

Autoconf/Automake

Eine kurze Ausarbeitung zur Open-Source-Software Vorlesung

Peter Novotnik
<*peternov1@gmx.de*>

19. März 2004

Inhaltsverzeichnis

1	Entstehungsgeschichte	3
1.1	UNIX Systeme	3
1.2	Autoconf	4
1.3	Automake	5
2	<i>./configure ; make ; make install</i>	5
2.1	Das Konfigurieren	6
2.2	Das Erstellen	8
2.3	Das Installieren	8
3	Programmierbeispiel	9
3.1	hello.c	9
3.2	Files	10
3.3	Makefile.am	12
3.4	configure.in	12
3.5	config.h.in	14
3.6	Erstellen des Pakets	15
3.7	Das Paket erstellen	16
4	Und was nun?	17

Einleitung

Mehr denn je sind Portabilität und Einfachheit beim Erstellen eines Softwareprojekts zu bedeutenden Faktoren geworden, um die Beliebtheit und den Erfolg eines Projektes anzustreben.

Viele Softwarepakete, die man sich von etlichen Servern im Internet herunterladen kann, werden auf eine standardisierte Art und Weise über den *./configure && make && make install* Zyklus in ein System installiert. Wer schon ein bißchen selbst programmiert hat, wird sicherlich über *Make* gestolpert sein und vielleicht die eine oder andere “Makefile” selbst geschrieben haben, um sich den Erstellungsprozess des entwickelten Programms erleichtert zu haben. Wer schon mal ein Programm auf mehreren Systemen erstellt hat, schätzt sicherlich das sog. “configure”-Skript, welches bei den Programmquellen dabei

war. Doch woher kommt dieses Skript? Und wieso ist es in Paketen enthalten? Die Antwort liegt in den beiden Tools *Autoconf* und *Automake*.

In der vorliegenden Ausarbeitung werden *Automake* und *Autoconf*, ihre Konzepte und Vorgehensweisen wie auch ihr geschichtlicher Werdegang vorgestellt, um eine Vorstellung davon zu geben, was diese Tools leisten, wozu sie gedacht und aus welchen Bedürfnissen sie entstanden sind.

Für den Neuling wird auf eine kurze Art und Weise der Erstellungszyklus eines Programms vorgestellt und einige Hinweise sowie Erklärungen gegeben.

Zum Schluß wird ein kurzes Beispiel gezeigt, wie man selbst den Quellcode seiner eigenen Programme mit *Autoconf* und *Automake* versieht, indem eine kleine “Hello World!” Anwendung entwickelt wird.

Das vorliegende Schriftstück erhebt keinen Anspruch auf Vollständigkeit! Es sei gleich darauf hingewiesen, dass der Leser dennoch zu den GNU Manuals¹ greifen sollte. Diese Ausarbeitung ist vielmehr dazu gedacht, den Leser zu animieren, das hier Besprochene selbst auszuprobieren und einen Anfang mit den beiden Tools zu wagen, denn *Automake* und *Autoconf* haben sich, zumindest in der UNIX-artigen Welt, als de facto Standard für die Verbreitung von Softwarepaketen im Sourcecodeformat etabliert.

1 Entstehungsgeschichte

Die Entstehungsgeschichte der Tools kann detaillierter in [4] nachgelesen werden.

1.1 UNIX Systeme

Autoconf wurde als erstes von den beiden hier vorgestellten Tools entwickelt. Seine Entstehung ist durch die Geschichte des Betriebssystems UNIX abhängig.

Die erste Version von UNIX wurde von Dennis Ritchie und Ken Thompson an den Bell Labs 1969 geschrieben. In den 70ern wurde es Bell Labs nicht erlaubt UNIX kommerziell zu vertreiben, aber für geringe Kosten an Universitäten zu verteilen. Die Berkeley Universität in Kalifornien arbeitete ihre Verbesserungen zu dem UNIX-Quellcode ein und machte ihr System später als *BSD* (Berkeley Software Distribution) bekannt. In den 80ern hatte AT&T ein Abkommen unterzeichnet, welches es der Firma wieder erlaubte, UNIX kommerziell zu verkaufen. Ihre erste Version von UNIX wurde als “System III” bekannt. Die Popularität des UNIX wuchs in den 80ern und etliche Firmen erstellten ihre eigenen Versionen. Obwohl alle UNIX-Varianten in ihren Fundamenten ähnlich waren, gab es verschiedene Unterschiede zwischen ihnen. Sie hatten leicht unterschiedliche Zusam-

¹Manuals zu vielen GNU Tools sind unter <http://www.gnu.org/manual/manual.html> zu finden.

menstellungen von Headerfiles und die Auflistungen von Funktionen in den Systembibliotheken wiesen auch leichte Unstimmigkeiten auf.

Variationen in den Systemen stellten Probleme für Programme dar. Sogar eine Funktion wie `memcpy` war nicht auf jedem System verfügbar; die BSD-Systembibliothek stellte zwar eine ähnliche Funktion namens `bcopy` zur Verfügung, aber die Aufrufparameter waren in umgekehrter Reihenfolge angeordnet. Programmierer, die ihre Programme auf mehreren unterschiedlichen UNIX-Varianten lauffähig halten wollten, mussten sich der einzelnen Unterschiede bewusst sein. Während es im Allgemeinen möglich war, die einzelnen Systeme und Versionen mittels `#ifdef` Präprozessoranweisungen zu identifizieren, wurde es extrem schwierig festzustellen welche Version welche Features anbot und welche nicht. Es wurde klar, dass eine besser organisierte Methode von Nöten war, um die Unterschiede zwischen den UNIX-Varianten handhabbar zu machen.

