

# Probleme durch rekursives Make

*Peter Miller*

pmiller@opensource.org.au

T} T{

Aus dem Amerikanischen  
ins Deutsche übersetzt von

*Ulrike Amoore*

uamoore@cmmagazin.de

## Vorwort

Zur Produktion großer UNIX-Projekte verwendet man traditionellerweise rekursives Make. Bei manchen Projekten führt das zu sehr langen Produktionszeiten, was insbesondere, wenn man nur eine Datei ändern möchte, unvertretbar ist. Als wir das Phänomen näher untersuchten, stellte sich heraus, dass eine Reihe von Problemen, die voneinander unabhängig zu sein schienen, gemeinsam für die Verzögerung verantwortlich waren. Eine genaue Analyse zeigte jedoch eine gemeinsame Ursache.

Dieser Artikel beschäftigt sich mit einigen der Probleme, die im Zusammenhang mit der Anwendung von rekursivem Make regelmäßig auftreten, und legt dar, dass sie alle Symptome desselben Problems sind. Diese Symptome haben die UNIX-Anwender lange als unumgängliche Tatsachen hingenommen, aber sie müssen nicht länger geduldet werden. Dazu gehört, dass ein rekursives Make oft sehr lange braucht, um herauszufinden, dass es nichts zu tun braucht, dass es zu viel oder zu wenig tut oder dass es übermäßig empfindlich auf Veränderungen von Quellcode reagiert und für eine ungestörte Funktion den ständigen Eingriff in das `Makefile` notwendig macht.

Man kann die Lösung für diese Probleme finden, indem man sein Augenmerk erst einmal darauf richtet, wie Make grundsätzlich arbeitet, und dann die Auswirkungen analysiert, die durch das Hinzufügen von rekursivem Make, hervorgerufen werden. Die Analyse zeigt, dass das Problem von der künstlichen Einteilung des Builds in gesonderte Teilmenen herrührt. Das wiederum führt zu den beschriebenen Symptomen. Um die Symptome zu vermeiden, ist es lediglich notwendig diese Einteilung zu verhindern, d. h. einen einzigen Make-Durchgang zu veranlassen, was nicht bedeutet, dass es nur ein einziges `Makefile` gibt.

Diese Schlussfolgerung widerspricht vielen bezüglich der Produktion großer Projekte angesammelten Volksweisheiten. Einige der von Vertretern dieser Volksweisheiten vorgebrachten Einwände werden hier untersucht und erweisen sich als unbegründet. Die praktische Umsetzung dieser Schlussfolgerung führt zu wesentlich ermutigenderen Ergebnissen. Wird diese Methode laufend weiterentwickelt, werden Verbesserungen der Effizienz erheblich schneller als erwartet sichtbar, ohne dass die Modularität aufgegeben werden muss. Die Durchführung eines Ganzprojekt-Makes ist nicht so schwierig umzusetzen, wie es zunächst erscheint.

## 1. Einführung

Die traditionellen Produktionsmethoden für große

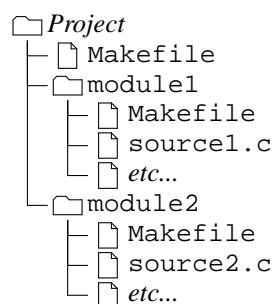
UNIX Softwareentwicklungsprojekte sind als *rekursives Make* bekannt geworden. Der Name verweist auf die Verwendung einer Hierarchie von Verzeichnissen, die die Quelldateien für die Module, aus denen das Projekt besteht, beinhalten, wobei jedes der Unterverzeichnisse ein `Makefile` enthält, das die Regeln und Anweisungen für das Make-Programm beschreibt. Das

Copyright © 1997 Peter Miller  
German translation Copyright © 2002 CM-Magazin.

vollständige Projekt-Build wird durchgeführt, indem man das Haupt-Makefile veranlasst, in allen Unterverzeichnissen wiederum Make aufzurufen.

Dieser Artikel untersucht ein paar wesentliche Probleme, auf die man stößt, wenn man bei der Entwicklung von Softwareprojekten mit dem rekursiven Make arbeitet. Außerdem wird eine einfache Lösung aufgezeigt und einige ihrer Auswirkungen werden untersucht.

Durch Rekursives Make erhält man einen Verzeichnisbaum, der ungefähr so aussieht:



Diese verschachtelte Modulhierarchie kann beliebig ausgeweitet werden. Reale Projekte haben oft zwei oder drei Ebenen.

### 1.1. Kenntnisse vorausgesetzt

Dieser Artikel setzt voraus, dass Sie mit Softwareentwicklung auf Unix, dem Make-Programm, den Grundsätzen von C-Programmierung und mit Dateiabhängigkeiten vertraut ist.

Weiterhin geht er davon aus, dass Sie GNU-Make auf ihrem System installiert haben und seine Funktionen mehr oder weniger gut kennen. Falls Sie eine eingeschränkte Version verwenden, kann es sein, dass Ihnen einige der unten beschriebenen Funktionen nicht zur Verfügung stehen.

## 2. Das Problem

Es gibt eine Vielzahl von Problemen mit rekursivem Make; in der Praxis begegnet man ihnen normalerweise täglich. Hier sind einige dieser Probleme aufgelistet:

- Es ist sehr schwierig, die Reihenfolge der Rekursion korrekt in die Unterverzeichnisse zu formulieren. Diese Reihenfolge ist nicht sehr stabil und ab und zu muss man sie von Hand korrigieren. Je mehr Verzeichnisse es gibt oder je mehr Ebenen dem Verzeichnisbaum hinzugefügt werden, desto unstabiler wird diese Reihenfolge.

- Es ist oft notwendig, die Unterverzeichnisse mehr als einmal zu durchlaufen, um das ganze System herzustellen. Das führt natürlich zu längeren Build-Zeiten.
- Da die Produktionszeiten sonst unverträglich lang wären, was mit einer Unproduktivität der Entwickler gleichbedeutend wäre, lässt man einige Informationen bezüglich der Abhängigkeiten der Verzeichnisse untereinander weg. Das führt in der Regel dazu, dass einige Produkte nicht aktualisiert werden, obwohl sie es sollten, wodurch häufige Produktionen von Null an erforderlich werden, um sicherzustellen, dass tatsächlich alles aufgebaut wird.
- Da die Abhängigkeiten zwischen den Verzeichnissen entweder weggelassen werden oder zu schwer auszudrücken sind, werden die Makefiles oft so geschrieben, dass sie zu viel machen, um sicherzustellen, dass wirklich nichts ausgelassen wurde.
- Die Ungenauigkeit der Abhängigkeiten oder einfach deren Fehlen kann zur Folge haben, dass ein Produkt sich nicht fehlerfrei bauen lässt. Dadurch wird eine sorgfältige Kontrolle des Build-Prozesses durch einen Entwickler erforderlich.
- Eine andere Folge des oben Beschriebenen ist, dass manche Projekte von den Möglichkeiten der Parallelisierung durch Make nicht profitieren können, weil das Build offensichtlich Unsinn macht.

Nicht jedes Projekt hat alle diese Probleme. Wenn sie auftauchen, tun sie das oft unregelmäßig und werden dann als unerklärbare einmalige Macken abgetan. In diesem Artikel sollen eine Reihe Symptome, die über einen langen Zeitraum in der Praxis beobachtet wurden, miteinander in Beziehung gesetzt werden. Es folgen eine systematische Analyse und ein Lösungsvorschlag.

It must be emphasized that this paper does not suggest that *make* itself is the problem. This paper is working from the premise that *make* does **not** have a bug, that *make* does **not** have a design flaw. The problem is not in *make* at all, but rather in the input given to *make* – the way *make* is being used.

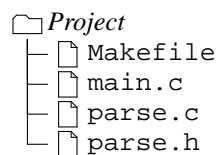
### 3. Analyse

Bevor es möglich ist, sich diesen scheinbar in keinerlei Beziehung zueinander stehenden Problemen zuzuwenden, ist es notwendig, zu verstehen, was Make bewirkt und wie es arbeitet. Dann erst kann man die Auswirkungen, die rekursives Make auf das Verhalten von Make hat, betrachten.

#### 3.1. Ganzprojekt-Make

Make ist ein Expertensystem. Sie versehen es mit einem Satz Regeln, nach denen gebaut werden soll, und einem Zielprodukt, das gebaut werden soll. Die Regeln können in paarweise geordnete Abhängigkeiten zwischen den Dateien zergliedert werden. Make liest die Regeln und ermittelt, wie das angegebene Zielprodukt gebaut werden soll. Sobald es entschieden hat, wie das Zielprodukt konstruiert werden soll, verfährt es entsprechend. Make ermittelt die Konstruktionsweise, indem es einen gerichteten azyklischen Graphen erstellt, den DAG (directed acyclic graph), der vielen Studenten der Computerwissenschaften vertraut ist. Die Knotenpunkte dieses Graphs sind die Dateien des Systems, die Kanten stellen die Abhängigkeiten zwischen den Dateien dar. Die Kanten des Graphen sind gerichtet, da die Abhängigkeiten paarweise geordnet sind, wodurch ein azyklischer Graph entsteht – was in einem ungerichteten Graphen wie ein Zyklus aussieht, wird im gerichteten Graphen durch die Richtung der Kanten aufgelöst.

