technische universität
dortmund

Bachelor-Arbeit

Unterstützung modularer WCET-Analyse durch annotierte Binärobjekte

Christian Günter
September 2013

Gutachter:
Prof. Dr. Peter Marwedel
Dipl.-Inform. Timon Kelter

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl XII
http://ls12-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Eingebettete Systeme, die heute weit verbreitet sind und in ihrer Bedeutung immer weiter zunehmen, sind meist Echtzeitsysteme, in denen es notwendig ist, Echtzeitbedingungen garantiert einhalten zu können [18]. Eine Echtzeitbedingung legt dabei fest, in welchem Zeitraum oder bis zu welchem Zeitpunkt die Ausführung eines Prozesses oder einer Aktion abgeschlossen sein muss. Dabei unterscheidet man zwischen weichen und harten Bedingungen: Bedingungen, deren Nicht-Einhaltung keine gravierenden Folgen hätte, nennt man weich, während bei harten Bedingungen die Nicht-Einhaltung katastrophale Folgen hätte und je nach System zu erheblichen Schäden führen kann.

Daher ist es bei einem Programm, das in einem Echtzeitsystem laufen soll, erforderlich, entsprechende Garantien für die Ausführungszeiten geben zu können. Typischerweise gibt man dazu eine obere Schranke für die maximale Laufzeit (*Worst-Case Execution Time*, kurz WCET) an, da die *maximale* Laufzeit bestimmt, ob eine Bedingung eingehalten werden kann und obere Schranken leichter zu finden sind als ein genauer Wert. Dabei sollte eine solche Schranke aus Gründen der Effizienz so nah wie möglich am genauen Wert liegen, denn bei einer zu großzügig gewählten Schranke kann man eine tatsächlich eingehaltene Bedingung nicht garantieren und würde grundlos das Programm optimieren oder die Leistung des Systems erhöhen. In diesem Zusammenhang ist es daher auch von Vorteil, wenn ein Compiler als Strategie verfolgt, vor allem die WCET so klein wie möglich zu halten.

Zu diesem Zweck wurde der Compiler Wcc [12] entwickelt, der aus einem mit Informationen über den Kontrollfluss – Flowfacts [21] – annotierten Quelltext ein ausführbares Programm erzeugt, das hinsichtlich seiner WCET optimiert ist und für dessen WCET er zumindest eine möglichst kleine obere Schranke angeben kann. Wcc unterstützt zum gegenwärtigen Zeitpunkt zwei Architekturen: ARM und TriCore. Die ARM-Variante, die Gegenstand dieser Arbeit ist, benutzt dabei mit GNU Gcc [3] einen anderen Compiler um aus einem bereits optimierten Quelltext eine maschinennahe Zwischendarstellung zu gewinnen, und analysiert das Ergebnis bzgl. der WCET mittels aiT [13], einer Software zur Vorhersage der Worst-Case Execution Time durch statische Programmanalyse. Dabei liegt

es in der Verantwortung des Programmierers, den Kontrollfluss im Quelltext durch Pragmas zu annotieren, die für die Analyse notwendige Aussagen über Ausführungshäufigkeiten und -abhängigkeiten machen.

Ganz allgemein ist es sinnvoll und notwendig, dass ein Compiler nicht nur Hochsprachen-Quelltexte verarbeiten kann, sondern auch bereits übersetzte Quellen zu einem Programm zusammenfügen kann – mit anderen Worten, die hier allgemein als Compiler bezeichnete Software beinhaltet auch einen Binder. Quellen liegen in bereits übersetzter Form als externe, binäre Objektdateien vor, wenn in einem Übersetzungslauf einzelne Module des Programms unabhängig übersetzt werden, wenn die Quellen in Assembler statt in einer Hochsprache geschrieben wurden oder wenn aus urheberrechtlichen Gründen der Quelltext gar nicht verfügbar ist.

Bisher wurde der gesamte Quelltext durch den WCC in einem einzigen Lauf übersetzt, wobei externe Objektdateien ausgeklammert wurden. Um Funktionen aus solchen Objekten trotzdem einbinden zu können, wurden sie erst nach dem Übersetzungslauf mit dem Kompilat gebunden. Anschließend wurde das fertige ausführbare Programm durch aiT analysiert. aiT erhält dabei alle aus dem Quelltext vorhandenen Annotationen des Kontrollflusses und ist so in der Lage, die WCET des als ausführbarem Programm vorliegenden Kompilats abzuschätzen. Musste in das Programm aber eine externe Objektdatei eingebunden werden, dann liegen für die daraus stammenden Funktionen keine Annotationen vor, da diese schwer festzustellen sind, d. h., deren einzige verlässliche obere Schranke ist ∞. Diese Schranke würde aber das gesamte Programm dominieren und es wäre keine sinnvolle Aussage über die WCET möglich. Daher wird für Funktionen aus externen Objektdateien pragmatisch der Wert 0 angenommen, wodurch die durch aiT ermittelte obere Schranke aber unter Umständen kleiner ist als die tatsächliche WCET des Programms. Dies ist zwar zulässig, da die WCETs in WCC nur als Optimierungskriterium genutzt werden. Für eine abschließende Bewertung muss dann aber wieder aiT unabhängig von WCC genutzt werden.

Deshalb nur mit vollständig im Quelltext vorliegenden Programmen zu arbeiten ist nicht praktikabel, da bereits der für die Code-Selection verwendete Compiler GCC eine Reihe von binär vorliegenden Bibliotheken einsetzt, beispielsweise für nicht in der Hardware der CPU vorhandene Funktionalitäten wie die Division oder Fließkomma-Operationen. Daraus ergibt sich die Notwendigkeit, auch externe Objektdateien annotieren zu können und diese Annotationen analog zu den Pragmas im C-Quelltext dauerhaft hinterlegen zu können.

1.1 Ziele

Übergeordnetes Ziel dieser Arbeit ist es, das Annotieren von externen binären Objektdateien zu ermöglichen, sodass der WCC ein erzeugtes Programm vollständig analysieren

und ggf. optimieren kann. Für die Analyse ist es notwendig, dass Kontrollflussannotationen sowohl erstellt als auch an aiT übergeben werden. Damit eine Objektdatei nicht für jeden Übersetzungslauf erneut annotiert werden muss, müssen die Annotationen dauerhaft hinterlegt werden. Um außerdem das gesamte Programm optimieren zu können, muss aus einer Objektdatei die Zwischendarstellung von WCC zurückgewonnen werden können. Daher bietet es sich an, ganz allgemein Objektdateien in die annotierte Zwischendarstellung zu überführen, um sie dann mit der ebenfalls annotierten Zwischendarstellung des Quelltextes zu einem vollständig annotierten Programm zu binden, das von aiT analysiert werden kann. In Teilschritte zerlegt, ergeben sich daraus folgende Ziele:

- Aus einer Objektdatei muss die Zwischendarstellung von WCC zurückgewonnen werden, die sowohl die nur noch im Maschinencode vorliegenden Funktionen als auch die dazugehörigen Daten umfasst.

- Diese Zwischendarstellung sollte, soweit es möglich ist, automatisch vervollständigt werden. Der Benutzer sollte nur wenige zusätzliche Informationen liefern müssen, wie beispielsweise die Sprungziele indirekter Sprünge, die sich nicht (ohne eine Datenflussanalyse) auflösen lassen.

- Es muss ein Editor entwickelt werden, mit dem ein Kontrollflussgraph mit Flowfacts annotiert werden kann, analog zu den Pragmas im Quelltext des Programms. Dabei muss der Editor die Annotationen in eine Form bringen können, in der sie einmal hinterlegt später wieder den Knoten eines Kontrollflussgraphen zugeordnet werden können. Auch andere Annotierungen wie die nicht auflösbaren Ziele indirekter Sprünge sollten damit durchgeführt werden können, um dem Benutzer die Möglichkeit zu geben, fehlende Informationen selbst beizusteuern.

- Die Annotationen sollen dauerhaft in den Objektdateien selbst hinterlegt werden, da so nicht nur der Programmcode und die Annotationen praktischerweise in ein und derselben Datei enthalten sind, sondern auch die Möglichkeit geschaffen wird, ein Programm leicht dateiweise, d. h. in Form von einzelnen Modulen zu übersetzen. Anzumerken ist hierbei aber, dass dabei manche Flowfacts gröber als nötig sein müssen, da keine allgemeingültige Aussage über Einschränkungen der möglichen Funktionsparameter gemacht werden kann – wird eine Funktion nur mit einer Teilmenge der zulässigen Parameterwerte aufgerufen, könnte beispielsweise die Anzahl der Durchläufe einer Schleife abnehmen oder bestimmte lange Ausführungspfade werden nie genommen.

- Da Flowfacts in einer Objektdatei hinterlegt werden können, sollte der WCC dies auch beim Erstellen von Objektdateien tun, um tatsächlich dateiweise übersetzen zu können.

1.2 Verwandte Arbeiten

Der in der Einleitung bereits erwähnte aiT Worst-Case Execution Time Analyzer [13] verarbeitet immer binäre Objektdateien. Dazu ist die Software ebenfalls in der Lage, eine Objektdatei zu disassemblieren. Anhand von beigesteuerten Flowfacts führt sie dann eine sinnvolle Abschätzung der WCET durch. aiT analysiert dabei aber nur vorhandene Objektdateien, d. h., die Software enthält keinen Compiler und macht ihre disassemblierte Darstellung des Programms nur eingeschränkt zugänglich. Außerdem liest die Software die Flowfacts immer aus Textdateien, die neben den binären Objektdateien erstellt und verwaltet werden müssen.

Neben aiT wurde auch ein anderes Werkzeug zur Bestimmung der WCET eines Programms betrachtet. Bound-T [23] kann ebenfalls ein als Binärobjekt vorliegendes Programm auswerten und die WCET bestimmen, und unterstützt dabei unter anderem auch ARM7. In der Dokumentation von Bound-T für ARM sind einige Details der Implementierung veröffentlicht, die in dieser Arbeit übernommen wurden.

Ein weiteres verwandtes Werkzeug, das Boomerang-Projekt [2], verfolgt das Ziel, einen vollständigen Decompiler zu erstellen, der aus einem binär in Maschinensprache vorliegenden Programm einen Hochsprachen-Quelltext gewinnt. Dieser Quelltext entspricht zwar nicht dem Ursprünglichen, würde aber durch Kompilieren zu einem äquivalenten Programm führen. Die Zielsetzung des Projekts erfordert ebenfalls die Auswertung von binären Objektdateien, um durch Disassemblieren die Kontrollstrukturen des Programms zu ermitteln. Dabei wird aber nicht die ARM-Prozessorfamilie unterstützt.

Die GNU Binutils [4] sind eine Sammlung von Werkzeugen, mit denen binäre Objektdateien in verschiedenen Formaten und für verschiedene Architekturen gesichtet und verarbeitet werden können. Die Sammlung unterstützt auch die ARM-Prozessorfamilie und enthält unter anderem einen Disassembler mit Textausgabe und eine eigene, nicht unabhängig veröffentlichte Bibliothek namens Libbfd, um Objektdateien zugänglich zu machen. Auf eine Reihe dieser Werkzeuge wurde bei der Erstellung dieser Arbeit zurückgegriffen.

In der Bachelor-Arbeit „Untersuchung des Code-Location-Problems in C-/C++-Programmen für eingebettete x86- und ARM-Architekturen“ [11] wird von einem Compiler erzeugter Assembler-Code für ARM- und x86-Prozessoren untersucht, der aus binären Objektdateien ausgelesen wird. Dabei wird der Disassembler aus den Binutils eingesetzt, der auch zur Grundlage dieser Arbeit gehört. Das Hauptaugenmerk liegt aber auf der Zuordnung des Assembler-Codes zum Quelltext, wozu die in den Kompilaten enthaltenen Debug-Informationen verwendet werden müssen. Im Gegensatz dazu wird bei dieser Arbeit ausdrücklich auf die Debug-Informationen verzichtet, da sie in den betroffenen Binärobjekten in der Regel nicht mehr enthalten sind, und es musste auf Basis des Binutils-Disassemblers ein eigener Disassembler entwickelt werden, um den Maschinencode aus binären Objekten in die von Wcc benötigte Form zu übersetzen.

Details zum benötigten Grundlagenwissen, die nicht fachspezifisches Allgemeinwissen sind, stammen im Wesentlichen aus zwei Lehrbüchern, deren Inhalte sich mit dem Thema dieser Arbeit überschneiden. In „Rechnerorganisation und -entwurf“ [19] wird detailliert dargelegt, worum es sich bei Assembler und Maschinensprache handelt, aber auch gezeigt, wie man aus einem Maschinenbefehl wieder einen Assembler-Befehl erhält und was der Zweck des Bindens von Objekten ist. „Linkers and Loaders“ [17] beschreibt – wie der Titel bereits aussagt – alle Aspekte des Bindens und Ladens, die auch bei der Auswertung von Programmcode aus einem binären Objekt von Bedeutung sind, da so wichtige Informationen über die Struktur des Programms gewonnen werden können.

1.3 Aufbau der Arbeit

Um das gesetzte Ziel zu erreichen, sind grundlegend betrachtet zwei Problemstellungen zu lösen. Zuerst muss eine binäre Objektdatei in LLIR-Darstellung überführt werden, und dann muss diese LLIR-Darstellung mit Flowfacts annotiert werden, um sie analysieren zu können. Die Kapitel 3 und 4 widmen sich der ersten Aufgabe, Kapitel 5 der zweiten. Kapitel 6 beschreibt dann, wie das Ergebnis überprüft und die hinzugefügte Software in das bestehende Framework eingebunden wurde. In Kapitel 7 findet eine abschließende Zusammenfassung statt.

Kapitel 2

Grundlagen

2.1 Einordnung in den WCC

Wie bereits in der Einleitung erwähnt, konnte der WCC-Compiler bisher beim Optimieren eines Programms nur auf die im Quelltext vorliegenden Module zugreifen. Dazu werden die Kontrollstrukturen eines in der Hochsprache C neu geschriebenen oder bereits vorliegenden Programms durch C-Pragmas mit Flowfacts annotiert. WCC liest dieses Programm ein, übersetzt es unter anderem in die Zwischendarstellung LLIR [12] („Low-Level Intermediate Representation“) und erzeugt daraus ein Assembler-Programm. Dabei wird das Programm auf verschiedenen Darstellungsebenen – auch in der LLIR-Darstellung – von WCC optimiert.

Dieser Vorgang und die Erweiterung für binäre Objektdateien werden im Flussdiagramm in Abbildung 2.1 gezeigt. Angefangen beim annotierten Quelltext „ANSI-C & Flow-Facts“, wird vom ICD-C Parser dieser Quelltext in die „High-Level ICD-C“-Darstellung übertragen und diese Darstellung des Programms wird unter Verwendung der Flowfacts hinsichtlich ihrer WCET optimiert. Anschließend wird das Programm durch den LLIR-Code Selector in die Low-Level LLIR als weitere, hardwarenahe Zwischendarstellung überführt. Im Fall der ARM-Architektur besteht der LLIR-Code Selector dabei aus dem GNU GCC-Compiler, der die ICD-C-Darstellung in ein Assembler-Programm übersetzt, und einem Programm zum Parsen der Ausgabe von GCC, das aus dem Assembler-Code die LLIR-Darstellung erzeugt. Der WCC-interne Flowfact-Manager transformiert dabei die C-Flowfacts auf die LLIR-Ebene.

Da es sich bei der LLIR-Darstellung um eine maschinenlesbare Form von ARM-Assembler handelt, die ein Programm in Form von Funktionen und deren Kontrollflussgraphen enthält, und binäre Objektdateien mit Maschinencode (und Daten) im Großen und Ganzen ebenfalls nicht weit von Assembler entfernt sind, setzt an dieser Stelle die Erweiterung von WCC an, die Gegenstand dieser Arbeit ist. Binäre Objektdateien („Binary Objects“) werden vom Parser/Disassembler gelesen und ebenfalls in die LLIR-Darstellung überführt.

Dazu wird vorher ihr Kontrollfluss durch den FlowFact-Editor genauso mit Flowfacts annotiert wie der Kontrollfluss des C-Quelltextes. Zwar ist Assembler-Code deutlich schwerer zu verstehen als C-Quelltext, aber dem Benutzer steht hier auch ein Kontrollflussgraph zur Verfügung.

Abschließend wird die LLIR-Darstellung des Programms wieder unter Verwendung der Flowfacts hinsichtlich ihrer WCET optimiert und der Code Generator erzeugt aus der LLIR-Darstellung ein Assembler-Programm („WCET-optimized Assembly & FlowFacts“), aus dem eine neue, ggf. ausführbare binäre Objektdatei erzeugt werden kann. An dieser Stelle ist außerdem vorgesehen, die Flowfacts aus der LLIR-Darstellung genauso in der Objektdatei abzulegen wie es der Flowfact-Editor tut, um einen modularisierten Übersetzungslauf mit WCC zu ermöglichen.

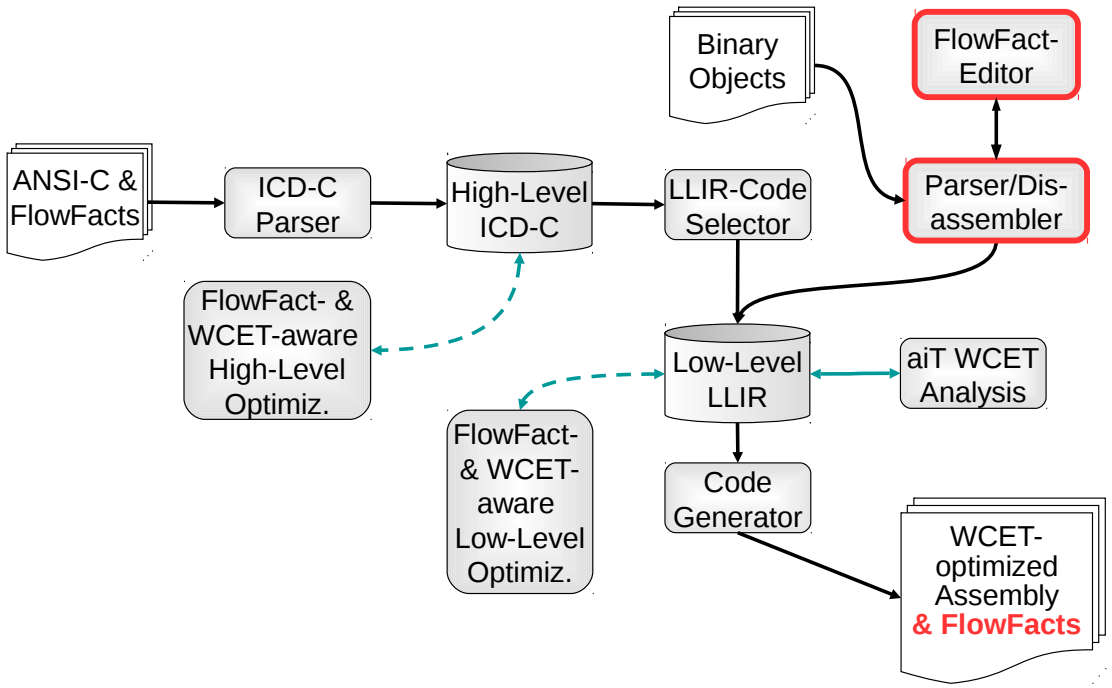

Abbildung 2.1: Einordnung in den WCC

2.2 Binäre Objektdateien

Um in einem System ausführbare, d. h. in der binären, für die Ausführung durch den Prozessor gestalteten Form vorliegende Programme und Funktionen dauerhaft bereithalten zu können, müssen Programmcode und Programmdaten in dazu geeigneten Formaten im System gespeichert sein. Für diesen Zweck werden binäre Objektdateien verwendet, die je nach Format sowohl vollständige, „fertige“ Programme als auch Bibliotheken von Funktionen enthalten können. Das dazu verwendete Dateiformat muss eine Reihe von Aspekten

berücksichtigen: Es müssen Informationen über das Programm vorgehalten werden (für welche Architektur und für welchen Prozessor es erstellt wurde), das Programm und seine Daten müssen in einer für die Speicherverwaltung des Systems geeigneten Form vorliegen und es sollte auch die Möglichkeit bestehen, beliebige weitere Daten speichern zu können, um beispielsweise Debug-Informationen hinzuzufügen. Es handelt sich also in aller Regel um ein Containerformat. Außerdem richtet sich das Format auch nach dem Einsatzzweck der Objektdatei – Programm oder Bibliothek –, wobei manche Formate wie das ELF-Format, das im Folgenden näher beschrieben wird, auch mehrere Einsatzzwecke gleichzeitig unterstützen.

