen dieser Arbeit nicht vorgestellt werden konnten. Die Zielstellung bei der Implementierung des Prototypen war die Realisierung eines Einzelschrittverfahrens. Unter Beachtung einiger Einschränkungen, die bereits im Vorfeld bekannt waren, ist dies gelungen. Die vorrangige Einschränkung bildet die mangelnde Einbeziehung der hierarchischen Struktur von Core Cells. Die Ursache dafür liegt in der Auflösung dieser Struktur durch den Compiler in einer frühen Phase des Übersetzungsvorgangs. Im Rahmen der Entwicklung des Prototypen war dies akzeptabel.

Aufgrund der einfachen Struktur von Reaktor Core Programmen konnten an verschiedenen Stellen der Konzeption Vereinfachungen vorgenommen werden. Dies trug dazu bei, dass die Implementierung der Ablaufkontrollmechanismen mit einem vertretbaren Aufwand möglich war. Im betrachteten Umfeld ist eine vergleichsweise einfache Programmstruktur häufig zu finden. Dies gilt für domänenspezifische Sprachen im Allgemeinen und für Sprachen der digitalen Signalverarbeitung im Besonderen. Die Konzeption setzt diese Einschränkungen jedoch nicht voraus. Ist ein entsprechend höherer Implementierungsaufwand akzeptabel, ist eine Verwendung auch in mächtigeren strukturierten Programmiersprachen denkbar.

Zusammenfassend lässt sich erkennen, dass sich der entwickelte Prototyp noch nicht für das Auffinden von Programmdefekten im Produktiveinsatz eignet. Er kann jedoch bereits jetzt bei der Einarbeitung in die Programmierung mit Reaktor Core behilflich sein. Die wesentlichen Problemstellungen konnten im Rahmen dieser Arbeit gelöst und im Prototypen implementiert werden. Dies bildet eine gute Grundlage für die zukünftige Weiterentwicklung.

Glossar

Echtzeit-Thread	Vom Scheduler des Betriebssystems speziell priorisierter Thread
Enkopplungspunkt	Funktion in der Ausführungsabfolge des Echtzeit-Threads, an der die Umleitung des Aufrufstacks stattfindet
Implementieren eines Breakpoints	Überschreiben eines Befehls im Maschinencode des Prüflings durch den Befehlscode des Breakpoints. Der ursprüngliche Befehlscode wird zuvor zwischengespeichert.
Instruction-Level	Programmcode auf Maschinenebene
Lineares Segment	Sequenzielle Folge von Maschinenbefehlen ohne Sprunganweisungen. Nur der erste Maschinenbefehl eines linearen Segments kann ein Sprungziel sein.
Programmdaten	Informationen zum Prüflingsprogramm, die eine Inspektion des Prüflings auf Quellcodeebene ermöglichen. Sie werden i. A. vom Compiler gesammelt und nach Abschluss des Übersetzungsvorgangs dem Debugger zur Verfügung gestellt.
Regulärer Stackspeicherbereich	Speicherbereich des Aufrufstacks eines Threads im Stack Segment des Prozesses

Separater Stackspeicher	Speziell allozierter Speicherbereich für die Aufnahme der Stackframes aller nach der Umleitung des Aufrufstacks aufgerufenen Funktionen (im Heap Segment des Prozesses)
Sequenzieller Befehlsblock	Sequenzielle Folge von Maschinenbefehlen, die an einem Sprungziel beginnt und alle Maschinenbefehle bis zum nächsten Sprungziel enthält. Gegenüber den Grundblöcken im Compilerbau können sequenzielle Befehlsblöcke Sprunganweisungen an beliebigen Stellen enthalten.
Setzen eines Breakpoints	Registrierung eines Breakpoints gegenüber dem Debugger
Source-Level	Programmcode auf dem Abstraktionsniveau einer höheren Programmiersprache
Umleitung des Aufrufstacks	Manipulaion der Stackregister des Prozessors. Die Stackframes aller nachfolgend aufgerufenen Funktionen werden in einem separaten Stackspeicher aufgebaut
Zeitschranke	Fest vorgegebener Zeitpunkt an dem die Ergebnisse einer Berechnung vorliegen müssen und die Ausführung im Echtzeit-Thread zurückgekehrt sein muss

Literaturverzeichnis

[ANCS00] Alpern, B., T. Ngo, J.D. Choi und M. Sridharan: *DejaVu: deterministic Java replay debugger for Jalapeño Java virtual machine.* Proceedings of OOPSLA 2000–Addendum, Seiten 165–166, 2000. 17