In diesem Artikel wird eine kleines Beispielprojekt für die Analyse benutzt. Obwohl die Anzahl der Dateien in diese Beispiel klein ist, ist es komplex genug, um alle oben beschriebenen Probleme mit rekursivem Make zu demonstrieren. Zuerst einmal wird das Projekt in einer nicht rekursiven Form vorgestellt.



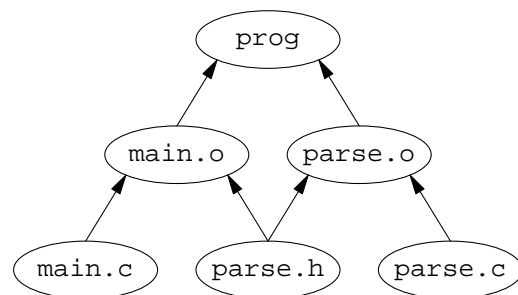
Das Makefile in diesem kleinen Projekt sieht so aus:

```

OBJ = main.o parse.o
prog: $(OBJ)
    $(CC) -o $@ $(OBJ)
main.o: main.c parse.h
    $(CC) -c main.c
parse.o: parse.c parse.h
    $(CC) -c parse.c
  
```

Ein paar der impliziten Regeln von Make sind hier explizit aufgeschrieben, um es für Sie einfacher zu machen, das Makefile in den zugehörigen DAG umzuformen.

Das oben genannte Makefile kann als DAG in folgender Weise dargestellt werden:



Aufgrund der Pfeile, die die Ordnung der Beziehungen der Dateien zueinander ausdrücken, handelt es sich um einen azyklischen Graphen. Wenn es den Pfeilen zufolge eine kreisförmige Abhängigkeit gäbe, läge ein Fehler vor.

Beachten Sie bitte, dass die Objektdateien (.o) von den Include-Dateien (.h) abhängig sind, obwohl es die Quelldateien (.c) sind, die das Einfügen vornehmen. Das hat folgenden Grund: Wird eine Include-Datei geändert, sind die Objektdateien nicht mehr aktuell, nicht die Quelldateien.

Der zweite Schritt des Make-Prozesses ist eine Postorder-Traversierung des DAG. Das bedeutet, dass die abhängigen Knotenpunkte zuerst besucht werden. Die eigentliche Reihenfolge der Traversierung ist nicht festgelegt, aber die meisten Make-Anwendungen gehen von oben nach unten und bei Kanten unter demselben Knotenpunkt von links nach rechts und die meisten Projekte verlassen sich stillschweigend auf dieses Verhalten. Die zuletzt geänderten Versionen aller Dateien werden untersucht und eine weiter oben liegende Datei wird als nicht mehr aktuell eingestuft, wenn irgendeine der darunterliegenden Dateien, von denen sie abhängig ist, jünger ist. Wenn eine Datei als nicht mehr aktuell klassifiziert wurde, wird die zu der entsprechenden Graphkante

gehörende Aktion ausgeführt (in dem oben aufgeführten Beispiel wäre das ein Kompilations- oder ein Bindeprozess).

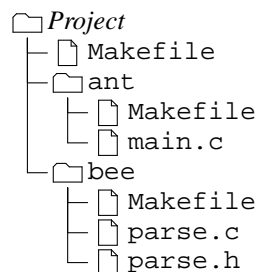
Die Anwendung von rekursivem Make beeinflusst beide Phasen der Make-Operation: Es veranlasst Make, einen ungenauen DAG zu erstellen und es zwingt Make, den DAG in einer unangebrachten Reihenfolge zu traversieren.

### 3.2. Rekursives Make

Um die Auswirkungen des rekursiven Makes zu untersuchen, teilen wir das obige Beispiel in zwei Module ein. Jedes Modul hat sein eigenes Makefile und ein Makefile für die oberste Ebene, deren Aufgabe es ist, jedes der Modul-Makefiles aufzurufen.

Dieses Beispiel ist absichtlich konstruiert und zwar durch und durch. Aber jede Modularität in Projekten ist in gewisser Weise ein Konstrukt. Bedenken Sie: Bei vielen Projekten ebnet der Linker am Ende alles wieder ein.

Die Verzeichnisstruktur ist folgendermaßen:



Das Makefile der obersten Ebene sieht oft sehr wie eine Shell-Befehlsdatei aus:

```

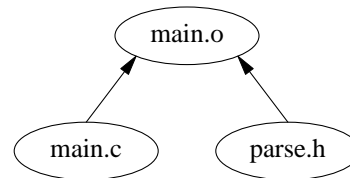
MODULES = ant bee
all:
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
  
```

Das ant/Makefile sieht so aus:

```

all: main.o
main.o: main.c ../bee/parse.h
$(CC) -I../bee -c main.c
  
```

und der entsprechende DAG sieht so aus:

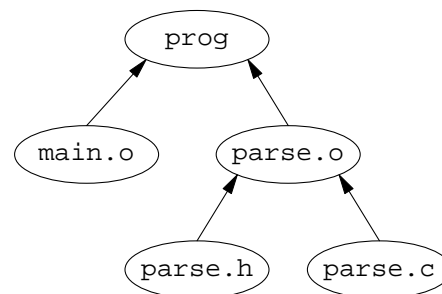


Das bee/Makefile sieht so aus:

```

OBJ = ../ant/main.o parse.o
all: prog
prog: (OBJ)
$(CC) -o $@ $(OBJ)
parse.o: parse.c parse.h
$(CC) -c parse.c
  
```

und der entsprechende DAG sieht so aus:



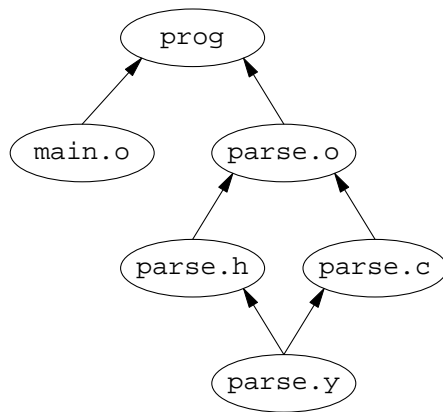
Schauen Sie sich die DAGs genau an. Sie stellen fest, dass keiner von ihnen komplett ist. Beiden DAGs fehlen Knoten und Kanten. Wenn die vollständige Produktion von der obersten Ebene ausgeführt wird, funktioniert alles.

Aber was passiert, wenn eine kleine Änderung eintritt? Was würde passieren, wenn `parse.c` und `parse.h` von einer `parse.y` Yacc Grammatik erzeugt würden? Dann würden folgende Zeilen dem `bee/Makefile` hinzugefügt:

```

parse.c parse.h: parse.y
$(YACC) -d parse.y
mv y.tab.c parse.c
mv y.tab.h parse.h
  
```

Und die entsprechenden Veränderungen des DAGs sähen so aus:



Diese Veränderung hat eine einfache Folge: Wenn `parse.y` editiert wird, wird `main.o` nicht korrekt aufgebaut. Das rührt daher, dass der DAG für `ant` nur einige der Abhängigkeiten von `main.o` kennt, während der DAG für `bee` sogar keine von ihnen kennt.

Um zu verstehen, warum das passiert, ist es notwendig, sich die Aktionen, die `Make` von der obersten Ebene aus unternimmt, anzuschauen. Nehmen Sie einmal an, dass das Projekt in sich konsistent ist. Jetzt editieren Sie `parse.y`, so dass die generierte `parse.h` nicht-triviale Veränderungen aufweist. Wenn nun das Haupt-`Make` aufgerufen wird, wird erst `ant` und dann `bee` besucht. Aber `ant/main.o` ist noch nicht rekompiliert, weil `bee/parse.h` noch nicht regeneriert wurde und deshalb noch nicht anzeigt, dass `main.o` nicht mehr aktuell ist. Das ist auch immer noch nicht der Fall, wenn das rekursive `Make` `bee` besucht, wobei `parse.c` und `parse.h` und zuletzt `parse.o` rekonstruiert werden. Wenn das Programm gelinkt wird, sind `main.o` und `parse.o` in erheblichem Maße nicht kompatibel. D. h. das Programm funktioniert nicht.

### 3.3. Herkömmliche Lösungen

Es gibt drei herkömmliche Korrekturmöglichkeiten für die oben beschriebene Störung.

#### 3.3.1. Umstrukturierung

Die erste ist, die Anordnung der Module in dem Haupt-`Makefile` von Hand zu berichtigen. Die Frage ist, warum diese Korrektur überhaupt notwendig ist. Schließlich soll `Make` ein Expertensystem sein. Hat `Make` irgendeinen Fehler oder ging etwas anderes schief?