Die Objektdatei eines vollständigen Programms muss neben dem Maschinencode des Programms und seinen Daten auch alle Informationen enthalten, die benötigt werden, um einen lauffähigen Prozess erzeugen und starten zu können. Der Maschinencode und die Daten müssen dazu vorbereitet sein, an eine geeignete Stelle im Arbeitsspeicher geladen zu werden, wobei ggf. im Programm enthaltene absolute Speicheradressen an die Ladeadresse angepasst werden müssen. Dieser Vorgang wird von einem sogenannten Lader (englisch „Loader“) durchgeführt und ist nicht Gegenstand dieser Arbeit.

Eine Objektdatei, die als Bibliothek Funktionen enthält, kann auf zwei Arten genutzt werden. Bei einer dynamischen Bibliothek wird die Objektdatei vom Lader eingelesen wenn ein Programm ausgeführt werden soll, das in der Bibliothek enthaltene Funktionen benötigt – auf diese Weise können gemeinsam genutzte Funktionen von verschiedenen Programmen geteilt werden. Im Gegensatz dazu ist eine statische Bibliothek dazu gedacht, bei der Erstellung eines ausführbaren Programms oder einer anderen Bibliothek ihre Funktionen beizusteuern, wobei dieser Vorgang von einem sogenannten Binder (englisch „Linker“) durchgeführt wird. Dabei sind Bibliotheken in der Regel Archivdateien, die eine Ansammlung einzelner Binärobjekte enthalten, die wiederum die Funktionen enthalten. Diese Binärobjekte können dabei ebenfalls als eigenständige Objektdateien auftreten und sind neben Quelltexten typische Ein- und Ausgaben von Compilern (sogenannte „Translation Units“). Dabei enthalten Binärobjekte immer eine Symboltabelle für die Zuordnung einzelner Komponenten und eine Relokationstabelle für die Speicherverwaltung, da sie ausschließlich dazu gedacht sind, mit anderen Objekten gebunden zu werden.

In dieser Arbeit werden ausschließlich die zuletzt genannten Binärobjekte betrachtet *und der Begriff „binäre Objektdatei“ wird sich immer auf diese Objekte beziehen.* Ganze Bibliotheken können damit aber ebenfalls durch Zerlegen und wieder Zusammensetzen bearbeitet werden.

Symbole kennzeichnen verschiedene Dinge wie Funktionsanfänge oder Datenobjekte mit einem Namen und Attributen, genauso wie es die Einträge der Symboltabelle eines Compilers, der das Objekt erzeugt haben könnte, tun (vgl. [6], Abschnitt 2.7). Binärobjekte kennen aber nur eine einzige Tabelle, d. h. es gibt keine Gültigkeitsbereiche außer einem

Attribut, das aussagt, ob ein Symbol global (sichtbar für andere Objekte) oder lokal (auf das Objekt beschränkt) ist. Wenn mehrere Binärobjekte gebunden werden, müssen daher alle globalen Symbole eindeutig sein, sonst schlägt das Binden fehl und dementsprechend kann auch hier von ihrer Eindeutigkeit ausgegangen werden.

Relokationseinträge teilen einem Binder oder Lader mit, an welchen Stellen absolute Adressen vorgesehen sind. Diese absoluten Adressen ändern sich, wenn mehrere Objektdateien zu einem größeren Objekt gebunden werden oder wenn ein Programm ausgeführt werden soll und dazu an eine im allgemeinen Fall nicht festgelegte oder vorhersagbare Speicheradresse geladen wird.

Zu diesem Zweck enthält ein typischer Relokationseintrag die Adresse einer Speicherstelle im Binärobjekt, die angepasst werden muss, die Zieladresse, die an der anzupassenden Speicherstelle einzutragen ist und Informationen darüber, in welcher Form diese Zieladresse einzutragen ist. Die Adressen innerhalb des Binärobjekts werden dabei durch Symbole mit Offsets ausgedrückt, da sie beim Binden immer von den absoluten Adressen unabhängig bleiben müssen. Über die Form der Zieladresse wird dem Binder mitgeteilt, wie er die tatsächliche, absolute Adresse am Ende an der Speicherstelle einfügen soll, beispielsweise in welchen Bits eines Befehlswortes sich die Adresse befindet und ob sie absolut oder PC-relativ ist.

Sowohl Symbole als auch Relokationseinträge liefern wertvolle Informationen zum Disassemblieren, da sie Aussagen über die Struktur des Programms machen. Auf die Verwendung von Debug-Informationen oder Ähnlichem wurde verzichtet, da zu erwarten ist, dass diese oft nicht vorhanden sein werden.

2.2.1 Das ELF-Format

Ein konkretes Beispiel für ein Dateiformat für Objektdateien ist ELF („Executable and Linking Format“), das auch im Zusammenhang mit dem WCC in der Regel eingesetzt wird, da es unter Linux und anderen Betriebssystemen – nicht aber Microsoft-Systemen – weit verbreitet ist. Das ELF-Format ist in [20] in Kapitel 4 allgemein dokumentiert und in [9] werden die ARM-spezifischen Eigenschaften und Erweiterungen von ELF festgelegt. Dieses Format ist auch deshalb von besonderer Bedeutung, da zum Lesen von Objektdateien in Kapitel 3 mit der LIBBFD eine Bibliothek gewählt wurde, die für viele verschiedene Objektdatei-Formate eine einheitliche Abstraktion bietet, deren Aufbau sich hauptsächlich an ELF und mit ELF verwandten Formaten orientiert.

Das Format unterstützt alle drei Arten von Objektdateien: ausführbare Programme, Bibliotheken, die durch Archive von Objekten dargestellt werden und relozierbare Binärobjekte. ELF-Dateien sind dabei wie in Abbildung 2.2 gezeigt so aufgebaut, dass sie je nach Anwendungszweck sowohl gebunden („Linking View“) als auch geladen werden können

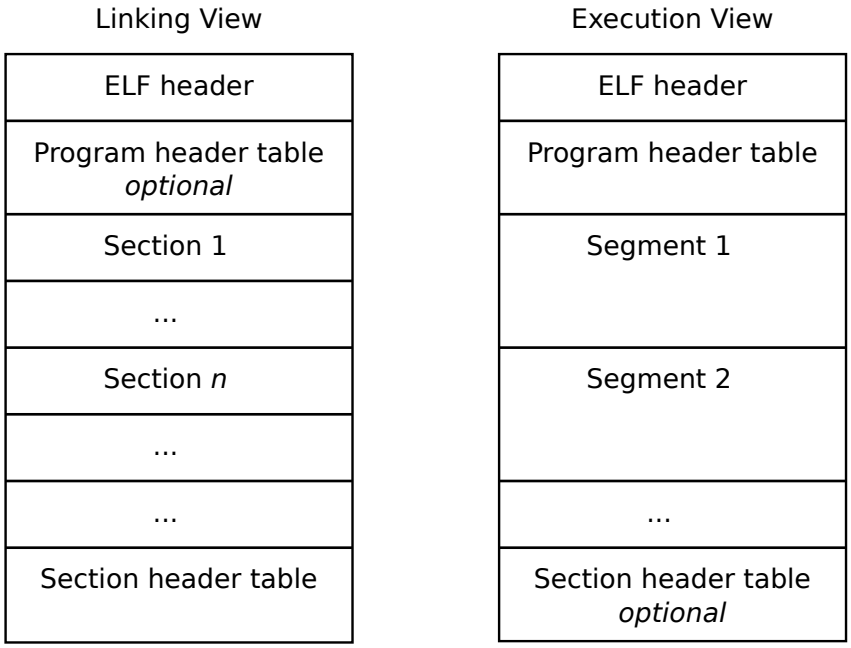

Abbildung 2.2: Aufbau eines ELF-Objekts (entnommen aus [20, Abbildung 4-1])

(„Execution View“) – oder sogar beides gleichzeitig. Beide Skizzen in der Abbildung sind *zwei verschiedene Sichtweisen auf ein und dieselbe Datei*, aber da hier mit Binärobjekten nur solche Objektdateien betrachtet werden, die dazu gedacht sind, mit anderen Objekten gebunden zu werden, ist nur die „Linking View“ relevant.

Am Anfang der Datei stehen im „ELF Header“ alle Meta-Informationen wie Größe und Lage des „Program header table“ und des „Section header table“ und Informationen über das enthaltene Programm wie beispielsweise die Architektur, in der es laufen soll. Der „Program header table“ beinhaltet alle Informationen, die der Lader benötigt um aus dem Inhalt der Objektdatei einen lauffähigen Prozess zu erzeugen. Der „Section header table“ beinhaltet analog dazu alle Informationen, die der Binder benötigt um die Objektdatei mit anderen Objekten binden zu können.

In der „Linking View“ ist eine ELF-Objektdatei ein Container, der aus den gezeigten Sections besteht. Eine solche Section ist ein zusammenhängender Bereich aus Bytes, der beispielsweise Programmcode, Daten oder Debug-Informationen enthalten kann. Auch die Symboltabelle und die Relokationstabelle sind jeweils in eigenen Sections gespeichert. Wenn mehrere Objekte gebunden werden, dann werden die Sections entsprechend ihrer Bedeutung zusammengelegt zu neuen Sections oder Segmenten (sofern sie Teile des Programms enthalten), je nach dem, ob wieder ein Objekt zum Binden oder ein ausführbares Programm erzeugt werden soll.

2.3 ARM Instruktionsformen

Die LLIR-Darstellung von Wcc für ARM unterstützt den ARM7TDMI-Prozessor [7] und damit die ARM-Prozessorfamilie ARM7 und deren Befehlssatz/Architektur ARMv4. Dieser RISC-Befehlssatz lässt sich in zwei Instruktionsarten unterteilen, die auch als unabhängige Befehlssätze verstanden werden können: konventionelle, 32-Bit-breite ARM-Befehle und 16-Bit-breite Thumb-Befehle, die eine kompakte Erweiterung mit einigen Einschränkungen sind. Der Prozessor verfügt dabei über die 13 Universalregister `R0-R12`, die drei wie Universalregister adressierbaren Register `SP` (`R13`), `LR` (`R14`) und `PC` (`R15`) und die zwei geschützten Statusregister `CPSR` und `SPSR`. `SP` ist der Stapelzeiger („Stack Pointer“), in `LR` wird die Rücksprungadresse eines Funktionsaufrufs hinterlegt („Link Register“) und `PC` ist der Befehlszähler („Program Counter“).

2.3.1 ARM 32-Bit-Befehlssatz

Bei dieser Instruktionsart handelt es sich um den konventionellen Befehlssatz der ARM-Prozessorfamilie. Die Befehle können alle 16 Register `R0-R15` adressieren und jeder Befehl kann bedingt ausgeführt werden, d. h. er kann einzeln ohne Zuhilfenahme von Sprungbefehlen übersprungen werden. Die Befehle werden in die folgenden Klassen semantisch zusammengefasst (vgl. [7], Abbildung 1-5 auf Seite 1-12), zu denen es jeweils eine binäre Kodierung gibt, die durch Flags und Bitcodes zwischen Unterformen unterscheidet (in einigen Fällen werden in der Definition des Befehlssatzes der Übersichtlichkeit halber mehrere nahezu identische Kodierungen angegeben). Diese Unterformen werden beim Assemblieren sowohl durch das Mnemonic als auch durch die Syntax der Operanden gewählt.

„Branch and exchange“ besteht aus einem einzigen Befehl, der einen indirekten Sprung mit einem beliebigen Register als Sprungziel durchführt. Dabei wird durch das niederwertigste Bit des Registers festgelegt, um welche Instruktionsart (konventionelle, 32-Bit-breite ARM-Befehle oder 16-Bit-breite Thumb-Befehle) es sich bei den Befehlen ab dem Sprungziel handelt. Dieser Befehl wird von GNU Gcc auch als Teil der Befehlsfolge für den Rücksprung aus einer Funktion verwendet.
(Assembler-Mnemonic `BX`)

„Branch“ besteht aus einem Befehl, der zu einem Ziel, das als vorzeichenbehaftete 26-Bit-Konstante im Maschinenbefehl enthalten ist, springt. Zusätzlich kann die Adresse des nachfolgenden Befehls im „Link Register“ `LR` hinterlegt werden, um einen der ARM-Konvention entsprechenden Funktionsaufruf zu realisieren. Durch die bedingte Ausführung dieses Befehls werden auch alle bedingten Sprünge realisiert.
(Assembler-Mnemonics `B` und `BL`)

„Data Processing“ repräsentiert alle verfügbaren ALU-Operationen bis auf die Multiplikation. Die Kodierung enthält sowohl die jeweilige ALU-Operation als auch zwei Quelloperanden und einen Zieloperanden. Auch im weitesten Sinne durch die ALU durchführbare Operationen wie Transfers zwischen Registern werden so realisiert. Dabei ist der Zieloperand und der erste Quelloperand immer ein Register, während der zweite Quelloperand sowohl ein Register als auch eine Konstante sein kann und bei Bedarf logisch oder arithmetisch geschoben werden kann. Die zu dieser Klasse gehörende Kodierung wird im Assembler durch eine große Zahl von verschiedenen Mnemonics erzeugt.
(Assembler-Mnemonics `ADC`, `ADD`, `AND`, `BIC`, `CMN`, `CMP`, `EOR`, `MOV`, `MVN`, `ORR`, `RSB`, `RSC`, `SBC`, `SUB`, `TEQ` und `TST`, außerdem die Pseudo-Instruktion `ADR`)

„FSR Transfer“ enthält Befehle, um die geschützten, nicht direkt adressierbaren Status-Register lesen und schreiben zu können.
(Assembler-Mnemonics `MRS` und `MSR`)

„Multiply“ und „Multiply Long“ enthalten die Multiplikationsbefehle, die auch Multiply-Accumulate unterstützen.
(Assembler-Mnemonics `MLA`, `MUL`, `SMLAL`, `SMULL`, `UMLAL` und `UMULL`)

„Single Data Transfer“ enthält die Befehle um 32-Bit breite Wörter aus dem Speicher zu lesen und in den Speicher zu schreiben.
(Assembler-Mnemonics `LDR`, `LDRB`, `LDRBT`, `LDRT`, `STR`, `STRB`, `STRBT` und `STRT`)

„Halfword Data Transfer“ und „Single Data Swap“ enthalten Befehle, um nur 8- oder 16-Bit-breite Wörter zu lesen oder zu schreiben, oder um zwei Speicherzugriffe atomar auszuführen. Beim Lesen kann dabei eine Vorzeichenerweiterung stattfinden. Durch „Single Data Swap“ kann in einem Multitasking-System eine Prozess-Synchronisation realisiert werden.
(Assembler-Mnemonics `LDRH`, `LDRSH`, `LDRSB`, `STRH`, `SWP` und `SWPB`)

„Software Interrupt“ enthält den SWI-Befehl, die eine Unterbrechung in Software auslöst – einen Sprung zur Adresse 0x08 analog zu „Branch with Link“, wobei der Prozessor vom „User“- in den „Supervisor“-Modus wechselt.
(Assembler-Mnemonic `SWI`)

2.3.2 Thumb 16-Bit-Befehlssatz

Die nur 16-Bit breiten Thumb-Befehle können eine kompaktere Darstellung eines Programms ermöglichen, wenn eine Reihe durch Einschränkungen dieser Instruktionsart fehlende Funktionalitäten bei einem konkreten Programm nur selten benötigt werden. Bei-

spielsweise können die meisten Thumb-Befehle nur die niederwertigsten 8 Register `R0-R7` adressieren und die bedingte Ausführung von Befehlen ist nur bei Sprungbefehlen möglich.

Im Gegensatz zu konventionellen 32-Bit-Befehlen werden Thumb-Befehle nicht nach ihrer Bedeutung klassifiziert, sondern in sogenannte Formate unterteilt, die jeweils der Kodierung einiger Maschinenbefehle entsprechen (vgl. [7], Abbildung 1-6 auf Seite 1-21). Dabei kennt der Befehlssatz 18 Formate, von denen jedes eine Teilmenge einer Klasse der 32-Bit-Befehle implementiert, sowie ein Format um die von den anderen Formaten nicht adressierbaren Register zu verwenden. Einige Befehle müssen dabei zwingend paarweise verwendet werden, d. h. sie bilden effektiv einen 32-Bit-Befehl.

Formate 1 bis 4 sowie 12 und 13 entsprechen verschiedenen „Data Processing"-Befehlen.
(Assembler-Mnemonics `ADC`, `AND`, `ASR`, `BIC`, `CMN`, `CMP`, `LSL`, `LSR`, `MOV`, `MUL`, `MVN`, `NEG`, `ORR`, `ROR`, `SBC`, `SUB` und `TST`)

Format 5 erlaubt für einige Operationen den Zugriff auf die sonst nicht oder nur teilweise adressierbaren höchstwertigen 8 Register.
(Assembler-Mnemonics `ADD`, `BX`, `CMP` und `MOV`)

Formate 6 und 11 entsprechen „Single Data Transfer"-Befehle.
(Assembler-Mnemonics `LDR` und `STR`)

Formate 7 und 9 entsprechen sowohl „Single Data Transfer"- als auch „Halfword Data Transfer"-Befehlen. (Assembler-Mnemonics `LDR`, `LDRB`, `STR`, `STRB`)

Formate 8 und 10 entsprechen „Halfword Data Transfer"-Befehlen.
(Assembler-Mnemonics `LDRH`, `LDSB`, `LDSH`, `STRH`)

Formate 16, 18 und 19 entsprechen „Branch"-Befehlen.
(Assembler-Mnemonics `B` und `BL`)

Format 17 entspricht dem „Software Interrupt"-Befehl.
(Assembler-Mnemonic `SWI`)

2.3.3 Bedingte Ausführung durch Conditional Codes

Mit dem sogenannten „Conditional Code" kann bei vielen Befehlen festgelegt werden, unter welcher Bedingung sie auszuführen sind, d. h. Befehle, deren Bedingung nicht erfüllt ist, ändern weder ein Prozessorregister noch den Speicher. Alle ARM 32-Bit-Befehle und das Format 16 des Thumb-Befehlssatzes („Conditional Branch") unterstützen eine solche

Bedingung. Alle bedingten Sprünge werden auf diese Weise aus einem einzelnen Sprungbefehl abgeleitet. Der „Conditional Code“ wird im Assembler an das Mnemonic angehängt (beispielsweise wird aus dem Befehl `ADD` und der Gleichheitsbedingung „`EQ`“ der Befehl `ADDEQ`) und belegt in kodierter Form vier Bits des Maschinenbefehls. Die Bedingung wird dabei an Flags des Statusregisters `CPSR` gestellt (Zero-Flag, Carry-Flag, Negative-Flag und Overflow-Flag), d. h. sie bezieht sich auf das Ergebnis des letzten vorangegangenen Befehls, der das Statusregister gesetzt hat.

Dabei überprüfen die Bedingungen `EQ`, `NE`, `CS`, `CC`,`MI`, `PL`, `VS` und `VC`, ob einzelne Flags aus `CPSR` entweder gesetzt oder nicht gesetzt sind. Die Bedingungen `HI`, `LS`, `GE`, `LT`,`GT` und `LE` nehmen anhand mehrerer Flags aus `CPSR` arithmetische Vergleiche vor. Für die unbedingte Ausführung steht die Bedingung `AL`, die automatisch angenommen wird, wenn keine Bedingung explizit angegeben wurde. Beispielsweise führen die Assembler-Befehle `B` und `BAL` zum selben Maschinenbefehl für `B` mit „Conditional Code“ `AL`.

Kapitel 3

Disassembling

Einer der wichtigsten Aspekte dieser Arbeit besteht darin, die nur noch in Form von Maschinenbefehlen vorliegenden Funktionen in die LLIR-Darstellung von Wcc zurück zu überführen, d. h. zuerst einmal an LLIR-Funktionen zu gelangen, die LLIR-Instruktionen mit Labeln enthalten statt Maschinenbefehlen mit Konstanten und Offsets. Dazu muss der Programmcode aus einer binären Objektdatei ausgelesen werden, die Funktionen müssen innerhalb des Programmcodes identifiziert werden und anschließend muss der Maschinencode jeder Funktion disassembliert werden, wobei außerdem viele in einem binär vorliegenden Programm nicht mehr vorhandene Informationen über Sprungziele und Ähnliches wiederhergestellt werden müssen. Außerdem müssen alle initialisierten Datenobjekte (Variablen, Arrays) ebenfalls aus der binären Objektdatei zurückgewonnen werden. Es stellen sich grundlegend also zwei Probleme, nämlich Objektdateien zu parsen und Maschinencode in die LLIR-Darstellung zu disassemblieren.

3.1 Parsen von Objektdateien

Das Ziel beim Parsen von Objektdateien ist es, überhaupt an die einzelnen Komponenten einer binären Objektdatei und ihre Inhalte zu gelangen. Für viele verschiedene Objektdateiformate gibt es frei verfügbare Bibliotheken und Werkzeuge zum Einlesen und auch zum Bearbeiten. Dabei sticht besonders das GNU Binutils-Paket hervor, da es eine Vielzahl von Werkzeugen, einen Assembler, einen Linker und mit der Libbfd eine eigene Bibliothek zum Lesen und Schreiben von Objektdateien enthält.