[App12] Apple Inc.: *Mac Developer Library, Kernel Programming Guide: Mach Scheduling and Thread Interfaces*, Februar 2012. `https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html#//apple_ref/doc/uid/TP30000905-CH211-TPXREF108`. 46

[ASL+08] Aho, A.V., R. Sethi, M.S. Lam *et al.*: *Compiler: Prinzipien, Techniken und Werkzeuge.* Pearson Deutschland GmbH, 2008. 40, 66

[Ben11] Bencina, Ross: *Real-time audio programming 101: time waits for nothing*, August 2011. `http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing`. 28

[BJ98] Bristow-Johnson, Robert: *Comp.realtime: Frequently Asked Questions (Version 3.5)*, Juli 1998. `http://www.faqs.org/faqs/realtime-computing/faq/`. 27

[CE00] Czarnecki, Krzysztof und Ulrich W. Eisenecker: *Generative programming: methods, tools and applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000, ISBN 0-201-30977-7. 11

[CFMP04] Costabile, Maria Francesca, Daniela Fogli, Piero Mussio und Antonio Piccinno: *Software environments for end-user development and tailoring.* Psychnology, 2:99–122, 2004. 14, 15

[CS03] Claus, Volker und Andreas Schwill: *Duden. Informatik. Ein Fachlexikon für Studium und Praxis.* Bibliographisches Institut, Mannheim, 2003. 28

[DD87] Date, C.J. und H. Darwen: *A Guide to the SQL Standard*, Band 3. Addison-Wesley Reading (Ma) et al., 1987. 12

[Don10] Donohue, Aran: *Debugging Domain-Specific Languages.* Diplomarbeit, University of Toronto, 2010. 13

[Dun88] Duncan, R.: *Advanced MS-DOS Programming.* Microsoft Press, 1988. 23

[Edw05] Edwards, J.: *Subtext: uncovering the simplicity of programming.* In: *ACM SIGPLAN Notices*, Band 40, Seiten 505–518. ACM, 2005. 13

[Fow05a] Fowler, Martin: *Language Workbenches: The Killer-App for Domain Specific Languages?*, Juni 2005. `http://martinfowler.com/articles/languageWorkbench.html`. 11, 13, 14

[Fow05b] Fowler, Martin: *Using the Rake Build Language*, August 2005. `http://martinfowler.com/articles/rake.html`. 12

[GHVJ09] Gamma, E., R. Helm, J. Vlissides und R. Johnson: *Entwurfsmuster.* Pearson Deutschland GmbH, 2009. 62

[Hag01] Hagenow, Henri: *Digitale Synthese komplexer Wellenformen zur Simulation akustischer, elektrischer und optischer Eigenzustände mehrdimensionaler Systeme.* Diplomarbeit, Optisches Institut der Technischen Universität Berlin, 2001. `http://www.brothers-in-music.de/klangsynthese/Klangsynthese&PhysicalModeling_HH.pdf`. 29

[Her11] Herrmanns, C.: *Entwicklung und Implementierung eines hybriden Debuggers für Java.* Wissenschaftliche Schriften der WWU Münster / 4. Westfälische Wilhelms-Universität, 2011. 7, 8, 17, 18

[Hug04] Hughes, T.P.: *American genesis: a century of invention and technological enthusiasm, 1870-1970.* University of Chicago Press, 2004. 7

[Int12a] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference*, 043. Auflage, Mai 2012. 39, 44, 73, 78

[Int12b] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, 043. Auflage, Mai 2012. 20, 21

[JG04] Jeff Gray, Hui Wu, Marjan Mernik: *Debugging Domain-Specific Languages in Eclipse*. Technischer Bericht, University of Alabama at Birmingham, University of Maribor, 2004. 13

[Kep93] Keppel, D.: *Fast Data Breakpoints*. Technischer Bericht TR-93-04-06, University of Washington, 1993. 25

[Kes90] Kessler, Peter B.: *Fast breakpoints: design and implementation*. SIGPLAN Not., 25(6):78–84, Juni 1990, ISSN 0362-1340. 23, 24, 36

[KOM+10] Kosar, Tomaž, Nuno Oliveira, Marjan Mernik, Maria J. V. Pereira, Matej Črepinšek, Daniela da Cruz und Pedro R. Henriques: *Comparing General-Purpose and Domain-Specific Languages: An Empirical Study*. Computer Science and Information Systems, 7(2):247–264, Mai 2010. 11

[KPKP06] Kolovos, Dimitrios S., Richard F. Paige, Tim Kelly und Fiona A. C. Polack: *Requirements for Domain-Specific Languages*. In: *Proc. 1st ECOOP Workshop on Domain-Specific Program Development (DSPD 2006)*, Nantes, France, Juli 2006. 13