Zur Beantwortung dieser Frage muss man nicht den Graphen, sondern die Anordnung der Traversierung des Graphen anschauen. Um fehlerlos zu arbeiten, muss `Make` eine Postorder-Traversierung vornehmen, aber dadurch, dass der DAG in zwei Teile geteilt wurde, war es `Make` nicht mehr möglich den Graphen in der notwendigen Reihenfolge zu traversieren – stattdessen wurde von dem Projekt eine Reihenfolge vorgeschrieben. Eine Reihenfolge, die, wenn man den Originalgraphen betrachtet, schlichtweg falsch ist. Indem man das Haupt-`Makefile` korrigiert, stellt man eine Reihenfolge her, die der, die `Make` hätte benutzen können, ähnlich ist. Solange, bis die nächste Abhängigkeit hinzugefügt wird...

Bitte beachten Sie, dass paralleles Build (`make -j`) viele der bei der manuellen Umstrukturierung gemachten Annahmen stillschweigend außer Kraft setzt, wodurch diese Lösung nutzlos wird. Außerdem führen alle untergeordneten `Makes` ebenfalls mehrere Aktionen gleichzeitig aus.

#### 3.3.2. Wiederholung

Bei der zweiten herkömmlichen Lösung lässt man das Haupt-`Makefile` mehrere Male durchlaufen. Das sieht ungefähr so aus:

```

MODULES = ant bee
all:
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
  for dir in $(MODULES); do \
    (cd $$dir; ${MAKE} all); \
  done
  
```

Dadurch wird die Produktionszeit verdoppelt. Aber das ist nicht alles: Es gibt keine Garantie, dass zwei Durchläufe ausreichen! Die Höchstzahl der Durchläufe ist nicht einmal proportional zu der Anzahl der Module, sondern proportional zu der Anzahl der Graphkanten, die Modulgrenzen kreuzen.

#### 3.3.3. Des Guten zu viel

Wir haben schon ein Beispiel gesehen, bei dem rekursives `Make` zu wenig baute, aber ein anderes verbreitetes Problem ist, dass zu viel gebaut wird. Bei der dritten herkömmlichen Lösung fügt man dem `ant/Makefile` sogar noch mehr Zeilen hinzu:

```
.PHONY: ../bee/parse.h
../bee/parse.h:
    cd ../bee; \
    make clean; \
    make all
```

Das bedeutet, dass immer wenn `main.o` gebaut wird, `parse.h` als nicht aktuell betrachtet wird. Alle Inhalte von `bee` werden jedesmal von neuem gebaut einschließlich `parse.h`. Auch `main.o` wird immer wieder neu gebaut, selbst wenn alles in sich konsistent war.

Beachten Sie bitte, dass bei dieser Lösung `make -j` (paralleles Build) viele der angenommenen Anordnungen stillschweigend außer Kraft setzt, wodurch diese Lösung nutzlos wird, weil alle der untergeordneten Makes ihre Builds gleichzeitig durchführen (`clean` und dann `all`), wobei sie sich ständig gegenseitig stören.

#### 4. Vorsorge

Die obige Analyse basiert auf einer einfachen Aktion: Der DAG wurde künstlich in unvollständige Stücke unterteilt. Diese Unterteilung hat all die Probleme, die man von rekursiv verwendetem Make kennt, zur Folge.

Hat Make es nicht richtig verstanden? Nein, das ist nicht der Grund. Hier liegt ein Fall des uralten GIGO-Prinzips (Garbage in, Garbage out) vor: Wo man Müll hereintut, kommt Müll heraus. Unvollständige Makefiles sind fehlerhafte Makefiles.

Wenn Sie diese Probleme vermeiden wollen, zerlegen Sie den DAG nicht in einzelne Teile. Verwenden sie stattdessen ein einziges Makefile für das ganze Projekt. Die Rekursion an sich ist nicht schädlich, sondern das verstümmelte Makefile, das man für die Rekursion verwendet, ist falsch. Es ist kein Fehler von Make, dass das rekursive Make nicht funktioniert; es macht das bestmögliche aus den mangelhaften Eingabeinformationen.

*"Aber, aber, aber... Das können Sie doch nicht machen!", höre ich Sie jammern, "ein einziges Makefile ist viel zu groß, man kann es gar nicht warten, es ist viel zu schwierig, die Regeln dafür zu schreiben, der Platz im Hauptspeicher wird nicht ausreichen, ich will nur meinen kleinen Teil bauen, das Build wird viel zu lange dauern. Es ist schlichtweg nicht praktikabel."*

Das sind stichhaltige Einwände und oft ziehen Make-Benutzer aus ihnen den Schluss, dass es keinerlei kurz- oder längerfristigen Nutzen für sie bringen würde, ihren Build-Prozess einmal zu überarbeiten. Diese Schlussfolgerung ist auf überholten, falschen Annahmen gegründet, die sich jedoch hartnäckig halten.

In den nächsten Abschnitten wird der Reihe nach auf jeden dieser Einwände eingegangen.

#### 4.1. Ein einziges Makefile ist zu groß

Wäre die vollständige Produktionsbeschreibung für das ganze Projekt in einem einzigen Makefile untergebracht, würde dieser Satz sicherlich zutreffen. Aber moderne Make-Implementierungen kennen Include-Anweisungen. Dadurch, dass ein entscheidendes Fragment jedes Moduls eingeschlossen wird, ist die Gesamtgröße des Makefiles und seiner Includes nicht unbedingt größer als die des beim rekursiven Vorgehen verwendeten Makefiles.

#### 4.2. Ein einziges Makefile kann man nicht warten

Ein Haupt-Makefile, das eine Referenz auf ein Fragment jedes Moduls enthält, ist nicht komplexer als das beim rekursiven Make verwendete. Da der DAG nicht zerstückelt ist, ist diese Art Makefile sogar weniger komplex und damit besser zu warten, einfach weil weniger Korrekturen notwendig sind, um seine Funktionstüchtigkeit zu erhalten.

Rekursive Makefiles beinhalten eine Menge Wiederholungen. Bei vielen Projekten wird dieses Problem durch die Verwendung von Include-Dateien gelöst. Wenn man ein einziges Makefile für das Projekt benutzt, fällt der Bedarf an diesen "allgemeinen" Include-Dateien weg – das eine Makefile ist der allgemeine Teil.

#### 4.3. Es ist zu schwierig, die Regeln zu formulieren

Man muss lediglich den Verzeichnisteil an einer Reihe von Stellen in die Dateinamen einfügen. Das ist notwendig, weil Make vom Verzeichnis der obersten Ebene ausgeführt wird; in dem aktuellen Verzeichnis erscheint die Datei nicht. Man muss sich nicht darum kümmern, wo die Ausgabedatei ausdrücklich in einer Regel genannt wird.

GCC erlaubt im Zusammenhang mit der `-c`-Option eine `-o`-Option und GNU-Make weiß

das. Daraus folgt eine implizite Kompilationsregel, die die Ausgabedatei an die richtige Stelle setzt. Ältere und weniger intelligente Compiler lassen die `-o`-Option mit der `-c`-Option jedoch vielleicht nicht zu und lassen die Objektdatei im Verzeichnis der obersten Ebene zurück (d. h. im falschen Verzeichnis). Es gibt drei Möglichkeiten, wie Sie diesen Fehler korrigieren können: Entweder Sie beschaffen sich GNU-Make und GCC, sie überschreiben die Build-Regel durch eine korrekt funktionierende oder Sie beschweren sich bei ihrem Händler.

Auch K&R; C-Compiler beginnen den in Anführungszeichen notierten Includepfad (`#include "filename.h"`) vom aktuellen Verzeichnis aus. Das bedeutet, dass sie nicht das ausführen, was Sie wollen. ANSI C-konforme C-Compiler aber beginnen den in Anführungszeichen notierten Includepfad von dem Verzeichnis aus, in dem die Quelldatei erscheint; hier sind keine Veränderungen der Quelle notwendig. Falls Sie keinen ANSI C-konformen C-Compiler besitzen, sollten Sie in Betracht ziehen, so bald wie möglich einen GCC auf Ihrem System zu installieren.

#### 4.4. Ich will doch nur mein kleines Teilprodukt bauen

Die meiste Zeit sind Entwickler mitten im Projektbaum beschäftigt. Sie bearbeiten ein oder zwei Dateien lassen dann Make durchlaufen, was ihre Veränderungen übersetzt, und probieren sie aus. Diesen Arbeitsschritt führen sie täglich Dutzende oder Hunderte Male aus. Es wäre absurd, wenn sie gezwungen wären jedesmal das ganze Projekt zu bauen.

Entwickler haben immer die Option, ein besonderes Zielprodukt für Make zu definieren. Das gilt immer, wir verlassen uns lediglich gewöhnlich auf das im Makefile des aktuellen Verzeichnisses vorgegebene Zielprodukt, um unsere Befehlszeile zu verkürzen. Man kann also auch mit einem Ganzprojekt-Makefile sein kleines Teilprodukt bauen, indem man einfach ein bestimmtes Zielprodukt definiert und, falls die Befehlszeile zu lang wird, ein Pseudonym verwendet.

Es stellt sich aber auch die Frage, ob es immer so absurd ist, das ganze Projekt zu bauen. Wenn z. B. eine in einem Modul vorgenommene Änderung in anderen Modulen Auswirkungen hat, weil eine Abhängigkeit existiert, die dem Entwickler nicht bewusst ist (aber dem Makefile ist sie

bewusst), wäre es dann nicht besser, wenn der Entwickler das so schnell wie möglich herausfindet? Solche Abhängigkeiten werden gefunden, weil der DAG vollständiger ist als beim rekursiven Vorgehen.