3.1.1 GNU Binutils

Die GNU Binutils sind eine Sammlung verschiedener Programme, die vor allem bei der Software-Entwicklung beim Umgang mit binären Objektdateien benötigt werden (oder dabei sehr hilfreich sind). Die wichtigsten Programme sind wohl GNU Ld und GNU As, der Binder und der Assembler, die beispielsweise auch von GNU Gcc verwendet werden. Daneben enthält die Sammlung aber auch eine Reihe von Programmen zum Anzeigen, Bearbeiten und Konvertieren von Objektdateien. Um dabei auf die Inhalte der Dateien zugreifen zu können, verwenden alle diese Programme die Libbfd-Bibliothek, die ebenfalls Teil der Binutils ist und die über eine abstrakte Schnittstelle Objektdateien zugänglich macht. Die Binutils unterstützen dabei eine Vielzahl von verschiedenen Objektformaten, Plattformen und Betriebssystemen. Da es sich um Open-Source-Software handelt, ist außerdem der Quelltext vollständig offengelegt.

Von besonderem Interesse für diese Arbeit waren dabei die zwei Programme `objdump` und `objcopy`, da sie es ermöglichen, alle Details einer Objektdatei einzusehen und Objektdateien durch Kopieren zu verändern. Außerdem enthält `objdump` einen textbasierten Disassembler für viele Prozessortypen, der unter anderem auch ARM unterstützt. Die für diese Arbeit implementierten Klassen und Programme verwenden Teile des Quelltextes von `objdump` und `objcopy`, weswegen auch dieser Code immer die Libbfd einsetzt.

Während der Implementierung stellten sich aber auch ein paar Nachteile der Binutils heraus. Vor allem zeigte sich, dass die Libbfd einige für diese Arbeit wichtige Informationen nicht über ihre Schnittstellen exportiert, sodass auch im Quelltext verschiedener Binutils-Programme immer wieder an der Libbfd vorbei direkt auf formatspezifische Datenstrukturen zurückgegriffen werden musste. Beispielsweise kann das ELF-Format für ein Symbol neben seiner Adresse in einer Section auch die Größe speichern, d. h. einen Bereich definieren, aber die Datenstrukturen der Libbfd enthalten nicht mehr die Größe eines Symbols. Diese direkten Zugriffe erfolgen über interne Datenstrukturen der Libbfd, die zu den sogenannten Backends gehören – Module, die für die Libbfd mit entsprechenden Bibliotheken der jeweiligen Objektformate arbeiten. Diese Datenstrukturen sind nicht öffentlich und damit auch nicht in den Header-Dateien enthalten, die für die Libbfd im Entwicklungssystem installiert sind.

Obendrein stellte sich heraus, dass die fertigen Libbfd-Pakete des Entwicklungssystems, die die Libbfd unabhängig von den Binutils enthalten, in Bezug auf die unterstützten Formate und Plattformen auf das Entwicklungssystem selbst reduziert sind (system-Fassung) oder versionsspezifisch sind (multiarch-Fassung), d. h. Wcc müsste bei jedem Versionswechsel der Libbfd neu übersetzt werden. Aus diesen Gründen war es unumgänglich, die Libbfd in das Wcc-Projekt zu importieren (`3RDPARTY/LIBBFD`) – und sich

damit auf eine Version festzulegen, diese Version als Teil vom WCC und unabhängig von den BINUTILS übersetzbar zu machen und sie bei jedem vollständigen Übersetzungslauf mit zu übersetzen.

3.1.2 Erzeugen von Objektdateien mit `objcopy`

In diesem Kapitel geht es zwar erst einmal darum, eine Objektdatei einlesen zu können, allerdings sollen auch die Flowfact-Annotationen des Kontrollflusses in Objektdateien hinterlegt werden können, um sie mit dem Programm zusammen laden zu können. Mehrere Werkzeuge aus den BINUTILS können Objektdateien erstellen: Für den Assembler und den Binder ist das zwingend erforderlich, aber mit `objcopy` steht auch ein Programm zur Verfügung, das speziell für das Verändern von Objektdateien durch Kopieren entwickelt wurde.

Objektdateien sind wie in Abschnitt 2.2 beschrieben Containerformate. Bei einem solchen Format ist es schwer und auch nicht vorgesehen, nachträglich Veränderungen an einer existierenden Datei vorzunehmen. Üblicherweise werden Objektdateien von einem Binder aus anderen Dateien zusammengestellt oder es werden einzelne binäre Objekte vom Assembler erzeugt. Um später in Kapitel 5 Flowfacts in einer binären Objektdatei hinterlegen zu können, muss daher die zu annotierende Objektdatei kopiert werden.

`objcopy` benutzt die LIBBFD, um eine Objektdatei einzulesen und abstrakt darstellen zu können. Mit verschiedenen Filtern für Symbole und Sections können dann beispielsweise existierende Sections entfernt oder auch neue Sections hinzugefügt werden, und auf diese Weise können auch existierende Sections ersetzt werden. Dabei wird im Arbeitsspeicher ein neues Objekt angelegt – auch die LIBBFD kann Objekte nur entweder einlesen oder neu anlegen. Die C++-Klasse `BinaryObjectCopy` aus Kapitel 5 wird basierend auf diesem Code die Flowfacts in einer eigenen Section zu einer Objektdatei hinzufügen können und existierende Flowfacts auch überschreiben können, um dem Flowfact-Editor aus Kapitel 5 das Schreiben von Flowfacts zu ermöglichen. Das Lesen von Flowfacts wird bereits durch die in diesem Kapitel vorgestellte Klasse `BinaryObject` realisiert, die ganz allgemein auf alle Sections einer Objektdatei lesend zugreifen kann.

3.1.3 Disassembler aus `objdump`

Das `objdump`-Programm aus BINUTILS enthält unter anderem einen Disassembler, der neben vielen verschiedenen Prozessortypen auch ARM-Code disassemblieren kann. Dieser Disassembler erzeugt aus dem Maschinencode der entsprechenden Sections einer binären Objektdatei Assembler-Code in Textform. Übersetzt man ein Beispielprogramm wie das in Abbildung 3.1 mit GNU GCC in Assembler, dann erhält man eine Ausgabe wie in Abbildung 3.2a. Disassembliert man ein ebenfalls aus dem Beispielprogramm erzeugtes Binärobjekt, dann erhält man wiederum den Assembler-Code in Abbildung 3.2b. Man sieht

bereits auf den ersten Blick, dass nahezu alle Assembler-Befehle wieder zurückgewonnen wurden – lediglich für den `MOV`-Befehl in Zeile 18 wurde ein äquivalenter Pseudo-Befehl gewählt. Allerdings sind für alle `PC`-relativen Sprungbefehle (`BNE` und `B` in den Zeilen 16 und 21) nur die Offsets erhalten. `objdump` disassembliert dabei in einem einzigen Durchlauf die Maschinenbefehle wörtlich, ohne Sprungziele auszuwerten oder Ähnliches – die Ausgabe dient zur Ansicht und ist nicht unmittelbar als Eingabe für einen Assembler gedacht. Da dieser Disassembler im Quelltext verfügbar ist, bot er sich als Ausgangspunkt für den LLIR-Disassembler an. Allerdings waren grundlegende Überarbeitungen notwendig, um statt der Textausgabe aus dem Maschinencode LLIR-Instruktionen zu erzeugen.

```
int a=3;
int b;

void main(void) {
    int c=7;

    if (a == 3) {
        b = c*2;
    } else {
        b = c*3;
    }
}
```

Abbildung 3.1: Beispielprogramm in C

Um zu der Textausgabe zu gelangen, identifiziert der Disassembler jeden Maschinenbefehl und übersetzt dessen Operanden in eine lesbare Form, indem er für jeden Befehl eine an die `printf()`-Funktion angelehnte Beschreibung zur Interpretation und Formatierung der Operanden verwendet. Dieses Vorgehen ist auch für die Erzeugung der LLIR-Darstellung sinnvoll, aber die Maschinenbefehle müssen bestimmten LLIR-Instruktionscodes zugeordnet werden und die Beschreibung der Operanden muss festlegen, welche LLIR-Parameter für die Operation der Instruktion zu erzeugen sind. Bei der Implementierung der Textausgabe konnten dabei einige Vereinfachungen vorgenommen werden, die für die LLIR-Darstellung problematisch sind.

Beispielsweise kennt der „Single Data Transfer"-Befehl `LDR` als Maschinenbefehl zwei Flags, die die Größe des Wortes und einen Teil der Adressierungsart festlegen, und die durch jeweils ein Bit kodiert werden. Im Assembler werden diese Flags aber über das Mnemonic gewählt – hier wird zwischen `LDR`, `LDRB`, `LDRBT` und `LDRT` unterschieden. Die LLIR-Darstellung orientiert sich am Assembler und kennt daher ebenso diese vier Instruktionscodes, d. h., in Abhängigkeit dieser Flags ist eine von vier *verschiedenen* Instruktionen

```
1   .data
2   a:
3   .word   3
4   .comm   b,4,4
5
6   .text
7   main:
8   str     fp, [sp, #-4]!
9   add     fp, sp, #0
10  sub     sp, sp, #12
11  mov     r3, #7
12  str     r3, [fp, #-8]
13  ldr     r3, .L4
14  ldr     r3, [r3, #0]
15  cmp     r3, #3
16  bne     .L2
17  ldr     r3, [fp, #-8]
18  mov     r2, r3, asl #1
19  ldr     r3, .L4+4
20  str     r2, [r3, #0]
21  b       .L1
22  .L2:
23  ldr     r2, [fp, #-8]
24  mov     r3, r2
25  mov     r3, r3, asl #1
26  add     r2, r3, r2
27  ldr     r3, .L4+4
28  str     r2, [r3, #0]
29  .L1:
30  add     sp, fp, #0
31  ldmfd   sp!, {fp}
32  bx      lr
33  .L4:
34  .word   a
35  .word   b
```

(a) gcc Assembler-Ausgabe

```
1
2
3
4
5   Disassembly of section .text:
6
7   00000000 <main>:
8   str     fp, [sp, #-4]!
9   add     fp, sp, #0
10  sub     sp, sp, #12
11  mov     r3, #7
12  str     r3, [fp, #-8]
13  ldr     r3, [pc, #64]  ; <main+0x5c>
14  ldr     r3, [r3]
15  cmp     r3, #3
16  bne     38 <main+0x38>
17  ldr     r3, [fp, #-8]
18  lsl     r2, r3, #1
19  ldr     r3, [pc, #44]  ; <main+0x60>
20  str     r2, [r3]
21  b       50 <main+0x50>
22
23  ldr     r2, [fp, #-8]
24  mov     r3, r2
25  lsl     r3, r3, #1
26  add     r2, r3, r2
27  ldr     r3, [pc, #16]  ; <main+0x60>
28  str     r2, [r3]
29
30  add     sp, fp, #0
31  ldmfd   sp!, {fp}
32  bx      lr
33  ...
34
35
```

(b) objdump Disassembler-Ausgabe

Abbildung 3.2: Assembler-Darstellung des kompilierten Beispielprogramms aus Abbildung 3.1

zu erzeugen, obwohl gemäß des Befehlsformats immer derselbe Maschinenbefehl vorliegt. Auch ein textbasierter Disassembler muss hier unterscheiden, kann aber in der Textausgabe einfach die Flags als Buchstaben an das Mnemonic anhängen und so wie im Befehlsformat alle vier Assembler-Befehle als einen betrachten – der Disassembler aus `objdump` tut das. An dieser Stelle muss also die Identifizierung des Befehls angepasst, d. h. aufgeteilt werden.

Bei der Auswertung der Operanden greift `objdump` in einigen Fällen auf Unterfunktionen zurück, wenn ein Maschinenbefehl verschiedene Operandentypen kennt, besonders bei Befehlen mit verschiedenen Adressierungsarten. Obwohl dieses Vorgehen auch hier angewendet werden könnte, erschien es angesichts der ohnehin notwendigen Anpassungen der Identifizierung sinnvoll, jeden Befehl mit jedem seiner verschiedenen Operandentypen unabhängig voneinander zu identifizieren. Ein Beispiel hierfür sind die „Data Processing“-Befehle, die einen Zieloperanden und bis zu zwei Quelloperanden kennen. Der zweite (manchmal einzige) Quelloperand ist dabei entweder ein Register oder eine Konstante, wobei der Wert des Registers wahlweise um einen konstanten Wert oder um den Wert eines weiteren Registers verschoben werden kann. Diese drei Varianten desselben Befehls werden unabhängig voneinander wie verschiedene Befehle identifiziert, sodass die Beschreibung zur Interpretation der Operanden jeweils nur genau einen Typ umfasst.

3.2 Disassembler für LLIR

Im Gegensatz zu einem Programm in Assembler-Darstellung oder als Maschinencode, das aus einzelnen Befehlen besteht die nur unterbrochen durch die Sprungbefehle sequenziell abgearbeitet werden, besteht die LLIR-Darstellung eines Programms aus Funktionen, die in Form von Kontrollflussgraphen vorliegen. Diese Art der Darstellung wird in [6], Abschnitt 8.4 als Flussgraph vorgestellt, der dort aus Grundblöcken besteht, die Drei-Adress-Anweisungen enthalten. Grundblöcke werden im Kontext der LLIR als Basisblöcke bezeichnet und statt der Drei-Adress-Anweisungen (vgl. [6], Abschnitt 6.2) werden LLIR-Instruktionen – eine maschinelle Darstellung von Assembler-Befehlen – verwendet. Die tatsächliche Rekonstruktion der Kontrollflussgraphen geschieht zwar erst in Kapitel 4, aber um sie vorzubereiten, werden bereits beim Disassemblieren die entsprechenden Datenstrukturen der LLIR verwendet.

Entscheidend für die LLIR-Darstellung ist hier, dass der Maschinencode auf Funktionen verteilt werden muss, dass Funktionen statt Instruktionen nur Basisblöcke enthalten, die innerhalb einer Funktion aber (auch) sequenziell geordnet sind, und dass erst die Basisblöcke immer sequenziell geordnete Instruktionen enthalten. Dabei kommen Label nur als Namen von Basisblöcken vor und können nicht mehr einzelne Instruktionen markieren bzw. eine Instruktion wird genau dann von einem Label markiert, wenn sie die erste Instruktion des ebenso benannten Basisblocks ist. Neben ihrem eigentlichen Aufbau aus Funktionen und ihren Kontrollflussgraphen ermöglicht ein LLIR-Objekt aber durch die se-

quenzielle Ordnung der Basisblöcke einer Funktion, dass man es auch als Container für ein im Ganzen sequenziell vorliegendes Programm aus Funktionen ohne Kontrollflussgraphen benutzt, und in genau dieser, in Abbildung 3.3 gezeigten Form soll der Disassembler den Maschinencode an die Kontrollflussrekonstruktion weitergeben.

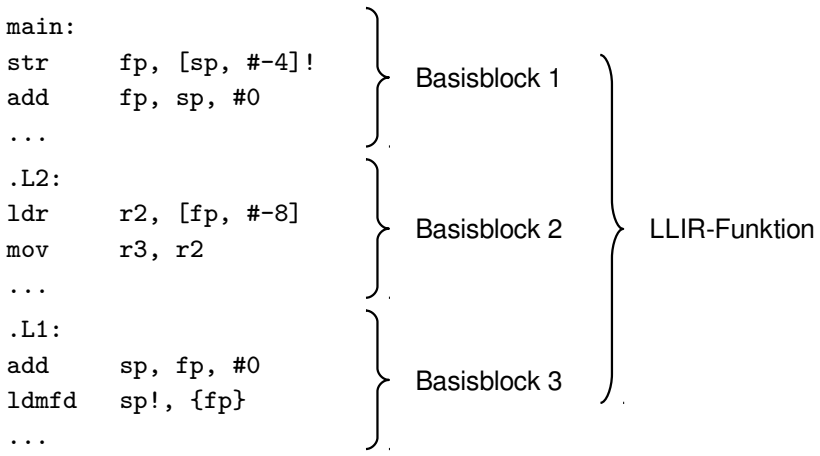

Abbildung 3.3: Sequenzielle Aufteilung eines Programms in Basisblöcke

Viele der Informationen, die man für ein LLIR-Objekt als Container benötigt, sind in einem binären Objekt noch erhalten, wenn es so erstellt wurde, dass man es binden kann. Beispielsweise wird zwischen Code-Sections und Daten-Sections unterschieden, Funktionen sind durch Symbole markiert und auch Datenobjekte haben oft ein eigenes Symbol, das sie als solche kennzeichnet. Andererseits liegen die Ziele von `PC`-relativen Sprungbefehlen nur noch in Form von Offsets vor, die in der LLIR-Darstellung sehr unhandlich sind, da sie sich auf Adressen im Maschinencode beziehen. Um später einen Kontrollflussgraphen rekonstruieren zu können, müssen daher die Ziele aller Sprungbefehle bereits als Label in den entsprechenden LLIR-Instruktionen eingetragen sein, und *dieselben* Label müssen auch für die Basisblöcke – die Sprungziele – verwendet worden sein.

In den folgenden Abschnitten werden alle Schritte erläutert, durch die ein LLIR-Disassembler basierend auf dem vorhandenen, textbasierten Disassembler aus `objdump` erstellt wurde und wie alle fehlenden Informationen so gut wie möglich wiederhergestellt wurden. Dieser Disassembler muss dabei nicht nur Maschinenbefehle umsetzen, sondern auch LLIR-Datenobjekte erzeugen und gemäß der Sprungziele eine Unterteilung in Basisblöcke vornehmen – im Gegensatz zum textbasierten Disassembler aus `objdump`, bei dem es vor allem darum geht, den Inhalt einer Section, die Maschinencode enthält, lesbar zu machen. Der `objdump`-Disassembler kann sich daher darauf beschränken, Maschinenbefehle zu dekodieren und braucht nicht alle Informationen auszugeben, die für die LLIR-Darstellung oder

beispielsweise auch für einen Assembler benötigt würden. Der LLIR-Disassembler schafft ganz allgemein die technische Voraussetzung für alle weiteren Schritte.

3.2.1 Wiederherstellung von Labeln

Der vom GCC erzeugte Assembler-Code unterscheidet sich von der Ausgabe des `objdump`-Disassemblers unter anderem darin, dass alle Variablen und Sprungziele mit Labeln markiert sind, die diesen Adressen einen Namen geben. Ein binäres Objekt enthält aber nur noch solche Informationen, die für das Binden (oder Laden) benötigt werden. Nur noch wenige Label sind in Form von Symbolen erhalten geblieben. Insbesondere fallen lokale Label weg, da sie keine Bedeutung mehr haben, während für Funktionen Symbole zum Binden erhalten bleiben müssen. Um den Kontrollflussgraphen einer Funktion erstellen zu können, müssen alle Sprungziele bekannt sein, d. h. es müssen zumindest die Zieladressen von Sprungbefehlen aufgelöst werden können. Aber auch bei Lade- und Schreibbefehlen ist es notwendig, die Zieladresse zu kennen, um einem Befehl ein Datenobjekt zuordnen zu können.

Bei Befehlen mit absoluten Adressen, die im ARM-Befehlssatz nicht vorgesehen sind, wäre für den Befehl ein Relokationseintrag vorhanden um die tatsächliche Adresse nach dem Binden oder Laden einfügen zu können. Das in diesem Relokationseintrag vorhandene Ziel-Symbol ermöglicht es, einen Label für den Adressoperanden im Befehl zu erhalten, gegebenenfalls mit einem Offset. Außerdem ist im Relokationseintrag die Adresse des Befehls enthalten, sodass sich diese Information beim Disassemblieren leicht zuordnen lässt. Tatsächlich legt aber beispielsweise der GCC-Compiler Adressen in Datenworten am Ende einer Funktion ab, die durch `PC`-relative Ladebefehle geladen werden, um so auf eine absolute Speicheradresse zuzugreifen.

Bei Befehlen mit indirekter Speicheradressierung ist eine Auflösung – wenn überhaupt – nur durch eine Datenflussanalyse möglich. Auch dann wird die Menge der möglichen Adressen aber in der Regel nur eingeschränkt, d. h., die tatsächlichen Adressen müssen vom Benutzer festgelegt werden. Da hier keine Datenflussanalyse stattfindet, können indirekte Sprungbefehle nicht aufgelöst werden. Stattdessen wird für bestimmte Formen von Sprungbefehlen heuristisch angenommen, dass es sich beispielsweise um den Rücksprung aus einer Funktion handelt. Andere Befehle bleiben unberücksichtigt und müssen immer vom Benutzer aufgelöst werden. Für Sprungbefehle ergibt sich hierbei eine praktische Einschränkung daraus, dass für einen Basisblock in der LLIR-Darstellung nur zwei Nachfolger erwartet werden, da der bisher durch den GCC erzeugte Code keine Sprünge mit mehr als zwei Zielen verwendet bzw. nur bedingte Sprungbefehle einsetzt, die entweder ausgeführt oder ignoriert werden.

