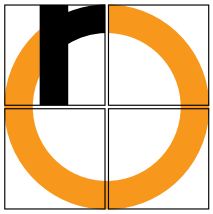

Hochschule Rosenheim
Fakultät für Informatik

Bachelorarbeit

Bachelorprüfung WS 2015 / 2016

Marius Schuller

Kaskadierung von Build-Systemen

Erstprüfer: Prof. Dr. Gerd Beneken
Zweitprüfer: Prof. Dr. Jochen Schmidt

Erklärung

Ich versichere, dass ich diese Arbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, den 9. März 2016

Marius Schuller

Kurzfassung

In dieser Arbeit werden Untersuchungen zur Abhängigkeitsauflösung kaskadierter Software-Projekte unternommen. Es werden Auswirkungen von falschen und fehlenden Abhängigkeiten ineinander verschachtelter Projekte herausgestellt. Unter diesen beiden Gesichtspunkten wurde verschiedene Build-Tools verglichen. Dabei wurden alternative Build-Managment-Werkzeuge neben dem traditionell verwendeten Make gefunden, die für eine Lösung des zugrunde liegenden Problems geeignet wären. Allerdings zeigte sich, dass ein Wechsel von Make-basierten Build-Infrastrukturen zu alternativen Build-Management-Werkzeugen mit hohem Migrationsaufwand einher geht. Um eine zweckmäßige Lösung für bestehende Make-basierte Build-Infrastrukturen bieten zu können wurde eine Erweiterung von Make angestrebt. Der in dieser Arbeit entwickelte Lösungsansatz verbessert Make hinsichtlich der Auflösung projektübergreifender Abhängigkeiten. Weiter zeigt sich, dass dieser Ansatz nicht nur einen geringen Migrations- und Pflegeaufwand für bestehende Make-basiert Build-Systeme mit sich bringt, sondern auch deren Performanz durch Vermeidung von Rekursion erheblich verbessert.

Schlagworte: Build-System, Kaskadierung, Abhängigkeiten, Make, Bazel, SCons

Inhaltsverzeichnis

Begriffe, die im Glossar erklärt werden, sind bei der ersten Nennung im Text *kursiv* dargestellt.
Dateinamen, Programm- und Funktionsaufrufe werden in `nichtproportionaler` Schriftart dargestellt.

Abbildungsverzeichnis

Quelltextausschnitte

1 Einleitung

Die Verwendung bestehender Software-Projekte für neue Projekte gewinnt immer mehr an Bedeutung, da viele Problemstellungen bereits gelöst wurden, und sich damit wertvolle Entwicklungszeit einsparen lässt. Dieser Umstand wird dadurch befördert, dass immer mehr Software unter freien und offenen Lizenzen veröffentlicht wird [Des08]. Dadurch erhöht sich im Gegenzug die Komplexität von Software-Projekten durch zusätzliche Abhängigkeiten. Diese Erhöhung der Komplexität beeinflusst die Performanz des Build-Systems dieser Projekte negativ. Dieser Einfluss wird noch verstärkt, wenn die neu hinzugekommenen Abhängigkeiten nicht korrekt integriert werden.

1.1 Kontext der Arbeit

In diesem Abschnitt soll nach einer kurzen Vorstellung der genua gmbh[1] der Kontext dieser Arbeit erklärt werden. Bei der Firma genua werden Sicherheitslösungen für Unternehmen und Behörden entwickelt, die auf dem offenen und sicheren Betriebssystem *OpenBSD* basieren. Die Produktpalette umfasst Firewall- und VPN-Appliances, Fernwartungslösungen, ein zentrales Managementsystem und ein Security-Notebook.

Derzeit wird von jedem Entwicklungsteam aus historischen Gründen eine eigene Kopie der OpenBSD-Quelltexte gepflegt. Jedes Team verfügt jeweils über

- die Quelltexte produktspezifischer Funktionen,
- die Quelltexte von OpenBSD, sowie
- ein eigenes Build-System.

Aus diesem Sachverhalt gestaltet sich die aktuelle Entwicklungsstruktur wie in Abbildung 1.1 beschrieben. Grau hinterlegt sind die Entwicklungsgruppen, die einzelnen Produkte sind mit grünen Rechtecken gekennzeichnet, in den gelben Rechtecken sind die unterliegenden Betriebssysteme und in blauen Rechtecken andere Software-Komponenten dargestellt.
Das Team Secure-Gateway-Development (SGD) ist für die Firewall *genugate* verantwortlich, Crypto-System-Development (CSD) für den Paketfilter *genuscreen*, die Fernwartungs-Appliance *genubox* und die Remote-Arbeitslösung *genucard*. Letztere drei sind in Abbildung 1.1 unter dem Begriff genuzoo zusammengefasst. Weder SGD, noch CSD haben eigene

1 `https://genua.de`

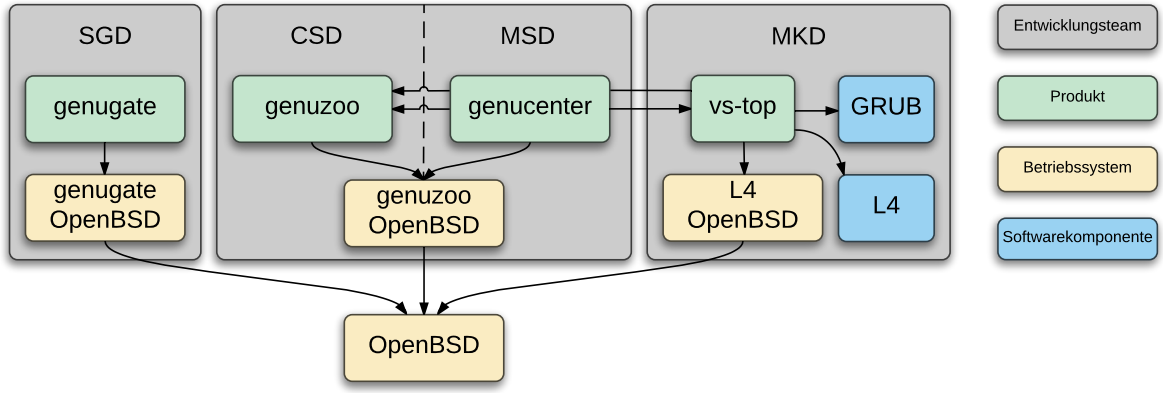

Abbildung 1.1 Entwicklungsstrukturen bei genua, Ist-Zustand

Abhängigkeiten zu anderen Produkten. Bei dem Team Micro-Kernel-Development (MKD) wird der Security-Laptop *vs-top* entwickelt, der eine genucard als Firewall verwendet und daher eine Abhängigkeit von CSD aufweist. Die Entwicklungsgruppe Management-System-Development (MSD) hingegen arbeitet am Produkt *genucenter*, der Management-Appliance, über die genuscreens, genuboxen und vs-tops verwaltet werden. Daraus ergeben sich produktbedingt stets Abhängigkeiten zu CSD und MKD, die auch durch eine Veränderung der Entwicklungsstrukturen nicht aufgehoben werden können.

Projektabhängigkeiten am Beispiel des vs-top

Am einfachsten lassen sich die Abhängigkeiten anhand des vs-top zeigen, die schematisch in Abbildung 1.2 dargestellt sind.

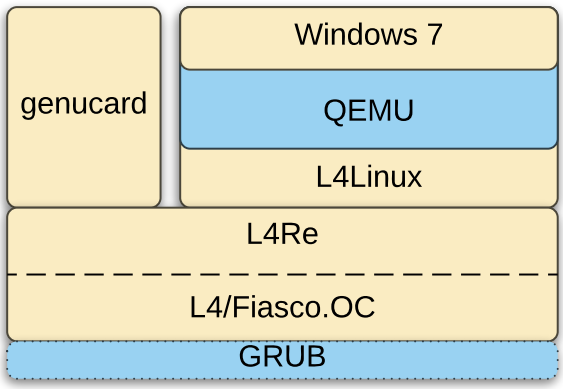

Abbildung 1.2 Software-Architektur des vs-top

Für den Boot des Systems kommt die Softwarekomponente GNU GRUB 2[2] zum Einsatz. Da diese Komponente lediglich beim Start zum Einsatz kommt und nicht dauerhaft in Betrieb ist, wurde sie punktiert umrahmt. Über GRUB 2 wird das *Microkernel*-basierte

2 `http://www.gnu.org/software/grub/`

Separationssystem *L4*[3] geladen. Darüber ist es möglich, mehrere Betriebsysteme auf einer Hardware, ähnlich zur Paravirtualisierung [Här08, Hei10], parallel zu betreiben. Hierfür werden sogenannte Compartments verwendet, die mit der Funktionalität von FreeBSD Jails[4] oder OpenVZ[5] für Linux vergleichbar sind.

Im Fall des vs-top wird in einem von zwei Compartments eine genucard installiert, die auf einem an die Separationsumgebung angepassten OpenBSD basiert. Weiter wird im zweiten Compartment ein an L4 angepasstes Linux betrieben, auf dem über die Virtualisierungs- und Emulationssoftware QEMU[6] ein Microsoft Windows 7 virtualisiert wird.

Für die Installation eines vs-tops muss ein bootfähiges Installationsmedium erzeugt werden, das Softwaremodule aus verschiedenen Entwicklungsbereichen und Quelltext-Repositories bezieht. Ein grober Überblick über die verwendeten Module wird in Abbildung 1.3 gegeben:

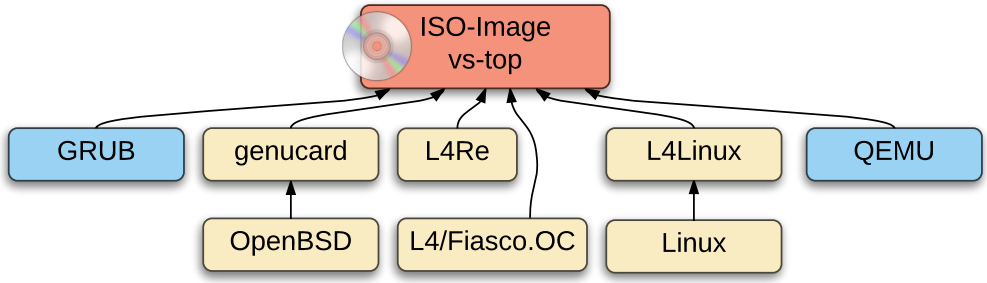

Abbildung 1.3 Modulabhängigkeiten des vs-top

Wie beschrieben, basieren alle Produkte auf OpenBSD und fügen jeweils eigene Funktionalitäten hinzu. Dieses zugrunde liegende Betriebssystem erscheint alle sechs Monate[7] in einer neuen Version, auf welche jedes Team die jeweiligen produktspezifischen Änderungen von neuem migrieren muss. Der zeitliche Aufwand für ein Upgrade entspricht je Team im Durchschnitt etwa zwei Wochen, und entspricht bei zwei Aktualisierungen pro Jahr demnach etwa einem ganzen Monat.

Veränderungen zur Reduzierung der Team-eigenen OpenBSD-Kopien werden von genua bereits vorgenommen. Dafür wurde durch die Verantwortlichen eine Vision, der Soll-Zustand, erarbeitet. Eine mögliche Veränderung der Strukturen ist in Abbildung 1.4 zu sehen. Wie zuvor sind Entwicklungsgruppen grau hinterlegt, Produkte in grünen, die geplante, zentralisierte Upgrade-Struktur des Betriebssystems in gelben und weitere Software-Komponenten in blauen Rechtecken dargestellt.

Der Grundstein struktureller Optimierung an dieser Stelle ist die Schaffung einer neuen Entwicklungsgruppe, die sich zur Entlastung anderer Teams ausschließlich mit dem

3 https://os.inf.tu-dresden.de/fiasco/
4 https://www.freebsd.org/doc/handbook/jails.html
5 https://openvz.org/Main_Page
6 http://wiki.qemu.org/Main_Page
7 http://www.openbsd.org/faq/faq1.html#Next

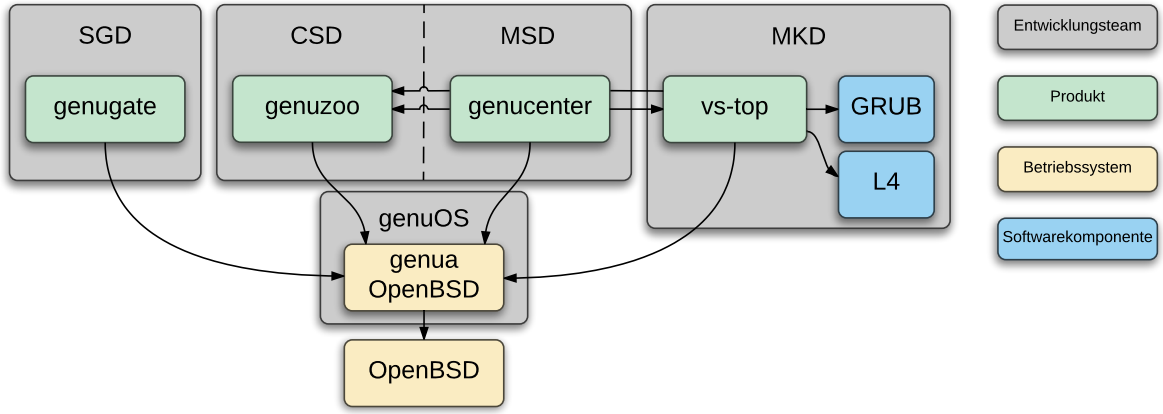

Abbildung 1.4 Vision der zukünftigen Entwicklungsstruktur

genua-eigenen OpenBSD, dem hier so genannten *genuOS*, dessen produktspezifischen Anforderungen und der halbjährlichen Aktualisierung von OpenBSD beschäftigt. Zusammen mit der Vereinheitlichung der Produktgrundlage entsteht damit sowohl die Herausforderung als auch die Chance von vielen Build-Systemen auf eine firmenweit einzige, und damit einheitliche Build-Umgebung zu wechseln und alle Quelltexte in einem zentralen Versionsverwaltungssystem (VCS) zu verwalten.

Zu diesem letzten Punkt wurde in den vergangenen Jahren unter anderem durch die Firma Google in diversen Vorträgen und Talks bereits mehrfach gezeigt, dass auch sehr große Mengen Code in einem monolithischen Ansatz mit einem Quelltextbaum und mit einem einzigen Build-System effizient verwaltet werden können [Web12, Web14]. Unter einem monolithischen Build-System ist eine zentrale Build-Struktur zu verstehen, an der alle Quelltextdateien, meist in einem einzigen VCS verwaltet und gebaut werden. Durch intelligentes Caching und verteilte Infrastruktur hat Google mit ihrem selbst entwickelten Build-Management-Werkzeug mit dem Namen Bazel[8] in den letzten Jahren trotz stetig steigender Zeilen Quelltext konstante und fallende Build-Zeiten erreicht.

Ein solcher Ansatz wird langfristig auch von genua angestrebt, wobei die kleinere Codebasis ein großer Vorteil ist, um einen ähnlichen Zustand mit deutlich weniger Aufwand zu erreichen.

1.2 Ziel der Arbeit

Es soll langfristig eine zentrale Build-Umgebung aufgebaut werden, die, trotz verschiedener, teilweise selbstständiger Unterprojekte, alle oder möglichst viele Abhängigkeiten atomar auflösen kann. Dafür sollen im Rahmen dieser Arbeit verschiedene bewährte, aber auch aktuelle Entwicklungen im Bereich der Build-Management-Werkzeuge untersucht werden.

8 `http://bazel.io`

Der aktuelle Zustand ist in Abbildung 1.5 abstrakt dargestellt. Darin wird das Grundproblem gezeigt, bei dem das Wissen der Abhängigkeiten innerhalb einzelner lokaler Projekte – grau hinterlegt – beschränkt ist und keine globale Kenntnis aller Abhängigkeiten existiert.

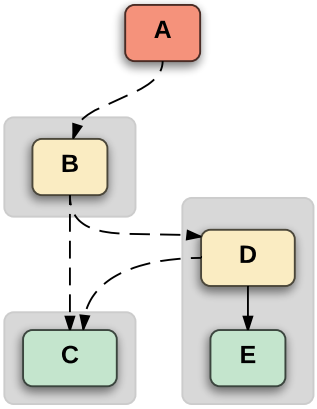

Abbildung 1.5 Mehrere, lokale Abhängigkeitssichten

In Abbildung 1.6 ist der Sollzustand abgebildet, bei dem jede Abhängigkeit eindeutig auch über die verschiedenen Projekte hinweg aufgelöst werden kann.