1.2 Autoconf

Im Laufe der Zeit entstanden verschiedene Systeme, die speziell dafür entworfen wurden, Entwicklern mit der Sourcecode-Portabilität zu helfen.

- *Metaconfig* von Larry Wall, Harlan Stenn und Raphael Manfredi.
- Das *Cygnus configure*-Skript von K.Richard Pixley und das *GCC configure*-Skript von Richard M. Stallman.
- Das *GNU Autoconf* Paket von David MacKenzie.
- *Imake*, das ein Teil des X Window Systems ist.

Jedes dieser Systeme spaltet das Erstellen einer Software in einen Konfigurations- und einen Erstellungsschritt. Der Erstellungsschritt stützt sich bei allen Systemen auf das Programm "Make". Dieses liest eine Reihe von Regeln aus einer sog. "Makefile", die während des Konfigurationsschrittes erzeugt wurden, anhand derer es den Quellcode kompiliert.

Metaconfig und *Autoconf* sind Programme, die von Programmierern benutzt werden, um ein Shell-Skript zu erstellen, welches üblicherweise im Softwarepaket enthalten ist. Ein Benutzer, der das Softwarepaket erstellen möchte, führt zuerst dieses Skript aus, um den Erstellungsschritt für sein System zu konfigurieren. Das von *Metaconfig* generierte Skript ist standarmäßig interaktiv.

Das *Cygnus* und *GCC "configure"* hatten zwar Unterstützung für das sog. "cross development" mussten aber für jedes neue Unix angepasst werden, damit sie saubere Arbeit leisten konnten. Dies war zwar keine schwierige Aufgabe, aber neue Benutzer eines Softwarepakets sollten sich damit nicht herumplagen müssen. Das selbe galt für *Imake*. Weil *Metaconfig* und *Automake* Eigenschaftstests selbst durchführten, generierten sie auch auf

neuen UNIX Varianten sehr oft korrekt arbeitende Skripte, die nicht mehr modifiziert werden mussten. Das machte sie flexibler und einfacher zu benutzen und führte zur großen Verbreitung von *Autoconf*. Später erweiterte David MacKenzie *Autoconf* um Features, die in den *Cygnus* und *GCC Skripten* enthalten waren, so dass es auch das “cross development” unterstützt.

Im Großen und Ganzen werden heute neue Projekte mit der Unterstützung von *Autoconf* begonnen und nicht mehr mit anderen Konfigurationswerkzeugen, obwohl diese doch noch bei einigen Paketen eingesetzt werden.

1.3 Automake

Nach dem man mit *Autoconf* ein stabiles Rahmenwerk hatte, um die Unterschiede zwischen verschiedenen UNIX Varianten handzuhaben, mussten die Entwickler jedoch große “Makefile.in”s schreiben, um *Autoconf* benutzen zu können. Das “configure”-Skript, welches durch *Autoconf* generiert wird, nimmt die “Makefile.in” entgegen und erstellt daraus eine “Makefile”, die von “Make” abgearbeitet wird.

Da die meisten Programme auf die gleiche Art und Weise kompiliert werden, ist es mühsam und langweilig immer wieder das Gleiche schreiben zu müssen. Ausserdem kann das Schreiben von Makefiles recht komplex werden, wodurch sich die Fehlerquote um einiges erhöht.

Diese Faktoren führten zu der Entwicklung von *Automake*. Es ist ebenso wie *Autoconf* ein Programm, das nur vom Entwickler benutzt wird, um sich die Arbeit im Wesentlichen zu erleichtern. Der Programmierer schreibt eine “Makefile.am”, die um vieles einfacher im Aufbau ist, und *Automake* erzeugt aus ihr eine “Makefile.in”, die von *Autoconf* weiter bearbeitet wird.

2 *./configure ; make ; make install*

Ein Softwarepaket, welches durch *Autoconf* unterstützt wird, beinhaltet meistens ein Skript namens “configure”. Will man das Paket installieren, ist im idealen Fall nur eine Folge von drei Kommandos einzugeben.

Das im Paket enthaltene “configure” muss immer als erstes aufgerufen werden; optional auch mit einer überwältigenden Menge an Kommandozeilenparametern, die mit dem Aufruf `./configure --help2` eingesehen werden können. Das Skript ermittelt einige Details zur Systemumgebung und bereitet den Quellcodeverzeichnisbaum auf die Kompilierung vor. Das Ergebnis eines erfolgreich durchgelaufenen “configure”s sind einige neu angelegte Dateien im Verzeichnisbaum des Softwarepakets, unter anderem auch

²Da das aktuelle “Working Directory” nicht im Suchpfad (PATH Umgebungvariable) eingetragen sein sollte, ist es notwendig, dem `configure` ein “./” voranzustellen.

die “Makefile”.