[Lab12] LabVIEW, National Instruments Webseite, Dezember 2012. `http://www.ni.com/labview/d/`. 9, 13, 29

[Lew03] Lewis, Bil: *Debugging Backwards in Time*. CoRR, cs.SE/0310016, 2003. 17

[Max12] Max/MSP, Cycling'74 Webseite, Dezember 2012. `http://cycling74.com/`. 9, 27, 29

[McC02] McCartney, J.: *Rethinking the computer music language: SuperCollider*. Computer Music Journal, 26(4):61–68, 2002. 27

[MHS05] Mernik, Marjan, Jan Heering und Anthony M. Sloane: *When and how to develop domain-specific languages*. ACM Comput. Surv., 37(4):316–344, Dezember 2005, ISSN 0360-0300. 12, 14

[Mic12a] Microsoft: *Dokumentation zur Verwendung der Funktion _aligned_malloc der Windows API*, Oktober 2012. `http://msdn.microsoft.com/en-us/library/8z34s9c6%28VS.80%29.aspx`. 78

[Mic12b] Microsoft: *Dokumentation zur Verwendung der Funktionen VirtualAlloc und VirtualProtect der Windows API*, Oktober 2012.

http://msdn.microsoft.com/en-us/library/windows/desktop/aa366887%28v=vs.85%29.aspx. 60

[MRB98] Martin, R.C., D. Riehle und F. Buschmann: *Pattern languages of program design 3.* Addison-Wesley, 1998. 62

[Nat10] Native Instruments: *Reaktor 5 Core Reference*, 2010. 30, 32

[PA90] Paxson, Vern und Prof Anderson: *A Survey of Support For Implementing Debuggers.* Technischer Bericht, Lawrence Berkeley Laboratory, Berkeley CA, 1990. 10, 23

[Rea12] Reaktor, Native Instruments Webseite, Dezember 2012. http://www.native-instruments.com/#/de/products/producer/reaktor-5/. 9, 28

[Ros96] Rosenberg, Jonathan B.: *How debuggers work: algorithms, data structures, and architecture.* John Wiley & Sons, Inc., New York, NY, USA, 1996, ISBN 0-471-14966-7. 16, 19

[SCC06] Simonyi, Charles, Magnus Christerson und Shane Clifford: *Intentional software.* In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, Seiten 451–464, New York, NY, USA, 2006. ACM, ISBN 1-59593-348-4. 14

[Sha82] Shapiro, E.Y.: *Algorithmic program debugging.* Dissertation Abstracts International Part B: Science and Engineering, 43(5):1982, 1982. 18

[Sim05] Simonyi, Charles: *Notations and Programming Languages*, May 2005. http://www.intentsoft.com/notations_and_p/. 12

[Sim12] Simulink, MathWorks Webseite, Dezember 2012. http://www.mathworks.de/products/simulink/. 13

[SPA92] SPARC International: *The SPARC Architecture Manual*, version 8 Auflage, 1992. 25

[Spi01] Spinellis, Diomidis: *Notable design patterns for domain-specific languages.* J. Syst. Softw., 56(1):91–99, Februar 2001, ISSN 0164-1212. 11, 13

[Ste06] Steiner, Hans Christoph: *bang pure data*, Kapitel Building your own instrument with Pd, Seiten 72–79. Wolke Verlag, 2006. `http://puredata.info/groups/pd-graz/label/book01`. 27, 29

[TA95] Tolmach, A. und A.W. Appel: *A debugger for Standard ML.* Journal of Functional Programming, 5(02):155–200, 1995. 17

[vDKV00] Deursen, Arie van, Paul Klint und Joost Visser: *Domain-specific languages: an annotated bibliography.* SIGPLAN Not., 35(6):26–36, Juni 2000, ISSN 0362-1340. 11

[vE01] Engelen, Robert A. van: *ATMOL: A domain-specific language for atmospheric modeling.* Journal of Computing and Information Technology (CIT), 9(4):289–303, 2001, ISSN 1330-1136. 13

[VS10] Voelter, M. und K. Solomatov: *Language modularization and composition with projectional language workbenches illustrated with MPS.* Software Language Engineering, SLE, 2010. 14

[War94] Ward, M.P.: *Language-oriented programming.* Software - Concepts and Tools, 15(4):147–161, 1994. 14

[Zav12] Zavalishin, Vadim: *The Art of VA Filter Design.* Native Instruments, 2012. `http://images-l3.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_1.0.3.pdf`. 32