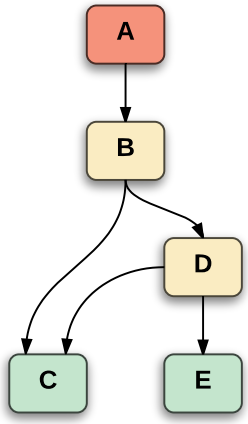

Abbildung 1.6 Einzelne Sicht auf alle Abhängigkeiten

Um in dem Prozess der Zentralisierung der Entwicklungsinfrastrukturen bei genua die Auswahl des Build-Systems zu vereinfachen, soll in dieser Arbeit untersucht werden, ob es mit aktuellen Build-Management-Werkzeugen möglich ist, Abhängigkeiten aufeinander aufbauender, sowie ineinander kaskadierter Softwareprojekte komplett zu erkennen und nichtrekursiv aufzulösen.

2 Grundlagen

Zunächst sollen die grundlegenden Konzepte von Build-Systemen erläutert und sprachliche Unterscheidungen aufgezeigt werden, um die verwendeten Begriffe klar zu unterscheiden. Oft werden im Kontext dieses Themas Begriffe untereinander austauschbar verwendet, was zu Unklarheiten führt. Die Terminologie dieser Arbeit orientiert sich an der des Buches "Software Build Systems" [Smi11].

Die Bezeichnung "Build-System" ist zu allgemein und beschreibt in dieser Arbeit den gesamten Software-Komplex, der für einen erfolgreichen Build notwendig ist. Unter dem Build versteht sich der Prozess der Erzeugung von ausführbaren Programmen aus Quelltextdateien. Gleichzeitig ist das Build-System aber selbst auch ein Programm in diesem Komplex. Aus diesem Grund wird das Kernstück des Build-Systems in dieser Arbeit als Build-Management-Werkzeug bezeichnet. Darunter ist jede Software zu verstehen, die in der Lage ist, einen azyklischen Baum von Abhängigkeiten zu erstellen und zu verarbeiten. Dem Build-Management-Werkzeug steht meist eine Menge aus verschiedenen weiteren Programmen, den sogenannten Kompilations-Werkzeugen, zur Seite.

Im Wesentlichen ist ein Build-Management-Werkzeug ein regelbasiertes Expertensystem. Darunter versteht man Computersysteme, die Menschen bei komplexen Problemen unterstützen, indem sie auf eine existierende Wissensbasis zurückgreifen. Dieses Wissen wird in Form von Regeln durch einen menschlichen Experten vorab bereitgestellt und ermöglicht es dem System darauf basierende Schlussfolgerungen zu treffen. Im Fall von Build-Management-Werkzeugen werden Regeln definiert, um Abhängigkeiten zwischen Quelltext, eventuellen Zwischenprodukten und ausführbaren Programmen abzubilden.

Funktional müssen Build-Management-Werkzeuge aufgrund der Existenz, Abwesenheit oder des Erzeugungszeitpunkts von Build-Produkten sowie deren Abhängigkeiten eine möglichst effiziente Bearbeitungsreihenfolge erarbeiten. Diese einzelnen Bearbeitungsschritte werden dann durch die in den Regeln beschriebenen Programme delegiert.

Der grundlegende Ablauf eines Builds besteht folglich aus drei Bearbeitungsschritten:

1. Bestimmung aller Abhängigkeiten und Erzeugung eines Abhängigkeitsgraphen
2. Identifizierung fehlender und veralteter Build-Produkte seit dem letzten Build
3. Ausführung der Programme zur Erneuerung zuvor identifizierter Build-Produkte

Wurde noch kein Build ausgeführt, so existieren auch keinerlei Build-Produkte, und das Build-Management-Werkzeug führt die entsprechenden Regeln aus, um diese zu erzeugen.

2.1 Komponenten eines Build-Systems

Der Begriff Build-System wird immer dann verwendet, wenn von einem Build-Management-Werkzeug und einer Sammlung diverser Programme gesprochen wird, mit denen menschenlesbarer Quelltext zu einem computerlesbaren Programm übersetzt werden kann. Daneben kann ein solches System aber noch viele weitere Funktionen bieten, wie etwa das Packen fertig übersetzter Programme zu installierbaren Softwarepaketen, automatischer Generierung von Quelltextdokumentation aus Kommentaren oder statischer Quelltextanalyse. In Abbildung 2.1 ist ein allgemeiner Aufbau eines Build-Systems, aus "Software Build Systems" [Smi11] entlehnt, dargestellt.

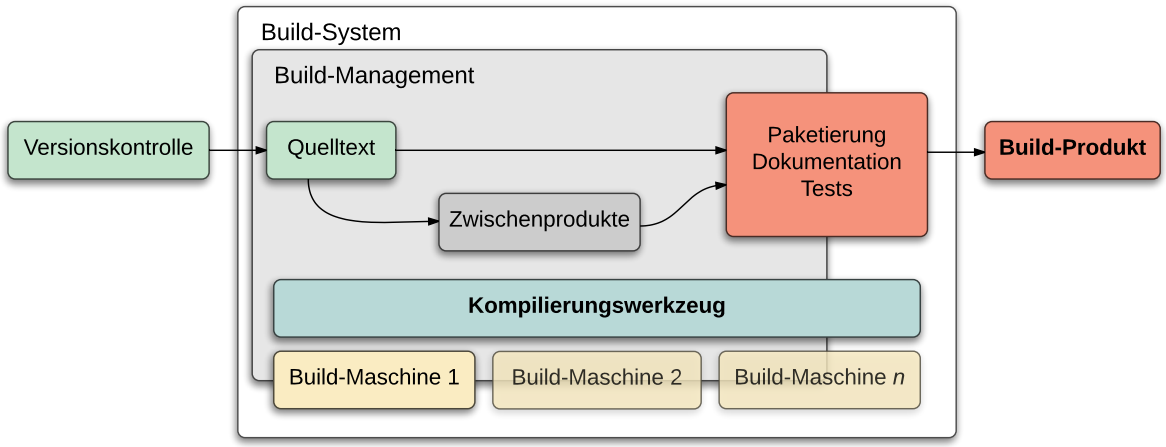

Abbildung 2.1 Generisches Build-System [Smi11, S. 20]

Der Quelltext-Baum wird aus einem VCS entnommen und durch den Aufruf des Build-Management-Werkzeugs erstellt dieses einen Abhängigkeitsgraphen, der nach den ihm bekannten Regeln die Quelldateien in Zwischen- und Endprodukte überführt. Das Build-System weiß nach der Verarbeitung des Abhängigkeitsgraphen, wie aus dem Quelltext das Build-Produkt erstellt werden kann, und ruft ein oder mehrere Kompilierungswerkzeuge auf, die die eigentliche Übersetzungsarbeit leisten.

Versionskontrollsysteme

Anhand von Versionskontrolle werden Änderungen an Dateien erfasst und mit einem Zeitstempel versehen. Zusätzlich wird der Entwickler gespeichert, der die Änderung vorgenommen hat. Anhand dieser Informationen ist es möglich, alle Versionen aus einem Archiv später wiederherzustellen. Wegen dieser Eigenschaften wird heute nahezu in jedem Softwareprojekt, aber auch außerhalb der reinen Entwicklung — beispielsweise im DevOp-Bereich — eine Art von Versionskontrolle eingesetzt. Durch dieses Werkzeug wird es auch erst möglich, effizient kollaborativ zu arbeiten. Neben der Nachvollziehbarkeit von Änderungen, oder der einfachen Verwaltung von verschiedenen Entwicklungszweigen zählt auch die Möglichkeit

der Versionierung zu den zahlreichen weiteren Funktionen. In der Vergangenheit wurden verschiedene Ansätze verfolgt, eine effektive Versionskontrolle zu realisieren. Beispiele für heutzutage verwendete VCS sind etwa git[9] , SVN[10] und Mercurial[11]. Allerdings werden auch noch historische Systeme zur Versionsverwaltung, wie CVS[12] im OpenBSD-Projekt, verwendet.

2.2 Abhängigkeitsgraph

Der Kern eines Build-Management-Werkzeugs ist ein gerichteter Graph ohne Zyklen der Beziehungen aufgrund vordefinierter Regeln darstellt. Damit werden die Abhängigkeiten zwischen den Quelltext-Dateien und ihren Zwischen- und Endprodukten abgebildet. Dabei sind Abhängigkeiten durch gerichtete Kanten und die Dateien mit Abhängigkeiten als Knoten beschrieben. Über den Kontext der Graphen ist es möglich, voneinander unabhängige Teile des Graphen zu erkennen und bei der Verarbeitung zu parallelisieren.

Aus der technischen Perspektive inkludieren Quelldateien ihre Header-Dateien meistens selbst, weswegen die in Abbildung 2.2 dargestellten Abhängigkeiten zwischen `foo.h` und den Zwischenprodukten `foo.o` sowie `bar.o` zunächst unlogisch erscheinten.

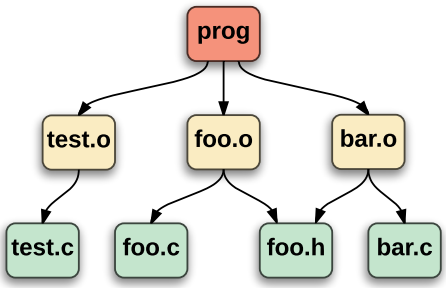

Abbildung 2.2 Konstruierter Abhängigkeitsgraph

Grund dafür ist jedoch, dass durch eine Änderung an Header-Dateien nicht die inkludierende C-Quelldatei neu generiert werden muss, sondern die Objektdatei, die aus beiden erstellt wird. Auf das als rotes Rechteck gekennzeichnete ausführbare Programm `prog` zeigt kein Abhängigkeitspfeil, da dies der Endpunkt des Build-Prozesses ist.

9 https://git-scm.com
10 https://subversion.apache.org/features.html
11 https://www.mercurial-scm.org
12 http://savannah.nongnu.org/projects/cvs

2.3 Grundregeln für Build-Systeme

Bei der Entwicklung und Konfiguration eines Build-Systems sollten nach Ludwig Hähne [Häh08] mindestens folgende vier Eigenschaften beachtet werden:

Einfach
Der Nutzen eines Build-Systems wird vor allem durch sein Maß an Unterstützung bei der Erstellung von Software bestimmt. Mit dazu gehört auch der Bereich Benutzerfreundlichkeit, der anhand des Funktionsumfangs, einer komfortablen Benutzerschnittstelle oder der Nachvollziehbarkeit von Entscheidungen bemessen wird.

Korrekt
Ein Build-System arbeitet dann korrekt, wenn es Kenntnis aller Abhängigkeiten hat und diese derart auflösen kann, dass Programmaufrufe nicht unnötig mehrfach oder überhaupt nicht ausgeführt werden. Kann nicht auf das richtige Verhalten des Build-Systems vertraut werden, wird meist ein kompletter Build durchgeführt um Probleme durch unvollständige Abhängigkeitsgraphen zu lösen [Mil98], wodurch das System an Performanz verliert.

Performant
Bei der Performanz eines Build-Systems spielt, die Laufzeit eines Builds, aber auch die Speichernutzung eine große Rolle. Dauert die Generierung des Abhängigkeitsgraphen sehr lang, etwa weil ein umfangreiches Regelwerk geprüft werden muss, so muss mindestens diese Zeit gewartet werden, um möglicherweise festzustellen, dass keine Datei verändert wurde und deswegen für das Build-Management-Werkzeug keine Aktion notwendig ist. Nach der Prüfung aller Regeln und der Erkennung und Sortierung von Arbeitsschritten wird die meiste Rechenleistung von Kompilierungswerkzeugen in Anspruch genommen.

Skalierbar
Software wird ständig weiterentwickelt, die unterliegenden Quelltexte werden mit der Zeit erweitert und die Programmlogik komplexer. Soll langfristig mit einem Build-System gearbeitet werden, muss es leicht erweiterbar und zuverlässig auch mit sehr komplexer Software umgehen können.

Wird bei der Entwicklung und Konfiguration eines Build-Systems nicht auf diese Eigenschaften bewusst Rücksicht genommen, kann das in der Zukunft zu hohen versteckten Kosten in Form von unnötig aufgewendeten Arbeitsstunden führen. Um Arbeiten wie Konfiguration, Fehlersuche oder Funktionserweiterungen an Build-Systemen vorzunehmen, verbringen Software-Entwickler laut Gary Kumfert und Tom Epperly im Durchschnitt 12% ihrer Arbeitszeit [Epp02]. Das heißt, bei einem Team von zehn Programmierern entspricht das einem Aufwand einer weiteren, ganzen Entwicklerstelle.

2.4 Build-Typen

In der Softwareentwicklung wird zwischen zwei Build-Arten unterschieden: Dem vollen Build, der alle Zwischen- und Endprodukte, auch Targets genannt, erzeugen muss und dem inkrementellen Build, welcher lediglich alle geänderten Dateien und deren Abhängigkeiten neu erzeugt. Weiter ist es möglich, den inkrementellen Build in einigen Fällen zu optimieren. Dieser dann sogenannte Subtarget-Build erzeugt schließlich nur die Programmbibliotheken neu, die von dem Endprodukte dynamisch eingebunden werden.

2.4.1 Voller Build

Wenn ein Quelltextbaum initial durch das Build-System übersetzt wird, werden alle Regeln ausgeführt werden, da keines der dort definierten Build-Produkte existiert. Somit werden alle Zwischen- und Endprodukte, wie zum Beispiel in Abbildung 2.3 zu sehen, erzeugt.

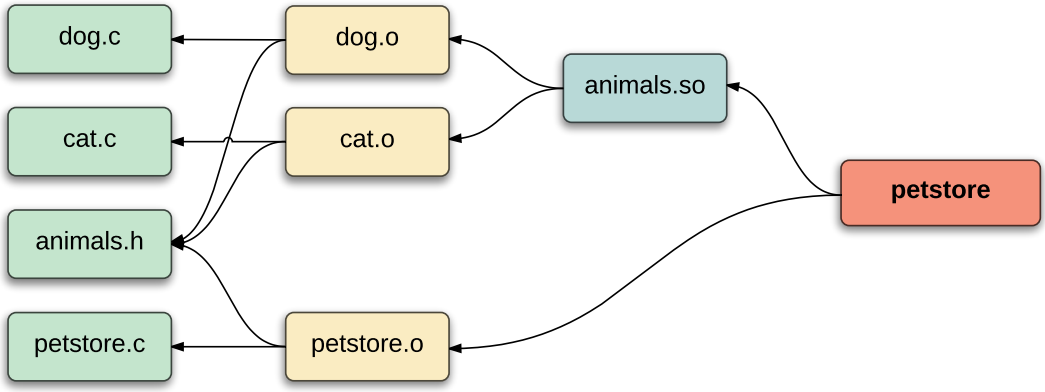

Abbildung 2.3 Generischer Abhängigkeitsgraph [Smi11, S. 307]

Die Dateien mit den Endungen `.c` und `.h` stammen aus dem VCS und sind die einzigen Dateien, an denen Änderungen von einem Entwickler vorgenommen werden sollen. Zwischenprodukte der Kompilierung, wie etwa `dog.o` und auch Bibliotheken und Programme, die aus diesen Zwischenprodukten entstehen, in Abbildung 2.3 die Dateien `animals.so` und `petstore`, werden nicht in einem VCS gespeichert.

2.4.2 Inkrementeller Build

Entwickler arbeiten in der Regel innerhalb eines komplett gebauten Software-Projekts und editieren dort einige wenige Quelltext-Dateien. Diese Änderungen müssen überprüft werden, wozu das Projekt erneut gebaut werden muss. Diese Schritte werden unter dem Begriff "Edit-Compile-Test"-Kreislauf geführt. Bei einem vollen Build werden also alle Zwischen- und Endprodukte erzeugt und Änderungen an Quelltextdateien führen nicht mehr zur Erzeugung aller dieser Produkte. Stattdessen wird ein sogenannter inkrementeller Build durchgeführt, der lediglich Abhängigkeiten der geänderten Dateien erneuert und damit

die Zeit eines Entwicklers in einem "Edit-Compile-Test"-Zyklus kurz hält. Bei größeren Projekten ist diese Art des Builds von großer Wichtigkeit, da dieser Build nur einen Bruchteil eines vollen Builds in Anspruch nimmt. In dem folgenden Beispiel in Abbildung 2.4 wird angenommen, dass ein voller Build im Vorfeld durchgeführt wurde.

