Programme bauen mit MAKE

Jan Hendrik Dithmar

10.11.2005

Einführung

Ziel des automatischen Programmbauens ist es, aus einer Menge von Quellkomponenten alle abgeleiteten Komponenten zu erzeugen, die Bestandteil des fertigen Softwareprodukts sind.

Bisherige Arbeitsweise: Arbeiten mit einem Shell-Skript

Nachteil: es wird bei jedem Start des Skriptes alles neu gebaut!

Das Softwarewerkzeug MAKE

Vorteile und Arbeitsweise:

- hat sich eine Komponente geändert, müssen alle von ihr abgeleiteten Komponenten neu gebaut werden
- es werden existierende abgeleitete Dateien so weit wie möglich weiterverwendet
- es wird das Änderungsdatum von Komponenten verwendet, um Änderungen zu erkennen
- Grundlage von MAKE ist ein Makefile
- Makefile: eine Systembeschreibung, die die Komponenten des Systems sowie deren Abhängigkeiten und Konstruktionsschritte aufzählt

Aufbau einer Makefile

Ein Makefile ist wie folgt aufgebaut:

$ziel_1\ ziel_2\ \ldots\ ziel_n :\ quelle_1\ quelle_2\ \ldots\ quelle_m$
$kommando_1$
$kommando_2$
$\ldots$

Beispiel 1: einfache Makefile

```
rechner: add.o sub.o
    cc -o rechner add.o sub.o

add.o: add.cpp
    cc -c -o add.o add.cpp

sub.o: sub.cpp
    cc -c -o sub.o sub.cpp
```

Achtung: Kommandos müssen mit Tabulatorzeichen eingerückt sein!

Variablen in MAKE

MAKE unterstützt Variablen:

$$\textit{variable} = \textit{wert}$$

Jedes Aufrufen von `$(variable)` oder `${variable}` wird dann automatisch durch den jeweiligen *wert* ersetzt.

Beispiel 2: Variablen in Makefiles

```
CC = cc
OBJECTS = add.o sub.o

rechner: $(OBJECTS)
    $(CC) -o rechner $(OBJECTS)

add.o: add.cpp
    $(CC) -c -o add.o add.cpp

sub.o: sub.cpp
    $(CC) -c -o sub.o sub.cpp
```

Mit dem Aufruf
$ **make CC=gcc rechner**
wird anstelle von cc der Compiler gcc verwendet.

Implizite Regeln in MAKE

Eine implizite Regel erfordert implizie Variablen:

- `$@` repräsentiert das aktuelle Ziel
- `$<` repräsentiert die erste Quelle

Beispiel:

```
.cpp.o:
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Die Variablen `CFLAGS` und `CPPFLAGS` werden zusätzlich zur Steuerung des C-Compilers und C-Präprozessors eingesetzt.
Die obige Regel ist in MAKE bereits mit geeigneten Werten für die benutzen Variablen vordefiniert.

Beispiel 3: Implizite Regeln in Makefiles

Um das Projekt `rechner` zu bauen reicht daher folgende Makefile völlig aus:

```
CC = cc
OBJECTS = add.o sub.o

rechner: $(OBJECTS)
    $(CC) -o rechner $(OBJECTS)
```

Die Regel für `add` und `sub` braucht man auf Grund der `.cpp.o`-Regel nicht mehr explizit anzugeben.

Andere Anwendungsgebiete von MAKE

- MAKE kann auch mit Hilfe der Definition des Pseudoziels `install` zum Installieren von Software benutzt werden
- MAKE kann über einen rekursiven Aufruf nicht nur zum Bauen eines Systems, sondern auch zum Bauen eines Subsystems verwendet werden

Automatische Bestimmung von Abhängigkeiten

Der UNIX C-Compiler stellt ein automatisches Verfahren zur Bestimmung von Abhängigkeiten zur Verfügung:

```
$ cc -M add.cpp
add.o: add.cpp
add.o: ./calc.h
$ _
```

Diese Eigenschaft kann in einem Makefile genutzt werden:

```
depend::
    (for file in $(SOURCES); do \
        $(CC) -M $$file; \
    done) > Makedeps
include Makedeps
```

`include Makedeps` wird beim Einlesen des Makefiles durch den Inhalt von `Makedeps` ersetzt, womit die bestimmten Abhängigkeiten Bestandteil des Makefiles werden.

MAKE im Internet

- http://www.gnu.org/software/make/make.html
- http://www.gnu.org/software/make/manual/make.html
- http://de.wikipedia.org/wiki/Make
- http://en.wikipedia.org/wiki/Make