In den seltensten Fällen sind Entwickler erfahrene, alte Hasen, die jede einzelne der Millionen von Zeilen Code des Produktes auswendig kennen. Meistens haben sie einen zeitlich begrenzten Vertrag oder sie sind jüngere Mitarbeiter. Sie wollen natürlich nicht, dass Auswirkungen wie die eben beschriebene entdeckt werden, nachdem Ihre Änderungen in den Hauptcode eingefügt wurden, sondern würden sie gerne ganz in Ruhe in ihrem lokalen Arbeitsbereich entdecken, weit weg vom Hauptcode.

Wenn Sie "nur Ihr kleines Teilprodukt" bauen wollen, weil Sie befürchten, dass ein Bau des ganzen Projekts infolge der Verzeichnisstruktur, die Sie in Ihrem Projekt verwendet haben, die Master Source des Projekts beschädigen könnte, lesen Sie bitte das Kapitel *Projekte im Vergleich zu Sandkästen*.

#### 4.5. Das Bauen dauert zu lange

Diese Aussage kann man für eine von zwei Situationen machen: 1. Die Durchführung des Make eines ganzen Projekts dauert, obwohl alles aktualisiert ist, unvermeidlich sehr lange. 2. Diese unvermeidbaren Verzögerungen sind unakzeptabel, wenn der Entwickler die eine Datei, die er verändert hat, schnell kompilieren und linken will.

##### 4.5.1. Builds von Projekten

Stellen Sie sich ein hypothetisches Projekt vor mit 1000 Quelldateien (.c), von denen jede ihre Aufrufschnittstelle hat, welche in der zugehörigen Include-Datei (.h) mit Definitionen, Typvereinbarungen und Funktionsdeklarationen definiert ist. Diese 1000 Quelldateien beinhalten ihre eigenen Interfacedefinitionen und zusätzlich die Interfacedefinitionen aller Module, die sie aufrufen können. Diese 1000 Quelldateien werden in 1000 Objektdateien übersetzt, die wiederum zu einem ausführbaren Programm gebunden werden. In diesem System gibt es etwa 3000 Dateien, über die Make informiert werden muss. Außerdem muss Make über die Includeabhängigkeiten informiert werden und man muss untersuchen, ob implizite Regeln (z. B. .y - .c) anwendbar sind.

Um den DAG zu erstellen, muss Make für 3000 Dateien deren Änderungsdatum ermitteln und

außerdem noch für etwa 2000 zusätzliche Dateien, abhängig davon, welche impliziten Regeln Ihr Make kennt und welche Ihr Makefile nicht ausgeschaltet hat. Auf dem bescheidenen 66MHz i486 des Authors dauert das etwa 10 Sekunden; auf systemeigenen Laufwerken auf schnelleren Hardwarebasen geht es sogar noch schneller. Mit NFS über 10MB Ethernet dauert es ebenfalls etwa 10 Sekunden, gleichgültig, von welcher Hardwarebasis die Aktion ausgeführt wird.

Das ist eine erstaunliche Statistik. Stellen Sie sich einmal vor, dass Sie ihn der Lage sind, eine einzige von 1000 Quelldateien in nur 10 Sekunden – zuzüglich der Zeit für das Kompilieren selbst – zu kompilieren.

Die Dateien auf 100 Module zu verteilen und den Prozess als ein rekursives Make durchzuführen, dauert immerhin 25 Sekunden. Die wiederholte Prozesserzeugung für die untergeordneten Make-Aufrufe nehmen eine relativ lange Zeit in Anspruch.

Aber warten Sie einen Moment! Bei realen Projekten mit weniger als 1000 Dateien dauert es sehr viel länger als 25 Sekunden bis Make herausgefunden hat, das es nichts zu tun hat. Für manche Projekte wäre es ein Fortschritt, wenn es nur 25 Minuten dauerte. Dieses Beispiel zeigt uns, dass es nicht die Anzahl der Dateien ist, die bremst (das dauert nur 10 Sekunden), und es ist auch nicht die wiederholte Prozesserzeugung für die untergeordneten Make-Aufrufe (die dauert nur 15 Sekunden). Was aber nimmt dann so viel Zeit in Anspruch?

Bei traditionellen Lösungen des durch rekursives Make entstandenen Problems werden die untergeordneten Make-Aufrufe oft über das hier beschriebene Minimum erhöht: z. B. um vielfältige Wiederholungen (3.3.2.) durchzuführen oder um Modulgrenzen überschreitende Abhängigkeiten (3.3.3.) übermäßig abzudecken. Das kann lange Zeit dauern, besonders, wenn beides zusammenkommt. Aber es ist nicht für die besonders langen Produktionszeiten verantwortlich. Was nimmt also noch soviel Zeit in Anspruch?

Die Komplexität des Makefiles ist so zeitaufwendig. Mehr darüber im Kapitel *Effiziente Makefiles*.

#### 4.5.2. Builds während der Entwicklung

Wenn es – wie bei dem Beispiel mit den 100 Dateien – nur 10 Sekunden dauert, herauszufinden welche Datei neu übersetzt werden muss, wird die Produktivität der Entwickler nicht bedeutend eingeschränkt, wenn sie ein Ganzprojekt-Make durchführen anstatt eines modulspezifischen Makes. Der Vorteil für das Projekt ist, dass der modulzentrierte Entwickler in entscheidenden Momenten (und nur in den entscheidenden) daran erinnert wird, dass seine Arbeit auch weitgehendere Auswirkungen hat.

Die ständige Verwendung von C-Include-Dateien, die genaue Interface-Definitionen (einschließlich Funktionsprototypen) enthalten, würde in vielen Fällen zu Übersetzungsfehlern führen, was wiederum ein fehlerhaftes Produkt zur Folge hätte. Werden Builds für das ganze Projekt durchgeführt, entdecken Entwickler solche Fehler sehr früh im Entwicklungsprozess und sind in der Lage Fehlerbehebungen dann vorzunehmen, wenn sie den geringsten Aufwand verursachen.

#### 4.6. Ihre Speicherkapazitäten sind erschöpft

Das ist der interessanteste Einwand. Irgendwann einmal vor langer Zeit auf einem Mikroprozessor weit, weit weg ist das vielleicht sogar vorgekommen. Als Feldman das erste Make entwickelte, schrieb man das Jahr 1978 und er benutzte eine PDP11. Unix-Prozesse waren auf 64B Daten beschränkt.

Solch ein Rechner würde bei dem oben genannte Beispiel mit seinen 3000 in dem Ganzprojekt-Makefile genau beschriebenen Dateien wahrscheinlich nicht zulassen, einen DAG und die Regeln im Hauptspeicher zu halten.

Aber wir benutzen keine PDP11 mehr. Die physische Speicherkapazität eines kleinen modernen Computers überschreitet 10MB und der virtuelle Speicher oft 100MB. Ein Projekt mit hunderttausenden Quelldateien ist notwendig, um die virtuelle Speicherkapazität eines kleinen modernen Rechners zu erschöpfen. Da das Beispielprojekt mit 1000 Quelldateien weniger als 100KB Speicherplatz in Anspruch nimmt (probieren Sie es aus, Sie werden sehen, dass es stimmt), ist es sehr unwahrscheinlich, dass irgendein Projekt, das man in einem einzigen Verzeichnisbaum auf einem einzigen Laufwerk verwalten kann, die Speicherkapazitäten ihres Rechners übersteigt.



#### 4.7. Warum erstellt man den DAG nicht in den Modulen?

Oben wurde erläutert, dass die Gründe für die Probleme bei rekursivem Make in dem unvollständigen DAG zu suchen sind. Dem zufolge kann das Problem gelöst werden, indem man die fehlenden Abhängigkeiten wieder hinzufügt, ohne die existierende Investition in das rekursive Make aufzugeben.

- Der Entwickler darf es jedoch nicht vergessen. Die Folgen trägt nicht er, sondern die Entwickler der anderen Module bekommen sie zu spüren. Es gibt keine bessere Methode, einen Entwickler daran zu erinnern, etwas zu tun, als den Zorn der Kollegen.
- Es ist schwierig, herauszufinden, an welcher Stelle die Änderungen vorgenommen werden müssen. Möglicherweise muss jedes Makefile im ganzen Projekt auf erforderliche Änderungen hin untersucht werden. Natürlich können Sie auch darauf warten, dass Ihre Kollegen die Stellen für Sie finden.
- Die Includeabhängigkeiten werden unnötigerweise neu berechnet oder werden nicht korrekt interpretiert. Das passiert, weil Make auf Zeichenketten basiert, wodurch `.` und `../ant` zwei verschiedene Stellen sind, selbst wenn Sie im `ant`-Verzeichnis stehen. Das ist von Bedeutung, wenn Includeabhängigkeiten automatisch berechnet werden, wie es bei allen großen Projekten der Fall ist.