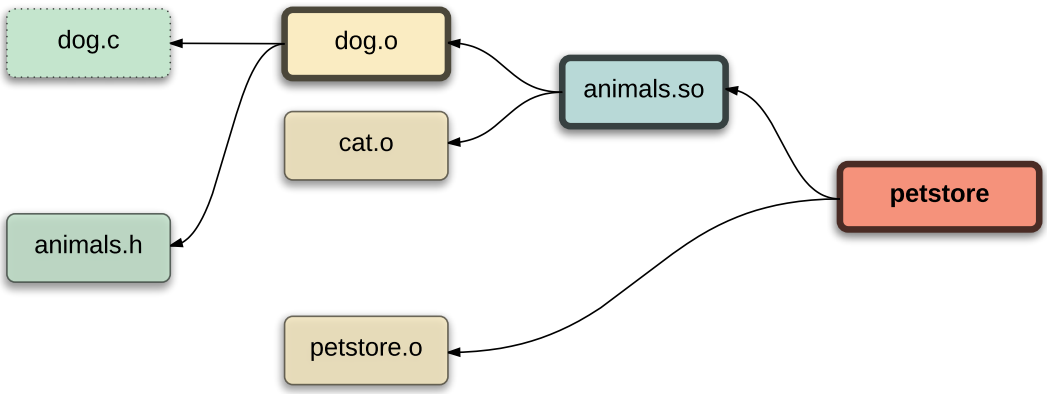

Abbildung 2.4 Abhängigkeitsgraph eines inkrementellen Builds [Smi11, S. 308]

Die Datei `dog.c` wurde verändert – durch eine gestrichelte Umrandung gekennzeichnet. Bei einem inkrementellen Build müssen nur `dog.o`, `animals.so` und `petstore` neu erzeugt werden. Die schattierten Dateien `cat.o` und `petstore.o` existieren bereits und werden lediglich verwendet, um die veralteten Build-Produkte neu zu übersetzen.

2.4.3 Subtarget-Build

Wie bereits beschrieben, ist der Subtarget-Build eine weitere Optimierung des inkrementellen Builds bei dem das ausführbare Programm nicht neu kompiliert wird, sondern sich auf die Neugenerierung der dynamischen Bibliotheken beschränkt. Im Beispiel in Abbildung 2.5 wurde dieses Mal die Datei `cat.c` verändert. Dies ist wieder an der gestrichelten Umrandung zu erkennen ist.

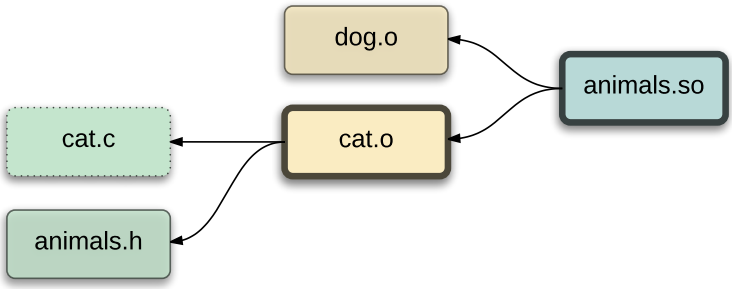

Abbildung 2.5 Abhängigkeitsgraph eines Subtarget-Builds [Smi11, S. 309]

Im Gegensatz zum inkrementellen Build wurde die ausführbare Datei `petstore` hier nicht neu erzeugt. Dazu muss das Programm dynamisch gelinkt sein, was bedeutet, dass ausserhalb des Build-Bereichs erst zum Ausführungszeitpunkt die nötigen Funktionen in den Arbeitsspeicher geladen werden. Diese Optimierung ist allerdings nur dann sinnvoll, wenn der inkrementelle Build zu lange dauert. Dies ist ein Zeichen von fehlender Modularität des Projekts und versucht dieses Problem zu umgehen, anstatt es zu beheben. Denkbar ist ein sinnvoller Einsatz bei komplexen Programmen, die über eine kompilierte GUI-Komponente verfügen. Sofern keine Änderung an der grafischen Oberfläche nötig ist, kann die Funktionalität im Hintergrund geändert werden, ohne die grafische Komponenten neu bauen zu müssen. Hierbei entsteht zusätzlich das Problem, dass zum Zeitpunkt der Kompilierung Änderungen an Programmierschnittstellen nicht in allen Teilen der Software bekannt sind. Erst bei der Ausführung treten schwer nachvollziehbare Fehler auf.

3 Untersuchte Build-Management-Werkzeuge

Nachdem die Grundlagen von Build-Systemen erläutert wurden, folgt in diesem Abschnitt eine Vorstellung von diversen Build-Management-Werkzeugen, die im Rahmen dieser Arbeit untersucht wurden. Um neben etablierten Programmen auch aktuelle Entwicklungen im Bereich der Build-Management-Werkzeuge vergleichen zu können, wurden jüngere Projekte, sowie auch Projekte mit neuartigen Ansätzen untersucht.

3.1 Definition von Anforderungen

Um untersuchte Programm anhand klarer Merkmale bewerten zu können werden folgenden Anforderungen definiert:

R.1 *Projektübergreifende Abhängigkeitsaufösung*
Wie in Abschnitt 1.2 beschrieben, muss es möglich sein, über verschiedene Projekte hinweg einen einzigen Abhängigkeitsgraph zu pflegen.

R.2 *Lizenzkosten*
Da zum aktuellen Zeitpunkt das frei lizensierte Make verwendet wird, sollen auch in Zukunft keine Lizenzkosten für den Build von genua-Produkten anfallen.

R.3 *Migrationsaufwand*
Durch die Verwendung eines neuen Build-Management-Werkzeugs entsteht ein bestimmter Aufwand. Dieser setzt sich aus Schulung von Entwicklern und der erstmaligen Konfiguration des Build-Systems in Form der Anpassung bestehender Build-Beschreibungen zusammen. Da sehr viele Projekte bei genua von einer Umstellung betroffen wären, soll dieser Aufwand möglichst klein sein.

R.4 *Pflegeaufwand*
Einige der bei genua verwendeten Softwaremodule Dritter (vgl. Abschnitt 1.1), wodurch in regelmäßigen Abständen neue Versionen erscheinen. Diese Aktualisierungen haben Einfluss auf die Build-Beschreibungen, sodass hier ein Pflegeaufwand entsteht. Aufgrund der Vielzahl an verschiedenen Softwareprojekte soll dieser Aufwand möglichst gering sein.

R.5 *Performanz*
Durch das zukünftige Build-Management-Werkzeugs soll die Build-Dauer reduziert werden.

R.6 *Ressourcen*
Die Ressourcenanforderungen des zukünftigen Build-Management-Werkzeugs soll in den Punkten Rechenleistung und Arbeitsspeicher nicht erhöht werden.

3.2 Make

Make ist mit seiner Veröffentlichung in 1978 eines der ältesten und heute noch immer am weitest verbreiteten, freien Build-Management-Werkzeuge [Fel79, Fel88]. Viele jüngere Build-Systeme arbeiten im Kern immer noch mit den gleichen Algorithmen zur Erzeugung des Abhängigkeitsgraphen wie Make. Die Grundfunktion gestaltet sich wie folgt: In einem sogenannten `Makefile` werden Regeln definiert, über die Abhängigkeiten zwischen Quelltext und daraus resultierenden Build-Produkten beschrieben sind. Zu jeder dieser Regeln wird zudem spezifiziert mit welchem Programmaufruf die Abhängigkeit aufgelöst werden kann. Make erzeugt aus den Regeln einen Abhängigkeitsgraph und arbeitet diesen ab. Mit dem Programm Make wird auf Unix-artigen Betriebssystemen historisch bedingt auch ein bereits vorgefertigtes, systemeigenes Standard-Regelwerk mitgeliefert [Fel79]. Unter dem Betriebssystem FreeBSD ist eine Sammlung von vordefinierten Regeln unter `/usr/share/mk/sys.mk` zu finden.

Der allgemeine Aufbau einer Make-Regel wird in der Quelltextauflistung 3.1 gezeigt, wobei der Unterstrich in Zeile 2 einen Tab darstellt. Fast alle Regelbeschreibungen von Build-Management-Werkzeugen werden nach diesem oder einem sehr ähnlichen Schema erstellt.

```
1 Target: Abhaengigkeiten
2 ␣␣␣␣␣␣␣␣Programmaufruf
```

Quelltext 3.1 Grundform einer Make-Regel

Unter Target versteht man den Dateinamen der Datei, die vom Programmaufruf dieser Regel erzeugt wird. Dies können beispielsweise ausführbare Dateien oder Vorprodukte dieser sein.

In derselben Zeile stehen nach einem Doppelpunkt und einem Leerzeichen eine Liste von Dateien von denen das Target abhängt. Anhand dieser, durch Leerzeichen getrennte, Dateiliste wird der Zeitstempel aller Abhängigkeiten mit dem Zeitstempel des Targets verglichen, sofern dieses bereits existiert.

In Zeile 2 steht nach einem Tab der Programmaufruf, mit Hilfe dessen ein Target neu erzeugt werden kann. Sollte eine Datei in der Abhängigkeitsliste dieses Targets einen neueren Zeitstempel aufweisen als das Target selbst, oder sollte das Target noch nicht existieren, so wird der Programmaufruf ausgeführt um das Target zu erzeugen.

In der Quelltextauflistung 3.2 ist eine minimale Regelkonfiguration von Make beschrieben. Das erste Target innerhalb eines `Makefile`s wird als Standard-Target verwendet und muss beim Aufruf von Make nicht explizit angegeben werden.

```
1 hello: hello.o
2 ␣␣␣␣␣␣␣␣gcc hello.o -o hello
3 hello.o: hello.c
4 ␣␣␣␣␣␣␣␣gcc -c hello.c
```

Quelltext 3.2 Zwei Make-Regeln

Wird `make` aufgerufen, soll also das ausführbare Programm `hello` erzeugt werden. Die Abhängigkeit zu `hello.o` ist noch nicht erfüllt, weswegen zunächst die zweite Regel in Zeile 3 bearbeitet wird. Da auch `hello.o` noch nicht existiert, wird anhand des Befehls `gcc -c hello.c` in Zeile 4 dieses Target erzeugt. Anschließend ist die Abhängigkeit der ersten Regel erfüllt und mit Hilfe des Befehls `gcc hello.o -o hello` wird das Zwischenprodukt `hello.o` zu dem ausführbaren Programm `hello` gelinkt.

Wird die Datei `hello.o` nicht von anderen Zwischen- oder Endprodukten benötigt, kann die Regelbeschreibung auf die in Quelltextauflistung 3.3 gezeigten zwei Zeilen reduziert werden.

```
1 hello: hello.c
2 ␣␣␣␣␣␣␣␣gcc -o hello hello.c
```

Quelltext 3.3 Minimale Make-Regel

3.2.1 Wichtige Derivate

Durch die frühe Entwicklung von Make gab es in der Vergangenheit viele Abwandlungen, die das Ziel hatten, einzelne Schwächen zu verbessern und neue Funktionen hinzuzufügen. Insgesamt hat das aber zu starker Fragmentierung geführt, denn auch trotz POSIX Standardisierung[13] der Grundfunktionen von Make sind verschiedene Derivate des Programms zueinander selten kompatibel.

> Currently over 30 different flavors of Make have been proposed and used, including Vmake, Imake, gnuMake and Odin.
> Few original propositions have been made. [...]
> There is a clear need to do better than Make; but it is a serious challenge.
>
> – J. Estublier, [Est00, Kapitel 2]

13 `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html`

GNU Make

GNU Make[14] ist das standardmäßig verwendete Build-Management-Werkzeug von GNU/-Linux Distributionen sowie Mac OS X. Namhafte Projekte, die GNU Make zur Erstellung verwenden sind zum Beispiel der Linux Kernel und der GNU C Compiler. Dieses Make Derivat ist zudem Teil des GNU-Build-Systems, der sogenannten Autotools[15,16]. Dieses Build-System besteht aus einer ganzen Reihe aufeinander aufbauender, abgestimmter einzelner Programme, die auch heute noch ein wichtiger Bestandteil der Entwicklung vieler Software-Projekte ist.

Dieses Derivat hat, neben eigener Funktionalitäten für bessere Kompatibilität mit anderen Autotools-Programmen, auch Funktionen des Make-Programm aus *System V* aber auch vielen nicht eindeutig spezifizierbaren anderen Derivaten übernommen.[17]

BSD Make

BSD Make ist der Grundbaustein für die aktuell in FreeBSD, NetBSD und OpenBSD verwendeten Make-Derivate, welches auf BSD 4.4 lite basiert. Die Alleinstellungsmerkmale gegenüber dem originalen Make von Stuart Feldman sind die Möglichkeit der Bedingungsprüfung und iterative Schleifen. Das heutige BSD Make wurde im Vergleich zu GNU Make nur wenig erweitert, wodurch die Quelltexte einfacher zu verstehen und die Menge an Quelltext geringer ist.

3.2.2 Falsche Verwendung: Rekursiver Aufruf von Make

Das Hauptproblem bei der Verwendung von Make ist, dass es in großen Projekten auf eine Art und Weise eingesetzt wurde, die alle in Kapitel 2.3 genannten Regeln für Build-Systeme missachtet. Unter dem Vorwand der Einfachheit werden in Software-Modulen einzelne, eigenständige Makefiles gepflegt und dazu ein übergreifendes Makefile erstellt, dass in jedes Verzeichnis mit untergeordnetem Makefile wechselt und lokal einen Make-Prozess startet. Obwohl dieses Vorgehen zu einer einfacheren Benutzung des Build-Systems im Sinne der Erweiterbarkeit führt, ist der erste Indikator, dass diese Lösung suboptimal ist, die stark erhöhte Build-Dauer. Dies wurde von Peter Miller bereits 1998 als Problem gesehen [Mil98] und von Ludwig Hähne im Rahmen einer Belegarbeit anhand eines nicht weiter spezifizierten, automatisch generierten Projekts nachgemessen [Häh08].

Hähne hat die Laufzeit von rekursiven und nicht-rekursiven Make-Builds unter dem Gesichtspunkt der Parallelität untersucht. Dabei wurde eindeutig festgestellt, dass eine

14 https://www.gnu.org/software/make/
15 https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html
16 https://www.gnu.org/software/autoconf/manual/autoconf-2.63/html_node/The-GNU-Build-System.html
17 https://www.gnu.org/software/make/manual/make.html#Features

nichtrekursive Verwendung von Make zu einer Halbierung der Build-Dauer führt, was durch eindeutige Auflösung aller Abhängigkeiten zu erklären ist. Dem Build-System ist in diesem Fall durch den globalen, gerichteten und azyklischen Abhängigkeitsgraphen (vgl. Kapitel 2.2) bekannt welche Abhängigkeiten unabhängig von anderen aufzulösenden Abhängigkeiten sind. Mit dieser Kenntnis können diese Aufgaben ohne unerwünschte Seiteneffekte parallel ausgeführt werden. In dem von Hähne verwendeten, generierten Projekt war sowohl bei der nichtrekursiven, wie auch bei der rekursiven Konfiguration bei mehr als vier gleichzeitig bearbeiteten Aufgaben kaum noch eine Veränderung in der Laufzeit zu messen. Diese Ergebnisse sind in Abbildung 3.1 zu sehen.

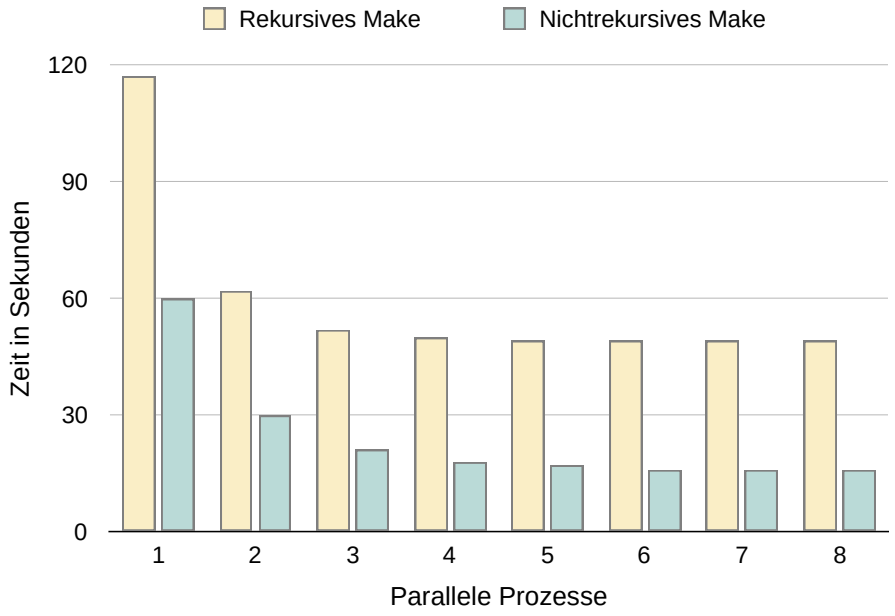

Abbildung 3.1 Parallele Ausführung von rekursivem und nichtrekursivem Make [Häh08, S. 32]

Im weiteren Verlauf seiner Arbeit wurde durch Hähne auch die Laufzeit der Builds unter dem Punkt der Skalierbarkeit untersucht. Durch einen nicht näher beschriebenen Code-Generator wurden Softwareprojekte in der Sprache C mit steigender Anzahl an Quelltextdateien erzeugt. Diese wurden mit rekursiven Make beziehungsweise nichtrekursivem Make als Build-System ausgestattet und die Build-Zeiten ohne parallele Abarbeitung verglichen. Die Ergebnisse zeigen ein ähnliches Bild wie schon in der Untersuchung zur Parallelität, dass rekursives Make im Vergleich zu nichtrekursivem Make etwa die doppelte Zeit für einen Build benötigt. Veranschaulicht ist dies in Abbildung 3.2. Der Grund hierfür ist die starke Verteilung der Build-Anweisungen im rekursiven Ansatz, bei dem für jedes Makefile ein neuer, jedoch im Gesamtbild unvollständiger Abhängigkeitsgraph erzeugt und abgearbeitet wird.