Ist eine “Makefile”³ im Verzeichnisbau des Quellcodes vorhanden, kann mit dem Aufruf `make` die Erstellung der Software erfolgen. “Make” liest sich die “Makefile” ein, und arbeitet die dort definierten Schritte ab, wobei meistens Compiler mit den entsprechenden Dateinamen und Optionen als Argumente aufgerufen werden.

Nach dem sich “Make” ohne Fehler beendet hat, sollte das ausführbare Programm nun irgendwo im Quellcodebaum vorliegen und gegebenen Falls auch schon gestartet werden können. Durch den Aufruf `make install` wird das Paket in den dafür vorhergesehenen Pfad kopiert, die Rechte der installierten Dateien werden gesetzt, und die letzten Vorbereitungen werden getroffen, um die Software lauffähig zu machen. Natürlich ist es wichtig, selbst entsprechende Rechte zu haben, wenn man etwas bspw. nach `/usr/bin` kopieren will; folglich sollte man das “install”-Target als “root” ausführen.

2.1 Das Konfigurieren

Wie schon erwähnt wurde, erkennt “configure” eine riesige Menge an Kommandozeilenparameter, mit Hilfe derer das zu erstellende Programm konfiguriert werden kann. Neben den Optionen, welche das zu erstellende Programm selbst zur Konfiguration anbietet, gibt es schon durch *Autoconf* standardisierte Parameter, von denen nur wenige hier vorgestellt werden sollen.

Um sich eine Liste aller Optionen anzeigen zu lassen, die von dem vorliegenden Skript unterstützt werden, ist lediglich die Kommandozeile

```
./configure --help
```

abzusetzen. Die Ausgabe dieses Kommandos unterscheidet sich von Paket zu Paket nur in den paketspezifischen Optionen. Es sei von daher empfohlen, die Liste der verfügbaren Optionen an das Konfigurationsskript vor dem Erstellen eines Pakets wenigstens zu überfliegen. In folgender Auflistung sind nun einige der wichtigen Optionen aufgeführt.

`--prefix`

Jedes Paket hat ein Installationsprefix. Sieht man sich einmal die Verzeichnisstruktur eines UNIX-artigen Systems an, wird man feststellen, dass es mindestens zwei ähnlich strukturierte Bäume im Dateisystem gibt. Zum einen den unter dem Wurzelknoten (`/`) und zum anderen unter `/usr` aber auch `/usr/local`. Die Software wird unterhalb des angegebenen Präfixes installiert. Folgender Aufruf ist denkbar:

```
./configure --prefix=/home/pete/test
```

³Das “Makefile” kann auch in Kleinbuchstaben geschrieben sein, für das “Make” Programm spielt es keine Rolle. Es hat sich allerdings eingebürgert das “M” groß zu schreiben, da bei einem Aufruf von `ls`, die großgeschriebenen Dateien als erstes aufgelistet werden und somit nicht in der Auflistung stören.

`--enable-feature[=arg]`

Nicht alle, aber die meisten Pakete bieten selbst definierte Optionen an, die vor der Erstellung gesetzt werden können. Eine Auflistung der Features, welche ein Paket zur Konfiguration anbietet, befindet sich in der Ausgabe des Kommandos `./configure --help`. Mit der `--enable-feature` Option kann eine Eigenschaft mit in die Software einkonfiguriert werden, falls sie es nicht schon standardmäßig wird. Bei einigen Optionen kann man auch ein optionales Argument übergeben. Bei welchen Optionen dies geht, sollte ebenfalls in der Hilfeausgabe des Skripts stehen. Die Übergabe des Arguments an die Option erfolgt wie im folgenden Beispiel gezeigt:

```
./configure --enable-buffers=128
```

Wird der “enable”-Option kein Argument übergeben, so wird es standardmäßig auf “yes” gesetzt. Eine Besonderheit stellt das das “no” Argument. Möchte man ein Feature gänzlich aus dem Program entfernen, so fern es der Quellcode unterstützt, kann man `--enable-feature=no` oder `--disable-feature` angeben.

`--disable-feature`

Mit dieser Option wird ein bestimmtes Feature ausgeschaltet. Beispielsweise könnte ein Programm zusätzlich eine GUI anbieten. Ist diese nicht erwünscht, könnte es über die `--disable-gui` Option abgeschaltet werden.

Das `--disable-feature` und `--enable-feature` gehören logisch zusammen. Das heißt, wenn es eine Optionen `--enable-gui` gibt, dann gibt es auch eine Option `--disable-gui`. Natürlich macht es keinen Sinn einen Aufruf des Skripts mit beiden Optionen zu machen.

`--with-package[=arg]`

Gelegentlich ist es notwendig anzugeben, unter welchem Pfad sich bestimmte Headerfiles und Libraries befinden. Dies ist vor allem dann der Fall, wenn benötigte Bibliotheken in einem exotischen Verzeichnis installiert wurden. Angenommen die GTK+ Bibliothek ist unter `/usr/local` installiert worden und das zu erstellende Programm benötigt diese. Sollte “configure” fehlerhaft abbrechen, kann man versuchen dem Skript die Option `--with-gtk=/usr/local` mit auf den Weg zu geben, um explizit auszudrücken, dass unter dem angegebenen Präfix die GTK+ Headers und Libs zu finden sind.