Indem man sicherstellt, dass jedes Makefile vollständig ist, kommt man an den Punkt, dass das Makefile wenigstens eines Moduls bereits die Informationen des Ganzprojekt-Makefiles umfasst (Sie dürfen nicht vergessen, dass diese Module ein einziges Projekt formen und daher miteinander verbunden sind), wodurch das rekursive Make überflüssig wird.

#### 5. Effiziente Makefiles

Das zentrale Thema dieses Artikels sind die semantischen Nebenwirkungen des künstlichen In-Stücke-Zerteilens eines Makefiles, das notwendig ist, um ein rekursives Make durchzuführen. Wenn Sie jedoch viele Makefiles haben, wird die Geschwindigkeit, in der Make diese Menge Dateien interpretieren kann, ebenfalls zum Thema.

Builds können aus zwei Gründen übermäßig lange dauern: Die herkömmlichen Korrekturen für den zerteilten DAG bauen zu viel oder Ihr

Makefile ist nicht effizient.

#### 5.1. Verzögerte Auswertung

Make muss den Text eines Makefiles irgendwie aus einer Textdatei lesen und verstehen, so dass ein DAG erstellt und die angegebenen Aktionen den Kanten zugeordnet werden können. All das wird im Speicher festgehalten.

Die Eingabesprache für Makefiles ist irreführend einfach. Sie ist textbasiert, im Gegensatz zu den tokenbasierten z. B. für C und AWK. Das ist ein entscheidender Unterschied, der Neulingen genauso wie Experten oft entgeht. Make tut das absolute Minimum, um die Eingabezeilen zu verarbeiten und sie im Hauptspeicher zu verstauen.

Folgende Zuordnung ist ein Beispiel dafür:

```
OBJ = main.o parse.o
```

Wenn Menschen das lesen, heißt das, der Variablen OBJ sind zwei Dateinamen `main.o` und `parse.o` zugeordnet. Aber Make versteht das ganz anders. OBJ wird die Zeichenfolge `main.o parse.o` zugeordnet. Und es wird noch schlimmer:

```
SRC = main.c parse.c
OBJ = $(SRC:.c=.o)
```

In diesem Fall erwartet der Mensch, dass OBJ durch Make zwei Dateinamen zugeordnet werden, aber Make ordnet in Wirklichkeit die Zeichenkette `$(SRC:.c=.o)` zu. Das rührt daher, dass es sich um eine Makrosprache mit verzögerter Auswertung handelt im Gegensatz zu einer mit Variablen und sofortiger Auswertung.

Wenn es Ihnen nicht zu schwierig erscheint, schauen Sie sich die folgende Makefile an:

```
SRC = $(shell echo 'Ouch!' \
1>&2 ; echo *.[cy])
OBJ = \
$(patsubst %.c,%.o,\
$(filter %.c,$(SRC))) \
$(patsubst %.y,%.o,\
$(filter %.y,$(SRC)))
test: $(OBJ)
$(CC) -o $@ $(OBJ)
```

Wie oft wird das Shell-Kommando ausgeführt? Autsch! Er wird, allein um den DAG zu konstruieren, zweimal ausgeführt und noch zweimal, wenn die Regel ausgeführt werden muss. Wenn dieser Shell-Befehl nichts komplexes oder zeitaufwendiges ausführt (was er aber

normalerweise tut), braucht er viermal so lang, wie erwartet.

Aber es lohnt sich, andere Teile dieses OBJ-Makros näher anzuschauen. Jedesmal, wenn es genannt wird, werden riesige Mengen von Vorgängen abgewickelt:

- Der Parameter für Shell ist eine einzige Zeichenkette (alle Built-in-Funktionen haben als Parameter eine einzige Zeichenkette). Die Zeichenkette wird in der untergeordneten Shell ausgeführt und die Standardausgabe dieses Kommandos wird wieder eingegeben, wobei Zeilenumbrüche in Leerschritte verwandelt werden. Das Ergebnis ist eine einzige Zeichenkette.
- Der Parameter zum Filtern ist eine einzige Zeichenkette. Dieser Parameter wird beim ersten Komma in zwei Zeichenketten zerlegt. Jede der beiden Zeichenketten wird dann in durch Leerschritte getrennte Teilketten aufspalten. Die erste Reihe entspricht den Mustern und die zweite den Dateinamen. Dann wird für jede Musterteilkette, mit der eine Dateiname-teilkette übereinstimmt, der Dateiname in die Ausgabe eingefügt. Sobald die Ausgabe vollständig ist, werden die Teilketten wieder zu einer einzigen durch Leerschritte unterteilten Zeichenkette zusammengesetzt.
- Der Parameter zur Musterersetzung ist eine einzige Zeichenkette. Dieser Parameter wird beim ersten und zweiten Komma in drei Ketten aufgespalten. Die dritte Kette wird dann in durch Leerschritte getrennte Teilketten zerteilt, die den Dateinamen entsprechen. Dann wird jeder Dateiname, der mit der ersten Kette übereinstimmt, gemäß der zweiten Kette ersetzt. Wenn ein Dateiname nicht mit der ersten Kette übereinstimmt, passiert er unverändert. Sobald die Ausgabe vollständig erzeugt wurde, werden die Teilketten wieder zu einer einzigen durch Leerschritte unterteilten Zeichenkette zusammengesetzt.

Sie sehen, wie oft diese Zeichenketten aufgespalten und wieder zusammengesetzt werden. Sie sehen, auf wieviel verschiedene Weisen das passiert. Das ist langsam. In unserem Beispiel gibt es nur zwei Dateien, aber stellen Sie sich vor, wie lange diese Prozeduren für 1000 Dateien dauern. Diese Aktion viermal auszuführen, ist sehr ineffizient.

Wenn Sie ein einfaches Make, das über keine Ersetzung und keine Built-in-Funktionen verfügt, benutzen, beeinträchtigt Sie das nicht. Aber ein

modernes Make hat viele Built-in-Funktionen und kann sogar nebenbei Shell-Kommandos ausführen. Die Semantik der Textmanipulation von Make führt, verglichen mit C oder AWK, zu einer sehr CPU-intensiven Zeichenkettenmanipulation in Make.

## 5.2. Unmittelbare Auswertung

Moderne Make-Ausführungen haben einen `:=` Zuordnungsoperator für unmittelbare Auswertung. Das oben genannte Beispiel kann folgendermaßen neu geschrieben werden:

```
SRC := $(shell echo 'Ouch!' \
1>&2 ; echo *.cy)
OBJ := \
$(patsubst %.c,%.o,\
$(filter %.c,$(SRC))) \
$(patsubst %.y,%.o,\
$(filter %.y,$(SRC)))
test: $(OBJ)
$(CC) -o $@ $(OBJ)
```

Beachten Sie, dass beide Zuordnungen Zuordnungen für unmittelbare Auswertung sind. Wäre die erste keine, würde das Shell-Kommando immer zweimal ausgeführt werden. Wäre die zweite keine, würden die aufwendigen Ersetzungen mindestens zweimal, möglicherweise sogar viermal durchgeführt werden.

Als Daumenregel gilt: Benutzen Sie immer Zuordnungen für unmittelbare Auswertung, es sei denn Sie wünschen bewusst eine verzögerte Auswertung.

## 5.3. Include-Dateien

Viele Makefiles führen denselben Textbearbeitungsvorgang (z. B. die o. g. Filter) bei jedem einzelnen Make-Durchlauf erneut durch; das Ergebnis dieser Prozeduren ändert sich jedoch selten. Wenn es auch praktisch ist, ist es doch effizienter, die Ergebnisse der Textbearbeitung einer Datei zu speichern und diese Datei in das Makefile einzufügen.

## 5.4. Abhängigkeiten

Seien Sie nicht geizig mit Include-Dateien. Man kann sie, verglichen mit `$(shell)`, mit relativ wenig Aufwand lesen und sie beeinträchtigen auch die Effizienz nur wenig.

Um ein Beispiel dafür zu zeigen, ist es erst einmal notwendig, eine nützliche Eigenschaft des GNU-Makes zu beschreiben: Wenn Make das Makefile gelesen hat und einige ihrer

Include-Dateien sind nicht mehr aktuell (oder existieren noch nicht), werden sie neu erzeugt und Make beginnt noch einmal. Das führt dazu, dass Make jetzt mit aktuellen Include-Dateien arbeitet. Diese Funktion kann genutzt werden, um eine automatische Berechnung der Include-Dateiabhängigkeiten für C-Quellen zu realisieren. Die naheliegendste Art, es auszuführen, hat jedoch einen kleinen Fehler.