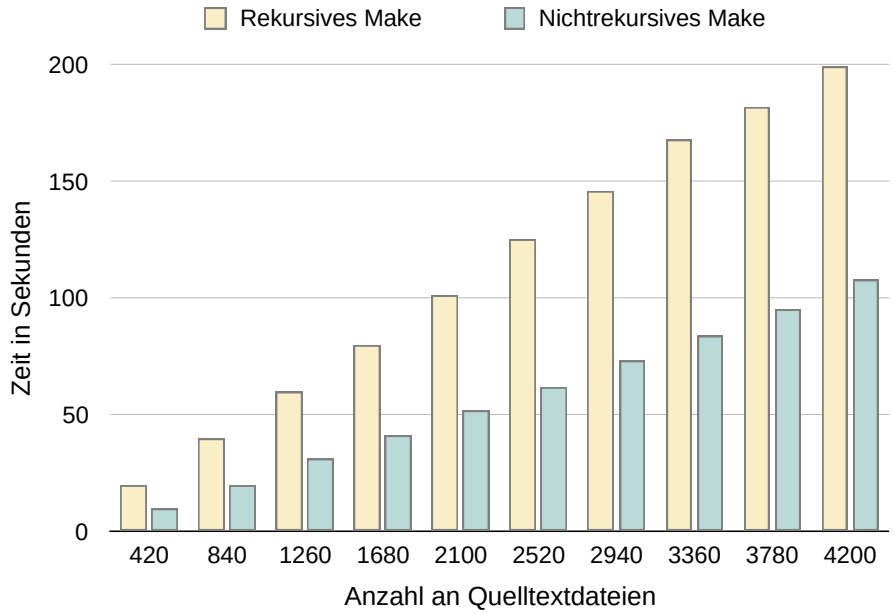

Abbildung 3.2 Skalierung von rekursivem und nichtrekursivem Make [Häh08, S. 33]

3.3 CMake

CMake selbst ist kein Build-Management-Werkzeug wie GNU Make oder BSD Make sondern ein Meta-Build-Werkzeug. Das bedeutet, es löst die Abhängigkeiten von Quelltext-Dateien nicht selbst auf, sondern verwendet eine Beschreibung der Abhängigkeiten zwischen Quelltextdateien in einer eigenen, zu Make nicht kompatiblen Syntax. Darauf aufbauend wird eine Build-Struktur mit GNU Make generiert. So können auf einer höheren Ebene als bei Make sehr einfach verschiedene, eigenständige Projekte in einem Projekt zusammengefasst werden. Die daraus generierte Build-Struktur verlässt sich allerdings sehr stark auf den rekursiven Aufruf von Make, was bereits in Kapitel 3.2 als für das Ziel dieser Arbeit ungeeignet herausgestellt wurde.

3.4 Sonstige

Es wurden weitere Build-Management-Werkzeuge als Alternativen für Make untersucht, die durch die Implementierung in Sprachen wie Java, Python und Ruby plattformunabhängig sind. Eine unvollständige Liste von Build-Management-Werkzeugen, die in der Lage sind, das in Kapitel 1.1 beschriebene Problem zu lösen, sind in Tabelle 3.1 dargestellt.

Programmname	Impl.-Sprache	Regelsprache	Fokus
Ant	Java	XML	Java-Projekte
Bazel	Java	Proprietär	Korrektheit, Geschwindigkeit
Gradle	Java	Groovy[18]	Multi-Projekt-Strukturen
Maven	Java	XML	Java-Projekte
Rake	Ruby	Ruby	Make-Ersatz für Ruby
SCons	Python	Python	Verbesserung von Make

Tabelle 3.1 Auflistung von Build-Systemen

Einige dieser Werkzeuge wurden auf bestimmte Ziel hin entwickelt, wie zum Beispiel Ant, welches initial für die Entwicklung von Java-Projekten geschrieben wurde.

> The main known usage of Ant is the build of Java applications.
>
> – `http://ant.apache.org`
> Besucht am 08.03.2016

Bei der Entwicklung von Gradle wurde expliziter Fokus auf die Möglichkeit gelegt, mehrere eigenständige Projekte unter einem neuen Projekt zusammen zu fassen.

> Out of the box, Gradle handles transitive dependencies that resolve across multiple repository types including Maven, Ivy, flat files.
>
> – `http://gradle.org/whygradle-build-automation/`
> Besucht am 08.03.2016

Bazel wurde von Google entwickelt, um einen sehr großen, monolithischen Build zu verwalten und dabei jederzeit eine Gesamtsicht auf alle Abhängigkeiten zu haben. Hierbei wird interne Software oft wiederverwendet, was bedeutet, dass Bazel zum Build kaskadierter Softwareprojekte geeignet ist.

Das in Python geschriebene Build-Management-Werkzeug SCons wurde als Verbesserung von Make entworfen. Um das Ziel zu erreichen wurden einigen vielversprechende Funktionen den Build und die Build-Beschreibung einfacher zu gestalten, wie eine automatische Abhängigkeitserkennung und die Generierung eines kompletten Abhängigkeitsgraphen implementiert.

Um zu untersuchen ob neue Entwicklungen im Bereich der Build-Management-Werkzeuge signifikante Verbesserungen von Builds kaskadierter Softwareprojekte erbringen, werden im weiteren Verlauf zwei dieser Werkzeuge näher untersucht. Zum einen wird Bazel untersucht, da es das Konzept von Make auf Client- und Server-Komponenten aufteilt und auf Build-Korrektheit ausgelegt ist. Als Zweites wird SCons betrachtet, da es bereits von Ludwig Hähne mit Make verglichen wurde [Häh08], und die bereits angesprochene Funktionalität vielversprechend wirkt.

18 JVM-basierte Skriptsprache; `http://groovy-lang.org`

3.5 Bazel

Das von der Firma Google stammende Bazel ist ein in Java geschriebenes Build-Management-Werkzeug, dass einen Client-Server-Ansatz verfolgt. Die Software befindet sich zum Zeitpunkt dieser Arbeit im Beta-Zustand, wird allerdings intern bei Google unter dem Namen Blaze produktiv eingesetzt. Bazel ist der Name der quelloffenen Version des Werkzeugs, dass bei Google unter dem Namen Blaze auf einer verteilten Infrastruktur mit großen Caches und starker Anpassung auf interne Bedürfnisse verwendet wird. Es wurde mit Fokus auf Korrektheit, Performanz und Reproduzierbarkeit entwickelt, und versteht unter Korrektheit, dass zwei identische Buildaufrufe zu exakt demselben Ergebnis führen. Gekoppelt mit einem VCS kann so auch die Reproduzierbarkeit garantiert werden. Anhand dieses Werkzeugs gelang es Google trotz stetig steigender Zeilen Quelltext gleichbleibende und sogar fallende Build-Zeigen vorzuweisen [Web12].

Im Gegensatz zu Blaze sind dem quelloffenen Bazel einige Funktionen vorenthalten, wie etwa die Möglichkeit den Server-Prozess auf mehrere Build-Maschinen zu verteilen:

> The open source Bazel code runs build operations locally.
> We believe that this is fast enough for most of our users.
>
> – Bazel FAQ, `http://bazel.io/faq.html`

Ein Beispiel für die Korrektheit und Reproduzierbarkeit ist in den Quelltextauflistungen 3.4 und 3.5 zu sehen. Wird etwa in einem C-Programm ein Makro wie `__DATE__` verwendet wird dies beim Aufruf des Kompilers ersetzt.

```
1 printf("Date: %s\n", __DATE__);
```

Quelltext 3.4 Reproduzierbarkeit von Builds: Quelltext

Im Fall von GNU GCC wird dies über die Kompiler-Option `-D__DATE__="redacted"` ermöglicht. Weitere Makros welche die Reproduzierbarkeit sowie Korrektheit beeinträchtigen und aus diesem Grund auf demselben Weg entfernt werden sind außerdem `__TIME__` und `__TIMESTAMP__`. Dadurch ergibt sich beim Aufruf des Programms folgende Ausgabe:

```
1 Date: redacted
2 Time: redacted
```

Quelltext 3.5 Reproduzierbarkeit von Builds: Programmausgabe

Die Beschreibung der Abhängigkeiten werden für Bazel in so genannten `BUILD`-Dateien geschrieben, die in jedem Verzeichnis und Unterverzeichnis existieren müssen. Ein zur Quelltextauflistung 3.1 äquivalentes Beispiel für eine `BUILD`-Datei Regel findet sich in der folgenden Quelltextauflistung 3.6:

```
1 genrule(
2     name = "Target",
```

```
3     srcs = ["Abhaengigkeiten"],
4     cmd = "Programmaufruf"
5 )
```

Quelltext 3.6 Grundform einer Bazel-Regel

Allerdings findet eine solche `genrule` nur selten Anwendung, da bereits vordefinierte Funktionen vorhanden sind, die tief in dem Build-Management-Werkzeug integriert sind und somit besser verarbeitet werden können.

Ein Beispiel für eine einfache Hello-World Build-Beschreibung findet sich in Quelltextauflistung 3.7. Es wird die Anweisung gegeben, eine ausführbare C-Binärdatei `hello` zu erzeugen. Die Erzeugung der Datei `hello.o` findet im Hintergrund statt und muss hier nicht explizit angegeben werden, denn der Funktionsaufruf `cc_binary` verfügt über die Kenntnis interner Regeln, die Konstruktion einer ausführbaren Datei aus einer C-Quelltextdatei beschreiben.

```
1 package(default_visibility = ["//visibility:public"])
2 cc_binary(
3     name = "hello",
4     srcs = ["hello.c"],
5 )
```

Quelltext 3.7 Beispiel einer Bazel `BUILD`-Datei

Wie bereits in Kapitel 2 erläutert und in der Quelltextauflistung 3.2 für das Build-Management-Werkzeug Make gezeigt, findet man auch bei Bazel dieselben Inhalte in der Build-Beschreibung. Lediglich die Programmaufrufe zur Erzeugung der Dateien sind in diesem Fall durch Abstraktion im Quelltext von Bazel untergebracht. Der Target-Name wird durch das `name`-Feld definiert, die dafür nötigen Quelltextdateien werden unter dem `srcs`-Feld aufgelistet. Abhängigkeiten von anderen Programmen oder Bibliotheken werden durch das `deps`-Feld angezeigt.

3.6 SCons

SCons[19] ist ein auf der Programmiersprache Python aufbauendes, quelloffenes Build-Management-Werkzeug, dass einige in Make nur sehr aufwändig zu implementierende Aufgaben bereits in der Grundkonfiguration bereitstellt und seinen Fokus auf die Korrektheit der Buildprodukte legt. So ist etwa die automatische Erkennung von Abhängigkeiten eine der Grundfunktionen von SCons. Weiter kann man in der Konfigurationsdatei eines SCons-Projekts, die dieselbe Funktion wie Makefiles oder BUILD-Dateien haben, auf alle Sprachmerkmale von Python zurückgreifen.

19 `http://www.scons.org`

In der bei SCons `SConstruct` genannten Build-Beschreibungsdatei wird, wie auch bei Make und Bazel, die Abhängigkeiten des zu bauenden Projekts beschrieben. Eine generische Regel wird in der Quelltextauflistung 3.8 gezeigt.

```
1 Command("Target", ["Abhaengigkeiten"], "Programmaufruf")
```

Quelltext 3.8 Grundform einer SCons-Regel

Wie schon bei Bazel (vgl. Abschnitt 3.5) wird dieser direkte Aufruf von Befehlen selten zum Einsatz kommen. Stattdessen werden auch bei SCons tiefer in das Build-Management-Werkzeug integrierte Funktionen verwendet. Als Beispiel ist eine zum Makefile in der Quelltextauflistung 3.2 äquivalente `SConstruct`-Datei in der Quelltextauflistung 3.9 dargestellt.

```
1 env = Environment()
2 hello_object = env.Object("hello.c")
3 env.Program("hello", hello_object)
```

Quelltext 3.9 Beispiel einer SCons `SConstruct` Datei

Die erste Zeile dient zur Erzeugung einer neuen Umgebung, die nicht von benutzerspezifischen Umgebungsvariablen beeinflussbar ist, um die Korrektheit des Builds zu verbessern. Diese Umgebung kann auch projektübergreifend ex- sowie importiert werden, um so Kenntnis von Abhängigkeiten von Unterprojekten zu erlangen. Anschließend wird SCons mitgeteilt, dass aus der Datei `hello.c` eine Objekt-Datei zu generieren ist. Diese Anweisung wird in der Python Variable `hello_object` gespeichert und in der nächsten Zeile als Abhängigkeit verwendet. In dieser letzten Zeile wird über die `Program`-Funktion bestimmt, dass ein ausführbares Programm mit dem Namen `hello` erzeugt werden soll anhand des Ergebnisses des zuvor definierten Arbeitsschrittes.

Wird die Objektdatei nicht explizit benötigt kann die `SConstruct`-Datei wie in Quelltextauflistung 3.10 verkürzt werden.

```
1 Program("hello", "hello.c")
```

Quelltext 3.10 Beispiel einer minimalen SCons `SConstruct` Datei

Das ist möglich, da SCons die Datei und etwaige, darin verwendete Header-Dateien, automatisch durchsucht und erkennt, und durch interne Regeln bereits das Vorgehen zur Kompilierung der ausführbaren Datei kennt.

Im Gegensatz zu Make verwendet SCons ausserdem bei der Prüfung auf veraltete Zwischen- und Endprodukte nicht den Zeitstempel, sondern berechnet standardmäßig den MD5-Hashwert der jeweiligen Dateien. Zudem ist es möglich selbst eigene Funktionen zu implementieren, die SCons mitteilen ob sich Dateien geändert haben.

3.7 Alternative Ansätze

Um grundlegende Veränderungen im Bereich der Build-Systeme nicht zu übersehen werden zwei experimentelle Build-Management-Werkzeuge untersucht. Beide setzen auf neue Ansätze um Abhängigkeiten aufzulösen, die vielversprechend für das Problem dieser Arbeit sind. Beide dieser Implementierungen sind allerdings in einem Zustand der nicht für einen produktiven Einsatz geeignet ist.

tup

In seinem Paper "Build System Rules and Algorithms" [Sha09] hat Mike Shal im Jahr 2009 eine Unterscheidung von Build-Systemen in Alpha- und Beta-Build-Systemen vorgenommen.

Ein Alpha-Build-System ist nach Shal jedes Build-System mit Kenntnis aller Abhängigkeiten des Software-Projekts, wie etwa nichtrekursives Make oder SCons. Dieser Typ von Build-Systemen basiert auf dem "Alpha Build System Algorithmus", der aus zwei Schritten besteht. Im ersten Schritt wird der Abhängigkeitsgraph aus den angegebenen Regeln und existierenden Quelltextdateien erstellt und im zweiten Schritt wird untersucht, welche Aktionen nötig sind um alle Zwischen- und Endprodukte zu erzeugen oder zu aktualisieren. Eine daraus resultierende, weitere Eigenschaft ist, dass Alpha-Build-Systeme bei einer Projektgröße von n Quelltextdateien eine Laufzeit von $O(n)$ aufweisen.

Als Beta-Build-System bezeichnet er Build-Systeme die partielle Abhängigkeitsgraphen verwenden um Abhängigkeiten zu erkennen und aufzulösen. Partielle Abhängigkeitsgraphen sind im Gegensatz zu unvollständigen Abhängigkeitsgraphen nicht unvollständig, sondern bilden einen bestimmten Teil von Abhängigkeiten eines Projektes konsistent ab. Werden alle partiellen Abhängigkeitsgraphen zu einem Graph zusammengeführt ergibt sich der komplette Abhängigkeitsgraph, den auch Alpha-Build-Systeme erzeugen. Weiter verwendet diese Art von Build-System den "Beta Build System Algorithmus". Bei diesem Algorithmus ist es nötig dem Build-System eine Liste von geänderten Dateien vor dem Build zu übergeben. Anhand dieser Liste sowie einer gespeicherten Version des globalen Abhängigkeitsgraphen wird ein minimaler partieller Abhängigkeitsgraph erstellt und anhand der vorher definierten Regeln abgearbeitet. Daraus ergibt sich die mögliche Laufzeit von $O(log^2 n)$ bei n Quelltextdateien.

Das Build-Management-Werkzeug tup wird als Beispiel für den "Beta Build System Algorithmus" verwendet und ist gegenüber dem klassischen Make in den Messungen des Autors[20] deutlich schneller. Das Projekt befindet sich unter aktiver Entwicklung und es existiert keine Angabe über die Produktionsreife des Programms. Aus diesem Grund und des sehr hohen Umstellungsaufwands von vorhandener Build-Beschreibungen wurde davon abgesehen das Programm tup näher zu untersuchen.

