Makefiles + Autotools + CMake

Semminar Effiziente Programmierung

Benedikt Deike

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

22. Dezember 2020

Gliederung

Automation im Software Build Process

Build Automation

build automation beschreibt die Automatisierung des Software-Erstellungsprozesses. Der Software-Erstellungsprozess umfasst unter anderem:

Build Automation

build automation beschreibt die Automatisierung des Software-Erstellungsprozesses. Der Software-Erstellungsprozess umfasst unter anderem:

- Kompilieren von Computerquellcodes in Binärcode

Build Automation

build automation beschreibt die Automatisierung des Software-Erstellungsprozesses. Der Software-Erstellungsprozess umfasst unter anderem:

- Kompilieren von Computerquellcodes in Binärcode
- Ausführen automatisierter Tests

Build Automation

build automation beschreibt die Automatisierung des Software-Erstellungsprozesses. Der Software-Erstellungsprozess umfasst unter anderem:

- Kompilieren von Computerquellcodes in Binärcode
- Ausführen automatisierter Tests
- Installation von Bibliotheken bzw. Binärcode

Build Automation

build automation beschreibt die Automatisierung des Software-Erstellungsprozesses. Der Software-Erstellungsprozess umfasst unter anderem:

- Kompilieren von Computerquellcodes in Binärcode
- Ausführen automatisierter Tests
- Installation von Bibliotheken bzw. Binärcode
- Erstellen zur Distribution geeigneten Softwarepakete

Make

- Make ist ein *build automation tool*

Make

- Make ist ein *build automation tool*
- Make leitet mithilfe von sogenannten Makefiles Programme und Bibliotheken aus Sourcecode ab

Make

- Make ist ein *build automation tool*
- Make leitet mithilfe von sogenannten Makefiles Programme und Bibliotheken aus Sourcecode ab
- Makefiles beschreiben wie Make beim Ableiten vorzugehen hat

Make

- Make ist ein *build automation tool*
- Make leitet mithilfe von sogenannten Makefiles Programme und Bibliotheken aus Sourcecode ab
- Makefiles beschreiben wie Make beim Ableiten vorzugehen hat
- Make wurde 1976 von Stuart Feldman entwickelt und hat bis heute viele Umschreibungen durchlaufen

Make

- Make ist ein *build automation tool*
- Make leitet mithilfe von sogenannten Makefiles Programme und Bibliotheken aus Sourcecode ab
- Makefiles beschreiben wie Make beim Ableiten vorzugehen hat
- Make wurde 1976 von Stuart Feldman entwickelt und hat bis heute viele Umschreibungen durchlaufen
- Neuimplementationen, die dasselbe Dateiformat und dieselben algorithmischen Grundprinzipien verwenden, sind in den letzten Jahrzehnten entstanden

GNU Make

- GNU Make ist die Standardimplementierung von Make für Linux und macOS

GNU Make

- GNU Make ist die Standardimplementierung von Make für Linux und macOS
- GNU Make bietet mehrere Erweiterungen gegenüber Make, wie z.B.:

GNU Make

- GNU Make ist die Standardimplementierung von Make für Linux und macOS
- GNU Make bietet mehrere Erweiterungen gegenüber Make, wie z.B.:
  - *conditionals*

GNU Make

- GNU Make ist die Standardimplementierung von Make für Linux und macOS
- GNU Make bietet mehrere Erweiterungen gegenüber Make, wie z.B.:
  - *conditionals*
  - *functions*

Eigenschaften von (GNU) Make

- (GNU) Make ist nicht spezifisch für eine bestimmte Programmiersprache

Eigenschaften von (GNU) Make

- (GNU) Make ist nicht spezifisch für eine bestimmte Programmiersprache
- (GNU) Make führt, im Makefile aufgelistete, Shell-Befehle aus

Eigenschaften von (GNU) Make

- (GNU) Make ist nicht spezifisch für eine bestimmte Programmiersprache
- (GNU) Make führt, im Makefile aufgelistete, Shell-Befehle aus

  Beliebige Programme wie z.B. Compiler und Linker lassen sich über Shell-Befehle aufrufen