Wie schon mehrmals erwähnt versteht “configure” noch viele andere Optionen. Der Leser sollte sich an dieser Stelle irgendein Paket seiner Wahl aus dem Internet herunterladen, dieses entpacken und nach einem Skript namens “configure” suchen. Sollte eines vorhanden sein, steht dem Experimentieren nichts mehr im Wege.

2.2 Das Erstellen

In diesem Stadium, nach dem “configure” erfolgreich abgelaufen ist, sollte man die “Makefile” im Verzeichnisbaum des Pakets vorfinden. Diese Datei, wie auch einige andere Dateien, sind vom vorhin gestarteten Skript generiert worden. Das “Makefile” enthält einige “Targets”, die üblich für die von *Autoconf*’s “configure” generierten “Makefile”s sind.

Es folgt eine Auflistung der gebräuchlichsten Aufrufe von “Make” im Zusammenhang mit der generierten “Makefile”.

`make all`

Dieser Aufruf sollte alle benötigten Dateien kompilieren und die Paketprogramme erstellen. Nach erfolgreichem Ablauf, sollten sich die “Binaries” unterhalb des Paketverzeichnisbaums befinden.

`make check`

“Make” sollte alle “Selbst-Tests” anstarten, die das Paket beinhalten kann.

`make install`

Instaliert die Software unterhalb des zuvor mit “configure” definierten Präfixes.

`make clean`

“Säubert” den Quellcodebaum. Dies resultiert üblicherweise darin, alle während der Kompilierung erstellten Objektdateien sowie die erstellten Programme zu löschen. Die von “configure” erstellten Dateien werden jedoch nicht gelöscht.

`make distclean`

“Säubert” noch sauberer als `make clean` :-). Löscht auch die von “configure” erstellten Dateien. “distclean” leitet sich aus dem Ausdruck “distribution clean” ab und soll den Verzeichnisbaum so säubern, dass alle generierten Files neu erstellt werden müssen.

Statt `make all` kann auch nur `make` eingegeben werden, da das Target “all” immer als erstes in der “Makefile” steht, wodurch sich somit die beiden Aufrufe entsprechen.

2.3 Das Installieren

Wie schon im vorhergehenden Abschnitt gezeigt, wird die erstellte Software mit dem Kommando `make install` ins System installiert und zwar unterhalb des mit der Option “prefix” angegebenen Pfades (z.B. `/usr/local`). Üblicherweise werden die ausführbaren Programme nach `bin` kopiert und die Rechte der Dateien gesetzt. Konfigurationsdateien laden oft im `etc` Verzeichnis. `share/prog` dient oft zur Ablage für zusätzliche Dateien, welche die Software benötigt. Natürlich wird all diesen Verzeichnissnamen das Präfix vorangestellt.

Zum Abschluß noch ein Tipp über das Erstellen eines Softwarepakets im Binärformat. Angenommen man will ein Paket erstellen, dieses Installieren, die installierten Dateien in ein Tar-Archiv packen und anschliessend dieses einem Freund zuschicken, damit er sich Arbeit spart. Da der Befehl `make install` die Dateien in das System installiert, ist es schwierig festzuhalten, welche Files wohin kopiert wurden. Eine Möglichkeit wäre es, das Präfix auf einen leeren Verzeichnisbaum zu setzen und die Software dorthin zu installieren. Das macht aber im allgemeinen keinen Sinn und kann unter Umständen zu Problemen führen. Der Trick besteht darin, die Software einfach mit dem gewünschten Präfix zu kompilieren, also bspw. mit `--prefix=/usr`, und vor dem Installieren eine Makevariable auf ein temporäres, leeres Verzeichnis zu setzen, in welches die Softwarekomponenten kopiert werden. Gibt es in der Makefile die `DESTDIR` Variable (mit `grep DESTDIR Makefile` ersichtlich), kann mit dem Kommando

```
make DESTDIR=/tmp/software install
```

das Paket in das angegebene Verzeichnis installiert werden. Dieses kann man “tarren” und seinen Freunden zuschicken.

3 Programmierbeispiel

Im vorliegenden Abschnitt werden (nun endlich) *Autoconf* und *Automake* als Entwicklerwerkzeuge betrachtet. An Hand des hier behandelten Beispiels sollen diese beiden Tools aus der Sicht eines Entwicklers gezeigt werden, wobei sichtbar werden sollte, wie stark die Werkzeuge den Entwicklern beim Erstellen eines Pakets unterstützen und den Erstellungsprozess eines Sourcecodepakets erleichtern. Zugegebener Massen mag der Anfang mit den Tools etwas undurchscheinbar erscheinen, der Leser sollte sich jedoch nicht davon abbringen lassen.