20 `http://gittup.org/tup/make_vs_tup.html`

redo

Die Funktionsweise des Build-Management-Werkzeugs redo wurde von D. J. Bernstein 2003 auf vier Web-Seiten beschrieben [Ber03]. Weitere Informationen oder Quelltexte des Autors zu diesem Werkzeug sind zu dem Zeitpunkt dieser Arbeit nicht bekannt. Der von redo verfolgte Ansatz ist ein Top-Down-Build. Das bedeutet, dass bei der Anforderung ein Target `hello.o` zu bauen, zuerst das zugehörige Build-Skript `hello.o.do` verwendet wird. Sollte diese Datei nicht existieren, so wird auf ein Standard-Build-Skript für diese Dateiendung mit dem Namen `default.o.do` zurückgegriffen. Abhängigkeiten sind in diesen Dateien mit der Endung `.do` definiert, die bei einem Aufruf rekursiv alle weiteren Abhängigkeiten des angeforderten Targets auflösen und bei Bedarf neu erzeugen.

Der Top-Down-Ansatz von redo bietet diverse Vorteile [Gro07, Kapitel 4] gegenüber von Make, angefangen mit einer Isolierung der Abhängigkeitsbeschreibungen. Wie bereits beschrieben werden die Regeln zu Erzeugung der Datei `hello.o` in der Datei `hello.o.do` definiert. Für allgemeine Regeln der Erzeugung von `.o`-Dateien aus `.c`-Dateien wird die Datei `default.o.do` eingelesen. Diese Funktionsweise führt dazu, dass nur genau die Regeln gelesen und ausgeführt werden, die zum Erzeugen des Targets wirklich nötig sind.

Wie auch schon tup ist von diesem Build-Management-Werkzeug keine Produktionsreife Version verfügbar. Alan Grosskurth hat in seiner Masterarbeit eine Version von redo in 250 Zeilen Shell Script implementiert [Gro07, Kapitel 5] und es existieren zahlreiche Forks unklarer Qualität einer minimalen Implementation in Python auf Github[21]. Da auch mit redo der Umstellungsaufwand der bereits bestehenden Build-Beschreibung zu hoch ist und auch die Qualität der bisherigen Implementationen des Programms unklar sind, wurde davon abgesehen redo im weiteren Verlauf der Arbeit als mögliche Alternative in Betracht zu ziehen.

21 `https://github.com/apenwarr/redo`

4 Unvollständige Abhängigkeitsgraphen

Die dieser Arbeit untersuchten Build-Systeme basieren alle auf der Kernidee des ursprünglichen Make [Fel79], Abhängigkeiten als gerichtete, azyklische Graphen abzubilden. Durch dieses Konzept ist eine häufige Ursache von Problemen bei Builds auf einen schlecht konstruierten oder unvollständigen Abhängigkeitsgraphen zurück zu führen [Smi11, S. 310].

> If *make* doesn't do what you expect it to, it's a good chance the makefile is wrong.
>
> – Adam de Boor [dB88]

Die resultierende Aussage dabei ist: Wenn die Beschreibung der Abhängigkeiten ungenau oder falsch ist, wird der Build nicht wie vorgesehen funktionieren.

Anhand des folgenden Beispiels sollen die zwei Probleme von unvollständigen Abhängigkeitsgraphen aufgezeigt werden, die unter anderem durch das rekursive Aufrufen von Make entstehen [Mil98]. Dabei soll ausgehend von drei eigenständigen Softwareprojekten **A**, **B** und **C** das Programm `prog` erstellt werden. Dieses Programm aus Projekt **A** hängt von der Datei `2` aus Projekt **B** ab. Zudem besteht zwischen der Datei `2` aus Projekt **B** eine weitere Abhängigkeit zu der Datei `5` aus Projekt **C**, die eine Programmbibliothek darstellen kann. Diese Situation ist in Abbildung 4.1 dargestellt.

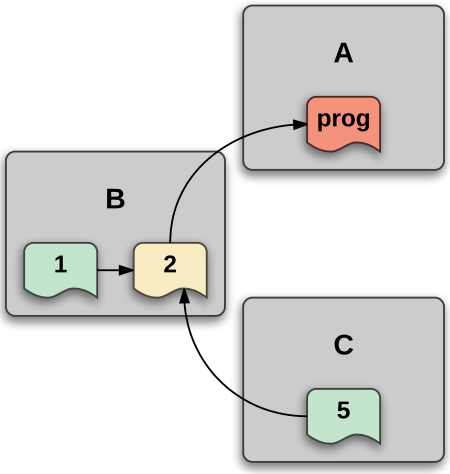

Abbildung 4.1 Abhängigkeiten zwischen drei Softwareprojekten

Über diese Projektstruktur kann der vollständiger Abhängigkeitsgraph in Abbildung 4.2 erstellt werden, der alle drei Projekte vereint.

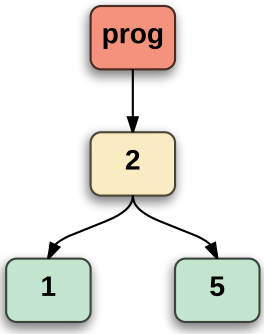

Abbildung 4.2 Vollständiger Abhängigkeitsgraph aus Abbildung 4.1

Der oben dargestellte Graph in Abbildung 4.2 beschreibt alle Abhängigkeiten über alle drei Projekte hinweg. Dazu muss vorausgesetzt sein, dass in Projekt **A** das Build-System Kenntnis über die Auflösung von Abhängigkeiten aus Projekt **B** sowie **C** hat. Diese Eigenschaft ist etwa bei einem rekursiven Aufruf von `make` nicht gegeben, da mit dieser Methode in jedem Unterprojekt lediglich die dort beschriebenen Abhängigkeitsregeln bekannt sind. Eine dort neu erzeugte Instanz von `make` kennt nur ihren lokalen Teil der Abhängigkeiten, ohne die Möglichkeit dieses Wissen an andere Instanzen weiter zu geben. In Abbildung 4.3 ist das Problem visuell dargestellt. Innerhalb der grauen Box befindet sich jeweils ein Make-Prozess mit eigenem Abhängigkeitsgraph. Gestrichelte Linien zeigen einen Aufruf eines weiteren Make-Prozesses und damit den Verlust der Kenntnis aller Abhängigkeiten. Die Notation **B / 2** bezeichnet demnach einen Aufruf von `make` im Projektverzeichnis **B** mit dem Target **2**.

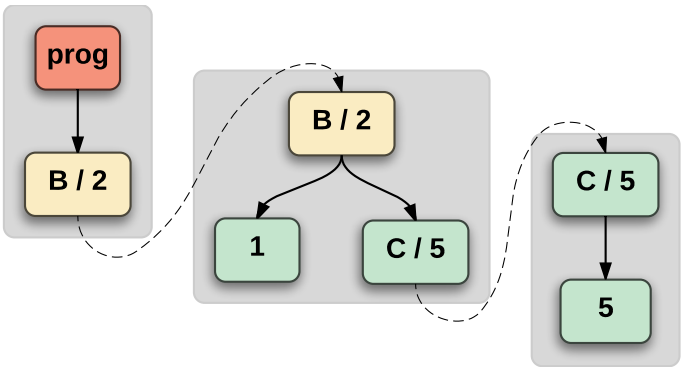

Abbildung 4.3 Unvollständige Abhängigkeitsgraphen aus Abbildung 4.1

4.1 Make wird zu oft aufgerufen

Zusätzlich zu der in Abbildung 4.1 dargestellten Situation wird nun angenommen, dass `prog` von einem weiteren Projekt mit dem Namen **D** abhängt. Weiter existiert eine zusätzliche Abhängigkeit der Datei `3` von Projekt **D** auf die projektinterne Datei `4` sowie auf Datei `5` von Projekt **C**, wie in Abbildung 4.4 zu erkennen ist.

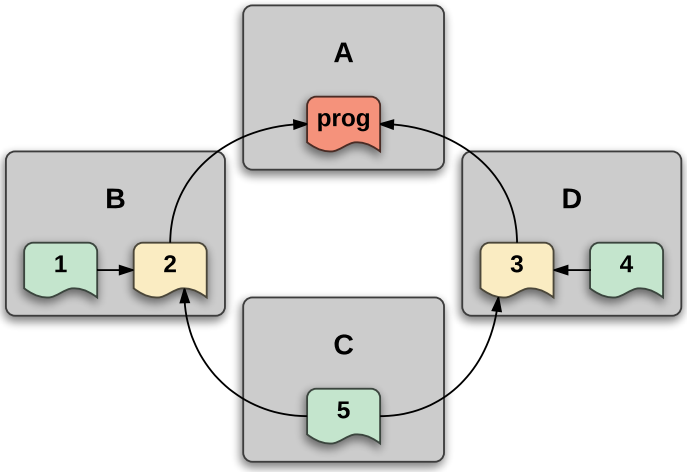

Abbildung 4.4 Abhängigkeiten zwischen vier Softwareprojekten

Um alle Abhängigkeiten korrekt auflösen zu können muss in diesem Fall der Graph der Form in Abbildung 4.5 entsprechen. Wie bereits eingangs erläutert ist das allerdings nur der Fall, wenn Projekt **A** Kenntnis über jede Abhängigkeit von jedem anderen Projekt hat.

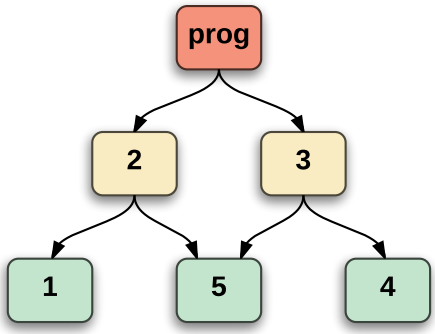

Abbildung 4.5 Vollständiger Abhängigkeitsgraph aus Abbildung 4.4

Nur in diesem Fall wird korrekt erkannt, dass die Zwischenprodukte `2` und `3` von derselben Datei `5` abhängig sind.

Wird nun ein Build-System mit rekursivem Make verwendet, entstehen statt einem vollständigen Graphen insgesamt fünf Make-Prozesse, mit je einem eigenen, unvollständigen Abhängigkeitsgraph. Dieses Verhalten ist in Abbildung 4.6 veranschaulicht. Die Ursache ist der identische Verweis auf **C / 5** in den Unterprojekten **B** und **D**.

Da keine Kommunikation zwischen den einzelnen Make-Prozessen stattfindet, wird das Target **C / 5** einmal zu oft aufgerufen. Durch die Funktionsweise von Make wird bei beiden Aufrufen von **C / 5** der komplette Abhängigkeitsgraph von **C** erstellt, was bei großen Projekten lange Zeit in Anspruch nehmen kann. Es ist in Make nicht vorgesehen und deswegen nicht trivial lösbar, dass die Make-Prozesse **B / 2** und **D / 3** aus diesem Beispiel Informationen teilen, damit erkannt werden kann, dass **C / 5** für beide dasselbe Ziel ist.

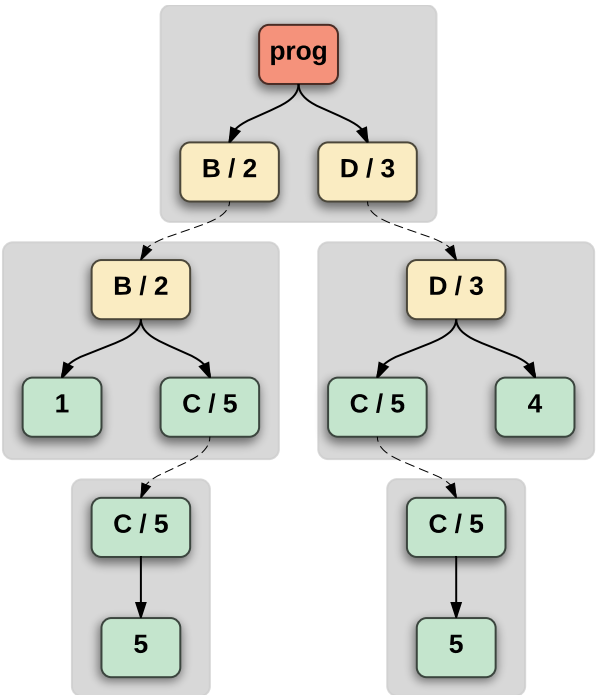

Abbildung 4.6 Unvollständige Abhängigkeitsgraphen aus Abbildung 4.4

4.2 Make kennt nicht alle Abhängigkeiten

Durch Fehler in der Beschreibung der Abhängigkeiten, oder fehlerhafte automatische Abhängigkeitserkennung, kann es vorkommen, dass das Build-Management-Werkzeug nicht ausreichende Informationen über vorliegende Quelldateien hat.

Ein konkretes Beispiel ist eine fehlende Abhängigkeit auf eine Header-Datei, in der etwa ein C `struct foo` definiert ist, welches aber seit dem letzten Build geändert wurde. Sollten Zwischenprodukte existieren, die die vorherige Version von `foo` nutzen, so werden diese nicht als veraltet erkannt und auch nicht neu erzeugt. Dies führt beim Build nicht zu Kompilierungsfehlern, und eine Erkennung des Problems ist damit erschwert. Erst bei der Ausführung des Programms, wenn etwa der Speicher für `foo` reserviert werden soll, um dort auf ein neu hinzugefügtes Element zuzugreifen wird das Programm wegen einer Speicherzugriffsverletzung beendet. Der Grund hierfür sind die nicht übereinstimmenden Datenstrukturen, und das durch die teilweise Neukompilierung des Programms inzwischen inkompatible Application Binary Interface (ABI).

Das Problem löst sich erst durch einen kompletten Build (vgl. Kapitel 2.4.1), oder wenn in der Beschreibung der Abhängigkeiten die fehlende Regel ergänzt wurde.

5 Lösungsansätze

In diesem Kapitel werden die zuvor in Abschnitt 3 ausgewählten Build-Management-Werkzeuge diskutiert und hinsichtlich der Problemstellung auf Tauglichkeit untersucht.

5.1 Bazel

Wie schon in Abschnitt 3.5 erläutert, ist dieses Build-Management-Werkzeug für sehr große Quelltextverzeichnisse entwickelt worden. Bazel wurde dafür konzipiert verschiedene, eigenständige Projekte in einem übergeordneten Build-System zusammenzufassen.

Zudem setzt Bazel auf einen Client-Server-Ansatz, der es erlaubt viele Optimierungen am Build-Prozess vorzunehmen, die mit einfach ausgeführten Build-Management-Werkzeugen aufwändig zu implementieren sind[22]. So werden etwa auf dem Server-Teil die vollständigen Abhängigkeitsgraphen von Projekten, `BUILD`-Dateien und Build-Meta-Daten in einem Cache gespeichert. Das bedeutet, dass der Abhängigkeitsgraph nicht bei jedem Build neu berechnet werden muss, wie es beispielsweise bei Make der Fall ist. Daraus resultiert, dass Builds sehr effizient auch mit mehreren, auf dem selben System arbeitenden, Entwicklern durchgeführt werden können.

Bei Bedarf ist es möglich jeden beliebigen Befehl über sogenannte `genrule`s, also generische Regeln, auszuführen, jedoch sind die bereitgestellten Funktionsaufrufe durch die zuvor beschriebene Aufteilung in Client und Server deutlich mächtiger. Beispielsweise weist die Funktion `cc_binary` aus Quelltextauflistung 3.7 im Bazel-Quelltext[23] Kenntnis der Semantik von GNU GCC auf. Diese Funktionen sind tief in den Server-Teil von Bazel integriert und können so besser auf Korrektheit, Reproduzierbarkeit und Performanz in Form von Skalierung und Parallelisierung optimiert werden.

Sollten Unterprojekte allerdings nicht mit Bazel als Build-Management-Werkzeug entwickelt worden sein, müssen die dort lokalen Build-Beschreibungen erst auf die Syntax von Bazels `BUILD`-Dateien übertragen werden.

Prüfung der Anforderungen

Die aktuelle Build-Infrastruktur von genua baut zumeist auf Make auf, da auch viele der Softwaremodule, etwa Grub und OpenBSD, auf Make setzen. Eine Umstellung aller bei

22 `http://bazel.io/docs/bazel-user-manual.html`
23 `src/main/java/com/google/devtools/build/lib/rules/cpp/CcBinary.java`

Anforderung	Erfüllt	Kapitel
R.1 (Projektübergreifende Abhängigkeitsaufösung)	Ja	3.5
R.2 (Lizenzkosten)	Ja	3.5
R.3 (Migrationsaufwand)	Nein	5.1
R.4 (Pflegeaufwand)	Nein	5.1
R.5 (Performanz)	k.A.	–
R.6 (Ressourcen)	Ja	5.1

Tabelle 5.1 Prüfung der Anforderungen an das Build-Management-Werkzeug Bazel

genua vorkommender Build-Systeme auf Bazel ist unrealistisch, da Abhängigkeiten auf externe Projekte bestehen, deren Pflege denselben oder mehr Aufwand bedeuten würden, als den aktuellen Zustand beizubehalten.