Für `PC`-relative Sprung- und Ladebefehle, die vom Compiler am häufigsten generiert werden, lässt sich die Zieladresse leicht berechnen. Dazu muss der Maschinenbefehl aber

zuerst als ein solcher Befehl identifiziert werden und dekodiert werden. Außerdem können PC-relative Sprungbefehle selbstverständlich negative Offsets enthalten, die dazu führen, dass ein Label *vor* dem Befehl eingefügt werden muss. Da der Disassembler anhand von Labeln eine Unterteilung in Basisblöcke vornimmt, werden daher zuerst in einem eigenen Durchlauf über den Maschinencode alle Label erzeugt und erst dann kann der Code vollständig disassembliert werden.

Befehlstyp	**Sprungart**	**Anmerkungen**	**Befehle**
„Branch“	PC-relativ	Ziel immer PC+Operand	B, BL
„Branch and Exchange“	indirekt	Ziel immer Register	BX
„Data Processing“	absolut	Rd=PC und Konstante als Operand (nicht sinnvoll)	MOV, MVN
	PC-relativ	Rd=Rn=PC und Konstante als 2. Operand	ADC, ADD, AND, BIC, EOR, ORR, RSB, RSC, SBC, SUB
	indirekt	Rd=PC, Rn≠PC oder Register als 2. Operand	
„Single Data Transfer“ oder „Halfword Data Transfer“	indirekt	Rd=PC, Ladebefehl	LDR, LDRB, LDRBT, LDRH, LDRSB, LDRSH, LDRT
„Block Data Transfer“	indirekt	PC in Registerliste, Ladebefehl	LDM
„Software Interrupt“	absolut	Ziel immer $(08)_{16}$	SWI

Abbildung 3.4: Sprungbefehle und solche Befehle, die zu Sprüngen führen

In der Tabelle in Abbildung 3.4 sind alle Sprungbefehle des ARM-Befehlssatzes aufgelistet. „Branch“, „Branch and Exchange“ und „Software Interrupt“ sind hierbei naheliegend, aber aufgrund des ungeschützten PC-Registers können auch Befehle aus den Klassen „Single Data Transfer“, „Halfword Data Transfer“ und „Block Data Transfer“ zu Sprüngen führen, wenn das PC-Register zu den Zielregistern gehört. Ungünstig ist vor allem, dass mit „Data Processing“-Befehlen Sprünge realisiert werden können, die entweder indirekt oder PC-relativ sind und bei denen das Sprungziel erst durch den Befehl berechnet wird.

3.2.2 Unterscheidung von Daten, ARM-Code und Thumb-Code

Grundsätzlich enthalten die Sections eines Binärobjekts immer Daten einer bestimmen Kategorie wie Programmcode, -Daten, Symbole oder Relokationen. Die Kategorie, in die

eine Section fällt, ist dabei eines von verschiedenen Attributen, das über die LIBBFD abgefragt werden kann. Trotzdem befinden sich in der „.text“-Section, die per Konvention den Programmcode enthält, auch Daten von Funktionen – insbesondere Adresswörter. Außerdem unterstützt die ARMv4-Architektur mit ARM-Code und Thumb-Code effektiv zwei Befehlssätze, die beim Disassemblieren unterschieden werden müssen, da die Maschinenbefehle denselben Wertebereich nutzen, d. h., der Zustand des Prozessors entscheidet, ob ein Wort als ARM- oder als Thumb-Befehl ausgeführt wird. Ungünstigerweise wechselt der Prozessor zwischen diesen beiden Zuständen ausschließlich durch den *indirekten* Sprungbefehl `BX`, bei dem das niederwertigste Bit der Zieladresse das Befehlsformat wählt. Ein Wechsel zwischen den Formaten kann theoretisch jederzeit erfolgen, beispielsweise durch Inline-Assembler. Trotzdem verbleiben in einem Binärobjekt eine Reihe von Informationen, die diese Unterscheidung in aller Regel ermöglichen.

In einer ELF-Objektdatei für ARM sind Funktionen mit Symbolen markiert, die aussagen, dass es sich um eine Funktion handelt und ob diese Funktion aus ARM-Code oder aus Thumb-Code besteht (vgl. [9], Abschnitt 4.5.3). Das Symbol lässt damit bereits eine Grundannahme zu. Außerdem werden in [9], Abschnitt 4.5.5 sogenannte Mapping-Symbole definiert, die anzeigen, von welchem Typ alle darauffolgenden Bytes bis zum nächsten Mapping-Symbol sind. Mapping-Symbole unterscheiden dabei zwischen Daten, ARM-Code und Thumb-Code, sind aber leider nicht zwingend vorgeschrieben.

Unabhängig vom Format der Objektdatei werden beim Dekodieren der Instruktionen `PC`-relative Ladebefehle erkannt, um ihren Zieladressen Label zuzuweisen. Da bei einem solchen Befehl sowohl die Adresse als auch die Wortlänge bekannt ist, identifiziert der Befehl einen Datenbereich in einer Code-Section. Hierfür muss aber bereits eine korrekte Unterscheidung zwischen ARM- und Thumb-Code erfolgt sein.

Da die Bedeutung eines Bytes oder Wortes einer Section also aus verschiedenen Quellen ermittelt werden muss, wird in der Implementierung jede Section partitioniert in Bereiche, die Daten, ARM-Code oder Thumb-Code enthalten. Durch die oben beschriebenen Vorgehensweisen wird eine Section dann Schritt für Schritt in entsprechende Bereiche unterteilt. Beim anschließenden Disassemblieren kann durch diesen rudimentären Typisierungs-Mechanismus für jedes Byte einer Section der Typ festgestellt werden.

3.2.3 Identifizierung von Befehlen

Um einen Maschinenbefehl zu identifizieren und zu dekodieren, muss analog zum Vorgehen des Instruktions-Dekoders eines Prozessors das Befehlswort auf bestimmte, den Befehl festlegende Bits hin überprüft werden. Auch die Form der Operanden wird auf diese Weise festgelegt. In [19] auf den Seiten 85-86 ist beispielhaft das dort als „entschlüsseln“ bezeichnete dekodieren eines MIPS-Befehls gezeigt. Das Vorgehen ist bei ARM-Befehlen analog und der Befehlssatz und dessen Kodierung sind in [8] dokumentiert. Auch ein Disassem-

bler basiert auf diesem Prinzip, daher soll der Vorgang hier kurz erläutert werden, um den Algorithmus des Disassemblers später zu beschreiben. Als Beispiel soll der folgende Assembler-Befehl

```
MOV R3, R2
```

dienen, bei dem es sich um einen „Data Processing“-Befehl handelt. Einmal assembliert liegt von diesem Befehl nur noch ein 32-Bit-breites Befehlswort vor, das wie in Abbildung 3.5 gezeigt beispielsweise zu $(\text{e1a03002})_{16}$ kodiert worden sein könnte (es gibt mehrere äquivalente Maschinenbefehle, beispielsweise kann der Typ der Schiebeoperation beliebig sein, da um 0 Bits verschoben wird). Dieses Befehlswort würde man dann in einem binären Objekt in der Code-Section „.text“ antreffen.

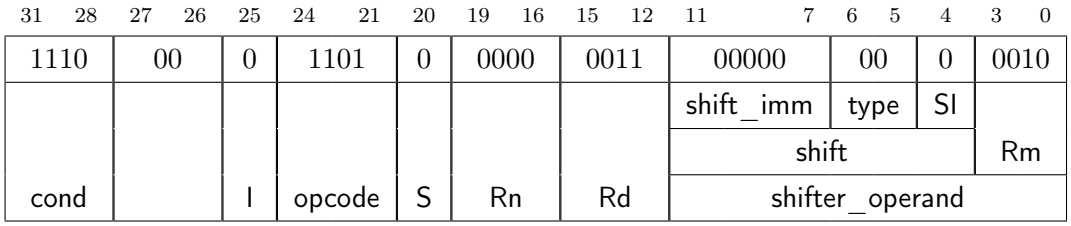

Abbildung 3.5: Binäre Darstellung eines Befehlswortes von „`MOV R3, R2`“ und Kodierung eines „Data Processing“-Maschinenbefehls mit einem Register als zweiten Operanden und einer konstanten Schiebe-Operation (vgl. [8], Abschnitte A3.4 und A5.1.5)

Betrachtet man die Interpretation der Kodierung in Abbildung 3.5, dann kann man den kodierten Befehl von Hand lesen. Durch Bits 26-27 ist erkennbar, dass es sich um einen „Data Processing“-Befehl handelt. Dies legt die Bedeutung weiterer Bits fest: „cond“ ist dementsprechend die Bedingung, unter der der Befehl ausgeführt wird, „opcode“ bestimmt die auszuführende Operation und „Rd“ legt das Zielregister fest. Da „opcode“ $(1101)_2$ ist, soll ein Register auf den Wert eines anderen Registers gesetzt werden (es ist also der „MOV“-Befehl). Für diese ALU-Operation gilt, dass „shifter_operand“ der einzige Operand ist und „Rn“ ignoriert wird, wobei „I“ auf 0 gesetzt bedeutet, dass „shifter_operand“ ein Register ist. Damit ist „Rm“ das Quellregister und „shift“ sagt aus, wie der Wert des Registers „Rm“ außerdem noch verschoben wird. Für „shift“ gilt aufgrund der 0 in „SI“, dass um die Konstante aus „shift_imm“ verschoben wird, wobei aufgrund der $(00)_2$ in „type“ binär nach links verschoben wird.

Es lässt sich also wie in Abbildung 3.6 gezeigt sofort anhand des Werts der Bits 26-27, „opcode“, „I“, „SI“ und „type“ erkennen, dass es sich um einen MOV-Befehl handelt, bei dem der Quelloperand ein Register ist, dessen Wert um eine Konstante nach links geschoben wird. Allgemeiner ausgedrückt lassen sich Maschinenbefehle durch einen Bitvergleich erkennen und korrekt auswerten, indem für einen Befehl und jede mögliche Form seiner Operanden ein binäres Muster aus einer UND-Maske und einem Wert erstellt wird, das

31 28	27 26	25	24 21	20	19 16	15 12	11 7	6 5	4	3 0
XXXX	00	0	1101	X	XXXX	XXXX	XXXXX	XX	0	XXXX
cond		I	opcode	S	Rn	Rd	shift_imm	type	SI	Rm

Abbildung 3.6: Für die Identifizierung eines „MOV“-Befehls mit einem Register als zweiten Operanden und einer konstanten Schiebe-Operation relevante Bits des Befehlswortes

festlegt, welche Bits 0 oder 1 sein müssen, oder egal sind. Die konkrete Maske und der erwartete Wert für den hier als Beispiel verwendeten „MOV“-Befehl ist in Abbildung 3.7 gezeigt.

	31 28	27 26	25	24 21	20	19 16	15 12	11 7	6 5	4	3 0
	1110	00	0	1101	0	0000	0011	00000	00	0	0010
∧	0000	11	1	1111	0	0000	0000	00000	00	1	0000
=	0000	00	0	1101	0	0000	0000	00000	00	0	0000
	cond		I	opcode	S	Rn	Rd	shift_imm	type	SI	Rm

Abbildung 3.7: Befehlswort von „MOV R3, R2“, mit UND-Maske und Wert zur Identifizierung, zeilenweise in der genannten Reihenfolge

3.2.4 Datenobjekte

Neben dem Programmcode enthält die LLIR-Darstellung auch alle Variablen, Arrays etc. in Form von Datenobjekten. In einer binären Objektdatei sind diese Daten verstreut über Sections wie „.data“ und „.rodata“, die ausschließlich Daten enthalten. Viele – aber nicht alle – Datenobjekte sind durch ein Symbol markiert, das den Bereich explizit als Datenobjekt deklariert. In manchen Objektdateiformaten wie ELF enthält das Symbol außerdem die Größe des Datenobjekts. Im Gegensatz dazu sind im Assembler-Code wie in Abbildung 3.2a gezeigt alle Datenobjekte mit Labeln versehen und ihre Größe ergibt sich aus dem Inhalt bzw. dem Abstand zum nächsten Datenobjekt.

Für alle mit einem Symbol markierten Datenobjekte wird daher ein Label erzeugt. Auch die nicht markierten Datenobjekte können dadurch gefunden werden, dass auf sie zugegriffen wird – dieser Zugriff erfolgt über eine absolute Adresse in einem anderen Datenobjekt oder als Datenwort am Ende einer Funktion, und absolute Adressen besitzen immer einen Relokationseintrag. Aus dem Ziel-Symbol des Relokationseintrags und seinem Offset können damit auch für diese Datenobjekte Label erzeugt werden.

Sobald alle Datenobjekte gefunden wurden, kann deren jeweilige Größe, sofern sie nicht in der binären Objektdatei angegeben ist, durch den Byte-Abstand zum nächsten Datenobjekt sinnvoll abgeschätzt werden. Dadurch werden genug Informationen wiederhergestellt, um LLIR-Datenobjekte anlegen zu können.

Es stellt sich aber die Frage, wie der Inhalt eines Datenobjekts zu interpretieren ist. Durch ein Symbol ist nicht mehr bekannt, ob es sich beispielsweise um einen 32-Bit-breiten Integer oder einen C-String handelt. Gerade bei Variablentypen, die länger als ein Byte sind, spielt auch die Byte-Endianness der Plattform bzw. des Prozessors eine wichtige Rolle. Aus diesem Grund werden *alle* LLIR-Datenobjekte ausschließlich mit Bytes initialisiert, d. h. die Bytes aus der entsprechenden Section der Objektdatei werden in derselben Reihenfolge einzeln in jedes Datenobjekt übernommen. Durch diesen kleinsten gemeinsamen Nenner ist zwar der Typ verloren, der im Kontext von Assembler aber ohnehin nicht mehr relevant ist, aber die Datenobjekte können verlustfrei nach der Übersetzung durch WCC wieder in einer binären Objektdatei abgelegt werden.

C-Strings

String-Variablen in C sind als Zeiger auf `char`-Arrays implementiert. Eine nicht initialisierte String-Variable ist damit nur eine einfache Variable, die wie jede andere als Datenobjekt existiert. Bei einer initialisierten String-Variable wird aber für den eigentlichen String (den Text) ein zweiter Speicherbereich in der „.rodata“-Section angelegt, wobei die String-Variable die Adresse dieses Speicherbereichs enthält, der nicht notwendigerweise über ein eigenes Symbol verfügt. Aufgrund der Adresse gibt es für die String-Variable aber einen Relokationseintrag, sodass der verlorene Label für den Text wiederhergestellt werden kann.

3.3 Umsetzung des Disassemblers

Da die LLIR-Darstellung aus einer binären Objektdatei gewonnen werden soll, wurde der Disassembler als Teil einer Klasse zum Parsen von Objektdateien implementiert. Das Klassendiagramm in Abbildung 3.8 zeigt die Klassenhierarchie und beinhaltet die wichtigsten Methoden, die hier und in den nachfolgenden Abschnitten erläutert werden.

Als abstrakte Oberklasse dient die Klasse `BinaryObject`, die sowohl eine Schnittstelle zum Inhalt einer binären Objektdatei bereitstellt als auch zusätzliche Funktionalitäten wie Label (vgl. Abschnitt 3.2.1), die nicht Teil einer Objektdatei sind, implementiert. Als Schnittstelle zur Objektdatei ist die Klasse ein erweiterter Adapter für LIBBFD, der speziell auf das Disassemblieren zugeschnitten ist. Die zusätzlichen Funktionalitäten dienen ebenfalls zum Disassemblieren, sind aber nicht darauf beschränkt. Die Klasse stellt dabei keinen Binder dar, da sie zwar mehrere Objektdateien in einer LLIR vereinen kann, aber im Gegensatz zu einem Binder aus dieser LLIR keine neuen Objektdateien erzeugt.

Wie bei der LIBBFD gibt es auch hier eine Klasse `Section` für die Sections der Objektdatei, die eine Schnittstelle zu den Attributen und Inhalten einer Section bietet. Auch diese Klasse implementiert mit der bereits in Abschnitt 3.2.2 erwähnten Partitionierung einer Section in Bereiche mit unterschiedlichen Typen eine zusätzliche Funktionalität, die für die Disassemblierung benötigt wird.

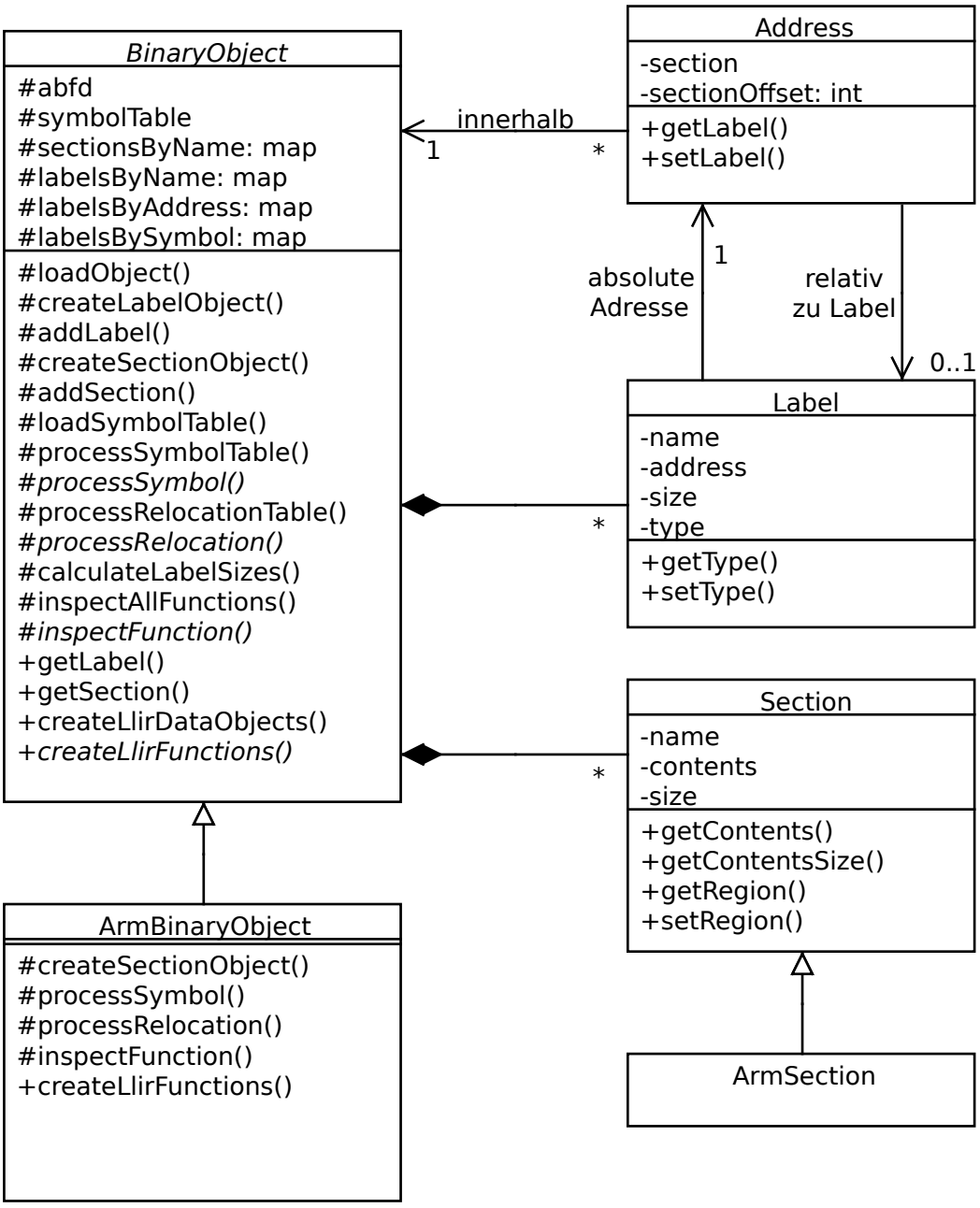

Abbildung 3.8: Klassendiagramm des Objektparsers

Für Label existiert die eigene Klasse `Label`, die die Adresse eines Labels, seinen Namen und seine Attribute enthält. Zu den Attributen gehört dabei der Typ eines Labels, der beispielsweise eine Funktion, ein Sprungziel, ein Datenobjekt oder ein Adressobjekt mitsamt einer Adresse sein kann. Aus allen Symbolen einer Objektdatei, die Label darstellen, werden dementsprechend Label-Objekte erstellt.