Eigenschaften von (GNU) Make

- (GNU) Make ist nicht spezifisch für eine bestimmte Programmiersprache
- (GNU) Make führt, im Makefile aufgelistete, Shell-Befehle aus

  Beliebige Programme wie z.B. Compiler und Linker lassen sich über Shell-Befehle aufrufen
- Im Makefile beschriebene Abhängigkeiten zwischen Programmdateien werden von (GNU) Make berücksichtigt

Eigenschaften von (GNU) Make

- (GNU) Make ist nicht spezifisch für eine bestimmte Programmiersprache
- (GNU) Make führt, im Makefile aufgelistete, Shell-Befehle aus

  Beliebige Programme wie z.B. Compiler und Linker lassen sich über Shell-Befehle aufrufen
- Im Makefile beschriebene Abhängigkeiten zwischen Programmdateien werden von (GNU) Make berücksichtigt
- Aktualisierte Quelldateien eines Programms werden von (GNU) Make erkannt

Eigenschaften von (GNU) Make

- (GNU) Make ist nicht spezifisch für eine bestimmte Programmiersprache
- (GNU) Make führt, im Makefile aufgelistete, Shell-Befehle aus

  Beliebige Programme wie z.B. Compiler und Linker lassen sich über Shell-Befehle aufrufen
- Im Makefile beschriebene Abhängigkeiten zwischen Programmdateien werden von (GNU) Make berücksichtigt
- Aktualisierte Quelldateien eines Programms werden von (GNU) Make erkannt
- Beim Ableiten von Programmen stellt (GNU) Make fest, welche Programmdateien in welcher Reihenfolge aktualisiert werden müssen

Makefile

Makefile - Beispiel

```
# Das Makefile im Verzeichnis extra enthält
# eine Definition der Variable 'var'.
include extra/Makefile

objects = main.o command.o display.o \
          insert.o search.o files.o
remove := rm

# Leite die ausführbare Datei 'edit' aus allen in
# objects' aufgelisteten Objektdateien ab.

edit : $(objects)
    gcc -o edit $(objects)

main.o : main.c defs.h
    gcc -c main.c
command.o : command.c defs.h command.h
    gcc -c command.c
display.o : display.c defs.h buffer.h
    gcc -c display.c
insert.o : insert.c defs.h buffer.h
    gcc -c insert.c
search.o : search.c defs.h buffer.h
    gcc -c search.c
files.o : files.c defs.h buffer.h command.h
    gcc -c files.c

# Gebe den Inhalt aller Varialben aus.
.PHONY : echo
echo :
    @echo $(objects) $(remove) $(var)

# Lösche alle in diesem Makefile beschriebenen Ziel.
.PHONY : clean
clean :
    $(remove) edit $(objects)
```

Alle Erwähnungen von Make beziehen sich ab sofort auf GNU Make!

Namen

- Auf der Suche nach einem Makefile wird standardmäßig das Vorhandensein folgender Dateien in der aufgeführten Reihenfolge durch Make geprüft:
  - `GNUmakefile`
  - `makefile`
  - `Makefile`

Namen

- Auf der Suche nach einem Makefile wird standardmäßig das Vorhandensein folgender Dateien in der aufgeführten Reihenfolge durch Make geprüft:
  - `GNUmakefile`
  - `makefile`
  - `Makefile`
- `Makefile` ist als Name empfohlen, da er am Anfang einer Verzeichnisliste aufgeführt wird

Namen

- Auf der Suche nach einem Makefile wird standardmäßig das Vorhandensein folgender Dateien in der aufgeführten Reihenfolge durch Make geprüft:
  - `GNUmakefile`
  - `makefile`
  - `Makefile`
- `Makefile` ist als Name empfohlen, da er am Anfang einer Verzeichnisliste aufgeführt wird
- Andere Make Implementationen ignorieren dem Namen `GNUmakefile`, weshalb dieser nicht empfohlen ist

Zeilenumbruch

- Makefiles verwenden eine zeilenbasierte Syntax, der Zeilenumbruch markiert das Ende einer Anweisung