5.2 SCons

Auch bei SCons ist der Trend modernerer Build-Systeme zur tieferen Integration von Funktionen wie auch schon in Abschnitt 5.1 bei Bazel zu erkennen. Wobei dieses Programm nicht auf einem Client-Server-Ansatz basiert sondern Make-ähnlich einen Build-Prozess pro Programmaufruf verwendet.

Die Nachteile dieser Vorgehensweise werden durch interessante Funktionen wieder ausgeglichen. Es ist etwa möglich SCons in einem interaktiven Modus zu starten, in welchem der Abhängigkeitsgraph des Projekts nur beim Start vollständig erzeugt wird. Änderungen von Quelltextdateien werden durch den laufenden Prozess beim Start eines Builds gegen den bereits im Arbeitsspeicher befindlichen Graphen abgeglichen und aufgelöst.

Für die Arbeit mehrerer Entwickler im selben Projekt ist ein Cache-Verzeichnis konfigurierbar, um Zwischenprodukte dort für alle Build-Aufrufe zur Verfügung zu stellen. Durch die Verwendung von MD5-Hashsummen können diese auf Unterschiede hin verglichen und wiederverwendet oder neu generiert werden.

Da SCons komplett in Python implementiert wurde, ist es zudem möglich Problemen bei der Geschwindigkeit mit Python Profilern genau zu untersuchen wann SCons welche Aktionen durchführt.

Allerdings ist Python eine interpretierte Sprache und auch die Berechnung vom Hash-Werts ist deutlich aufwändiger als das einfache Auslesen von Dateiattributen. Beides sind Gründe, weswegen SCons im Vergleich zu nichtrekursivem Make deutlich langsamer ist, was auch Ludwig Hähne im Rahmen seines Großen Belegs[Häh08] nachgewiesen hat (vgl. Abbildungen 5.1 und 5.2).

Ein weiterer Nachteil der durch die Nutzung einer interpretierten, objektorientierten Spra-

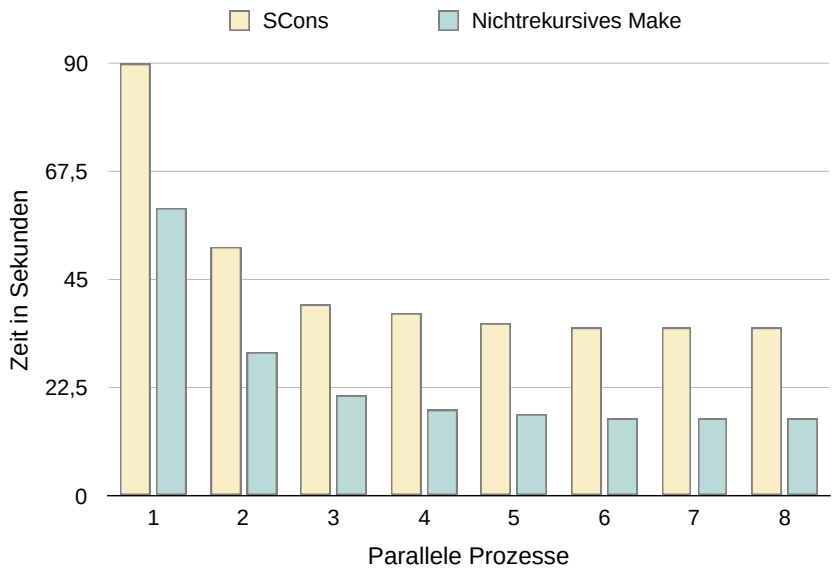

Abbildung 5.1 Parallele Ausführung von SCons und Make [Häh08, S. 32]

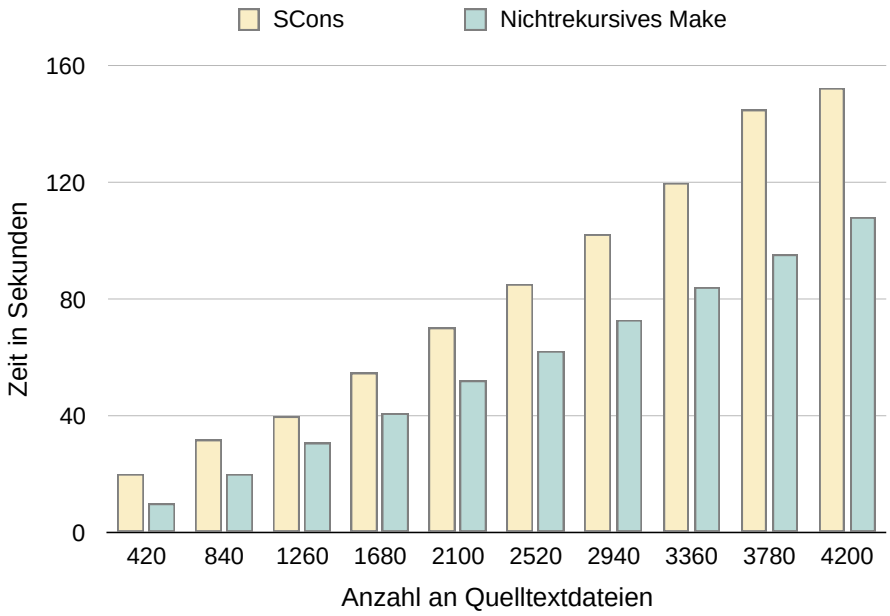

Abbildung 5.2 Skalierung von SCons und Make [Häh08, S. 33]

che entsteht, ist die Speichernutzung im Vergleich zu Make. In dem von Hähne generierten, synthetischen Projekt beträgt die Differenz des Speicherbedarfs fast eine Größenordnung. Der Grund dafür ist, dass SCons selbst zuerst in den Arbeitsspeicher geladen werden muss.

> The offset memory consumption of approximately 10MB accounts for the SCons source which must be loaded in advance.
>
> – Ludwig Hähne [Häh08]

Der linear, stark steigende Speicherbedarf, veranschaulicht in Abbildung 5.3, ist dadurch

zu erklären, dass jede Abhängigkeit selbst ein Python Objekt ist. Zusätzlich werden daran weitere Meta-Daten für den Build, wie etwa Kompiler-Optionen und Beziehungen zu anderen Objekten, gespeichert.

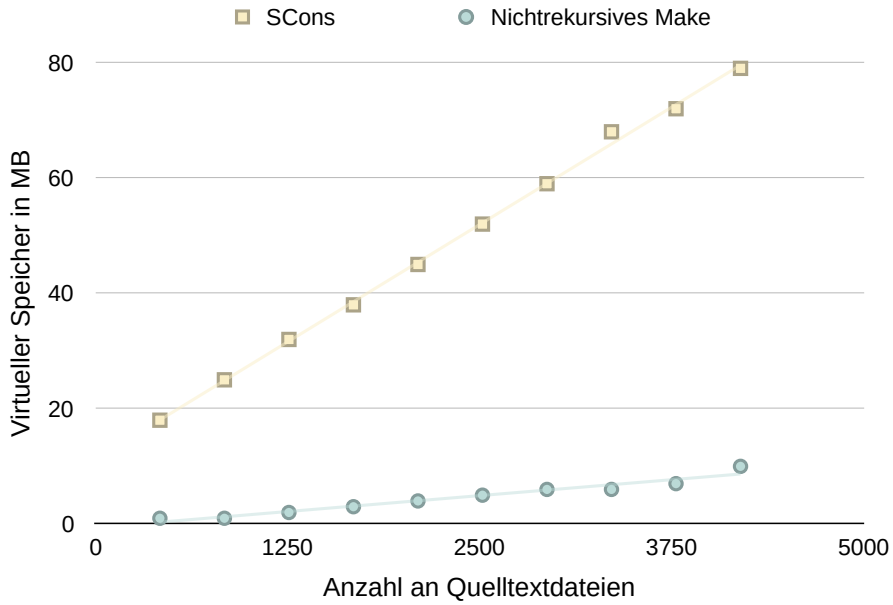

Abbildung 5.3 Vergleich der Speichernutzung von SCons und Make [Häh08, S. 34]

Die hohe Speichernutzung fällt allerdings erst bei größeren Projekten ins Gewicht. Sollte der Linux Kernel in der Version 3.2, der aus etwa 37.000 Quelltextdateien besteht[24], mit SCons als Build-System gebaut werden, so würden pro Build-Prozess mehr als 600MB an Arbeitsspeicher benötigt, während Make nur um 100MB beanspruchen würde. Dazu kommt, dass sich die Build-Geschwindigkeit zwischen rekursivem und nichtrekursivem Make einordnet.

Prüfung der Anforderungen

Wie bereits bei Bazel in Abschnitt 5.1 erläutert, werden die von genua genutzten Projekte nicht mit SCons als Build-System ausgeliefert, sodass hier ein sehr hoher Aufwand durch den Wechsel zu SCons sowie die andauernde Pflege dieses Deltas entstehen würde. Aus diesem Gund und den zuvor aufzeigten schlechten Performanz wird SCons als untauglich für die Lösung des Problems dieser Arbeit eingestuft.

24 `http://go.linuxfoundation.org/who-writes-linux-2012` (PDF)

Anforderung	Erfüllt	Kapitel
R.1 (Projektübergreifende Abhängigkeitsaufösung)	Ja	3.6
R.2 (Lizenzkosten)	Ja	3.6
R.3 (Migrationsaufwand)	Nein	5.2
R.4 (Pflegeaufwand)	Nein	5.2
R.5 (Performanz)	Nein	5.2
R.6 (Ressourcen)	Nein	5.2

Tabelle 5.2 Prüfung der Anforderungen an das Build-Management-Werkzeug SCons

5.3 Make

Anhand des Build-Management-Werkzeuges Make wird bereits der Großteil der Softwareprojekte bei genua verwaltet und gebaut. Allerdings wird dabei zumeist auch auf den rekursiven Aufruf des Programms gesetzt. Zum Teil begründet sich dieser Umstand darin, dass zum Beispiel OpenBSD ebenfalls mit rekursivem Make gebaut wird, und genua bereits vor 20 Jahren das Produkt genugate darauf aufsetzend entwickelt hat.

Die Probleme des rekursiven Aufrufs von Make in großen Projekten hat Peter Miller in seiner Veröffentlichung "Recursive Make Considered Harmful" [Mil98] bereits 1998 aufgezeigt. Auf diese wurde in Abschnitt 3.2 anhand der Arbeit von Ludwig Hähne [Häh08] unter Performanz-Gesichtspunkten eingegangen.

> For large UNIX software development projects, the traditional methods of building the project use what has come to be known as "recursive make".
>
> – Peter Miller [Mil98]

Die Ansicht der Veröffentlichung Millers, dass große Projekte lediglich Quelltextdateien verwenden, die für diese Projekte geschrieben wurden ist nichtmehr zeitgemäß, wie bereits in der Einleitung in Abschnitt 1 geschildert wurde. Heutige Entwicklung findet stark modularisiert statt, um bereits gelöste Probleme nicht erneut lösen zu müssen. Viele dieser Module sind eigenständige, oft auch externe Projekte, auf deren Build-System nur in begrenztem Umfang Einfluss genommen werden kann. Die Grundidee hinter "Recursive Make Considered Harmful" bleibt allerdings bestehen, denn nur bei Kenntnis aller Abhängigkeiten kann ein Build korrekt und effizient durchgeführt werden.

Der Aufbau eines `Makefile`s für eine alle Abhängigkeiten umfassende Build-Beschreibung wird in "Recursive Make Considered Harmful" in Kapitel 7 gegeben. Dabei wird die Direktive `include` verwendet um tiefer in der Ordnerstruktur befindliche Build-Instruktionen neben den zugehörigen Quelltextdateien abzulegen. Über diese Direktive wird an der Stelle des Auftretens von `include` der Inhalt der übergebenen Datei kopiert. Der Nachteil bei dieser Lösung ist, dass diese Build-Beschreibungsfragmente – von Miller `module.mk` genannt –

derart gestaltet sein müssen, sodass alle Pfade relativ zum Wurzelverzeichnis des Projektes korrekt beschrieben sind.

Dieser Ansatz ist allerdings nur bei Projekten sinnvoll bei dem die Entwickler volle Kontrolle über alle Quelltextdateien haben. Bei einem Projekt aufbauend auf mehreren Modulen und Projekten externer Entwickler müssten die verwendeten `module.mk` Dateien an anderer Stelle als innerhalb der Module gepflegt werden. Sollte das externe Projekt auf eine neue Version aktualisiert werden, würden sonst interne Abhängigkeitsbeschreibungen entfernt. Dazu kommt, dass bei Änderungen in externen Modulen die jeweiligen `module.mk` Dateien angepasst werden müssen, um weiterhin einen korrekten und effizienten Build zu ermöglichen.

Prüfung der Anforderungen

Anforderung	Erfüllt	Kapitel
R.1 (Projektübergreifende Abhängigkeitsaufösung)	Nein	4
R.2 (Lizenzkosten)	Ja	3.2
R.3 (Migrationsaufwand)	Nein	5.3
R.4 (Pflegeaufwand)	Nein	5.3
R.5 (Performanz)	Ja	5.3
R.6 (Ressourcen)	Ja	5.3

Tabelle 5.3 Prüfung der Anforderungen an das Build-Management-Werkzeug Make

Aufgrund der Gegebenheit, dass bei genua viele der verwendeten Softwareprojekte bereits auf Make setzen, jedoch der Pflegeaufwand des `include`-Ansatzes zu hoch ist, kann auch die Lösung aus “Recursive Make Considered Harmful” nicht direkt angewendet werden. Eine Erweiterung der Funktionalität der `include`-Direktive, die den Pfadkontext einer inkludierten Datei speichern könnte, wäre denkbar.

5.4 Fazit: Erweiterung der Make-Funktionalität

Es wurden die drei Build-Management-Werkzeuge Bazel, SCons und Make im Kontext der Problemstellung untersucht. Bazel lagert den Abhängigkeitsgraph in einen lokalen Server-Prozess aus, der über den Client gesteuert wird. So können Optimierungen und Anpassungen der Abhängigkeiten an einer zentralen Stelle stattfinden. Bei der Entwicklung von SCons wurde sich funktional eher an Make orientiert, jedoch sinnvolle Verbesserungen vorgenommen, wie etwa einen interaktiven Modus, um den Abhängigkeitsgraph, ähnlich wie Bazels Server-Prozess, nur einmal komplett zu berechnen.

Beide sind für moderne, modulare Softwareprojekte geeignet, sofern die verwendeten Module ausreichend klein sind, sodass sich der Pflegeaufwand der Build-Beschreibung in Grenzen hält. Bei genua ist diese Bedingung nicht gegeben, da OpenBSD als Grundprojekt das Gros der Quelltextdateien stellt und dabei selbst auf rekursives Make als Build-Management-Werkzeug setzt. In Anbetracht der Umstände, wird eine Erweiterung der Make-Funktion `include` verfolgt, da hierdurch eine Verbesserung der bislang nicht erfüllten Anforderungen erwartet wird.

6 Prototypische Implementierung einer neuen Direktive in fmake

Für die Implementierung einer neuen Funktion in Make wird exemplarisch das in FreeBSD 10.1-RELEASE verfügbare `fmake` verwendet. Dieses Derivat von Make ist der Vorgänger des aktuell in OpenBSD verwendeten `make` sowie des standardmäßig von FreeBSD verwendeten `bmake`. Es wurde `fmake` ausgewählt, da diese Implementation als Vorgänger aktueller Derivate geringeren Funktionsumfang aufweißt und aus weniger Quelltext besteht[25] was eine Implementierung und deren weitere Untersuchung im Rahmen dieser Arbeit vereinfacht.

6.1 Funktionsweise

Die in dieser Arbeit implementierte `import`-Funktion baut auf der `include`-Funktion auf, um den Entwicklungsaufwand gering zu halten. Das führt dazu, dass nur wenige Änderungen an der ursprünglichen Implementierung vorgenommen werden müssen, was eine Integration in das Upstream-Projekt vereinfacht.

Um den Unterschied zwischen `include` und `import` an einem konkreten Beispiel zu erläutern, wird ein Projekt bestehend aus zwei `Makefile`s, einer Quelltextdatei und einem Unterverzeichnis erstellt.

```
1 Beispielprojekt
2 `-- Makefile
3     subdir
4     |-- Makefile
5     `-- hello.c
```

Quelltext 6.1 Beispielhafte Ordnerstruktur

Innerhalb des Ordners `Beispielprojekt` befindet sich das erste `Makefile`, dass sich im Wurzelverzeichnis befindet, und den Build steuert. Innerhalb eines Unterverzeichnisses `subdir` befindet sich ein zweites `Makefile`, dass für die Erzeugung des Programms `hello` alle nötigen Regeln (vgl. Quelltextauflistung 3.3) enthält.