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
    $(CC) -o $@ $(OBJ)
include dependencies
dependencies: $(SRC)
    depend.sh $(CFLAGS) \
        $(SRC) > $@
```

Das `depend.sh`-Skript druckt seine Zeilen in folgender Weise:

```
Datei.o: Datei.c Include.h...
```

Die einfachste Art, das umzusetzen, ist, GCC zu verwenden, aber Sie brauchen eine entsprechende AWK-Skript oder C-Programm, wenn Sie einen anderen Compiler verwenden:

```
#!/bin/sh
gcc -MM -MG "$@"
```

Diese Methode, C-Includeabhängigkeiten aufzufinden, hat mehrere ernste Mängel, aber der verbreitetste Mangel ist, dass die Abhängigkeitsdatei selbst nicht von den C-Include-Dateien abhängig ist. Das bedeutet, dass sie nicht neu erzeugt wird, wenn sich Include-Dateien verändern. Es gibt im DAG keine Kante zwischen einem Knotenpunkt der Abhängigkeiten und einem Knotenpunkt der Include-Dateien. Wenn sich eine Include-Datei ändert, weil sie eine andere Datei aufnimmt (verschachtelte Include), werden die Abhängigkeiten nicht neu bestimmt und die C-Datei wird möglicherweise nicht neu übersetzt, wodurch das Programm nicht korrekt neu gebaut wird.

Das ist ein klassischer Fall von *zu wenig bauen*. Das Problem wird dadurch verursacht, dass man Make unzulängliche Informationen gibt und es dadurch veranlasst, einen unzureichenden DAG zu erstellen und das falsche Ergebnis zu liefern.

Die traditionelle Lösung ist, zu viel zu bauen:

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
    $(CC) -o $@ $(OBJ)
include dependencies
.PHONY: dependencies
dependencies: $(SRC)
    depend.sh $(CFLAGS) \
        $(SRC) > $@
```

Jetzt werden selbst, wenn das Projekt aktualisiert ist, die Abhängigkeiten neu ermittelt. Wenn Projekte groß sind, ist das eine erhebliche Zeitverschwendung und kann einer der Hauptgründe dafür sein, dass Make sehr lange braucht, um herauszufinden, dass nichts zu tun ist.

Es gibt ein zweites Problem: Wenn sich irgendeine der C-Dateien verändert, werden wieder für alle C-Dateien die Includeabhängigkeiten neu berechnet. Das ist genauso wenig effizient, als hätte man ein Makefile, das folgendermaßen aussieht:

```
prog: $(SRC)
    $(CC) -o $@ $(SRC)
```

In exakter Entsprechung zum C-Fall benötigt man hier eine Zwischenform. Dieser verleiht man gewöhnlich ein `.d`-Suffix. Dank der Tatsache, dass man in einer Include-Anweisung mehr als eine Datei benennen kann, existiert keine Notwendigkeit alle `.d`-Dateien zu verknüpfen:

```
SRC := $(wildcard *.c)
OBJ := $(SRC:.c=.o)
test: $(OBJ)
    $(CC) -o $@ $(OBJ)
include $(OBJ:.o=.d)
%.d: %.c
    depend.sh $(CFLAGS) $* > $@
```

Diese Form erfordert noch eine Korrektur: Genau wie die Objektdateien (`.o`) hängen die Abhängigkeitsdateien (`.d`) von den Quelldateien und den Include-Dateien ab.

```
Datei.d   Datei.o:   Datei.c
Include.h
```

Das bedeutet, dass man wieder an dem `depend.sh`-Skript herumbasteln muss:

```
#!/bin/sh
gcc -MM -MG "$@" |
sed -e 's@^\(.*\)\.o:@\1.d \1.o:@'
```

Durch dieses Vorgehen bei der Bestimmung der Abhängigkeiten der Include-Dateien erhöht sich die Anzahl der Dateien, die das Makfile beinhaltet, im Gegensatz zur herkömmlichen Methode. Dateien zu öffnen ist jedoch weniger aufwendig, als jedesmal wieder alle Abhängigkeiten zu ermitteln. Normalerweise bearbeitet ein Entwickler ein oder zwei Dateien, bevor er ein Re-Build durchführt; bei dieser Methode wird genau die betroffene Abhängigkeitsdatei wiederhergestellt (oder mehr als eine, wenn Sie eine Include-Datei bearbeitet haben). Unterm Strich erfordert diese Vorgehensweise weniger Rechenarbeit durch die CPU und weniger Zeitaufwand.

Muss bei einem Build nichts gemacht werden, tut Make tatsächlich nichts und findet das auch sehr schnell heraus.

Die beschriebene Technik nimmt jedoch an, dass Ihr Projekt vollständig in ein Verzeichnis hineinpasst. Das ist bei großen Projekten gewöhnlich nicht der Fall. Also muss noch einmal an dem depend.sh-Skript herumgebastelt werden:

```
#!/bin/sh
DIR="$1"
shift 1
case "$DIR" in
  "" | ".")
    gcc -MM -MG "$@" |
    sed -e 's@^\(.*\)\.o:@\1.d \
        \1.o:@'
    ;;
  *)
    gcc -MM -MG "$@" |
    sed -e "s@^\(.*\)\.o:@$DIR/\
        \1.d $DIR/\1.o:@"
    ;;
esac
```

Und die Regel muss auch geändert werden, damit das Verzeichnis, wie das Skript annimmt, als erster Parameter übergeben wird.

```
%.d: %.c
    depend.sh `dirname $*` \
    $(CFLAGS) $* > $@
```

Bitte beachten Sie, dass die Namen der .d-Dateien relativ zum Verzeichnis der obersten Ebene sind. Man kann sie auch so schreiben, dass sie von jeder Ebene aus benutzt werden, aber das geht über den Rahmen dieses Artikels hinaus.

## 5.5. Multiplikator

Alle in diesem Kapitel beschriebenen Ineffizienzen hängen zusammen. Wenn Sie 100 Makefile-Interpretationen für jedes Modul einmal durchführen, kann es sehr lange dauern, 1000 Quelldateien zu überprüfen; ebenso wenn die Interpretation komplexe Bearbeitungsprozesse erfordert oder unnötige Arbeiten ausführt, oder beides. Das Ganzprojekt-Make muss auf der anderen Seite nur ein einziges Makefile interpretieren.

## 6. Projekte im Vergleich zu Sandkästen

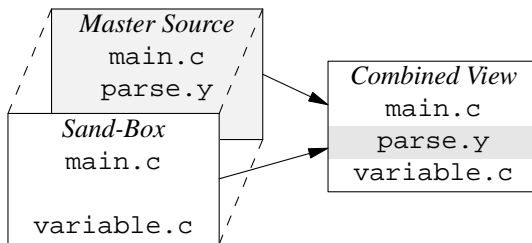
Das oben Besprochene setzt voraus, dass ein Projekt sich im Rahmen eines einzigen Verzeichnisbaums befindet und das ist in der Regel der Idealfall. Die Arbeitswirklichkeit in großen Softwareprojekten führt dagegen oft zu bizarren und wundervollen Verzeichnisstrukturen, damit die Entwickler in verschiedenen Bereichen des Projekts arbeiten können, ohne vollständige Kopien des Projekts zu benötigen und damit wertvollen Speicherplatz zu verschwenden.

Sie können das hier vorgeschlagene Ganzprojekt-Make unpraktisch finden, weil es nicht zu den entwickelten Methoden Ihres Entwicklungsprozesses passt.

Das hier vorgestellte Ganzprojekt-Make hat eine Auswirkung auf die Entwicklungsmethoden: Es schafft eine sauberere und einfachere Produktionsumgebung für Ihre Entwickler. Indem man die VPATH-Funktion von Make anwendet, ist es möglich, dass Sie nur jene Dateien, die Sie bearbeiten müssen, in Ihren privaten Arbeitsbereich, oft auch Sandkasten genannt, kopieren.

Die einfachste Weise, zu erklären, was VPATH tut, ist, eine Analogie zu dem Suchpfad für Include-Dateien herzustellen, indem man -I-Pfadoptionen für den C-Compiler benutzt. Diese Optionen beschreiben, wo man nach Dateien suchen muss, genauso wie VPATH-Make anweist, wo es nach Dateien suchen muss.

Durch die Verwendung von VPATH wird es möglich, die sich im Sandkasten befindenden Dateien oben auf der Master-Source des Projekts *aufzustapeln*. Auf diese Weise bekommen die Dateien, die sich im Sandkasten befinden, Vorrang. Make verwendet jedoch den gesamten Dateienverband, um ein Build durchzuführen.



In dieser Umgebung hat der Sandkasten dieselbe Baumstruktur wie die Master-Source (Stammdatenquelle) des Projekts. Das ermöglicht den Entwicklern, gefahrlos modulübergreifenden Code zu verändern z. B. bei der Änderung einer Modulschnittstelle.

Es ermöglicht außerdem, den Sandkasten physikalisch von der Master-Source zu trennen, ihn z. B. auf einem anderen Laufwerk oder in Ihrem privaten Arbeitsbereich anzulegen und der Master-Source des Projekts einen schreibgeschützten Status zu verleihen, sofern Sie ein strenges Check-in-Verfahren haben (oder sich zulegen wollen).

Bitte beachten Sie: Sie müssen nicht nur Ihrem Entwicklungs-Makefile eine VPATH-Reihe hinzufügen, sondern auch die `-I`-Optionen dem `CFLAGS`-Makro, damit der C-Compiler denselben Pfad wie Make benutzt. Das kann man einfach mit einem 3-Zeilen-Makefile in Ihrem Arbeitsbereich durchführen – Sie definieren erst ein Makro, dann `VPATH` und fügen schließlich das Makefile der Master-Source des Projekts ein.