Adressen werden wiederum durch Objekte der Klasse `Address` angegeben, die eine Adresse Section-relativ durch ein Section-Objekt und ein Offset innerhalb dieser Section enthalten. Dabei kann eine Adresse auch relativ zu einem Label dargestellt werden, indem in einem `Address`-Objekt zusätzlich ein Label aus derselben Section als Bezugspunkt gesetzt wird.

Alle diese Klassen haben gemeinsam, dass sie nicht auf eine bestimmte Architektur wie ARM festgelegt sind – insbesondere `BinaryObject`, aber auch `Section` sind dazu gedacht, für konkrete Architekturen abgeleitet zu werden. Diese Trennung zwischen Allgemeinem und ARM-spezifischem ist dazu gedacht, später andere vom Wcc unterstützte Architekturen (derzeit TriCore) leichter hinzufügen zu können.

Konkret für die ARM-Architektur gibt es die beiden Klassen `ArmBinaryObject` und `ArmSection`. `ArmBinaryObject` beinhaltet den Disassembler und Implementierungen der virtuellen Methoden von `BinaryObject`, beispielsweise zum ARM-spezifischen Auswerten von Symbolen und Relokationseinträgen. `ArmSection` erweitert vor allem die Klasse `Section` um die Fähigkeit, nicht nur zwischen Code und Daten, sondern auch zwischen ARM-Code und Thumb-Code zu unterscheiden.

3.3.1 Anfängliches Parsen der binären Objektdatei

Bereits bei der Instanziierung öffnet eine `BinaryObject`-Unterklasse die Objektdatei mit der Libbfd, die die Datei einliest, und wertet sofort das gesamte Binärobjekt aus. Für alle Sections werden `Section`-Objekte angelegt und die Section-Inhalte geladen. Dabei folgt `BinaryObject` dem Entwurfsmuster Fabrik und verwendet zum Anlegen der Objekte die virtuelle `createSectionObject()`-Methode, die im Fall von `ArmBinaryObject` beispielsweise Instanzen von `ArmSection` statt von `Section` zurückgibt.

Danach findet das eigentliche Parsen des Binärobjekts statt, das im Sequenzdiagramm in Abbildung 3.9 veranschaulicht ist. Es werden sowohl die Symbole und Relokationseinträge als auch die im Maschinencode enthaltenen Sprungbefehle ausgewertet. Der jeweilige Ort der Implementierung einer virtuellen Methode ist im Diagramm dadurch gekennzeichnet, das zwischen den Klassen `BinaryObject` und `ArmBinaryObject` unterschieden wird.

Alle Symbole werden betrachtet und für jedes Symbol, das als Adresse eine Speicherstelle in einer Section markiert, wird ein entsprechender Label erzeugt. Die konkrete Auswertung von Symbolen und Relokationseinträgen wird dabei über die virtuellen Methoden `processSymbol()` und `processRelocation()` der jeweiligen Unterklasse ebenfalls zur Ver-

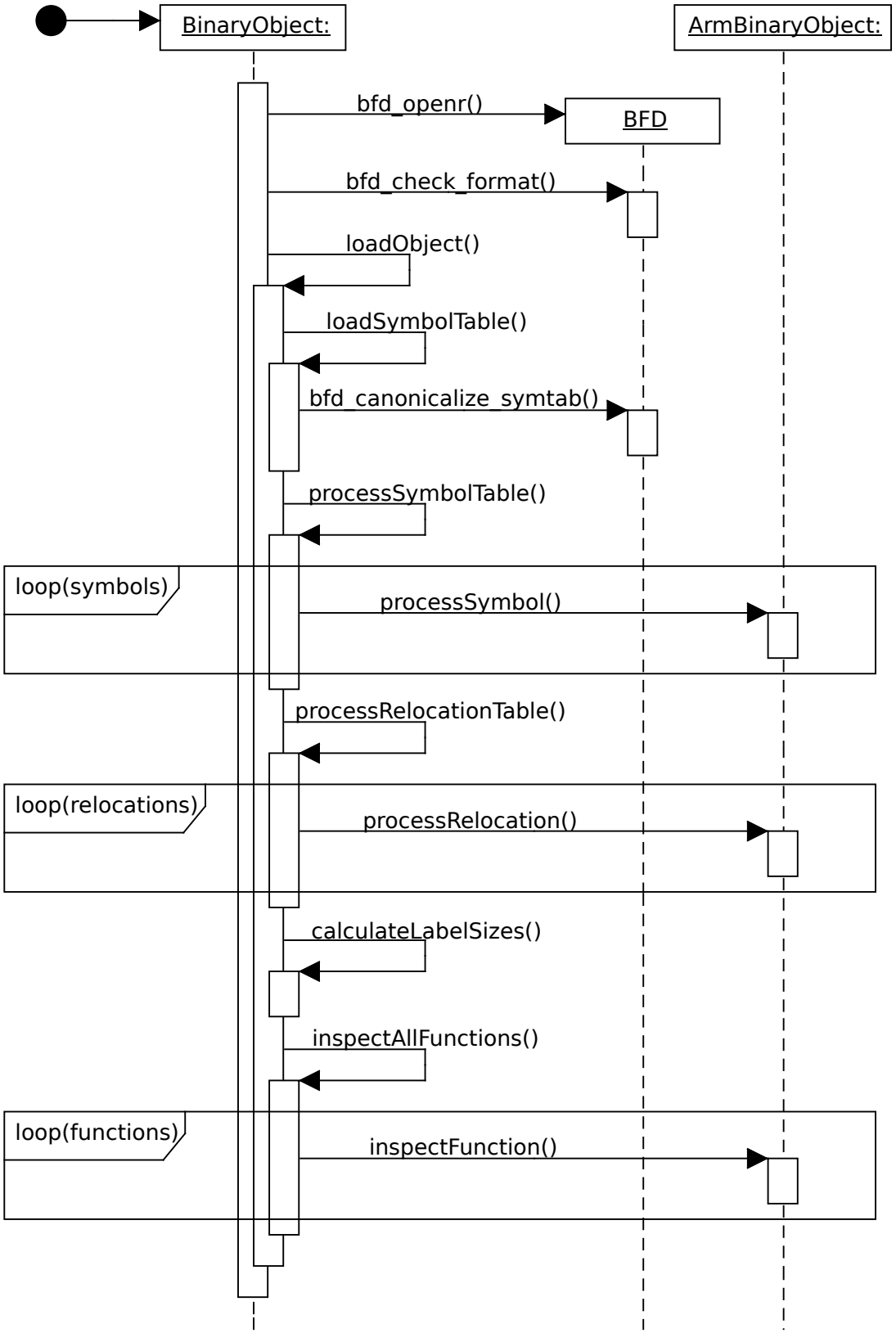

Abbildung 3.9: Sequenzdiagramm des Parsing-Vorgangs für Binärobjekte

fügung gestellt. Auf diese Weise erhält `ArmBinaryObject` beispielsweise die ARM-spezifischen Mapping-Symbole, die Aussagen über den genauen Inhalt einzelner Bereiche einer Section machen. Die beiden Methoden legen dabei für alle geeigneten Symbole ggf. auch für Relokationseinträge die Label an.

Nachdem damit eine Vielzahl von Labeln bereits wiederhergestellt wurde, wird durch die `calculateLabelSizes()`-Methode sichergestellt, dass für alle bisher erzeugten Label neben ihrer Adresse auch ihre Größe bekannt ist. Bei einigen Objektdateiformaten ist die Größe für Symbole bereits vorgegeben, aber für nachträglich aus Relokationseinträgen erzeugte Label ist keine Größe eingetragen. `calculateLabelSizes()` schätzt für alle Label ohne Größenangabe die Größe als den Abstand zum nächsten Label.

Auch die Methode `inspectAllFunctions()` wird bereits vom Konstruktor aufgerufen. Diese Methode ruft wiederum für jede Funktion die abstrakte, hier von `ArmBinaryObject()` implementierte Methode `inspectFunction()` auf, die alle Maschinenbefehle einer Funktion vorab identifiziert, um Sprung- und Ladebefehle zu erkennen und deren Ziele mit Labeln zu versehen. Dazu implementiert `inspectFunction()` den Algorithmus 3.1, der nebenbei auch das eigentliche Disassemblieren, d. h. das Erzeugen der LLIR-Funktionen ein Stück weit vorbereitet, indem er die Bedeutung aller Maschinenbefehle bereits speichert.

Algorithmus 3.1 Dekodierung der Maschinenbefehle und Wiederherstellung von Sprungzielen

```
1:  for all mIns ∈ Maschinenbefehle(objFunktion) do
2:    // Zuerst den Befehl dekodieren (hier wird angenommen, dass alle Maschinenbefehle
      gültig sind)
3:    for all muster ∈ MusterFuerAlleKodierungen do
4:      if (mIns.wort ∧ muster.UND-Maske) = muster.wert then
5:        befehl ← muster.bedeutung
6:      end if
7:    end for
8:    speichereDekodiertenBefehl(objFunktion, befehl)
9:
10:   ziel ← Zieladresse(befehl)
11:
12:   if istSprungbefehl(befehl) then
13:     erzeugeSprunglabel(ziel)
14:   else if istLadebefehl(befehl) then
15:     erzeugeDatenlabel(ziel)
16:     markiereDatenbereich(ziel.section, ziel.offset)
17:   end if
18: end for
```

Algorithmus 3.1 basiert auf dem Konzept zur Identifizierung von Maschinenbefehlen aus Abschnitt 3.2.3, aber da ein Disassembler eine ganze Funktion oder ein ganzes Programm umsetzen können muss, das in Form von Maschinencode vorliegt, d. h. eine Sequenz von nicht vorab bekannten Maschinenbefehlen dekodieren können muss, besitzt der Disassembler daher wie in Abschnitt 3.1.3 beschrieben eine eigene UND-Maske mit Testwert für jeden Maschinenbefehl und jeden Operandentyp dieses Befehls.

Nachdem eine `BinaryObject`-Unterklasse instanziiert wurde, ist damit bereits das gesamte binäre Objekt geparst und alle Label stehen sofort zur Verfügung, sodass anschließend die Datenobjekte und die Funktionen sofort ausgelesen und in ein LLIR-Objekt übertragen werden können.

3.3.2 Erstellen der Datenobjekte

Mit der Methode `createLlirDataObjects()` werden die Datenobjekte aus der binären Objektdatei ausgelesen. Diese Methode ist bereits in der Oberklasse `BinaryObject` implementiert, da sie nicht architekturspezifisch ist. Die Methode iteriert über alle Sections, die gemäß ihren Attributen Daten statt Code enthalten, und erzeugt für jeden vorgefundenen Label ein LLIR-Datenobjekt. Der Inhalt eines initialisierten Datenobjekts besteht normalerweise wie in Abschnitt 3.2.4 bereits beschrieben aus einzelnen Bytes (LLIR-Initialisierungstyp `INITDATA_BYTE`). Eine Ausnahme bilden Label, die vom Typ Adresse sind. Bei solchen Labeln wird stattdessen mittels `INITDATA_SYMBOL` das Datenobjekt mit der im Label enthaltenen Adresse in Form von einem weiteren Label initialisiert – dies entspricht einer initialisierten Zeigervariable, deren absolute Adresse in der binären Objektdatei in einem Relokationseintrag festgehalten wurde und so wieder in einen Label übersetzt werden konnte.

3.3.3 Disassemblieren in LLIR-Funktionen

Da bereits beim Parsen des Binärobjekts alle Befehle identifiziert und alle Sprungziele durch Label markiert wurden, beschränkt sich die Aufgabe der `createLlirFunctions()`-Methode, die den Maschinencode des Binärobjekts in LLIR-Funktionen überträgt, darauf, den Code wie in Abbildung 3.3 auf Seite 23 gezeigt auf LLIR-Funktionen und deren Basisblöcke zu verteilen. Dazu implementiert die Methode den Algorithmus 3.2, der immer alle Befehle der Reihe nach zu einem Basisblock hinzufügt bis ein Label angetroffen wird – dann wird ein neuer Basisblock angelegt, der bis zum nächsten Label für alle weiteren Befehle benutzt wird. Damit bilden die Algorithmen 3.1 und 3.2 zusammen den Disassembler.

Algorithmus 3.2 Erzeugen einer LLIR-Funktion aus dem bereits dekodierten Maschinencode

```
1: llirFunktion ← neueFunktion(funktion.name)
2: llirBB ← ersterBasisblock(llirFunktion)
3:
4: for all befehl ∈ DekodierteBefehle(funktion) do
5:    if existiertLabelFuerBefehl(befehl) then
6:       llirBB ← neuerBasisBlock(label)
7:       llirFunktion.basisBlockHinzufuegen(llirBB)
8:    end if
9:
10:   if existiertLabelFuerOperanden(befehl) then
11:      llirIns ← erzeugeLLIR-InstruktionMitLabel(befehl, label)
12:   else
13:      llirIns ← erzeugeLLIR-InstruktionMitKonstanten(befehl)
14:   end if
15:
16:   llirBB.instruktionHinzufuegen(llirIns)
17: end for
```

Kapitel 4

Kontrollflussrekonstruktion

Nach der Disassemblierung liegt zwar der gesamte Maschinencode in Form von LLIR-Instruktionen vor, aber das LLIR-Objekt wurde nur als Container benutzt. Vor allem sind die Basisblöcke der Funktionen noch nicht miteinander verknüpft – es wurden lediglich Basisblöcke angelegt in der Reihenfolge, in der Label in einer Funktion auftraten und nach diesen Labeln benannt, und die Instruktionen wurden ebenfalls der Reihe nach zum entsprechenden Basisblock hinzugefügt. Diese ungültige Form der Darstellung ist in Abbildung 4.1 für die in Abbildung 3.2a auf Seite 21 enthaltene Assembler-Ausgabe des Beispielprogramms aus Abbildung 3.1 gezeigt. Eine *gültige* LLIR-Darstellung eines Programms besteht aber aus Funktionen, die in Form von Kontrollflussgraphen wie in [6], Abschnitt 8.4 definiert vorliegen. Daher ist eine Rekonstruktion des Kontrollflusses als zweiter, abschließender Schritt notwendig, um zu einem gültigen Graphen wie in Abbildung 4.2 zu gelangen.

Abgesehen von den fehlenden Verknüpfungen ist die Darstellung aus Abbildung 4.1 auch in einer zweiten Hinsicht ungültig: Basisblöcke sind gemäß der Definition eines Kontrollflussgraphen atomar, d. h. nur der letzte Befehl darf ein Sprungbefehl sein und alle Sprünge zu einem Basisblock haben dessen ersten Befehl als Ziel. Diese Bedingung wird vom Basisblock 1 verletzt, da der Disassembler eine Funktion nur nach Labeln, d. h. nach Sprungzielen zerlegt, nicht aber nach Sprungbefehlen. Um Basisblöcke auch nach Sprungbefehlen zu zerlegen, müssen zusätzliche Basisblöcke mit *eindeutigen Labeln* erzeugt werden, d. h. einige der vom Disassembler erzeugten Basisblöcke müssen gespalten werden. Um alle Sprungziele auflösen zu können und Label-Kollisionen zu vermeiden, müssen sowohl das Verknüpfen als auch das Spalten von bestehenden Basisblöcken nachträglich stattfinden, wenn alle Label bekannt sind.

Wie in Abbildung 2.1 auf Seite 8 zu sehen ist, wird die gültige LLIR-Darstellung des Quelltextes vom LLIR-Code Selector erzeugt. Dabei wird wie in Abschnitt 2.1 beschrieben GNU Gcc als Code Selector eingesetzt, indem C-Quelltext vom Compiler *ohne jegliche Optimierung* in ein Assembler-Programm übersetzt wird, das von dem eigens dafür entwickelten Programm `gcc2llir` eingelesen und in die LLIR-Darstellung überführt wird.

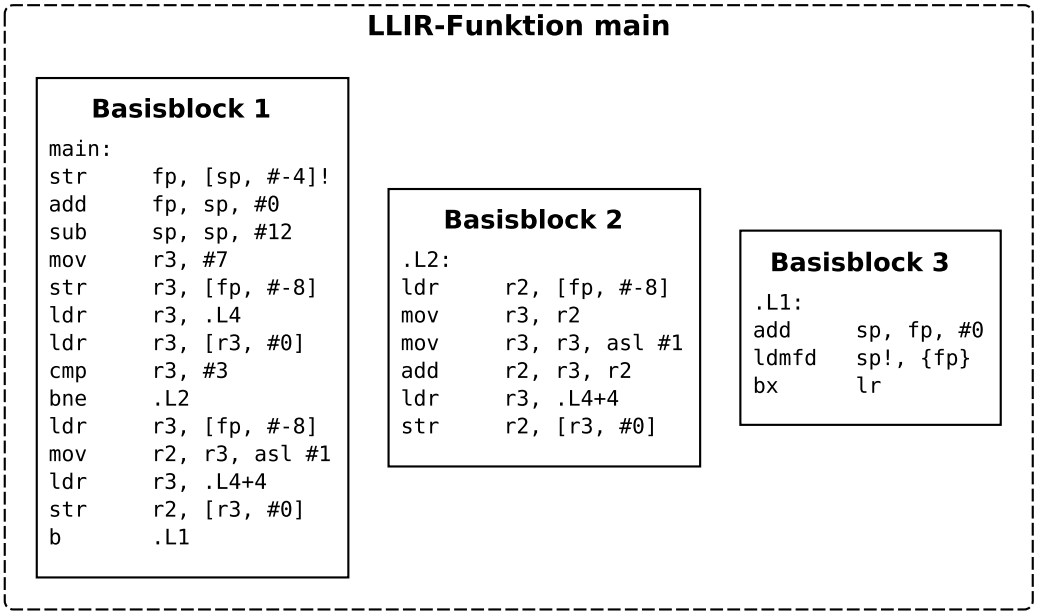

Abbildung 4.1: Lose Basisblöcke des Beispielprogramms 3.1

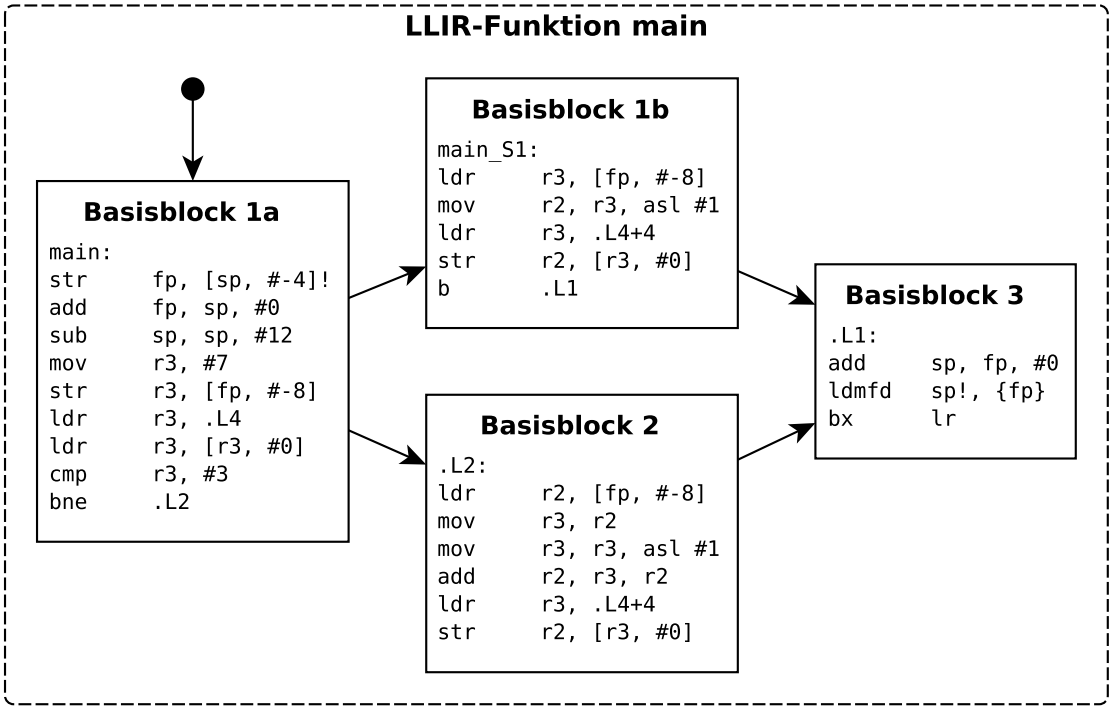

Abbildung 4.2: Gültiger Kontrollflussgraph des Beispielprogramms 3.1

`gcc2llir` geht dabei in zwei Schritten vor: Zuerst wird der Assembler-Quelltext geparst und *in dieselbe ungültige LLIR-Darstellung* übertragen, die auch der Disassembler erzeugt. Anschließend wird der ungültige Graph aus Abbildung 4.1 durch Spalten und Verknüpfen in den gültigen Kontrollflussgraphen aus Abbildung 4.2 transformiert. Aufgrund der Gemeinsamkeiten von `gcc2llir` und dem Disassembler liegt es daher nahe, durch eine Modularisierung von `gcc2llir` den entsprechenden Code wiederzuverwenden.