25 Anzahl Zeilen Quelltext (Stand 03.03.2016):

bmake	OpenBSD make	fmake
18.814	12.632	10.950

Die neue Direktive ist analog zu `include` zu verwenden. Dem Funktionsnamen folgt ein Pfad zu einer Datei, die beliebig benannt sein kann und gültige Make-Regeln enthalten muss.

```
1 .import "subdir/Makefile"
```

Quelltext 6.2 Inhalt der Datei `Beispielprojekt/Makefile`

Sollte in `Beispielprojekt/Makefile` die Datei `subdir/Makefile` über die `include`-Funktion eingebunden sein, wird die Regel ohne Veränderung übernommen und im Wurzelverzeichnis wie in Quelltextauflistung 6.3 ersichtlich ausgeführt werden.

```
1 hello: hello.c
2 ␣␣␣␣␣␣␣␣cc -o hello hello.c
```

Quelltext 6.3 Make-Regel nach `include`

Das führt zu einem Ausführungsfehler in Make, da im Wurzelverzeichnis das Programm `hello` nicht existiert. Aus diesem Grund versucht Make anhand des Programmaufrufs dieses zu erzeugen. Da jedoch die Quelltextdatei ebenfalls nicht im Wurzelverzeichnis zu finden ist, beendet Make sich mit folgendem Fehler:

```
1 [user@fbsd ~/Beispielprojekt]$ fmake
2 make: don't know how to make hello.c. Stop
```

Quelltext 6.4 Fehlermeldung von Make nach `include`

Damit auch im Wurzelverzeichnis Abhängigkeiten tiefer in der Projektstruktur aufgelöst werden können, wird in der `import`-Funktion der Pfad der zu importierenden Datei bei Programmaufrufen beachtet. Daher entsteht bei Verwendung von `import` folgende Sicht auf die im Wurzelverzeichnis importierte Datei:

```
1 subdir/hello: subdir/hello.c
2 ␣␣␣␣␣␣␣␣cc -o hello hello.c
```

Quelltext 6.5 Make-Regel nach `import`

Wird das Target erzeugt oder regeneriert, wird vor der Ausführung des Programmaufrufs in das Verzeichnis `subdir` gewechselt. Da hierbei in den richtigen Kontext für die erfolgreiche Ausführung des Befehls gewechselt wurde, wird `hello` korrekt erzeugt.

```
1 [user@fbsd ~/Beispielprojekt]$ fmake
2 cc -o hello hello.c
```

Quelltext 6.6 Ausführung von `fmake` mit `import`

Ein alternativer Ansatz wäre das Verändern des Programmaufrufs, jedoch kann in einem Programmaufruf eine zu komplexe Ansammlung von Shell-Eigenschaften auftreten.

Etwa können Umgebungsvariablen mit Pfaden gesetzt werden, die Einfluss auf die Programmaufrufe haben, und ein Wechsel in einen anderen Pfadkontext könnte zu Fehlern führen.

Im Detail erzeugt Make für die Ausführung jedes Programmaufrufs einer Regel einen neuen Shell-Prozess. Programmintern werden dafür die Systemaufrufe `fork` und `exec` verwendet. Anhand von `fork` wird ein neuer Prozess erzeugt, der ein exaktes Duplikat des Vaterprozesses ist.[26] Durch den Systemaufruf `execve` wird der aktuelle Prozess durch einen neuen Prozess ersetzt.[27]

Um für das auszuführende Kommando der Make-Regel den richtigen Pfadkontext zu setzen wird zwischen `fork` und `execve` in den zuvor gespeicherter Pfad mittels `chdir` gewechselt. Es ist auch möglich einen Wechsel des Pfadkontexts vor dem `fork` Systemaufruf zu vollziehen, allerdings muss dann nach dessen Ausführung im Vaterprozess in den ursprünglichen Pfadkontext zurück gewechselt werden. Ein Pfadwechsel nach dem `execve` ist ohne die oben bereits abgelehnte Anpassung des Programmaufrufs oder der ausführenden Shell nicht möglich.

Implementierungsdetails

Durch die Wiederverwendung der `include`-Implementation konnte die Anzahl nötiger Änderungen gering gehalten werden. Folgende Dateien wurden verändert:

`GNode.h`

In dieser Datei wird die Datenstruktur definiert, die die Knoten in dem Abhängigkeitsgraph abbildet. Diese Struktur wird um einen Pfad erweitert, welcher nur im Fall einer `import`-Verwendung gesetzt wird.

```
1 char     *import_path;
```

Quelltext 6.7 `GNode.h`: Möglichkeit einen Pfad zu speichern

`job.c`

Hier wurde der Wechsel des Pfadkontexts implementiert. Sofern ein importierter Pfad gesetzt ist, wird an dieser Stelle in diesen Pfad gewechselt. Sollte dies nicht möglich sein, wird eine Fehlermeldung ausgegeben und das `fmake` beendet.

```
1 if (gn->import_path != NULL) {
2         if (chdir(gn->import_path) == -1) {
3                 perror("chdir");
4                 exit(1);
5         }
6 }
```

Quelltext 6.8 `job.c`: Wechsel des Pfadkontexts

26 Manpage zu “fork”, Kategorie 2, FreeBSD 10.1-RELEASE
27 Manpage zu “execve”, Kategorie 2, FreeBSD 10.1-RELEASE

`parse.c`

Von der `include`-Direktive wird im Quelltext die interne Funktion `xparse_include` aufgerufen. Diese wurde für die Implementierung der `import`-Direktive wiederverwendet und erweitert.

```
1 static void
2 parse_import(char *file, int code __unused, int lineno __unused
     )
3 {
4         xparse_include(file, 2);
5 }
```

Quelltext 6.9 `parse.c`: Wiederverwenden vorhandener Funktionen

`hash_tables.c`

In dieser automatisch generierten Datei befinden sich Informationen für die Zuordnung von Direktiven aus der Datei `parse.c`. Zu diesen Direktiven gehören unter anderem `if`, `for` und `include`. Durch die Erweiterung der `parse.c` um `import` musste diese Datei neu generiert werden. Die Zuordnung dieser Direktiven auf deren jeweilige Implementierung basiert auf minimalen, perfekten Hashes [Cic80]. Diese dient dazu, einen Zugriff auf die Implementierungsfunktion in konstanter Zeit zu ermöglichen.

`targ.c`

An dieser Stelle wird der zuvor in `GNode.h` definierte Zeiger `import_path` initialisiert.

```
1 gn->import_path = NULL;
```

Quelltext 6.10 `targ.c`: Initialisierung des Pfades für Targets

6.2 Validierung

Um die Funktionalität der Implementierung der `import`-Funktion nachzuweisen, wird aufbauend auf der Projektstruktur, die Quelltextauflistung 6.11 beschrieben ist, Make mit rekursivem Ansatz dem erweiterten Make mit nicht-rekursiven Ansatz gegenüber gestellt.

```
Math
|-- Makefile
`-- exp
    |-- Makefile
    |-- exp.c
    |-- exp.h
    |-- expo.c
    `-- mult
        |-- Makefile
        |-- mult.c
        |-- mult.h
        |-- multi.c
        `-- sum
            |-- Makefile
            |-- add.c
            |-- sum.c
            `-- sum.h
```

Quelltext 6.11 Verschachtelte Beispielprojekte

Dieses Beispiel zeigt drei ineinander kaskadierte Software-Projekte. In diesen werden die mathematischen Funktionen Addition, Multiplikation sowie die Potenzierung aufeinander aufbauend implementiert. Das bedeutet beispielsweise, dass innerhalb des `mult`-Projekts auf die Implementierung der Addition aus dem `sum`-Projekt verwendet wird. Jedes Projekt ist dabei für sich eigenständig, so werden die jeweiligen Funktionen in den Programmen `add`, `multi` und `expo` abgebildet. In den Dateien `sum.c`, `mult.c` und `exp.c` (siehe Anhang A.1.1) werden die eigentlichen Funktionen implementiert, dabei greift `exp.c` auf `mult.c` und `mult.c` auf `sum.c` zurück.

6.2.1 Make mit rekursivem Ansatz

Das `Makefile` des rekursiven Ansatzes im Wurzelverzeichnis ist in Quelltextauflistung 6.12 dargestellt.

```
all: exp/expo exp/mult/multi exp/mult/sum/add

exp/expo:
```

```
4 ␣␣␣␣␣␣␣␣fmake -C exp expo
5
6 exp/mult/multi:
7 ␣␣␣␣␣␣␣␣fmake -C exp/mult multi
8
9 exp/mult/sum/add:
10 ␣␣␣␣␣␣␣␣fmake -C exp/mult/sum add
11
12 clean:
13 ␣␣␣␣␣␣␣␣fmake -C exp clean
```

Quelltext 6.12 Rekursiver Ansatz: Inhalt des `Makefile` im Wurzelverzeichnis

Die Ausgabe des Abhängigkeitsgraphen des Wurzel-`Makefile`s dieses Ansatzes ist in Anhang A.2.1 zu finden. Da in diesem Ansatz nicht alle Abhängigkeiten zu jedem Zeitpunkt bekannt sind, wurden aus Gründen der Vollständigkeit auch die Ausgaben der Abhängigkeitsgraphen des `exp`-Projekts im Anhang A.2.2, des `mult`-Projekts in Anhang A.2.3 sowie des `sum`-Projekts in Anhang A.2.4 abgebildet. Hier kann sehr gut die Ähnlichkeit zu der in Abschnitt 1.2 beschriebenen Abbildung 1.5 erkannt werden.

Der in den Abschnitten 3.2.2 und 4.1 beschriebene Bruch des Abhängigkeitsgraphen kann in der Ausgabe in Quelltextauflistung 6.13 anhand der Zeilen 2, 5, 7, 10 und 13 nachvollzogen werden. An diesen Stellen wird ein weiterer Make-Prozess erzeugt, der eine eigene, rein lokale Sicht auf seine Regeln besitzt.

```
1 [user@fbsd ~/Math]$ fmake
2 fmake -C exp expo
3 cc -c -o expo.o expo.c
4 cc -c -o exp.o exp.c
5 fmake -C mult mult.o
6 cc -c -o mult.o mult.c
7 fmake -C mult/sum sum.o
8 cc -c -o sum.o sum.c
9 cc -o expo expo.o exp.o mult/mult.o mult/sum/sum.o
10 fmake -C exp/mult multi
11 cc -c -o multi.o multi.c
12 cc -o multi multi.o mult.o sum/sum.o
13 fmake -C exp/mult/sum add
14 cc -c -o add.o add.c
15 cc -o add add.o sum.o
```

Quelltext 6.13 Rekursiver Ansatz: Ausgabe von `fmake`

6.2.2 Erweitertes Make mit nicht-rekursivem Ansatz

Im Gegensatz zum rekursiven Ansatz, muss mit dieser Methode und der in dieser Arbeit implementierten Funktion `import` das `Makefile` im Wurzelverzeichnis lediglich den Inhalt aus Quelltextauflistung 6.14 enthalten.

```
1 all: exp/expo exp/mult/multi exp/mult/sum/add
2
3 clean: exp/clean
4
5 .import "exp/Makefile"
```

Quelltext 6.14 `Makefile` vom Projekt `exp`

Bei der Ausgabe des Abhängigkeitsgraphen im Wurzelverzeichnis (siehe Anhang A.3) ist der Unterschied zum rekursiven Ansatz, dass alle nötigen Abhängigkeiten zur Erstellung von jedem Zwischen- und Endprodukt in einem, projektübergreifenden Abhängigkeitsgraphen vorhanden sind. Dies ist an der Ausgabe eines vollen Builds in Quelltextauflistung 6.15 ersichtlich, da der Build durch nur einem Make-Prozess erfolgt.

```
1  [user@fbsd ~/Beispielprojekt]$ fmake
2  cc -c -o expo.o expo.c
3  cc -c -o exp.o exp.c
4  cc -c -o mult.o mult.c
5  cc -c -o sum.o sum.c
6  cc -o expo expo.o exp.o mult/mult.o mult/sum/sum.o
7  cc -c -o multi.o multi.c
8  cc -o multi multi.o mult.o sum/sum.o
9  cc -c -o add.o add.c
10 cc -o add add.o sum.o
```

Quelltext 6.15 Nicht-rekursiver Ansatz: Ausgabe von `fmake`

Wieder kann durch das Ergebnis dieses Abschnitts die Ähnlichkeit zu der in Abschnitt 1.2 beschriebenen Abbildung 1.6 erkannt werden.

6.2.3 Schwächen der Implementierung

In dem aktuellen Zustand der Implementierung ist es nicht möglich Variablen gezielt zu kontrollieren. Sollten diese in importierten Makefiles verwendet werden, so überschreiben sie Variablen übergeordneter Build-Beschreibungen ohne Warnung.

Weiter ist es nicht möglich innerhalb von Regeln Spezialvariablen wie `$@` (das Target) oder `$<` (alle Abhängigkeiten einer Regel) zu verwenden, da diese den vollen Pfad des Targets aus Sicht des Wurzelverzeichnis enthalten. Dies führt zu Konflikten, da zum Ausführungszeitpunkt bereits in das Unterverzeichnis gewechselt wurde.

Ergebnis

Anhand dieses Beispiels wurde gezeigt, dass mit der Implementierung der `import`-Funktion es nun möglich ist, Abhängigkeiten von Unterprojekten im Wurzelverzeichnis eines übergreifenden Projektes komplett aufzulösen. Dadurch ist die Anforderung R.1 erfüllt.

Der Migrationsaufwand aus Anforderung R.3 wird durch die `import`-Direktive minimal gehalten. Die redundante Angabe von Subtargets, die einen rekursiven Make-Aufruf enthalten entfällt hierdurch. Weiter können bereits bestehende, auch rekursive, Make-Infrastrukturen schrittweise migriert werden. Zudem müssen importierte Makefiles nicht verändert, wodurch auch der Aufwand für Pflege aus Anforderung R.4 minimal gehalten wird.

Durch die Einsparung rekursiver Aufrufe von Make, wie in Quelltextauflistung 6.15 im Vergleich zu Quelltextauflistung 6.13 ersichtlich, ist eine Verbesserung der Performanz auch nach [Häh08] zu erwarten.

Anforderung	Erfüllt	Kapitel
R.1 (Projektübergreifende Abhängigkeitsaufösung)	**Ja**	6.2
R.2 (Lizenzkosten)	Ja	3.2
R.3 (Migrationsaufwand)	**Ja**	6.2
R.4 (Pflegeaufwand)	**Ja**	6.2
R.5 (Performanz)	**Ja**	3.2.2
R.6 (Ressourcen)	Ja	5.3

Tabelle 6.1 Validierung der Anforderungen an das Build-Management-Werkzeug Make

7 Schlussbetrachtung

In diesem Kapitel werden die Erkenntnisse sowie die Ergebnisse zusammengefasst und ein Ausblick gegeben, der mögliche Erweiterungen und weiteres Vorgehen aufzeigt.

7.1 Zusammenfassung

Bei dem Ziel dieser Arbeit handelte es sich um die Untersuchung von Build-Management-Werkzeugen für die zukünftige Build-Struktur von genua.

Dafür wurden zunächst die Grundlagen der Build-System-Thematik erläutert, wobei zunächst eine klare, sprachliche Unterscheidung geschaffen wurde. Es folgte eine Einführung in den Aufbau und die Funktionsweise von Build-Management-Werkzeugen.

Nach der Einführung in das Thema wurde eine Auswahl von Build-Management-Werkzeugen vorgestellt, die unter den Gesichtspunkten der Problemstellung untersucht wurden. Dabei wurde auf die Fähigkeit einen Multi-Projekt-Build durchzuführen sowie eine zentrale Stelle für die Erzeugung eines vollständigen Abhängigkeitsgraphs geachtet. Für eine nähere Untersuchung wurden Make, Bazel und SCons herangezogen. Für Bazel sprach der Client-Server-Ansatz, für SCons die Ähnlichkeit zu Make bei deutlich besserer Funktionalität. Make wurde untersucht, da es bereits bei genua verwendet wird.

Es wurde festgestellt, dass modernere Build-Management-Werkzeuge auf tiefe Integration und Kenntnis der Build-Werkzeug-Semantik setzen, um so bestimmte Aspekte wie Reproduzierbarkeit oder Skalierbarkeit zu verbessern. Weiter wurde erkannt, dass Make sich mit der aktuellen Funktionalität nur knapp nicht zur Bewältigung des Problems dieser Arbeit eignet.

Durch die große Verbreitung von Make und die Nachteile einer Verwendung eines anderen Build-Management-Werkzeugs wurde entschieden in Make eine `import`-Funktion zu implementieren.

Letztlich wurde die Implementierung vorgestellt und anhand eines Beispielprojekts mit einem rekursiven Make-Ansatz verglichen.