Zeilenumbruch

- Makefiles verwenden eine zeilenbasierte Syntax, der Zeilenumbruch markiert das Ende einer Anweisung
- Make hat keine Begrenzung für die Länge einer Anweisungszeile

Zeilenumbruch

- Makefiles verwenden eine zeilenbasierte Syntax, der Zeilenumbruch markiert das Ende einer Anweisung
- Make hat keine Begrenzung für die Länge einer Anweisungszeile
- Mit einem durch das Backslash `\` maskierten Zeilenumbruch können Makefiles zur besseren Lesbarkeit mit Zeilenumbrüchen formatiert werden

Zeilenumbruch

- Makefiles verwenden eine zeilenbasierte Syntax, der Zeilenumbruch markiert das Ende einer Anweisung
- Make hat keine Begrenzung für die Länge einer Anweisungszeile
- Mit einem durch das Backslash `\` maskierten Zeilenumbruch können Makefiles zur besseren Lesbarkeit mit Zeilenumbrüchen formatiert werden
- *recipe lines* mit maskierten Zeilenumbrüchen werden als eine logische Zeile an die Shell übergeben

Zeilenumbruch

- Makefiles verwenden eine zeilenbasierte Syntax, der Zeilenumbruch markiert das Ende einer Anweisung
- Make hat keine Begrenzung für die Länge einer Anweisungszeile
- Mit einem durch das Backslash `\` maskierten Zeilenumbruch können Makefiles zur besseren Lesbarkeit mit Zeilenumbrüchen formatiert werden
- *recipe lines* mit maskierten Zeilenumbrüchen werden als eine logische Zeile an die Shell übergeben
- Das Backslash sowie der Zeilenumbruch werden in *recipe lines* nicht entfernt, sondern an die Shell weitergeleitet

Inhalt

Makefiles bestehen aus fünf Komponenten:

- Kommentaren
- Variablen (Definitionen)
- Direktiven
- *explicit rules*
- *implicit rules (Diese werden nicht besprochen!)*

Kommentar - Definition

- Das Hashtag `#` in einer *non-recipe line* startet einen Kommentar

Kommentar - Definition

- Das Hashtag `#` in einer *non-recipe line* startet einen Kommentar
- Kommentare werden von Make ignoriert

Kommentar - Definition

- Das Hashtag `#` in einer *non-recipe line* startet einen Kommentar
- Kommentare werden von Make ignoriert
- Ein Backslash, welcher den Zeilenumbruch am Ende einer Kommentarzeile maskiert, setzt den Kommentar über die nächste Zeil hinweg fort

Kommentar - Definition

- Das Hashtag `#` in einer *non-recipe line* startet einen Kommentar
- Kommentare werden von Make ignoriert
- Ein Backslash, welcher den Zeilenumbruch am Ende einer Kommentarzeile maskiert, setzt den Kommentar über die nächste Zeil hinweg fort
- In Variablenreferenzen und Funktionsaufrufen ignoriert Make keine Kommentare, weshalb sie dort nicht verwendet werden können

Kommentar - Definition

- Das Hashtag `#` in einer *non-recipe line* startet einen Kommentar
- Kommentare werden von Make ignoriert
- Ein Backslash, welcher den Zeilenumbruch am Ende einer Kommentarzeile maskiert, setzt den Kommentar über die nächste Zeil hinweg fort
- In Variablenreferenzen und Funktionsaufrufen ignoriert Make keine Kommentare, weshalb sie dort nicht verwendet werden können
- Kommentare innerhalb einer *recipe line* werden an die Shell übergeben

Kommentar - Definition

- Das Hashtag `#` in einer *non-recipe line* startet einen Kommentar
- Kommentare werden von Make ignoriert
- Ein Backslash, welcher den Zeilenumbruch am Ende einer Kommentarzeile maskiert, setzt den Kommentar über die nächste Zeil hinweg fort
- In Variablenreferenzen und Funktionsaufrufen ignoriert Make keine Kommentare, weshalb sie dort nicht verwendet werden können
- Kommentare innerhalb einer *recipe line* werden an die Shell übergeben
- Die Shell entscheidet, wie der Kommentar interpretiert wird