### 6.1. VPATH-Semantik

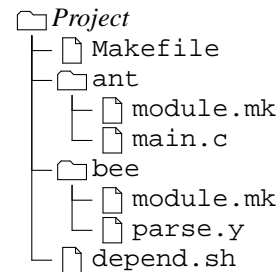
Um das eben Erläuterte anzuwenden, müssen Sie GNU-Make 3.76 oder eine neuere Version verwenden. Für frühere GNU-Make-Versionen benötigen Sie Paul Smith's VPATH+ Patch. Den können Sie bei <ftp://ftp.wellfleet.com/netman/psmith/gmake/> beziehen.

Die POSIX-Semantiken von VPATH sind unzulänglich genauso wie viele existierende Make-Ausführungen. Vielleicht überlegen Sie sich, GNU-Make zu installieren.

## 7. Das Gesamtbild

In diesem Kapitel werden alle auf den vorangegangenen Seiten erörterten Aspekte zusammengefügt und das Beispielprojekt mit seinen getrennten Modulen wird vorgestellt, aber mit einem Gesamtprojekt-Makefile. Die Verzeichnisstruktur unterscheidet sich wenig von der

rekursiven, außer dass die tieferliegenden Makefiles durch modulspezifische Include-Dateien ersetzt werden:



Das Makefile sieht folgendermaßen aus:

```

MODULES := ant bee
# look for include files in
#   each of the modules
CFLAGS += $(patsubst %,-I%,\
$(MODULES))
# extra libraries if required
LIBS :=
# each module will add to this
SRC :=
# include the description for
#   each module
include $(patsubst %, \
%/module.mk,$(MODULES))
# determine the object files
OBJ := \
$(patsubst %.c,%.o, \
$(filter %.c,$(SRC))) \
$(patsubst %.y,%.o, \
$(filter %.y,$(SRC)))
# link the program
prog: $(OBJ)
$(CC) -o $@ $(OBJ) $(LIBS)
# include the C include
#   dependencies
include $(OBJ:.o=.d)
# calculate C include
#   dependencies
%.d: %.c
depend.sh `dirname $*.c` $(CFLAGS) $*.c > $@
  
```

Das erscheint sehr lang, aber es enthält alle üblichen Elemente an einer Stelle, so dass die Make-Include-Dateien in den einzelnen Modulen sehr kurz sein können.

Die `ant/module.mk` Datei sieht folgendermaßen aus:

```
SRC += ant/main.c
```

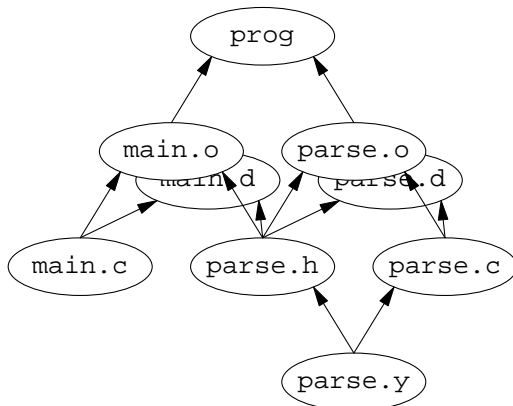
Die bee/module.mk Datei sieht so aus:

```
SRC += bee/parse.y
LIBS += -ly
%.c %.h: %.y
    $(YACC) -d $*.y
    mv y.tab.c $*.c
    mv y.tab.h $*.h
```

Beachten Sie bitte, dass die Built-in-Regeln für die C-Dateien verwendet werden, aber wir benötigen ein spezielles yacc-Verfahren, um die erzeugte .h-Datei zu bekommen.

Die Einsparungen in diesem Beispiel wirken unerheblich, weil das Makefile der obersten Ebene so groß ist. Aber stellen Sie sich vor, es handele sich um 100 Module, jedes mit ein paar funktionellen, modulspezifischen Zeilen. Insgesamt erreicht man oft, ohne an Modularität zu verlieren, einen geringeren Zeitaufwand als bei Verwendung eines rekursiven Makes.

Der entsprechende DAG des Makefiles sieht, nachdem man alle Einfügungen vorgenommen hat, folgendermaßen aus:



Die Knotenpunkte und Kanten für die Include-Dateiabhängigkeitsdateien sind ebenfalls vorhanden, da sie wichtig sind, damit Make fehlerlos funktionieren kann.

### 7.1. Nebeneffekte

Es gibt eine Reihe wünschenswerter Nebeneffekte, wenn man ein einziges Makefile verwendet.

- Die `-j`-Option des GNU-Makes für parallel laufende Builds funktioniert besser als vorher. Sie kann sogar mehr voneinander unabhängige Vorgänge zugleich ausführen und leidet nicht

mehr an subtilen Fehlern.

- Die `-k`-Option des allgemeinen Makes, die dafür sorgt, dass Make selbst angesichts von Fehlern so weit wie möglich weiter arbeitet, funktioniert ebenfalls besser als vorher. Sie findet sogar mehr Material, mit dem sie weiterarbeiten kann.

## 8. Literaturstudie

Wie ist es möglich, dass wir 20 Jahre lang Make falsch eingesetzt haben? Wie ist es möglich, dass das Verhalten von Make, das wir bisher seiner begrenzten Funktionalität zugeschrieben haben, sich nun als falsche Anwendung von Make herausstellt?

Der Autor begann über die Ideen, die in diesem Artikel vorgestellt werden, nachzudenken, als er sich mit einer Reihe häßlicher Build-Probleme in gänzlich unterschiedlichen Projekten aber mit gemeinsamen Symptomen konfrontiert sah. Indem er von den einzelnen Projekten Abstand nahm und die Gemeinsamkeiten der Probleme eingehend untersuchte, wurde ihm möglich, eine Regelmäßigkeit zu erkennen. Die meisten von uns sind zu sehr mit den Flickarbeiten für eine fehlerfreie Funktion des mangelhaften Builds beschäftigt, als dass sie Zeit finden würden, die Sache einmal mit Distanz zu begutachten und sich einen Gesamteindruck der Schwierigkeiten zu verschaffen. Besonders dann, wenn das fragliche Produkt offensichtlich arbeitet und das seit 20 Jahren.

Es ist interessant, dass die Probleme des rekursiven Makes in den einschlägigen Büchern, auf die sich Unixprogrammierer verlassen, wenn sie präzisen praktischen Rat benötigen, kaum erwähnt werden.

### 8.1. Das Original

Das originale Make-Handbuch [feld78] enthält keinen Hinweis auf rekursives Make, und erst recht keine Erörterung der relativen Vorteile des Ganzprojekt-Makes gegenüber dem rekursiven Make.

Es überrascht nicht, dass das Originalhandbuch rekursives Make nicht erwähnte. Damals passten Unix-Projekte gewöhnlich in eine einziges Verzeichnis.

Das ist vielleicht auch ein Grund, warum sich das "ein Makefile in jedem Verzeichnis"-Konzept so in der kollektiven Unix-Entwicklungsdenkweise festsetzte.

## 8.2. GNU-Make

Das GNU-Make-Handbuch [stal93] beschäftigt sich auf mehreren Seiten mit dem rekursiven Make; die Erläuterung seiner Vorzüge oder der Technik ist auf folgende Bemerkung reduziert:

"Diese Technik ist nützlich, wenn Sie getrennte Makefiles für verschiedene Teilsysteme, die zusammen ein größeres System bilden, anlegen wollen."

Kein Wort über die Schwierigkeiten, auf die Sie stoßen könnten.

## 8.3. Projektverwaltung mit Makefiles

Das Nutshell-Make-Handbuch [talb91] preist das rekursive Make besonders als dem Ganzprojekt-Make überlegen an:

Der "sauberste" Weg, ein Build zu erstellen, ist, indem man in jedem Verzeichnis ein gesondertes Makefile anlegt und sie durch ein Haupt-Makefile verbindet, das eine rekursive Make-Funktion hervorruft. Wenn diese Technik auch umständlich ist, ist sie doch leichter zu verwalten als eine einzige, riesenhafte Datei, die mehrere Verzeichnisse abdeckt." (Seite 65)

Das widerspricht genau dem Rat, den das Buch nur zwei Paragraphen weiter vorne erteilt:

"Make ist am glücklichsten, wenn Sie all seine Dateien in einem Verzeichnis lassen" (Seite 64)

Aber das Buch versäumt es, den Widerspruch in diesen beiden Aussagen zu erörtern, und fährt stattdessen fort, eine der herkömmlichen Möglichkeiten zu beschreiben, mit der man die Symptome eines durch rekursives Make verursachten unvollständigen DAGs zu umgehen versucht.

Dieses Buch bietet einen Anhaltspunkt, warum rekursives Make seit so vielen Jahren in dieser Weise verwendet wurde. Sie sehen, wie die beiden o. g. Aussagen das Konzept eines Verzeichnisses mit dem Konzept eines Makefiles verwechseln.

Dieser Artikel legt eine einfache Änderung der Denkweise nahe: In Verzeichnisbäumen, egal wie verzweigt sie sind, werden Dateien gespeichert; Makefiles dagegen sind dazu da, die Beziehungen dieser Dateien untereinander zu

beschreiben, gleichgültig wieviele Dateien es sind.