3.1 hello.c

Es soll eine sehr einfache Applikation, in C geschrieben, erstellt werden, die nichts anderes durchführen soll, als “Hallo Welt” auszugeben. Der Code für das Programm wird einfach in einer Datei “hello.c” gespeichert. Dabei soll der Code so entworfen sein, dass er mit den *Autoconf*-üblichen Mitteln zu einem Paket gepackt werden kann.

```
#include <stdio.h>

#ifdef HAVE_CONFIG_H
# include <config.h>
#endif
```

```

int main( int argc, char ** argv )
{
#ifdef CONFIG_REPEAT
    int i;
    for( i = 0; i < 10; i++ )
#endif
        puts( "Hello World!" );

    return 0;
}

```

Die Headerdatei `stdio.h` wird wegen der im Code verwendeten `puts`-Funktion inkludiert (siehe man 3 `stdio`). Eine kleine Erweiterung des Programms soll gegeben sein, um eine Demonstration für *Autoconf* zu haben. Wird zum Zeitpunkt des Kompilierens die Präprozessorvariable “`CONFIG_REPEAT`” definiert, so soll der auszugebende String zehn mal auf dem Bildschirm erscheinen. Dazu ist der dafür verantwortliche Code durch die Präprozessorvariable isoliert. Die Datei `config.h` wird nur gegebenen Falls eingebunden. In ihr befinden Definitionen, die sich von Kompilation zu Kompilation unterscheiden können, wie z.B. die Definition von “`CONFIG_REPEAT`”. Ansonsten sind im Quelltext keine Besonderheiten.

3.2 Files

Da der Quellcode für das Programm geschrieben ist, ist es an der Zeit das Skript namens “`configure`” zu erstellen. Dabei ist es wichtig zu verstehen welche zusätzlichen Dateien benötigt werden, um aus “`hello.c`” mit Hilfe von *Autoconf* und *Automake* ein richtiges Softwarepaket zu erstellen. Die im Folgenden aufgelisteten Dateien werden von einem *Autotools*-konformen Paket benötigt.

`configure.in` enthält die Definitionen, mit deren Hilfe *Autoconf* das “`configure`”-Skript erstellt. Diese Datei wird vom Entwickler selbst geschrieben.⁴

`Makefile.am` enthält die Definitionen, mit deren Hilfe *Automake* die sog. “`Makefile.in`” erstellt, aus welcher letztlich mittels “`configure`” die entscheidende “`Makefile`” wird.

`config.h.in` ist optional. Ist jedoch die vorliegende Software zur Kompilierzeit konfigurierbar (wie bspw. im vorliegenden Beispiel mittels `CONFIG_REPEAT`), ist es sinnvoll die Konfiguration mittels “`config.h.in`” aufzuziehen.

⁴Seit *Autoconf* Version 2.50 wird in den GNU Manuals zu *Autoconf* und *Automake* von “`configure.ac`” gesprochen. Dies ist der neue Name der Datei. Allerdings wird der Name “`configure.in`” immer noch aus Kompatibilitätsgründen unterstützt und wird immer noch sehr häufig vorgefunden.

“Maintainerfiles” Es gibt Dateien, die nicht unbedingt notwendig sind, um “Makefile” und “configure” zu erstellen, jedoch aber Bestandteil eines jeden Softwarepakets sein sollten. Fehlen die Dateien “NEWS”, “README”, “AUTHORS” und “ChangeLog”, so beklagt sich *Automake* über ihre Abwesenheit.

Nach dem der Quellcode für das Programm in nur einer Datei ist, sollte somit “hello.c” zur Zeit die einzige Datei in Projektverzeichnis sein. Die “Maintainerfiles” in obiger Auflistung sollen hier nicht von großer Bedeutung sein und werden deshalb mit den “touch” Befehl angelegt:

```
$ pwd
/home/pete/hello
$ touch NEWS README AUTHORS ChangeLog
$ ls -F
AUTHORS  ChangeLog  NEWS  README  hello.c
```

Die beiden Dateien “configure.in” und “Makefile.am” spielen bei den *Autotools* eine zentrale Rolle. Zwischen den beiden Dateien besteht ein impliziter Zusammenhang. Sehr vereinfacht dargestellt könnte man ihn wie folgt darstellen (dabei ist in eckigen Klammern das jeweils auszuführende Programm angegeben):

```
(1)                                     [autoconf]
configure.in ----- > configure

(2)                                     [automake]
configure.in + Makefile.am ----- > Makefile.in

(3)                                     [configure]
Makefile.in ----- > Makefile
```

Im logisch ersten Schritt wird die Datei “configure.in” vom Programmierer geschrieben, woraus mittels *Autoconf* das Skript “configure” erstellt wird. Dieses Skript ist oft über 4000 Zeilen lang, während die hier im Beispiel vorgestellte Datei “configure.in” (der Quellcode für “configure”) gerade mal 25 Zeilen haben wird.

Als zweites benötigt der Programmierer die Datei “Makefile.am”. Diese beschreibt unter anderem in sehr einfacher Form, was für Programme erstellt werden sollen und welche Quelldateien dafür jeweils notwendig sind. Da bestimmte Definitionen für *Automake* auch aus den Definitionen in “configure.in” hervorgehen, werden beide Dateien benötigt, um eine “Makefile.in” zu erstellen. Auch hier ist *Automake* eine große Hilfe, denn die generierte Datei wird über 500 Zeilen haben, während die von Hand geschriebene Datei zwei Zeilen lang sein wird.

Im logisch letzten Schritt (3) wird mittels “configure” mit optionalen Parametern eine “Makefile” erstellt, die sich aus den Definitionen in der Vorlage “Makefile.in” und den aus der Umgebung festgestellten Systemeigenschaften ableitet. Jedgliche weitere Arbeit übernimmt das Programm “Make”.