7.2 Ausblick

In dieser Arbeit wurden erste Schritte in der Konsolidierung der Build-Strukturen von genua gemacht. Es wurde eine Änderung im Quelltext von `fmake` vorgenommen, die es möglich macht Abhängigkeiten in kaskadierten Projekten besser aufzulösen.

Implementierung zur Produktionsreife führen
Durch die zeitliche Beschränkung der Arbeit war es nicht möglich die entstandene Implementierung zur Produktionsreife zu führen. Um die Produktionsreife zu erreichen, müssen die in Abschnitt 6.2.3 benannten Schwächen behoben werden.

Implementierung in alternativen Versionen von Make
Die Änderung dieser Arbeit sollte in Make-Varianten von OpenBSD sowie GNU Make integriert werden, sodass die in dieser Arbeit entstandenen Ergebnisse im Build-System von genua vollumfänglich genutzt werden können. Eine Umstellung von rekursivem Make auf die Nutzung der `import`-Direktive ist dabei schrittweise möglich.

Umstrukturierung von komplexen Build-Umgebungen
Anhand der Ergebnisse dieser Arbeit sollte eine Umstrukturierung bestehender, rekursiver Build-Systeme, wie beispielsweise der OpenBSD-Build, untersucht werden. Da konkret der OpenBSD-Base-Build massiv auf den Einsatz rekursiven Makes setzt, ist hier großes Verbesserungspotential zu erkennen. Zudem ist das OpenBSD-Ports-System ein gutes Beispiel für ein kaskadiertes Build-Systems von Dritt-Software. Daher kann auch dieses von den Ergebnissen dieser Arbeit profitieren.

A Anhang

A.1 Quelltext des verschachtelten Beispielprojekts

A.1.1 Implementierungen

```
1 int sum(int a, int b) {
2     return(a+b);
3 }
```

Quelltext A.1 `sum.c`

```
1  #include "sum/sum.h"
2
3  int mult(int a, int b) {
4      int i,tmp = 0;
5
6      for (i = 0; i < a; i++)
7          tmp = sum(tmp, b);
8
9      return(tmp);
10 }
```

Quelltext A.2 `mult.c`

```
1  #include "mult/mult.h"
2
3  int _exp(int base, int exp) {
4      int i,tmp = 1;
5
6      for (i = 0; i < exp; i++)
7          tmp = mult(tmp, base);
8
9      return(tmp);
10 }
```

Quelltext A.3 `exp.c`

A.1.2 Makefiles des rekursiven Ansatzs

```
1 add: add.o sum.o
2 ␣␣␣␣␣␣␣␣cc -o add add.o sum.o
3
4 add.o: add.c sum.h
5 ␣␣␣␣␣␣␣␣cc -c -o add.o add.c
6
7 sum.o: sum.c sum.h
8 ␣␣␣␣␣␣␣␣cc -c -o sum.o sum.c
9
10 clean:
11 ␣␣␣␣␣␣␣␣rm -f add.o sum.o add
```

Quelltext A.4 `Makefile` vom Projekt `sum`

```
1 multi: multi.o mult.o sum/sum.o
2 ␣␣␣␣␣␣␣␣cc -o multi multi.o mult.o sum/sum.o
3
4 multi.o: multi.c mult.h sum/sum.h
5 ␣␣␣␣␣␣␣␣cc -c -o multi.o multi.c
6
7 mult.o: mult.c sum/sum.h
8 ␣␣␣␣␣␣␣␣cc -c -o mult.o mult.c
9
10 sum/sum.o:
11 ␣␣␣␣␣␣␣␣fmake -C sum sum.o
12
13 clean:
14 ␣␣␣␣␣␣␣␣fmake -C sum clean
15 ␣␣␣␣␣␣␣␣rm -f mult.o multi.o multi
```

Quelltext A.5 `Makefile` vom Projekt `mult`

```
1 expo: expo.o exp.o mult/mult.o mult/sum/sum.o
2 ␣␣␣␣␣␣␣␣cc -o expo expo.o exp.o mult/mult.o mult/sum/sum.o
3
4 expo.o: expo.c mult/mult.h mult/sum/sum.h
5 ␣␣␣␣␣␣␣␣cc -c -o expo.o expo.c
6
7 exp.o: exp.c exp.h
8 ␣␣␣␣␣␣␣␣cc -c -o exp.o exp.c
9
10 mult/mult.o:
11 ␣␣␣␣␣␣␣␣fmake -C mult mult.o
12
```

```
13 mult/sum/sum.o:
14 ␣␣␣␣␣␣␣␣fmake -C mult/sum sum.o
15
16 clean:
17 ␣␣␣␣␣␣␣␣fmake -C mult clean
18 ␣␣␣␣␣␣␣␣rm -f exp.o expo.o expo
```

Quelltext A.6 `Makefile` vom Projekt `exp`

A.1.3 Makefiles des nicht rekursiven Ansatzs

```
1  multi: multi.o mult.o sum/sum.o
2  ␣␣␣␣␣␣␣␣cc -o multi multi.o mult.o sum/sum.o
3
4  multi.o: multi.c mult.h sum/sum.h
5  ␣␣␣␣␣␣␣␣cc -c -o multi.o multi.c
6
7  mult.o: mult.c sum/sum.h
8  ␣␣␣␣␣␣␣␣cc -c -o mult.o mult.c
9
10 clean: sum/clean
11 ␣␣␣␣␣␣␣␣rm -f mult.o multi.o multi
12
13 .import "sum/Makefile"
```

Quelltext A.7 `Makefile` vom Projekt `mult`

```
1  expo: expo.o exp.o mult/mult.o mult/sum/sum.o
2  ␣␣␣␣␣␣␣␣cc -o expo expo.o exp.o mult/mult.o mult/sum/sum.o
3
4  expo.o: expo.c mult/mult.h mult/sum/sum.h
5  ␣␣␣␣␣␣␣␣cc -c -o expo.o expo.c
6
7  exp.o: exp.c exp.h
8  ␣␣␣␣␣␣␣␣cc -c -o exp.o exp.c
9
10 clean: mult/clean
11 ␣␣␣␣␣␣␣␣rm -f exp.o expo.o expo
12
13 .import "mult/Makefile"
```

Quelltext A.8 `Makefile` vom Projekt `exp`

A.2 Rekursiver Ansatz: Auszug aus der Ausgabe von `fmake -rdg1`

A.2.1 Abhängigkeitsgraph des Ausgangs-`Makefile`

```
[...]
#*** Input graph:
#
# *** MAIN TARGET ***
all             : exp/expo exp/mult/multi exp/mult/sum/add

# parents: all
exp/expo        :
        fmake -C exp expo

# parents: all
exp/mult/multi  :
        fmake -C exp/mult multi

# parents: all
exp/mult/sum/add:
        fmake -C exp/mult/sum add

clean           :
        fmake -C exp clean
[...]
```

A.2.2 Abhängigkeitsgraph des `exp-Makefile`

```
[...]
#*** Input graph:
#
# *** MAIN TARGET ***
expo            : expo.o exp.o mult/mult.o mult/sum/sum.o
        cc -o expo expo.o exp.o mult/mult.o mult/sum/sum.o

# parents: expo
expo.o          : expo.c mult/mult.h mult/sum/sum.h
        cc -c -o expo.o expo.c

# parents: expo
exp.o           : exp.c exp.h
        cc -c -o exp.o exp.c

# parents: expo
mult/mult.o     :
```

```
18          fmake -C mult mult.o
19
20  # parents: expo
21  mult/sum/sum.o  :
22          fmake -C mult/sum sum.o
23
24  clean           :
25          fmake -C mult clean
26          rm -f exp.o expo.o expo
27  [...]
```

A.2.3 Abhängigkeitsgraph des `mult-Makefile`

```
1   [...]
2   #*** Input graph:
3   #
4   # *** MAIN TARGET ***
5   multi           : multi.o mult.o sum/sum.o
6           cc -o multi multi.o mult.o sum/sum.o
7
8   # parents: multi
9   multi.o         : multi.c mult.h sum/sum.h
10          cc -c -o multi.o multi.c
11
12  # parents: multi
13  mult.o          : mult.c sum/sum.h
14          cc -c -o mult.o mult.c
15
16  # parents: multi
17  sum/sum.o       :
18          fmake -C sum sum.o
19
20  clean           :
21          fmake -C sum clean
22          rm -f mult.o multi.o multi
23  [...]
```

A.2.4 Abhängigkeitsgraph des `sum-Makefile`

```
1   [...]
2   #*** Input graph:
3   #
4   # *** MAIN TARGET ***
5   add             : add.o sum.o
6           cc -o add add.o sum.o
7
```

```
# parents: add
add.o           : add.c sum.h
        cc -c -o add.o add.c

# parents: add
sum.o           : sum.c sum.h
        cc -c -o sum.o sum.c

clean           :
        rm -f add.o sum.o add
[...]
```

A.3 Nicht-rekursiver Ansatz: Auszug aus der Ausgabe von `fmake -rdg1`

```
[...]
#*** Input graph:
#
# *** MAIN TARGET ***
all             : exp/expo exp/mult/multi exp/mult/sum/add

# parents: all
exp/expo        : exp/expo.o exp/exp.o exp/mult/mult.o exp/mult/sum/sum.o
        cc -o expo expo.o exp.o mult/mult.o mult/sum/sum.o

# parents: all
exp/mult/multi  : exp/mult/multi.o exp/mult/mult.o exp/mult/sum/sum.o
        cc -o multi multi.o mult.o sum/sum.o

# parents: all
exp/mult/sum/add: exp/mult/sum/add.o exp/mult/sum/sum.o
        cc -o add add.o sum.o

clean           : exp/clean

# parents: clean
exp/clean       : exp/mult/clean
        rm -f exp.o expo.o expo
# parents: exp/mult/sum/add
exp/mult/sum/add.o: exp/mult/sum/add.c exp/mult/sum/sum.h
        cc -c -o add.o add.c

# parents: exp/mult/sum/add exp/mult/multi exp/expo
exp/mult/sum/sum.o: exp/mult/sum/sum.c exp/mult/sum/sum.h
```

```
30         cc -c -o sum.o sum.c
31
32 # parents: exp/mult/clean
33 exp/mult/sum/clean:
34         rm -f add.o sum.o add
35
36 # parents: exp/mult/multi
37 exp/mult/multi.o: exp/mult/multi.c exp/mult/mult.h exp/mult/sum/sum.h
38         cc -c -o multi.o multi.c
39
40 # parents: exp/mult/multi exp/expo
41 exp/mult/mult.o : exp/mult/mult.c exp/mult/sum/sum.h
42         cc -c -o mult.o mult.c
43
44 # parents: exp/clean
45 exp/mult/clean  : exp/mult/sum/clean
46         rm -f mult.o multi.o multi
47
48 # parents: exp/expo
49 exp/expo.o      : exp/expo.c exp/mult/mult.h exp/mult/sum/sum.h
50         cc -c -o expo.o expo.c
51
52 # parents: exp/expo
53 exp/exp.o       : exp/exp.c exp/exp.h
54         cc -c -o exp.o exp.c
55 [...]
```

Abkürzungsverzeichnis

ABI Application Binary Interface 30

CSD Crypto-System-Development 1, 2

MKD Micro-Kernel-Development 2

MSD Management-System-Development 2

SGD Secure-Gateway-Development 1

VCS Versionsverwaltungssystem 4, 8, 9, 11, 22

Glossar

BSD

Unix-Derivate mit dem Namen Berkeley Software Distribution. In 1977 entstanden durch einen Fork einer Vorgängerversion von System V durch die Universität von Kalifornien in Berkeley. Heute versteht man unter dem Begriff BSD die Klasse aller Unix-Derivate, die aus den eigentlichen BSDs stammen. Mac OS X ist die derzeit kommerziell erfolgreichste Unix-Variante.
Wichtigster Punkt von BSD ist die Lizenz unter der die Quelltexte veröffentlicht wurden. Diese ist eine freie Lizenz, die es erlaubt, den Quelltext auch bei propertiären Programmen zu verwenden.
18, 20, 59, 60

genubox

Fernwartungs-Appliance mit umfangreichen Monitoring-Funktionen zur Absicherung von Wartungsarbeiten.
1, 2, 59

genucard

Personal Security Gerät zur Verwendung mit Notebook oder Computer zur sicheren Anbindung von Geräten ans Firmennetz.
1–3, 59

genucenter

Zentrale Verwaltungsplattform für genua-Produkte wie genuscreen, genucard, genubox oder vs-top.
2

genugate

Zweistufige Firewall mit Application Level Gateway und Paketfilter auf getrennter Hardware.
1, 35

genuOS

Idee eines unternehmensinternen OpenBSD-Forks zur Zentralisierung der halbjährlichen Betriebssystem-Upgrades von OpenBSD.
4

genuscreen

Paket-Filter-Firewall und VPN-Appliance für die Netzwerkabsicherung von verschiedenen Standorten.
1, 2, 59

L4

Microkernel-basierte Separationstechnologie, die von genua genutzt wird, um auf derselben Hardware Betriebssysteme getrennt voneinander zu betreiben.
3, 60

Microkernel

Bezeichnet den Kern eines Betriebssystem, der nur notwendigste Funktionen für einen Betrieb besitzt. Zusätzliche Funktionalität wird über Prozesse im Benutzer-Modus implementiert. [Lie95]
2, 60

OpenBSD

Ein auf Sicherheit ausgelegtes Unix-Derivat unter der freien BSD-Lizenz. Das Projekt ist bekannt für das Beharren auf Quelloffenheit, freie Dokumentation und seine kompromisslose Stellung gegenüber Software-Lizenzen sowie der Korrektheit von Quelltext.
1, 3, 4, 9, 18, 37, 59

System V

Das erste kommerzielle Unix Betriebssystem und Konkurrent von BSD. Entwickelt von AT&T und veröffentlicht im Jahr 1983. Verfolgte anderen Philosophien als BSD, was sich in verschiedenen, grundlegenden Verhaltens- und Entwicklungsarten auszeichnete.
18, 59

vs-top

Sicherer Laptop mit dem Separationssystem L4 um eine Trennung von sensiblen Daten und Netzen zu ermöglichen.
2, 3, 59

Literaturverzeichnis

[Ber03] D. J. Bernstein. Rebuilding target files when source files have changed. `http://cr.yp.to/redo.html`, Juli 2003. (Besucht am 19.02.2016).

[Cic80] R. J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[dB88] A. de Boor. PMake - A Tutorial. `https://www.freebsd.org/doc/en/books/pmake/book.html`, 1988. (Besucht am 08.03.2016).

[Des08] A. Deshpande und D. Riehle. The total growth of open source. In *Open Source Development, Communities and Quality*, S. 197–209. Springer, 2008.

[Epp02] G. K. T. Epperly. Software in the DOE: The hidden overhead of the build. *Lawrence Livermore National Laboratory, CA, USA, Technical Report*, 2002.

[Est00] J. Estublier. Software configuration management: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, S. 279–289. ACM, 2000.

[Fel79] S. I. Feldman. Make - a program for maintaining computer programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[Fel88] S. I. Feldman. Evolution of MAKE. In *Proceedings of the International Workshop on Software Version and Configuration Control, January 27-29, 1988, Grassau, Germany*, S. 413–416. 1988.

[Gro07] A. Grosskurth. *Purely top-down software rebuilding*. Masterarbeit, University of Waterloo, Januar 2007.

[Häh08] L. Hähne. *Empirical Comparison of SCons and GNU Make*. Großer Beleg, Technische Universität Dresden, August 2008.

[Här08] H. Härtig, M. Roitzsch, A. Lackorzynski, B. Döbel und A. Böttcher. L4-virtualization and beyond. *Korean Information Science Society Review*, 2, 2008.

[Hei10] G. Heiser und B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proc. ACM First Asia-Pacific Workshop on Systems (APSys 2010)*, S. 19–24. New Delhi, India, Aug. 2010.

[Lie95] J. Liedtke. *On micro-kernel construction*, Bd. 29. ACM, 1995.

[Mil98] P. Miller. Recursive Make Considered Harmful. *Australian UNIX Systems User Group Newsletter; Journal of AUUG Inc.*, 19(1):14–25, 1998.

[Sha09] M. Shal. Build System Rules and Algorithms. `http://gittup.org/tup/build_system_rules_and_algorithms.pdf`, 2009. (Besucht am 08.03.2016).

[Smi11] P. Smith. *Software Build Systems: Principles and Experience*. Addison Wesley, 2011.

[Web12] Building Software at Google Scale Tech Talk. `https://www.youtube.com/watch?v=2qv3fcXW1mg`, März 2012. (Besucht am 05.01.2016).

[Web14] The Motivation for a Monolithic Codebase: Why Google Stores Billions of Lines of Code in a Single Repository. `https://www.youtube.com/watch?v=W71BTkUbdqE`, September 2014. (Besucht am 05.01.2016).