## 8.4. BSD-Make

Die Anleitung für BSD-Make [debo88] erwähnt das rekursive Make überhaupt nicht, aber sie ist eines der wenigen Handbücher, das, wenn auch sehr kurz, tatsächlich die Beziehung zwischen Makefile und DAG beschreibt (Seite 30). Daher stammt auch dieses wunderbare Zitat:

"Falls Make nicht das macht, was Sie erwarten, ist die Wahrscheinlichkeit sehr groß, dass das Makefile falsch ist." (Seite 10)

Das ist eine kurze und prägnante Zusammenfassung dieses Artikels.

## 9. Zusammenfassung

Dieser Artikel erläutert einige miteinander in Beziehung stehende Probleme und zeigt, dass es sich nicht, wie allgemein angenommen, um Make anhaftende Unzulänglichkeiten handelt, sondern dass sie eine Folge davon sind, dass Make falsche Informationen eingegeben wurden. Hier arbeitet das alte "Garbage in, Garbage out"-Prinzip (wo Müll hineinkommt, kommt Müll heraus). Der Fehler besteht darin, dass man das Makefile in unvollständige Teilstücke zerlegt, denn Make kann nur mit einem vollständigen DAG fehlerfrei arbeiten.

Das erfordert ein Umdenken. In Verzeichnisbäumen werden lediglich Dateien gespeichert, in Makefiles hingegen werden Hinweise auf die Informationen über die Beziehungen der Dateien untereinander gespeichert. Bringen Sie das nicht durcheinander, denn es ist genauso wichtig die Beziehungen zwischen Dateien in verschiedenen Verzeichnissen wiederzugeben, wie die Beziehungen zwischen Dateien im gleichen Verzeichnis. Daraus folgt, dass es genau ein Makefile für ein Projekt geben sollte, aber der Großteil der Beschreibung kann man durch Verwendung von Make-Include-Dateien in den einzelnen Verzeichnissen, die die Teilmenge der Projektdateien in den jeweiligen Verzeichnissen beschreiben, handhaben. Dieses Vorgehen ist genauso modular, als wäre ein Makefile in jedem Verzeichnis.

Es wurde gezeigt, dass bei Verwendung des Ganzprojekt-Make eine Entwicklungsproduktion und eine Vollproduktion gleich kurze Laufzeiten haben. Angesichts der gleichen Laufzeiten wiegen die durch die genaueren Abhängigkeiten erzielten

Vorteile umso schwerer und bewirken, dass dieser Prozess tatsächlich schneller und genauer läuft, als wenn das rekursive Make angewendet würde.

### 9.1. Projektübergreifende Abhängigkeiten

In Unternehmen mit einer starken Neigung zur Mehrfachverwendung kann die Verwirklichung eines Ganzprojekt-Makes eine Herausforderung darstellen. Sich dieser Herausforderung zu stellen, mag erfordern, dass man sich einen Gesamtein- druck der Situation verschafft.

- Es kann sein, dass ein Modul von zwei Programmen gemeinsam verwendet wird, weil die Programme nahe verwandt sind. Natürlich gehören die zwei Programme und das gemeinsam genutzte Modul zu demselben Projekt (das Modul kann auch unabhängig sein, die Programme jedoch nicht). Die Abhängigkeiten müssen ausdrücklich angegeben werden und Änderungen des Moduls ziehen nach sich, dass beide Programme entsprechend neu übersetzt und neu gebunden werden müssen. Vereinigt man sie alle in einem einzigen Projekt, kann das Ganzprojekt-Make dies leisten.
- Es ist möglich, dass ein Modul von zwei Projekten gemeinsam genutzt wird, weil ihr Wirkungsbereich ineinander greift. Möglicherweise ist Ihr Projekt größer, als Ihre gegenwärtige Verzeichnisstruktur aufnehmen kann. Die Abhängigkeiten müssen ausdrücklich angegeben werden und Änderungen am Modul ziehen nach sich, dass beide Projekte entsprechend neu übersetzt und neu gebunden werden müssen. Vereinigt man sie alle in einem einzigen Projekt, kann das Ganzprojekt-Make dies leisten.
- Es ist normal, die Kanten zwischen Ihrem Projekt und ihrem Betriebssystem oder dritten installierten Werkzeugen wegzulassen. Das ist so normal, dass sie in den Makefiles in diesem Artikel und bei den vordefinierten Regeln im Make-Programm ignoriert werden. Module, die von mehreren Projekten gemeinsam genutzt werden, könnten in diese Kategorie fallen. Werden sie geändert, bauen Sie Ihre Projekte extra neu oder beziehen die Änderungen stillschweigend bei der nächsten Produktion mit ein. In beiden Fällen geben Sie die Abhängigkeiten nicht ausdrücklich an und das Ganzprojekt-Make findet keine Anwendung.
- Es ist der Wiederverwendung möglicherweise dienlich, wenn das Modul als Schablone

verwendet und Abweichungen zwischen den Projekten als normal angesehen werden. Kopiert man das Modul für jedes Projekt, könnte man die Abhängigkeiten ausdrücklich angeben. Es hat jedoch einen zusätzlichen Aufwand zur Folge, wenn an dem gemeinsamen Teil Änderungen erforderlich werden.

In einer von Mehrfachverwendung stark geprägten Umgebung wird das Strukturieren von Abhängigkeiten zu einer Übung in Risikomanagement. Welche Gefahr besteht, dass fehlende Stücke des DAGs Ihr Projekt schädigen? Wie wichtig ist es, neu zu bauen, wenn ein Modul sich verändert? Welches sind die Folgen, wenn es nicht automatisch neu produziert wird? Woher wissen Sie, wann ein Neubau notwendig ist, wenn Abhängigkeiten nicht ausdrücklich genannt sind? Welches sind die Konsequenzen, wenn man vergisst, neu zu bauen? ...

### 9.2. Die Rendite

Einige der Techniken, die in diesem Artikel beschrieben wurden, beschleunigen ihren Buildvorgang sogar dann, wenn Sie rekursives Make beibehalten. Aber das ist nicht das Anliegen dieses Artikels, eher ein nützlicher Umweg. Die Hauptaussage ist, dass Sie korrektere Produktionen Ihres Projekts erhalten, wenn Sie das Ganzprojekt-Make anstatt des rekursiven Makes anwenden.

- Make bedarf nicht mehr und oft sogar weniger Zeit, um herauszufinden, das es nichts zu tun braucht.
- Die gesamte Eingabe für Make ist nicht umfangreicher und komplexer, oft sogar weniger umfangreich und weniger komplex.
- Im Ganzen sind die Eingabedaten für Make nicht weniger modular, als bei Anwendung des rekursiven Makes.
- Die Pflege des Makefiles ist nicht aufwendiger, oft sogar geringer.

Die angeblichen Nachteile der Anwendung eines Ganzprojekt-Makes gegenüber dem rekursiven Make sind oft nicht überprüft. Wieviel Zeit wird damit verbracht, herauszufinden, warum Make etwas Unerwartetes getan hat? Wieviel Zeit wird damit verbracht, an dem Build-Prozess herumzubasteln? Diese Tätigkeiten werden oft als normaler Entwicklungsaufwand angesehen.

Die Projektproduktion ist eine wesentliche Tätigkeit. Wenn sie schlecht funktioniert, werden auch die Entwicklung, die Fehlerbehebung und



das Testen in Mitleidenschaft gezogen. Die Projektproduktion sollte so einfach sein, dass sie der neuste Mitarbeiter – ohne eine noch so kurze schriftliche Anleitung – sofort durchführen kann. Sie sollte so einfach sein, dass sie so gut wie keinen Entwicklungsaufwand erfordert. Ist Ihr Produktionsprozess so einfach?

## 10. Literaturverweise

**debo88:** Adam de Boor (1988). *PMake – A Tutorial*. University of California, Berkeley

**feld78:** Stuart I. Feldman (1978). *Make – A Program for Maintaining Computer Programs*. Bell Laboratories Computing Science Technical Report 57

**stal93:** Richard M. Stallman and Roland McGrath (1993). *GNU Make: A Program for Directing Recompilation*. Free Software Foundation, Inc.

**talb91:** Steve Talbott (1991). *Managing Projects with Make, 2nd Ed.* O'Reilly & Associates, Inc.

Miller, P.A. (1998), *Recursive Make Considered Harmful*, AUUGN Journal of AUUG Inc., 19(1), pp. 14-25.

See <http://www.cmmagazin.de> December 2002 issue for the HTML version of the translation.

## 11. Über den Autor

Peter Miller hat viele Jahre lang in der Software R&D Industrie gearbeitet, vor allem auf UNIX-Systemen. In dieser Zeit schrieb er Werkzeuge wie Aegis (ein System des Software Configuration Management) und Cook (eine weitere Make-Version), beide kann man frei über das Internet beziehen. Die Betreuung bei der Verwendung dieser Werkzeuge auf vielen Internet Sites ermöglichte den Einblick, der zu diesem Artikel führte.

Wenn Sie Interesse am Bezug der freien Software des Autors haben, besuchen Sie bitte: <http://-miller.emu.id.au/pmiller/>