3.3 Makefile.am

Eine “Makefile” beschreibt die Art und Weise, sowie die Reihenfolge, wie ein Programm erstellt werden soll. Je mehr Quelldateien ein Programm als Quelle hat, um so mehr Abhängigkeiten bestehen zwischen diesen Dateien. Dies gilt es beim Erstellen zu berücksichtigen. Das ist jedoch die Aufgabe des Programms “Make”, welches die “Makefile” als Eingabedatei abarbeitet. Eine korrekte und alle Abhängigkeiten berücksichtigende “Makefile” von Hand zu erstellen ist gerade bei grösseren Projekten nicht gerade trivial. Das manuelle Erstellen von größeren “Makefile”s ist von Projekt zu Projekt recht ähnlich, ermüdend und somit recht fehleranfällig.

Abhilfe schafft *Automake*, welches eine “Makefile.am” einliest und daraus eine “Makefile.in” erstellt. Die “Makefile.am”, welche der Programmierer von Hand erstellt, ist sehr einfach und übersichtlich. Die generierte “Makefile.in” ist eine Vorlage für “configure”, welches in der Datei einige Ersetzungen durchführt und damit die gewünschte “Makefile” erstellt.

Die Syntax der “Makefile.am” ist sehr einfach und intuitiv. Näheres dazu ist im *Automake*-Manual [2] beschrieben, welches es in verschiedenen Formaten gibt. Die “Makefile.am” im vorliegenden Beispiel besteht aus nur zwei Zeilen:

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

Die erste Zeile meldet *Automake*, dass ein Program namens “hello” erstellt werden soll, während die zweite, die Quelldateien angibt, die zur Erstellung von “hello” benötigt werden. Um alles andere kümmert sich *Automake*.

3.4 configure.in

Es fehlt noch die “configure.in”, um das Paket startklar zu machen. *Autoconf* liest diese Datei ein, expandiert dort die stehenden Makros und erstellt letztlich ein portables Shell-Skript namens “configure”.

Die Eingabesprache von *Autoconf* ist nicht besonders complex, allerdings ist der Anfang mit *Autoconf* nicht gerade leicht. Im Wesentlichen werden Makros verwendet, deren Bedeutung man in [1] nachlesen kann. Das Manual ist sehr empfehlenswert, da es den Leser in einer Art Tutorial mit *Autoconf* bekannt macht.

Einen sehr guten Anfang bietet das Tool “Autoscan”. Ruft man es ohne Kommandozeilenparameter auf, so wird das aktuelle Verzeichnis genommen, um die darin enthaltenen Dateien zu durchsuchen und eine Datei “configure.scan” anzulegen. Diese Datei ist ein Vorschlag für eine “configure.in” von “Autoscan”. Nach dem man Verbesserungen oder Erweiterungen in dieser Datei vorgenommen hat, kann man sie in “configure.in” umbenennen.

Die hier folgende “configure.in” wurde mit `autoscan` erstellt, worauf hin einige Punkte von Hand hinzugefügt wurden.

```
AC_PREREQ(2.59)
AC_INIT([hello],[1.0],[peternov1@gmx.de])
AM_INIT_AUTOMAKE([hello],[1.0])

AC_ARG_ENABLE([repeat],
  AS_HELP_STRING([--enable-repeat],
    [Repeat the output of 10 times]),
  [if test "x$enableval" = "xyes" ; then
    AC_DEFINE(CONFIG_REPEAT,,
      [Repeat the output of the programm 10 times.])
  fi])

AC_CONFIG_SRCDIR([hello.c])
AC_CONFIG_HEADER([config.h])
AC_CONFIG_FILES([Makefile])

AC_PROG_CC
AC_OUTPUT
```

Das erste Makro `AC_PREREQ`, mit “2.59” als Argument, stellt sicher, dass eine gleiche oder höhere Version von *Autoconf* benutzt wird, um “configure” zu erstellen. Ist auf dem System eine niedrigere Version installiert, wird sich *Autoconf* mit einer Warnung beenden. Somit kann man sicherstellen, dass kein, aus Versionsgründen, falsches “configure” erstellt wird. `AC_PREREQ` ist das einzige Makro, welches vor dem `AC_INIT` vorkommen darf.

`AC_INIT` ist das *Autoconf*-Initialisierungsmakro. Es ist das Makro, welches, mit der Ausnahme von `AC_PREREQ`, als erstes in der “configure.in” vorkommen muss. Es nimmt drei Parameter entgegen: Den Projektnamen, die Projektversion und die E-Mail Adresse, an welche “Bugreports” geschickt werden sollen. Hier sieht man, dass alle Parameter mit den eckigen Klammern umgeben sind. Dies ist das “Quoten”.

Nachdem auch *Automake* die “configure.in” während seinem Ablauf analysiert, ist es notwendig das `AM_INIT_AUTOMAKE` Makro darin zu plazieren; am besten gleich nach

der *Autoconf*-Initialisierung. Die beiden Parameter sind wie zuvor der Projektname und die Projektversion.