4.1 Transformation des Graphen

Um die Basisblöcke einer Funktion miteinander zu verknüpfen, wird von `gcc2llir` der Algorithmus 4.1 eingesetzt. Hierbei wird *jeder* Befehl in einem Basisblock bezüglich seiner Sprungeigenschaften klassifiziert als direkter Sprung (`J_UNCOND`, `J_COND`), Funktionsaufruf (`JL_UNCOND`, `JL_COND`), Funktionsrücksprung (`RET_UNCOND`, `RET_COND`) oder kein Sprungbefehl (`NO_JUMP`), und es wird außerdem zwischen unbedingten (...`_UNCOND`) und bedingten Befehlen (...`_COND`) unterschieden. Handelt es sich um einen Sprungbefehl, der nicht der letzte Befehl des Basisblocks ist, dann muss der Basisblock gespalten werden, d. h. die restlichen Befehle *nach* dem Sprungbefehl werden in einen eigenen Basisblock verschoben. Die Unterscheidung von Sprungklassen ist deshalb von Bedeutung, da sie die Nachfolger des Basisblocks festlegt, wobei als Nachfolger nur das Sprungziel und der in Ausführungsreihenfolge nächste Basisblock infrage kommen. Die Nachfolger eines Basisblocks werden von Algorithmus 4.1 folgendermaßen gewählt:

- Bei einem unbedingten Sprung ist ausschließlich das Sprungziel Nachfolger des Basisblocks.
- Bei einem bedingten Sprung ist neben dem Sprungziel auch der nächste Basisblock ein möglicher Nachfolger, da keine allgemeingültige Aussage über den Wahrheitsgehalt der Bedingung gemacht werden kann.
- Funktionsaufrufe haben immer den nächsten Basisblock zum Nachfolger, da pauschal davon ausgegangen wird, dass die Funktion zurückspringt. Die aufgerufene Funktion wird im LLIR-Kontrollflussgraphen einer Funktion nicht vermerkt, da jede Funktion ihren eigenen, von den anderen Funktionen unabhängigen Kontrollflussgraphen hat und es deshalb keine Kanten in eine andere Funktion geben kann.
- Funktionsrücksprünge haben keinen Nachfolger, es sei denn, sie sind bedingt – dann ist der nächste Basisblock ein möglicher Nachfolger.

Einen Befehl als Sprungbefehl zu erkennen ist einfach: der ARM-Befehlssatz kennt eine Reihe von Sprungbefehlen und gibt außerdem direkten Zugriff auf das `PC`-Register, d. h. ein Befehl bzw. die entsprechende LLIR-Instruktion führt zu einem Sprung, wenn es ein expliziter Sprungbefehl ist oder wenn das `PC`-Register der Zieloperand ist (vgl. Tabelle 3.4

auf Seite 25). Schwieriger gestaltet sich aber die Erkennung der Sprungklasse, d. h., ob der Sprung beispielsweise ein direkter Sprung oder ein Funktionsaufruf ist. Noch schwieriger und hier nicht algorithmisch gelöst ist die Erkennung der Ziele eines indirekten Sprungbefehls. Mögliche Lösungen dafür finden sich beispielsweise in [22] , [15, 16] und [10], die aber alle auf Datenflussanalysen basieren, die in dieser Arbeit nicht implementiert wurden.

Die in `gcc2llir` enthaltene Kontrollflussrekonstruktion, die hier wiederverwendet werden soll, klassifiziert Sprünge heuristisch anhand der Tabellen 4.3 und 4.4. Da die Eingabe von `gcc2llir` aus außerordentlich gut strukturiertem Assembler-Code besteht, genügt es die von GNU Gcc gewählten Assembler-Konstrukte zu erkennen. Indirekte Sprünge treten dabei nur in Funktionsrücksprüngen auf für die kein Ziel benötigt wird bzw. deren Ziel immer der Aufrufer ist. Außerdem kann davon ausgegangen werden, dass Funktionen eindimensional und in sich abgeschlossen sind, d. h., es existieren keine anonymen Funktionen innerhalb einer anderen Funktion und Funktionen teilen sich auch keinen Code (keine Basisblöcke) mit anderen Funktionen. Das LLIR-Objekt, das vom Assembler-Parser von `gcc2llir` als Container für Instruktionen, Sprungziele und Funktionslabel verwendet wird, ist also bereits richtig aufgeteilt in Funktionen und deren Basisblöcke und muss nur noch wie oben beschrieben in einen gültigen Kontrollflussgraphen transformiert werden.

Befehl	**Operanden**	**Anmerkungen**	**Sprungklasse**
`B`	bedingt	—	`J_COND`
	unbedingt		`J_UNCOND`
`BL`	bedingt	—	`JL_COND`
	unbedingt		`JL_UNCOND`
`BX`	egal	—	`RET_UNCOND`
`LDM`	`PC` ist auf Registerliste, bedingt	—	`RET_COND`
	`PC` ist auf Registerliste, unbedingt		`RET_UNCOND`
`LDR`	Zielregister `Rd`=`PC`, bedingt	—	`RET_COND`
	Zielregister `Rd`=`PC`, unbedingt		`RET_UNCOND`
`MOVS`, `MVNS`	Zielregister `Rd`=`PC`	—	`RET_UNCOND`

Abbildung 4.3: Sprungklassen für 32-Bit ARM-Befehle in von GNU Gcc erzeugtem Code

Algorithmus 4.1 Algorithmus aus `gcc2llir` zur Verlinkung der Basisblöcke einer Funktion

```
1:  for all bb ∈ funktion.Basisblöcke do
2:    for all ins ∈ bb.Instruktionen do
3:      klasse ← Sprungklasse(ins)
4:      zielBB ← Sprungziel(ins)
5:
6:      if istLetzteInstruktion(bb, ins) then
7:        bb2 ← nächsterBasisblock(funktion, bb)
8:        if klasse ∈ { J_UNCOND, J_COND } and not istAusserhalb(funktion, zielBB)
          then
9:          // nur bei einem Sprungbefehl ist das Ziel ein Nachfolger (nicht bei Funktionsaufrufen)
10:         setzeNachfolger(bb, zielBB)
11:       end if
12:       if klasse ∉ { J_UNCOND, RET_UNCOND } then
13:         // bei einem unbedingten Sprung oder Funktionsrücksprung wäre der nächste Basisblock kein Nachfolger mehr
14:         setzeNachfolger(bb, bb2)
15:       end if
16:
17:     else if klasse ≠ NO_JUMP then
18:       // es wurde ein Sprungbefehl mitten in einem Basisblock angetroffen
19:       bb2 ← spalteBasisblockAb(bb, ins)
20:       if not istErsterBasisblock(funktion, zielBB) and not istAusserhalb(funktion, zielBB) then
21:         setzeNachfolger(bb, zielBB)
22:       end if
23:       if klasse ≠ RET_UNCOND then
24:         setzeNachfolger(bb, bb2)
25:       end if
26:     end if
27:
28:   end for
29: end for
```

Befehl	Operanden	Anmerkungen	Sprungklasse
`B` (Formate 16, 19)	bedingt	—	`J_COND`
	unbedingt		`J_UNCOND`
`BL` (Format 19)	—	—	`JL_UNCOND`
`BX` (Format 5)	—	—	`RET_UNCOND`
`LDMIA` (Format 15)	`PC` ist auf Registerliste	nicht möglich	`RET_UNCOND`
`MOV` (Format 5)	`PC` ist Zielregister	—	`RET_UNCOND`
`POP` (Format 14)	`PC` ist auf Registerliste	—	`RET_UNCOND`

Abbildung 4.4: Sprungklassen für 16-Bit Thumb-Befehle in von GNU Gcc erzeugtem Code

4.2 Erzwingen einer prozeduralen Struktur

Der Maschinencode aus einer binären Objektdatei ist nicht notwendigerweise genauso strukturiert wie die Assembler-Ausgabe von Gcc, da seine Herkunft beliebig ist und es durchaus auch von Hand erstellter Assembler-Code sein kann. Um die Problemstellung trotzdem lösen zu können, kann man aber annehmen, dass der Maschinencode einer Vielzahl von Objektdateien auch strukturiert ist und den einschlägigen Konventionen der ARM-Architektur folgt, ganz besonders, wenn diese von einem Compiler erzeugt wurden. Zumindest kann man viele Verletzungen dieser Annahmen erkennen und den betroffenen Maschinencode verändern. Folgende Vorkommnisse werden dabei berücksichtigt:

1. Direkten Sprüngen in Basisblöcke anderer Funktionen wird durch das Kopieren des Inhalts der angesprungenen Basisblöcke begegnet. Ebenso wird mit Nachfolgern verfahren, die nicht mehr in derselben Funktion liegen (beispielsweise den Nachfolgern eines kopierten Basisblocks). Algorithmus 4.1 enthält deshalb als Erweiterung zur ursprünglichen Fassung aus `gcc2llir` eine Überprüfung der Zugehörigkeit des Sprungziels. Gehört dieser Basisblock zu einer anderen Funktion, dann wird er nicht als Nachfolger gesetzt. Nachdem in allen Funktionen alle zulässigen Nachfolger eingetragen wurden, werden alle Querverbindungen zwischen Funktionen von Algorithmus 4.2 durch Kopieren der betroffenen Basisblöcke aufgelöst. Der Algorithmus prüft solange den jeweils letzten Befehl aller Basisblöcke einer Funktion, bis keiner der Basisblöcke mehr einen Basisblock außerhalb der Funktion als Nachfolger hätte. Solange Basisblöcke anderer Funktionen Ziele sind, werden diese kopiert.

2. Funktionsaufrufe, die nicht den ersten Basisblock einer Funktion zum Ziel haben, sind hier kein Problem, da LLIR-Kontrollflussgraphen grundsätzlich keine Kanten zu anderen Funktionen enthalten. Sie könnten aber ebenfalls durch die Erstellung einer Kopie des angesprungenen Teils einer Funktion als eigenständige Funktion ermöglicht werden.

3. Für indirekte Sprünge wird in Kapitel 5 die Möglichkeit gegeben, die Sprungziele von Hand zu annotieren.

4. Die Annahme, dass alle Befehle einer Funktion „unterhalb“ ihres Labels stehen, d. h. dass der Label auf den Befehl einer Funktion zeigt, der die niedrigste Adresse hat, darf von einem Programm verletzt werden. Ist dies der Fall, dann erscheint der „oberhalb“ stehende Maschinencode zuerst als Teil einer anderen Funktion, deren Basisblöcke aus einer fremden Funktion direkt angesprungen zu werden scheinen. Gemäß Punkt 1 werden diese Basisblöcke dann in ihre eigentliche Funktion kopiert. Dadurch verbleiben in der anderen Funktion nicht erreichbare Basisblöcke, die aber leicht erkennbar sind und entfernt werden können.

5. Die heuristische Klassifikation von Sprungbefehlen wird um das erste Konstrukt aus Tabelle 4.5 erweitert. Alle diese Konstrukte werden von Bound-T, einer weiteren Software zur WCET-Analyse verwendet um Maschinencode auszuwerten, wobei die Konstrukte für indirekte Sprünge aber keine Anwendung finden können, da diese Art von Sprungbefehlen hier ganz allgemein nur von Hand aufgelöst werden kann. Die Unterscheidung zwischen einem direkten Sprung und einem Funktionsaufruf erfordert trotzdem weiterhin, dass die ARM-Konvention eingehalten wird, im `LR`-Register die Rücksprungadresse zu hinterlegen, und es ist nicht möglich, eine gelegentliche Nicht-Einhaltung zu erkennen.

6. Bei Zugriffen auf Datenobjekte muss immer angenommen werden, dass niemals durch Zeigerarithmetik die Grenzen des jeweiligen Datenobjekts überschritten werden. Dies würde aber funktionieren, da an ein geladenes Programm die Erwartung gestellt werden kann, dass einzelne Sections immer als Ganzes, d. h. zusammenhängend in den Speicher geladen werden, wodurch man über einen einzigen Label aus einer Section auf die *gesamte* Section zugreifen könnte. In der von Wcc schlussendlich erzeugten Objektdatei werden die Datenobjekte einzelner, als Quellen benutzter Objektdateien aber nicht mehr zwangsläufig in derselben Anordnung sein. Eine radikale Lösung für dieses Problem könnte darin bestehen, bei betroffenen Objektdateien gesamte Sections jeweils als ein einziges Datenobjekt zu betrachten und alle Label innerhalb der Sections durch einen Label am Anfang und ein entsprechendes Offset zu ersetzen – dies ist aber nicht umgesetzt.

Insgesamt wird der vorliegende Maschinencode durch diese Maßnahmen in die von der LLIR-Darstellung verwendete prozedurale Struktur gezwungen. Das in einigen Situationen notwendige Kopieren von Basisblöcken erhöht die Codegröße und kann damit die Effizienz von Caches reduzieren, ist aber aufgrund der Anforderungen an die Datenstruktur der LLIR unvermeidbar. Es wäre aber möglich, Basisblöcke zu einem späteren Zeitpunkt bei der Optimierung durch den Compiler wieder zu deduplizieren.

Algorithmus 4.2 Kontrolle der Sprungziele im Kontrollflussgraphen

kontrolliereZiele(funktion)(bbAlt, neueFunktion)

```
 1: inOrdnung ← true
 2:
 3: for all bb ∈ funktion.Basisblöcke do
 4:    ins ← bb.letzteInstruktion
 5:    klasse ← Sprungklasse(ins)
 6:    zielBB ← Sprungziel(ins)
 7:
 8:    if klasse ∈ { J_UNCOND, J_COND } and istAusserhalb(funktion, zielBB) then
 9:       bbKopie ← kopiereBB-Inhalt(zielBB, funktion)
10:       ersetzeLabel(ins, bbKopie.label)
11:       setzeNachfolger(bb, bbKopie)
12:       inOrdnung ← false
13:    end if
14: end for
15:
16: // solange wiederholen, bis alle Ziele in Ordnung sind
17: if not inOrdnung then
18:    kontrolliereZiele(funktion)
19: end if
```

4.3 Strukturumbau

Abbildung 4.6 zeigt schematisch als Flussdiagramm, wie man beim Laden von Objektdateien oder Assembler-Code von der jeweiligen Eingabe zu einer gültigen LLIR-Darstellung gelangt. Binäre Objektdateien müssen zuerst selbst geparst werden, d. h. Symbole und Relokationseinträge werden ausgewertet und nicht mehr vorhandene Label werden wiederhergestellt wie in Abschnitt 3.2 dargelegt. Anschließend können die Section-Inhalte in Funktionen disassembliert (Abschnitt 3.3.3) oder in Datenobjekte übertragen werden (Abschnitt 3.2.4), die in einem LLIR-Objekt abgelegt werden, das erst mal nur als Container dient. `gcc2llir` fängt mit Dateien an, die von Gcc erzeugten Assembler-Code enthalten, parst diesen Assembler-Code und überträgt in dann ebenfalls in ein LLIR-Objekt als Container. In beiden Fällen werden dann die Kontrollflussgraphen der Funktionen rekonstruiert und damit erhält man eine gültige LLIR-Darstellung. Das Flussdiagramm zeigt dabei bereits das Ziel des Strukturumbaus, wobei die eingabespezifischen Teile durch die entsprechenden Gruppierungen markiert sind.

1st instruction	2nd instruction	Role
A DP instruction that sets LR to PC, or sets LR to PC ±constant	B	Static call with return to LR
	BX Rn but not BX LR	DC: Dynamic call with return to LR
	A DP instruction with Rd = PC but not MOV PC, LR.	
	LDR PC, ...	
LDR Rn, ... where Rn ≠ PC	An unconditional DP instruction that sets PC to some constant base (perhaps the PC itself) plus some constant factor times Rn	STO: Dynamic branch using a scaled offset from a table, indexed by the LDR operands.

Abbildung 4.5: Kontrollfluss-Muster aus [23], Tabelle 14 auf Seite 28

`gcc2llir` ist zu diesem Zweck als ein einziges, großes Programm aufgebaut, dass in einer einzelnen C++-Datei enthalten ist und eine prozedurale Struktur hat. Der Assembler-Parser und die Kontrollflussrekonstruktion – hier Basisblock-Linking genannt – sind implementiert durch zwei große, aufeinanderfolgende Schleifen in der `main()`-Funktion, von denen die Erste den Assembler-Code der Eingabedateien parst und die Zweite im vollständig gefüllten LLIR-Objekt die Kontrollflussgraphen der einzelnen Funktionen wiederherstellt. Zwischen den beiden Schleifen bestehen dabei Abhängigkeiten in Form von gemeinsam genutzten Datenstrukturen zur Verwaltung der Label im Assembler-Code und im LLIR-Objekt.

Das konkrete Ziel des Strukturumbaus war es, die Kontrollflussrekonstruktion wie in Abbildung 4.6 gezeigt gemeinsam nutzen zu können. Außerdem sollten die in der Abbildung als Prozesse gezeigten Komponenten von `gcc2llir` so modularisiert werden, dass sie als Bibliotheksfunktionen genutzt werden können, um zu einem späteren Zeitpunkt keine externen Programme mehr starten zu müssen, sondern die jeweiligen Funktionen direkt in Wcc aufrufen zu können.

Der erste Schritt bestand daher darin, die `main()`-Funktion von `gcc2llir` in zwei unabhängige Funktionen zu unterteilen und diese Funktionen in eigene Module auszugliedern. Jede der beiden Schleifen nutzt dabei eine Reihe von Hilfsfunktionen, die ebenfalls entsprechend auf die Module aufgeteilt wurden. Die Abhängigkeit zwischen den Schleifen besteht im Wesentlichen darin, dass beim Spalten von Basisblöcken während der Kontrollflussrekonstruktion Namenskollisionen verhindert werden müssen – über die in der LLIR-Klasse enthaltene Funktion `LLIR::getUniqueLabel()` lassen sich aber ebenfalls eindeutige Namen für die Label erzeugen.

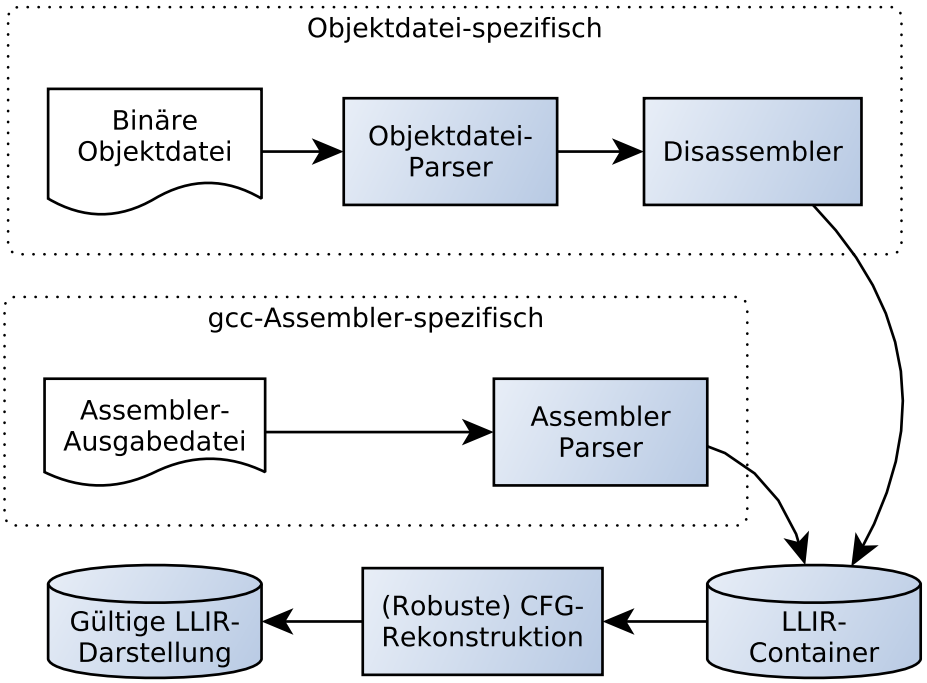

Abbildung 4.6: Flussdiagramm der Erstellung der LLIR-Darstellung

Im nächsten Schritt wurde der bestehende Algorithmus zur Verknüpfung der Basisblöcke zu Algorithmus 4.1 erweitert, um den durch binäre Objektdateien gestellten Anforderungen aus Abschnitt 4.2 besser gerecht zu werden.

Anschließend wurden auf Basis dieser Module und der `BinaryObject`-Klassenhierarchie als Ersatz für `gcc2llir` zwei neue Programme erstellt, um auch binäre Objektdateien direkt in Wcc verarbeiten zu können.