Kommentar - Beispiel

```
# Lorem ipsum dolor sit amet, consectetur \
  adipiscing elit. Morbi mattis pretium   \
  nunc vitae posuere. Pellentesque        \
  habitant morbi tristique senectus et    \
  netus et malesuada fames ac turpis      \
  egestas.

# Morbi ut tellus fermentum,
# gravida magna a, hendrerit augue.
```

Variablen - Definition

- Eine Variable in einem Makefile ist ein Name, welcher einen Wert repräsentiert

Variablen - Definition

- Eine Variable in einem Makefile ist ein Name, welcher einen Wert repräsentiert
- Variablen können durch Referenzierung in *targets*, *prerequisites*, *recipes* und andere Teile des Makefiles substituiert werden

Variablen - Definition

- Eine Variable in einem Makefile ist ein Name, welcher einen Wert repräsentiert
- Variablen können durch Referenzierung in *targets*, *prerequisites*, *recipes* und andere Teile des Makefiles substituiert werden
- Variablennamen können eine beliebige Folge von Zeichen sein, die kein Doppelpunkt `:`, Hashtag `#`, Gleichheitszeichen `=` oder Leerzeichen enthalten

Variablen - Definition

- Eine Variable in einem Makefile ist ein Name, welcher einen Wert repräsentiert
- Variablen können durch Referenzierung in *targets*, *prerequisites*, *recipes* und andere Teile des Makefiles substituiert werden
- Variablennamen können eine beliebige Folge von Zeichen sein, die kein Doppelpunkt `:`, Hashtag `#`, Gleichheitszeichen `=` oder Leerzeichen enthalten
- Make unterscheidet bei Variablennamen zwischen Groß- und Kleinschreibung

Variablen - Definition

- Eine Variable in einem Makefile ist ein Name, welcher einen Wert repräsentiert
- Variablen können durch Referenzierung in *targets*, *prerequisites*, *recipes* und andere Teile des Makefiles substituiert werden
- Variablennamen können eine beliebige Folge von Zeichen sein, die kein Doppelpunkt `:`, Hashtag `#`, Gleichheitszeichen `=` oder Leerzeichen enthalten
- Make unterscheidet bei Variablennamen zwischen Groß- und Kleinschreibung
- Der Variablenname kann Funktions- und Variablenreferenzen enthalten, die beim Lesen des Variablennames substituiert werden

Variablen - Wertezuweisung

- Eine Zeile im Makefile, die mit einem gültigen Variablennamen beginnt und von einem Zuweisungsoperator (`=`, `:=`, `::=`) gefolgt wird, definiert eine Variable

Variablen - Wertezuweisung

- Eine Zeile im Makefile, die mit einem gültigen Variablennamen beginnt und von einem Zuweisungsoperator (`=`, `:=`, `::=`) gefolgt wird, definiert eine Variable
- Der Variablenwert einer Variable steht nach dem Zuweisungsoperator

Variablen - Wertezuweisung

- Eine Zeile im Makefile, die mit einem gültigen Variablennamen beginnt und von einem Zuweisungsoperator (`=`, `:=`, `::=`) gefolgt wird, definiert eine Variable
- Der Variablenwert einer Variable steht nach dem Zuweisungsoperator
- Der Wert einer Variablen kann zur besseren Lesbarkeit in mehrere physische Zeilen aufgeteilt werden

Variablen - Wertezuweisung - Beispiel

```
# Die Variable objects speichert die
# Zeichenkette "main.o utils.o edit.o"

objects = main.o utils.o edit.o
```

Variablen - Referenzierung

- Ein Dollarzeichen `$` gefolgt vom Namen einer Variablen in eckigen oder runden Klammern referenziert diese

Variablen - Referenzierung

- Ein Dollarzeichen `$` gefolgt vom Namen einer Variablen in eckigen oder runden Klammern referenziert diese
- Beim Lesen des Makefiles durch Make werden alle referenzierten Variablen durch deren konkrete Werte substituiert, mit Ausnahme von:

Variablen - Referenzierung

- Ein Dollarzeichen `$` gefolgt vom Namen einer Variablen in eckigen oder runden Klammern referenziert diese
- Beim Lesen des Makefiles durch Make werden alle referenzierten Variablen durch deren konkrete Werte substituiert, mit Ausnahme von:
  - Referenzierten Variablen in *recipes*

    Variablenreferenzen in Rezepten werden erst beim Ausführen des Rezepts substituiert!