Mit dem Makro `AC_CONFIG_SRCDIR` überprüft *Autoconf*, ob das richtige Verzeichnis für den Quellcode angegeben wurde, da es zum Zeitpunkt des Aufrufs von “configure” möglich ist, das Quellverzeichnis explizit anzugeben. Dem Makro wird eine Datei übergeben, so dass *Autoconf* feststellen kann, ob sich diese Datei in dem Ordner befindet. Kann sie nicht gefunden werden, ist offensichtlich das falsche Verzeichnis angegeben worden.

Das Makro `AC_CONFIG_FILES`, mit dem Argument “Makefile”, weist *Autoconf* an, nach einer Datei names “Makefile.in” zu suchen, alle Variablen dort mit den entsprechenden Werten zu ersetzen und die Ausgabe in “Makefile” zu speichern.

Das `AC_PROG_CC` führt mehr oder weniger Checks im System durch, um sicherzustellen, dass ein funktionierender C Compiler vorhanden ist und setzt Umgebungsvariablen, damit Ersetzungen in der “Makefile.in” vorgenommen werden können.

Das letzte Makro heißt `AC_OUTPUT`. Dieses bildet immer den Abschluß einer “configure.in”. Dieses Makro ist neben `AC_INIT` das einzige, welches definiert sein muss.

Neben den bereits besprochenen Makros, ist unter anderem noch `AC_ARG_ENABLE` von Hand eingefügt worden. Dieses Makro ist dafür verantwortlich, dass das Konfigurationskript bspw. die `--enable-repeat` Option versteht. Obwohl das Makro ziemlich kompliziert aussieht, besteht es im Grunde genommen nur aus drei Argumenten. Das erste übergebene Argument ist der Name der Option. Als zweites kommt, ein Hilfetext, der bei dem einem Aufruf von “configure” mit der Option `--help` ausgegeben wird. Als drittes bekommt das Makro einen Shell-code, der dann ausgeführt wird, wenn die Option (hier `--enable-repeat`) beim Konfigurieren angegeben wurde. In diesem Fall wird eine interne Shellvariable auf einen Wert gesetzt, der ohne Argumente an die Option “yes” ist (siehe `--enable-feature=[arg]` auf Seite 7). Wird das Feature beim Konfigurieren ausgewählt, so wird *Autoconf* angewiesen die Präprozessorvariable `CONFIG_REPEAT` zu definieren. Dies geschieht über das Makro `AC_DEFINE`.

Last but not least ... Im Absatz zuvor wurde eine Präprozessorvariable definiert. Die Frage ist jedoch: Wo wird sie definiert? Wie wird sie dem Quelltext bekannt gemacht? Prinzipiell ist es möglich solche Variablen über die “-D” Option eines Compilers dem Quelltext bekannt zu geben. Dies kann jedoch zu Problemen führen. Deshalb wird eine Datei names “config.h” erstellt, die die Definitionen wie z.B. `CONFIG_REPEAT` enthält. Die “config.h” inkludiert der Quelltext im hiesigen Beispiel. Das Makro `AC_CONFIG_HEADER` läßt die Definitionen in die Datei “config.h” einfließen, vorausgesetzt die entsprechende Datei “config.h.in” existiert.

3.5 config.h.in

“config.h.in” ist ein Template für die durch “configure” zu entstehende “config.h”. Wie *Autoscan* für “configure.in”, so ist *Autoheader* für die Erstellung von “config.h.in” ein

Werkzeug, das die gesamte Operation automatisiert. `autoheader` scant die zuvor erstellte “`configure.in`” nach Definitionen für den Präprozessor und erstellt eine entsprechende “`config.h.in`”.

3.6 Erstellen des Pakets

Nun sind alle erforderlichen Dateien (siehe Abschnitt Files auf Seite 10) erstellt und man kann das “`configure`” erstellen. Das Kommando `ls` sollte die folgenden Ausgabe produzieren:

```
$ pwd
/home/pete/hello
$ ls -F
AUTHORS      Makefile.am  README      configure.in
ChangeLog    NEWS         config.h.in  hello.c
```

Die letzten Schritte zu einem eigenen Softwarepaket “Hello 1.0” sind nur noch eine Kleinigkeit.

1. Es ist notwendig das Tool `aclocal` aufzurufen. Dieses führt einige wichtige Schritte durch, um sicherzustellen, dass die in “`configure.in`” verwendeten Makros richtig umgesetzt werden.
2. Nun kann `autoconf` aufgerufen werden. Sollten sich Fehler in der Eingabedatei “`configure.in`” befinden, wird *Autoconf* dies melden und die Arbeit abbrechen. Nach dem erfolgreichen Ablauf des Programms sollte im Verzeichnis das lang ersehnte Skript “`configure`” liegen.
3. “`configure`” kann jedoch noch nicht erfolgreich aufgerufen werden, da immer noch die “`Makefile.in`” fehlt. Diese wird einfach mit `automake` aus der Eingabedatei “`Makefile.am`” erstellt. In hiesigen Beispiel wird sich *Automake* darüber beklagen, dass immer noch einige Dateien fehlen. Diese kann *Automake* automatisch anlegen, in dem man dem Tool die Option `--add-missing` auf den Weg gibt.