`asm2llir` ersetzt das bisher eingesetzte, monolithische Programm `gcc2llir`. Dabei benutzt `asm2llir` denselben Code, der jetzt aber nicht mehr im Programm enthalten ist, sondern auf die oben genannten zwei Module für Assembler-Parsing und Kontrollflussrekonstruktion verteilt ist. `asm2llir` ist damit nur noch ein Front-End für Bibliotheksfunktionen, das sich aufgrund gleicher Schnittstellen aber trivial in Wcc integrieren lässt, sodass keine Änderungen an Wcc notwendig sind. Im Flussdiagramm in Abbildung 4.6 entspricht `asm2llir` also dem Pfad von „Assembler-Ausgabedatei“ nach „Gültige LLIR-Darstellung“.

`obj2llir` soll dieselbe Schnittstelle und Funktionalität wie `gcc2llir` für binäre Objektdatei bieten. Wie `asm2llir` ist auch `obj2llir` nur ein Front-End, nämlich für die `BinaryObject`-Klassenhierarchie und für das Kontrollflussrekonstruktions-Modul. Auch dieses Programm lässt sich daher mit minimalem Aufwand in Wcc integrieren. Im Flussdia-

gramm 4.6 entspricht `asm2llir` dem Pfad von „Binäre Objektdatei“ nach „Gültige LLIR-Darstellung“.

Algorithmus 4.3 rekursives Kopieren von Basisblöcken

kopiereBB-Inhalt(bbAlt, neueFunktion)

```
1: if istSchonKopiert(bbAlt) then
2:    // (Kontrollflussgraphen enthalten naturgemäß Schleifen)
3:    return  vorherigeKopie(bbAlt)
4: end if
5:
6: // den Inhalt des alten Basisblocks in einen neuen kopieren
7: bbNeu ← erzeugeNeuenBB(neueFunktion)
8: kopiereInstruktionen(bbAlt, bbNeu)
9:
10: // Ausführungsreihenfolge beachten wegen fall-through-Nachfolgern
11: for all nfAlt ∈ Nachfolger(bbAlt) do
12:    if istDirekterNachfolger(bbAlt, nfAlt) then
13:       nfNeu ← kopiereBB-Inhalt(nfAlt, neueFunktion)
14:       fuegeNachfolgerHinzu(bbNeu, nfNeu)
15:    end if
16: end for
17:
18: for all nfAlt ∈ Nachfolger(bbAlt) do
19:    if not istDirekterNachfolger(bbAlt, nfAlt) then
20:       nfNeu ← kopiereBB-Inhalt(nfAlt, neueFunktion)
21:       fuegeNachfolgerHinzu(bbNeu, nfNeu)
22:    end if
23: end for
24:
25: return  bbNeu
```

Kapitel 5

Flowfacts

Durch den in Kapitel 3 beschriebenen Disassembler kann zwar eine LLIR-Darstellung des in einem binären Objekt enthaltenen Programmcodes zurückgewonnen werden, aber die im C-Quelltext in Form von Pragmas enthaltenen Flowfacts [21] müssen ebenfalls beigesteuert werden um eine Auswertung hinsichtlich der WCET durchführen zu können. Wie beim C-Quelltext ist auch hier eine Annotierung von Hand erforderlich – entlang von Assembler-Befehlen, aber unter Zuhilfenahme der rekonstruierten Kontrollflussgraphen. Außerdem kann es aufgrund von nicht auflösbaren indirekten Sprüngen in einigen Fällen unmöglich sein, einen (vollständigen) Kontrollflussgraphen zu erzeugen, was eine weitere Form der Annotation erfordert. Dies bedeutet, dass Möglichkeiten geschaffen werden müssen, um eine bereits vorhandene LLIR-Darstellung nachträglich zu annotieren und zu vervollständigen, und diese Informationen dauerhaft auf eine Art zu hinterlegen, dass sie beim Laden des Binärobjekts wieder zur Verfügung stehen.

Bei den Flowfacts, die vom WCC unterstützt werden, handelt es sich um sogenannte Loopbounds und Flowrestrictions. Ein Loopbound legt für eine wohlgeformte Schleife Grenzen für die Anzahl der Durchläufe und den Kontrolltyp fest, d. h., ob die Schleifenbedingung am Kopf oder am Ende überprüft wird. Dabei ist eine Schleife wohlgeformt, wenn alle anderen Schleifen des Programms entweder vollständig außerhalb oder vollständig innerhalb der Schleife liegen. Nicht wohlgeformte Schleifen können in durch einen Compiler stark optimiertem Code und in von Hand geschriebenem Assembler-Code vorkommen.

Flowrestrictions beschreiben lineare Abhängigkeiten der Ausführungshäufigkeiten verschiedener Basisblöcke. Seien $n, m, a_1 \dots a_{n+m} \in \mathbb{N}^+$, $BB_1 \dots BB_{n+m} \in LLIR$ und die Funktion count definiert als die Ausführungshäufigkeit von Basisblock BB. Dann ist eine Flowrestriction definiert durch

$$\sum_{i=1}^{n} a_i \cdot \text{count}(BB_i) \leq \sum_{j=n+1}^{n+m} a_j \cdot \text{count}(BB_j)$$

Auf diese Weise können beispielsweise auch für nicht wohlgeformte Schleifen Grenzen festgelegt werden.

Alle diese Annotationen könnte man in der einfachsten Form mittels eines Text-Editors speichern. Hierbei wird aber bereits eine Reihe von Fragen aufgeworfen: In welchem Format sollen die Annotationen ausgedrückt werden? Wie können Basisblöcke, die in einem Binärobjekt überhaupt nicht vermerkt sind, referenziert werden? Wo werden die Textdateien gespeichert und wie werden sie mit einem Binärobjekt assoziiert? Wie wird mit gerade in von Hand erstellten Annotationen unweigerlich auftretenden syntaktischen Fehlern umgegangen?

Einige dieser Probleme lassen sich aus Benutzersicht leicht lösen, wenn man einen eigenen Editor für Flowfacts entwirft, der das Annotieren deutlich erleichtert: Annotationen werden in Form von Kommandozeilen-Befehlen ausgedrückt, Basisblöcke können mit ihren Labeln aus der LLIR-Darstellung referenziert werden, statt in einer unabhängigen Textdatei werden die Flowfacts innerhalb des Binärobjekts abgelegt und syntaktisch falsche Annotationen werden sofort abgelehnt. Dabei ist weiterhin nur ein Format für Annotationen notwendig, da die Flowfacts auch in Form von den Befehlen des Editors gespeichert werden können, d. h. beim Laden wird einfach ein vom Editor erzeugtes Skript von Befehlen abgearbeitet.

Zum Speichern beliebiger Daten in einem vorhandenen Binärobjekt kann man zusätzliche, eigene Sections anlegen – wie bereits in Abschnitt 3.1.2 auf Seite 19 beschrieben, ist der dafür erforderliche Quelltext in `objcopy` aus den GNU Binutils schon vorhanden. Das Referenzieren von Basisblöcken in einem Binärobjekt und ein Format für Annotationen sind aber weiterhin erforderlich und werden im Folgenden beschrieben.

5.1 Bezeichner in binären Objektdateien

In einer binären Objektdatei befinden sich üblicherweise nur noch Symbole für die einzelnen Funktionen innerhalb einer Section mit Maschinencode und Relokationseinträge für bestimmte Instruktionen und Adressworte. Eine Funktion in der LLIR-Darstellung lässt sich also leicht mittels ihres Symbolnamens referenzieren, aber für Basisblöcke und für einzelne Instruktionen muss eine eigene Form gefunden werden.

Es bietet sich an, einen Adressabstand relativ zu einem vorhandenen Symbol zu wählen, das aber *immer* in genau diesem Abstand bleiben muss. Damit kommen nur der Adressabstand zum Anfang der Section oder zum Symbol für die Funktion infrage. Als Einheit für den Abstand ist dabei die Nummer der Instruktion innerhalb ihrer Funktion statt beispielsweise eines Byte-Abstandes sinnvoll, da die Nummer genauso wie der Byte-Abstand immer gleich bleiben wird, aber die Nummer der Instruktion etwas handlicher ist.

Es wurde daher für die Bezeichner von Funktionen, Basisblöcken und Instruktionen die folgende Syntax und Semantik festgelegt:

<Symbolname einer Funktion> bezeichnet eine Funktion, die im binären Objekt mit dem genannten Symbol markiert ist. Der Symbolname ist bei einer aus dem Objekt erzeugten LLIR-Darstellung weiterhin bekannt, d. h., es ist leicht möglich, zwischen einem Symbolnamen und dem dazugehörigen LLIR-Label zu übersetzen.

?<Symbolname der Funktion>:<Zeilennummer> bezeichnet einen Basisblock über die Funktion, die ihn enthält. Die Zeilennummer ist dabei die Nummer einer Instruktion des Basisblocks, angefangen mit 0 für die erste Instruktion einer Funktion. Beim Auflösen dieses Bezeichners wird der Basisblock gewählt, der eben diese Instruktion enthält.

!!<Symbolname der Funktion>:<Zeilennummer> bezeichnet eine einzelne Instruktion. Die Bedeutung ist analog zum Bezeichner für einen Basisblock.

5.2 Editor für Flowfacts

Als eigentlicher Editor für Flowfacts dient eine textbasierte, interaktive Benutzerschnittstelle, mit der Flowfacts sowohl erzeugt als auch gelöscht werden können und mit der die möglichen Ziele indirekter Sprünge festgelegt werden können. Durch die Möglichkeit zu löschen können Flowfacts damit auch rudimentär editiert werden, indem sie gelöscht und in veränderter Form wieder neu erzeugt werden.

5.2.1 Bezeichner im Editor

Um in einem Befehl Funktionen, Basisblöcke und Instruktionen angeben zu können, wird die in Abschnitt 5.1 bereits festgelegte Form für einen Bezeichner verallgemeinert und erweitert:

Bezeichner	**Bedeutung**
<Name einer Funktion>	Funktion
<Name des Basisblocks>	Basisblock
?<Name der Funktion>:<Zeilennummer>	
!<Name des Basisblocks>:<Zeilennummer>	Instruktion
!!<Name der Funktion>:<Zeilennummer>	
<Nummer>	Flowfact

Hierbei bezeichnet das Wort „Name“ entweder einen Symbolnamen, falls die Befehle aus einem binären Objekt stammen, oder einen LLIR-Label, falls vom Anwender Befehle interaktiv eingegeben werden. Es können daher in einem Bezeichner nur solche Namen verwendet werden, für die es tatsächlich ein Symbol bzw. einen LLIR-Label gibt.

Dabei sind die Formen „<Name einer Funktion>“, „<Name des Basisblocks>“ und „`!`<Name des Basisblocks>`:`<Zeilennummer>“ für die Verwendung durch den Benutzer im Editor gedacht, während die bereits in Abschnitt 5.1 eingeführten Formen zum Speichern von Annotationen verwendet werden. Beide Formen können aber überall verwendet werden, sofern der jeweilige Name bekannt ist.

Flowfacts (Loopbounds und Flowrestrictions) werden wiederum durch ihre Nummer auf der Liste aller Flowfacts bezeichnet.

5.2.2 Befehle für Loopbounds

`new loopbound` **Basisblock [min] max [Kontrolltyp]**
`delete loopbound` **Basisblock [min] max [Kontrolltyp]**
`delete loopbound` **Nummer**

Basisblock	bezeichnet einen Basisblock (siehe Abschnitt 5.2.1)
min	minimale Anzahl an Schleifendurchläufen
max	maximale Anzahl an Schleifendurchläufen
Kontrolltyp	Kontrolltyp der Schleife: „`head`“, „`tail`“ oder „`unknown`“
Nummer	Nummer eines Loopbounds auf der Liste aller Flowfacts

`new` erzeugt einen neuen Loopbound mit den angegebenen Werten. Dabei ist vom Benutzer darauf zu achten, dass nur genau *ein* Basisblock einer Schleife annotiert wird. Idealerweise sollte dies der Basisblock sein, der den Schleifenkopf enthält.

Falls kein Minimum angegeben wird, wird standardmäßig der Wert 0 benutzt. Falls kein Kontrolltyp angegeben wird, ist der Kontrolltyp des Loopbounds „`unknown`“.

`delete` entfernt den Loopbound, der exakt den Angaben entspricht oder der auf der Liste aller Flowfacts die angegebene Nummer hat. Dabei führt das Weglassen des Minimums oder des Kontrolltyps zu denselben Standardvorgaben wie beim `new`-Befehl, denen der zu entfernende Loopbound entsprechen muss. Falls unter der Nummer ein anderer Typ von Flowfact abgelegt ist, wird dieser *nicht* entfernt.

5.2.3 Befehle für Flowrestrictions

`new flowrestriction` **Summanden {<=|>=} Summanden**
`delete flowrestriction` **Summanden {<=|>=} Summanden**
`delete flowrestriction` **Nummer**

Summanden	**Summand [+ Summanden]**
Summand	**Faktor∗Basisblock**
Nummer	Nummer einer Flowrestriction auf der Liste aller Flowfacts

`new` erzeugt eine neue Flowrestriction mit den angegebenen Basisblöcken und ihren Faktoren. Alle Faktoren müssen größer null sein.

`delete` entfernt die Flowrestriction, die exakt den Angaben entspricht oder die auf der Liste aller Flowfacts die angegebene Nummer hat. Falls unter der Nummer ein anderer Typ von Flowfact abgelegt ist, wird dieser *nicht* entfernt.

5.2.4 Befehle für indirekte Sprünge

`new target` **Sprungblock {Zielblock|Instruktion}**
`delete target` **Sprungblock {Zielblock|Instruktion}**

Sprungblock	ein Basisblock, der mit einem indirekten Sprungbefehl endet
Zielblock	ein Basisblock, der indirekt angesprungen wird
Instruktion	eine Instruktion, die indirekt angesprungen wird

`new` fügt im Kontrollflussgraphen einer Funktion ihrem Basisblock Sprungblock eine Kante zum Basisblock Zielblock oder zu dem Basisblock, in dem die angegebene Instruktion liegt, hinzu. Wenn eine Instruktion als Ziel angegeben wird und diese Instruktion nicht die erste ihres Basisblocks ist, wird der Basisblock entsprechend gespalten, um die Definition eines Kontrollflussgraphen einhalten zu können (vgl. [6], Abschnitt 8.4).

`delete` entfernt analog zu `new` aus dem Kontrollflussgraphen einer Funktion die Kante vom Basisblock Sprungblock zum Basisblock Zielblock oder zu dem Basisblock, in dem die angegebene Instruktion liegt. Dabei werden vormals durch `new` gespaltene Basisblöcke aber *nicht* wieder zusammengelegt.

5.2.5 Interaktive Befehle im Editor

`ls {flowrestrictions|loopbounds|branches}`
`help`
`write`
`quit`

`ls` listet alle Flowrestrictions, alle Loopbounds oder alle Basisblöcke mit indirekten Sprungbefehlen auf. `help` gibt eine Hilfestellung aus. `write` schreibt alle Annotationen in die binäre Objektdatei. `quit` beendet den Editor.

Diese Befehle sind für den Kommandozeilen-Editor gedacht und nur dort implementiert. Zwar dürfen sie auch in der Annotations-Section eines binären Objekts vorkommen, aber dort werden sie beim Einlesen ignoriert.

5.3 Technische Umsetzung

Der Umgang mit Flowfacts erfordert insgesamt im Wesentlichen drei Komponenten: einen Parser für das oben beschriebene Befehlsformat für Flowfacts und andere Annotationen, einen interaktiven Editor und eine Speicherkomponente zum Lesen und Schreiben der Annotationen in einem binären Objekt. Da eine eigene Section im jeweiligen binären Objekt der Speicherort sein soll, bietet sich zum Lesen der Annotationen die `BinaryObject`-Klasse aus Abschnitt 3.3 an. Zum Erzeugen und Ersetzen dieser Section wird aber eine weitere Klasse benötigt, die auf dem Code von `objcopy` aus den GNU Binutils aufbaut, wie bereits in Abschnitt 3.1.2 auf Seite 19 angekündigt. Zunächst soll aber auf den Parser eingegangen werden, der sowohl für den interaktiven Editor als auch für die Speicherkomponente benötigt wird.

Das oben beschriebene Befehlsformat wurde als kontextfreie Grammatik entworfen, um eine konsistente Struktur und Eindeutigkeit der Ausdrücke zu erreichen. Daher konnte zum Parsen der Parsergenerator Yacc [14, 1] eingesetzt werden, der aus einer mit sogenannten Aktionen (kurzen Fragmenten von C-Code) angereicherten kontextfreien Grammatik ein C-Programm erzeugt. Dieses Programm erhält die eingegebenen oder eingelesenen Befehle, parst sie und führt entsprechend der Produktionen der Grammatik Aktionen aus. Dabei sind diese Produktionen die Bezeichner und Befehle des Formats für Annotationen, und die Aktionen ändern die Flowfacts der LLIR gemäß den Befehlen, d. h. sie erzeugen neue Annotationen, entfernen existierende Annotationen oder führen andere Aufgaben aus, wie beispielsweise Textausgaben für den Benutzer zu erzeugen. Um den Parser überschaubar zu halten, ist der LLIR- und Flowfact-spezifische Code so gering wie möglich gehalten, d. h. es werden zum Bearbeiten außerhalb des Parsers liegende Funktionen eingesetzt.

Um denselben Parser gleichzeitig durch den Editor und die Speicherkomponente nutzen zu können, wurde eine Klassenhierarchie bestehend aus der abstrakten Oberklasse `FlowfactEdit` und den beiden Unterklassen `FlowfactConsole` für den Editor und `FlowfactBlockReader` zum Einlesen entworfen, wie im Klassendiagramm in Abbildung 5.1 veranschaulicht ist. Die Oberklasse `FlowfactEdit` stellt dabei außerdem auch dem Parser alle zum Bearbeiten der Annotationen benötigten Methoden zur Verfügung, d. h., alle Änderungen am Code zum Bearbeiten der Flowfacts würden in der `FlowfactEdit`-Klasse vorgenommen werden. Daneben enthält `FlowfactEdit` auch virtuelle Stub-Methoden für alle interaktiven Befehle, die ebenfalls von Aktionen des Parsers benutzt werden. Diese Methoden können erst durch die Unterklassen entsprechend der Funktionalität der Unterklasse implementiert werden, aber der Parser muss sie aufrufen können. Bei den Methoden zum Bearbeiten der Flowfacts handelt es sich vor allem um `newLoopbound()`, `deleteLoopbound()`, `newFlowrestriction()`, `deleteFlowrestriction()`, `newBranchTarget()` und `deleteBranchTarget()`. Die Methoden `findFunction()`, `findBB()`, `findInstruction()`

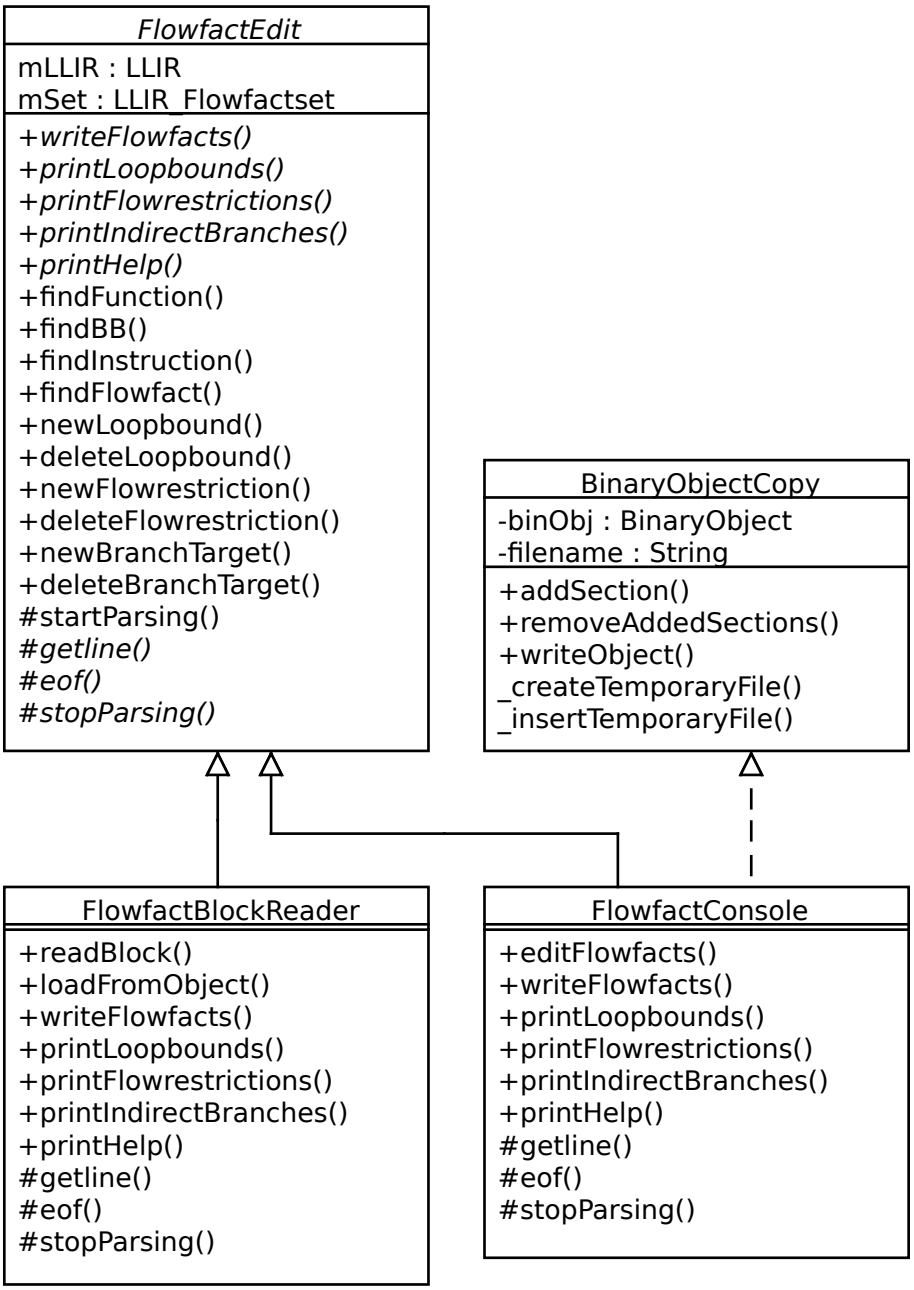

Abbildung 5.1: Klassendiagramm des Flowfact-Editors

und `findFlowfact()` werden aber ebenfalls vom Parser verwendet, um Bezeichner aufzulösen.