Jetzt sollte das Projektverzeichnis wie folgt aussehen:

```
$ pwd
/home/pete/hello
$ ls -F
AUTHORS      Makefile.in  config.h.in  install-sh@
COPYING@     NEWS        configure*   missing@
ChangeLog    README      configure.in
INSTALL@     aclocal.m4  depcomp@
Makefile.am  autom4te.cache/  hello.c
```

Voilà ... es liegt ein Projekt nach allen Regeln der Kunst im Zeichen des GNUs⁵ vor. Allerdings ist mit all diesen Schritten nicht sichergestellt, dass es Fehler nicht gibt. Das Testen und Verbessern bleibt nicht aus.

Da nun ein “configure” und die “Makefile.in” vorliegen, kann die Software erstellt werden. Das Paket kann mit `./configure [options]` konfiguriert werden. Das Skript nimmt nun die als Kommandozeilenparameter übergebenen Optionen entgegen und erstellt zusammen mit den aus der Umgebung ermittelten Systemeigenschaften eine “Makefile” an Hand der Vorlage aus “Makefile.in”.

Wie versprochen, sollte nun die Software mit einem einfachen `./configure && make && make install` zu erstellen und zu installieren sein. Wird dem “configure” in vorliegendem Beispiel die Option `--enable-repeat` mit auf den Weg gegeben, sollte das erstellte Programm zehn mal den String “Hello World!” ausgeben.

3.7 Das Paket erstellen

Beim Erstellen eines Sourcecodepakets stellt sich grundsätzlich die Frage: Welche Dateien gehören in das resultierende “tar.gz” Archiv? Hier spaltet sich die Meinung der Experten stark. Während auf der einen Seite die Theorie steht, dass man nur die von Hand erstellten Dateien in die Archive packen sollte, ist es doch sehr üblich die Pakete schon mit dem fertigen “configure” und der “Makefile.in” auszuliefern. Die letztere Methode, Pakete, die an Normaluser gehen, mit dem generierten “configure” zu erstellen, hat sich stark verbreitet, so dass der Benutzer nur noch `./configure && make && make install` ausführen muss und *Autoconf* wie *Automake* nicht zu installieren braucht. Werden die Dateien dagegen z.B. für ein Backup oder in einem CVS-Repository abgelegt, sollten sich dort keine automatisch generierbaren Files befinden.

Um nun ein fertiges, benutzerfreundliches Softwarepaket im Sourcecodeformat zu erstellen, packt man es am besten in ein “zipped tape archive” mit der üblichen Endung “tar.gz”. Der Name des Archivs sollte in vorliegendem Beispiel “hello-1.0.tar.gz” lauten, wobei das Verzeichnis nach dem entpacken auch “hello-1.0” lauten sollte. Auf solche Kleinigkeiten und weitere Details jedes mal von Hand zu achten macht keinen Sinn und ist sehr fehleranfällig. Deshalb braucht man auch hierfür eine automatisierte Methode. Was leicht zu übersehen ist, ist die Tatsache, dass *Automake* in der generierten “Makefile.in” ein Target mit dem Namen “dist” reserviert, welches das Erstellen des Archivs übernimmt. Ein einfaches `make dist`⁶ erstellt das gewünschte Paketarchiv und legt es im Projektverzeichnis ab. Dieses kann man nun auf die Projektseite zum Download anbieten oder auf einen FTP-Server stellen oder direkt an seine Freunde verschicken. Man darf jedoch nicht vergessen, dass das `make` Kommando eine “Makefile” braucht, also muss man zuvor “configure” ausführen, so lange keine “Makefile” im Projektordner vorhanden ist.

⁵Die Anspielung hier meint das GNU Projekt von Richard M. Stallman, der mit anderen Programmierern die GNU Coding Standards entworfen hat.

⁶“dist” steht für “distribution”

4 Und was nun?

Nach dem der Leser einen Überblick über *Autoconf* und *Automake* bekommen hat, sollte er sich animiert fühlen, mit den Tools zu experimentieren, die Manuals zu lesen und sich selbst weiterzubilden.

Besonders empfehlenswert ist “Magic Happens Here” von David MacKenzie (siehe [4]), welches auch das sog. *Libtool* zur Erstellung von dynamischen Bibliotheken beschreibt.

Weiter ist ein sehr interessantes Tutorium unter [3] zu finden, das sich darauf konzentriert, dem Leser einen Weg im Umgang mit den GNU Entwicklertools zu bahnen.

Letztlich sollte man auf keinen Fall die beiden Manuals zu *Autoconf* (siehe [1]) und *Automake* (siehe [2]) vergessen. Sie bieten stets sehr gute Hilfe beim Nachschlagen von Makros und Vorgehensweisen, da sie sehr viele kleine Beispiele beinhalten.

Literatur

- [1] Free Software Foundation. *Autoconf Manual*. World Wide Web, <http://www.gnu.org/software/autoconf/manual/autoconf-2.57/autoconf.html>, 2002.
- [2] Free Software Foundation. *Automake Manual*. World Wide Web, <http://www.gnu.org/software/automake/manual/automake.html>, 2003.
- [3] Eleftherios Gkioulekas. *Learning Autoconf and Automake*. World Wide Web, http://www.amath.washington.edu/~lf/tutorials/autoconf/toolsmanual_toc.%html, 1999.
- [4] David MacKenzie. *Magic Happens Here*. World Wide Web, <http://docs.linux.cz/autobook-1.2/autobook.html>, 2000.