`printLoopbounds()`, `printFlowrestrictions()`, `printIndirectBranches()` und `printHelp()` bilden die interaktiven Methoden, die von der jeweiligen Unterklasse zu implementieren sind. `writeFlowfacts()` ist ebenfalls eine virtuelle Methode, da es der Unterklasse überlassen ist, ob sie Flowfacts auch speichern kann – dies sollte beim Einlesen aus einer Objektdatei nicht möglich sein, d. h. die Methode sollte dann leer sein. Die geschützten Methoden `startParsing()`, `stopParsing()`, `getline()` und `eof()` dienen zur Kommunikation mit dem YACC-Parser und werden von der jeweiligen Unterklasse gesteuert.

Die Unterklasse `FlowfactConsole` wird für interaktive Benutzereingaben, d. h. für den Editor verwendet. Die Klasse implementiert alle interaktiven Befehle so, dass Textausgaben für den Benutzer erzeugt werden, und ist damit für einen Kommandozeilen-Editor geeignet. Außerdem ist mit `writeFlowfacts()` auch eine Methode zum Schreiben der Flowfacts vorhanden, die ganz allgemein und unabhängig vom Editor benutzt werden kann, beispielsweise auch durch WCC um zu einer erzeugten binären Objektdatei die Flowfacts hinzuzufügen. Die Methode `editFlowfacts()` führt das kommandozeilenbasierte Editieren aus. Dabei werden mit der GNU READLINE-Bibliothek [5] Befehle zeilenweise vom Benutzer entgegengenommen und an den Parser weitergereicht. READLINE ist eine verbreitete Bibliothek für Texteingaben, mit der die Eingabe komfortabel editiert werden kann und die außerdem eine History-Funktion besitzt.

Die Unterklasse `FlowfactBlockReader` ist im Gegensatz dazu nur in der Lage, blockweise im Speicher vorliegende Annotationen zu parsen und in eine bestehende LLIR-Darstellung einzutragen. Alle interaktiven Kommandos sind nur leere Stubs, auch `writeFlowfacts()` ist eine leere Funktion. Dafür kann die Klasse mit der `loadFromObject()`-Methode aus einem binären Objekt die Section mit den Annotationen auslesen.

Von der eingangs erwähnten Komponente zum Speichern wird durch `FlowfactBlockReader` das Lesen der Annotationen implementiert. Um Annotationen auch schreiben zu können, besitzt die `FlowfactConsole`-Klasse die `writeFlowfacts()`-Methode, deren Implementierung auf eine weitere Klasse zurückgreift: `BinaryObjectCopy`. Diese auf das Programm `objcopy` aufbauende Klasse erlaubt es, durch Kopieren eines binären Objekts Sections hinzuzufügen und zu ersetzen. Dazu werden die Sections mit der `addSection()`-Methode hinzugefügt und anschließend mit `writeObject()` eine Kopie des binären Objekts in eine Datei geschrieben. Als Objekt kann dabei sowohl eine Instanz von `BinaryObject` als auch eine externe Objektdatei benutzt werden, wodurch die Klasse sowohl beim Editieren als auch beim nachträglichen Einfügen von Flowfacts in kompilierte Objekte eingesetzt werden kann. Zusätzlich enthält die Klasse mit `createTemporaryFile()` und `insertTemporaryFile()` Hilfsmethoden, um den Umgang mit temporären Dateien zu erleichtern.

Als Front-End für den Benutzer wurde abschließend der eigentliche Flowfact-Editor in Form des Kommandozeilenprogramms `objff` geschrieben, das die Datei eines binären ARM-Objekts mit der `BinaryObject`-Unterklasse `ArmBinaryObject` öffnet, mit `FlowfactBlockReader` alle bereits eingetragenen Flowfacts ausliest und mit `FlowfactConsole` das Editieren und Schreiben ermöglicht.

Kapitel 6

Funktionstest

6.1 Vergleich von `obj2llir` mit `asm2llir`

Für den direkten Vergleich von `asm2llir` mit `obj2llir` wurde als Test-Eingabe ein Quelltext entworfen, der die Instruktions-Codes aller unterstützten ARM-Befehle und jedes ihrer Operandentypen erzeugt. Dieses Programm ist nicht dazu gedacht, ausgeführt zu werden, sondern ist schlicht eine Liste dieser Maschinenbefehle. Übersetzt man den Quelltext, dann kann man mit dem Kompilat (hier entweder ein Binärobjekt oder Assembler-Code) die folgenden zwei Tests durchführen:

1. Benutzt man die Kompilate als Eingabe für `asm2llir` und `obj2llir`, dann sollten beide Programme eine LLIR erzeugen, die dieselbe Folge von LLIR-Instruktionen mit denselben Operanden enthält. Unterschiede weisen auf einen Fehler in einem der Programme hin, wobei zu beachten ist, dass der GNU Assembler weitere Transformationen auf dem Assembler-Code durchgeführt haben könnte (beispielsweise die Übersetzung von Pseudobefehlen).

2. Die binäre Objektdatei kann außerdem zum Testen der Disassembler-Muster verwendet werden. Dazu sind alle Instruktions-Codes in der Test-Eingabe doppelt vorhanden, wobei für die Parameter wie den „Condition Code“, Flags und Operanden möglichst unterschiedliche Werte gewählt wurden. Viele Fehler in den Mustern wie falsche Bereiche lassen sich leicht entdecken, indem man die ausgegebene LLIR mit dem Quelltext oder der Ausgabe eines anderen Disassembler wie dem von `objdump` vergleicht. Setzt man voraus, dass `asm2llir` korrekt funktioniert, dann zeigen sich diese Fehler bereits im ersten Test.

Der relevante Inhalt des Quelltextes der Test-Eingabe besteht aus willkürlich angeordneten, handgeschriebenen Assembler-Befehlen, da dies der naheliegendste Weg ist, um die gewünschten Instruktions-Codes zu erhalten. Da `asm2llir` aber dazu entworfen wurde, die Assembler-Ausgabe eines mit GNU Gcc übersetzten C-Quelltextes zu verarbeiten, wurde

das Testprogramm trotzdem in C geschrieben und besteht im Wesentlichen aus Inline-Assembler-Ausdrücken.

Um für beide Parser eine möglichst hohe Test-Abdeckung zu erreichen, muss jede einzelne LLIR-Instruktion erzeugt werden. Da der Disassembler für jede Instruktion und jeden Operandentyp dieser Instruktion ein eigenes Muster benutzt, wird für jeden Operandentyp ein Beispiel erzeugt. Beim oben in Punkt 2 genannten Verdoppeln der Instruktionen zur Erhöhung der Test-Abdeckung beim Disassembler wurden vor allem – wenn möglich – binär inverse Werte gewählt. Dabei ist sichergestellt, dass jedes Bit von jedem dieser Werte in der Test-Eingabe sowohl auf 0 als auch auf 1 gesetzt wird. Beim Verdoppeln wurden die Parameter des zweiten Befehls dabei folgendermaßen aus dem Ersten abgeleitet bzw. für beide Befehle zueinander passend gewählt:

- Für jedes Register `R`(n) wird `R`(15-n) verwendet.

- Der „Conditional Code“, der die Bedingung für die Ausführung eines Befehls festlegt (vgl. Abschnitt 2.3.3), ist für doppelte Befehle immer aus der folgenden Menge der Kombinationen binär inverser Codes gewählt:

$$\{(\texttt{NE},), (\texttt{CS},\texttt{LE}), (\texttt{CC},\texttt{GT}), (\texttt{MI},\texttt{LT}), (\texttt{PL},\texttt{GE}), (\texttt{VS},\texttt{LS}), (\texttt{VC},\texttt{HI}),$$
$$(,\texttt{NE}), (\texttt{LE},\texttt{CS}), (\texttt{GT},\texttt{CC}), (\texttt{LT},\texttt{MI}), (\texttt{GE},\texttt{PL}), (\texttt{LS},\texttt{VS}), (\texttt{HI},\texttt{VC})\}$$

  Das erste Element jedes Tupels ist dabei der „Conditional Code“ für den ersten, ursprünglichen Befehl und das zweite Element der Code für den verdoppelten Befehl. Leider gibt es kein binär inverses Gegenstück für den Code „`EQ`“, sodass auch die nicht inversen Paare (`EQ`, `AL`) und (`AL`, `EQ`) gelegentlich verwendet werden, was aber erst ab der dritten Wiederholung desselben Befehls geschieht. „`AL`“ ist der „Conditional Code“ für die unbedingte Ausführung, der die Standardvorgabe ist falls kein Code angegeben wird. „`AL`“ explizit zu verwenden testet `asm2llir` auf Vollständigkeit.

- Für eine Konstante als zweiten (oder einzigen) Operanden einer „Data Processing“-Instruktion werden die zwei Werte #0x00000082 und #0x00000174 verwendet, da im kodierten Befehl Konstanten durch einen nur 8-bit-breiten Betrag angegeben werden, der auf zwei binären Stellen genau nach rechts rotiert werden kann – eine beliebige 32-Bit-breite Konstante kann nicht in einem selbst nur 32-Bit-breiten Maschinenbefehl kodiert werden. Durch die gewählten Konstanten soll erreicht werden, dass die beiden in den Maschinenbefehl kodierten 8-Bit- und 4-Bit-Beträge vom Assembler auf die binär inversen Werte (0x82,0) und (0x7d,15) gesetzt werden.

- Für ein Register als zweiten Operanden einer „Data Processing“, „FSR Transfer“ oder „Single Data Transfer“-Instruktion mit einer Konstante als Shift-Betrag werden #1 und #30 für den 5 Bits breiten Betrag verwendet.

- Für Konstanten mit einem zusätzlichen Vorzeichen-Bit („Up/DownBit'), wie sie als 8-Bit-Wert in „Halfword Data Transfer"-Instruktionen oder als 12-Bit-Wert in „Single Data Transfer"-Instruktionen vorkommen, wird jeweils #+1 und #-254 bzw. #-4094 verwendet.
- Für die Register-Menge eines „Block Data Transfer" werden jeweils {`R0`,`R2-R14`} und {`R1`,`R15`} verwendet.
- Das S-bit und andere Flag-Bits werden gekippt.

6.2 Framework-Einbindung

Da bei der technischen Umsetzung darauf geachtet wurde, bis auf weiteres die bestehenden Schnittstellen beizubehalten, d. h. mit `asm2llir` ein externes Programm anzubieten, das genauso wie `gcc2llir` die Assembler-Ausgabe von GCC einliest und in der Text-Repräsentation der LLIR wieder ausgibt, war `asm2llir` unmittelbar integriert durch eine entsprechende Änderung des aufzurufenden Kommandos bzw. durch Setzen eines symbolischen Links von `gcc2llir` auf `asm2llir`. `obj2llir` ließ sich ebenfalls leicht in WCC integrieren, da das Programm dieselbe Schnittstelle wie `asm2llir` bietet, d. h. es musste vor allem das Einbinden der externen Objektdateien verhindert werden, da deren Programmcode bereits in der LLIR-Darstellung enthalten ist. Auch die Ausgabe der Flowfacts über die `FlowFactConsole`-Klasse aus Abschnitt 5.3 ist möglich.

Kapitel 7

Fazit

7.1 Zusammenfassung

Das Gesamtziel dieser Arbeit ist es, die Analyse von binären Objektdateien zu erleichtern: Es sollte eine Möglichkeit geschaffen werden, den Programmcode aus Objektdateien genauso mit Flowfacts zu annotieren wie der zu kompilierende Quelltext bereits annotiert wird, um die Optimierung eines Programms durch WCC zu verbessern.

Dazu wurde ein Flowfact-Editor entwickelt, mit dem ein Binärobjekt eingelesen und annotiert werden kann und die Annotationen außerdem in der entsprechenden Objektdatei in einer eigenen Section abgelegt werden. Der Editor ist kommandozeilenbasiert und kann sowohl neue Annotationen erstellen als auch die bereits existierenden Annotationen einer Objektdatei bearbeiten. Dadurch ist es möglich, nahezu beliebige Objektdateien zu annotieren.

Durch den Objekt-Parser und den Disassembler, die auch vom Flowfact-Editor verwendet werden, kann der Maschinencode aus einem Binärobjekt im Allgemeinen wieder in die LLIR-Zwischendarstellung des WCC übersetzt werden. Dabei können in der Objektdatei vorliegende Flowfacts ebenfalls ausgelesen und der LLIR-Darstellung hinzugefügt werden, sodass der Programmcode aus einem annotierten Binärobjekt sowohl durch das Analysewerkzeug aiT während der Übersetzungsläufe analysiert als auch durch WCC selbst optimiert werden kann, d. h. im Gegensatz zu vorher können nun auch binär vorliegende Code-Teile optimiert werden.

Der Disassembler arbeitet dabei weitestgehend automatisch, aber es zeigt sich auch, dass durch die verwendeten Heuristiken dem Disassembling Grenzen gesetzt sind. Nicht nur indirekte Sprünge, die keiner der Heuristiken entsprechen, können nicht aufgelöst werden. Auch die Unterscheidung zwischen direkten Sprüngen, Funktionsaufrufen und Funktionsrücksprüngen muss sich darauf verlassen können, dass gewisse Konventionen eingehalten wurden. Außerdem benötigt der Disassembler zwingend Informationen darüber, welche

Speicherbereiche ARM-Code und welche Thumb-Code enthalten und erwartet, dass Bereichsgrenzen von Datenobjekten eingehalten werden.

Einer Reihe dieser Probleme ließ sich durchaus begegnen: Indirekte Sprünge lassen sich sicher erkennen und der Flowfact-Editor bietet zusätzlich die Fähigkeit, auch Sprungbefehle mit Sprungzielen zu annotieren. Die Nicht-Einhaltung einer prozeduralen Struktur lässt sich bei der Kontrollflussrekonstruktion durch Kopieren beheben. Andererseits kann beispielsweise vollkommen frei von Hand programmierter Assembler-Code, der nicht ausreichend strukturiert ist, zum gegenwärtigen Zeitpunkt *nicht* eingelesen werden, und es ist außerdem möglich, Code so zu gestalten, dass unerkannte Fehler auftreten, die zu einer LLIR-Darstellung führen die zwar syntaktisch gültig ist, aber semantisch nicht dem Binärobjekt entspricht. Von Compilern wie Wcc oder GNU Gcc erzeugte Binärobjekte lassen sich aber verarbeiten.

7.2 Ausblick

Der derzeitige Disassembler führt zwar erfolgreich einen „Linear Sweep“ mit compilerabhängigen Heuristiken durch, wobei bereits erprobte Heuristiken speziell für Binärobjekte etwas verallgemeinert und die Rekonstruktion des Kontrollflussgraphen etwas robuster gestaltet wurden, aber wie bereits in der Zusammenfassung erwähnt ist die Anzahl der unterstützten Binärobjekte trotzdem deutlich eingeschränkt. Der nächste Schritt könnte daher darin bestehen, durch eine Datenflussanalyse noch mehr Informationen über die Funktionen aus einem Binärobjekt zu erlangen. Indirekte Sprünge würden sich so eingrenzen lassen und manche Überschreitungen von Speicherbereichen würden erkannt. Auch die Unterscheidung zwischen ARM- und Thumb-Code könnte dadurch weiter verbessert werden, sodass insgesamt mehr Binärobjekte erfolgreich eingelesen oder zumindest abgelehnt werden könnten. Dem Ziel, den Benutzer so weit wie möglich zu entlasten, käme eine automatisierte Analyse entgegen, da beispielsweise der Flowfact-Editor bei indirekten Sprüngen eine Auswahl möglicher Sprungziele anbieten könnte.

Abbildungsverzeichnis

Literaturverzeichnis

[1] *Bison - GNU parser generator.* http://www.gnu.org/software/bison/, September 2013.

[2] *Boomerang: A general, open source, retargetable decompiler of machine code programs.* http://boomerang.sourceforge.net/, August 2013.

[3] *GCC, the GNU Compiler Collection.* http://gcc.gnu.org/, September 2013.

[4] *GNU Binutils.* http://www.gnu.org/software/binutils/, August 2013.

[5] *The GNU Readline Library.* http://cnswww.cns.cwru.edu/php/chet/readline/rltop.html, September 2013.

[6] Aho, Alfred V.: *Compiler: Prinzipien, Techniken und Werkzeuge.* Pearson Studium, 2. Auflage, 2008.

[7] ARM Limited: *ARM7TDMI Technical Reference Manual, Revision r4p1*, 2004.

[8] ARM Limited: *ARM Architecture Reference Manual*, 2005.

[9] ARM Limited: *ELF for the ARM® Architecture*, Nov. 2012.

[10] Bardin, Sébastien, Philippe Herrmann und Franck Védrine: *Refinement-based CFG Reconstruction from Unstructured Programs.* Technischer Bericht, CEA, LIST, Gif-sur-Yvette CEDEX, 91191 France, 2011.

[11] Böckenkamp, Adrian: *Untersuchung des Code-Location-Problems in C-/C++-Programmen für eingebettete x86- und ARM-Architekturen.* Bachelor-Arbeit. Technische Universität Dortmund, September 2011.

[12] Falk, Heiko und Paul Lokuciejewski: *A compiler framework for the reduction of worst-case execution times.* Journal on Real-Time Systems, 46(2):251–300, Oktober 2010. DOI 10.1007/s11241-010-9101-x.

[13] Heckmann, Reinhold und Christian Ferdinand: *Worst-Case Execution Time Prediction by Static Program Analysis.* Technischer Bericht, AbsInt Angewandte Informatik GmbH, 2004.

[14] Johnson, Stephen C. und Ravi Sethi: *UNIX Vol. II.* Kapitel Yacc: a parser generator, Seiten 347–374. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[15] Kinder, Johannes und Helmut Veith: *Jakstab: A Static Analysis Platform for Binaries.* Technischer Bericht, Technische Universität Darmstadt, 2008.

[16] Kinder, Johannes, Florian Zuleger und Helmut Veith: *An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries.* Technischer Bericht, Technische Universität Darmstadt, Technische Universität München, 2009.

[17] Levine, John R.: *Linkers and Loaders.* Morgan Kaufmann, 2000.

[18] Marwedel, Peter: *Eingebettete Systeme.* Springer, Berlin [u.a.], 2008.

[19] Patterson, David A. und John L. Hennessy: *Rechnerorganisation und -entwurf*, Kapitel 2: Befehle: Die Sprache des Rechners, Seiten 36–131. Elsevier, Spektrum Akad. Verl., 3 Auflage, 2005.

[20] The Santa Cruz Operation: *System V Application Binary Interface*, 1997.

[21] Schulte, Daniel: *Flow Facts für WCET-optimierende Compiler - Modellierung und Transformation.* VDM Verlag, November 2007.

[22] Theiling, Henrik: *Extracting Safe and Precise Control Flow from Binaries.* Technischer Bericht, Universität des Saarlandes and AbsInt Angewandte Informatik GmbH, 2000.

[23] Tidorum Limited: *Bound-T time and stack analyser, Application Note ARM7*, Februar 2010.