Variablen - Referenzierung

- Ein Dollarzeichen `$` gefolgt vom Namen einer Variablen in eckigen oder runden Klammern referenziert diese
- Beim Lesen des Makefiles durch Make werden alle referenzierten Variablen durch deren konkrete Werte substituiert, mit Ausnahme von:
  - Referenzierten Variablen in *recipes*

    Variablenreferenzen in Rezepten werden erst beim Ausführen des Rezepts substituiert!
  - Der rechten Seite von Variablen, die mit dem Zuweisungsoperator `=` definiert wurden

Variablen - Referenzierung

- Ein Dollarzeichen `$` gefolgt vom Namen einer Variablen in eckigen oder runden Klammern referenziert diese
- Beim Lesen des Makefiles durch Make werden alle referenzierten Variablen durch deren konkrete Werte substituiert, mit Ausnahme von:
  - Referenzierten Variablen in *recipes*

    Variablenreferenzen in Rezepten werden erst beim Ausführen des Rezepts substituiert!
  - Der rechten Seite von Variablen, die mit dem Zuweisungsoperator `=` definiert wurden
  - Den Körpern von Variablen, die mit der Direktive `define` definiert wurden

Variablen - Referenzierung - Beispiel

```
# $(foo) oder ${foo} sind gültige
# Referenzen auf die Variable foo.

foo = Hallo

# Hallo = Welt

$(foo) = Welt

# HalloWelt := Hallo Welt

HalloWelt := ${foo} ${$(foo)}
```

Recursively Expanded Variables

- *recursively expanded variables* werden durch die `define` Direktive oder den Zuweisungsoperator `=` definiert

Recursively Expanded Variables

- *recursively expanded variables* werden durch die `define` Direktive oder den Zuweisungsoperator `=` definiert
- Referenzen im Wert von *recursively expanded variables* werden erst zu deren Substitution aufgelöst!

Recursively Expanded Variables - Beispiel

```
# Dieses Makefile gibt Huh?
# auf der Konsole aus.

foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
	echo $(foo)
```

Recursively Expanded Variables - Beispiel

```
# Beim Substituieren der Variablen
# CFLAGS würde es zu einer
# Endlosschleife kommen. Make
# erkennt dies und meldet einen
# Fehler.

CFLAGS = $(CFLAGS) -O
```

Simply Expanded Variables

- *simply expanded variables* werden durch den Zuweisungsoperator `:=` oder `::=` definiert

Simply Expanded Variables

- *simply expanded variables* werden durch den Zuweisungsoperator `:=` oder `::=` definiert
- Referenzen im Wert von *simply expanded variables* werden beim Lesen des Makefiles aufgelöst

Simply Expanded Variables

- *simply expanded variables* werden durch den Zuweisungsoperator `:=` oder `::=` definiert
- Referenzen im Wert von *simply expanded variables* werden beim Lesen des Makefiles aufgelöst
- Der Wert einer *simply expanded variable* steht schon zu deren Definition fest!

Simply Expanded Variables - Beispiel

```
# Diese Variablendefinition ist
# äquivalent mit

x := one
y := $(x) two
x := later

# folgender Variablendefinition.

y := one two
x := later
```

Simply Expanded Variables - Beispiel

```
# Diese Variablendefinition ist
# äquivalent mit

CFLAGS := $(CFLAGS) -O

# folgender Variablendefinition,
# falls die Variable CFLAGS nicht
# bereits definiert ist.

CFLAGS := -O
```

Automatic Variables

- *automatic variables* besitzen nur innerhalb von *recipes* einen Wert

Automatic Variables

- *automatic variables* besitzen nur innerhalb von *recipes* einen Wert
- Der Wert einer *automatic variable* bezieht sich entweder auf die *prerequisites* oder das *target* einer *rule*

Automatic Variables

- *automatic variables* besitzen nur innerhalb von *recipes* einen Wert
- Der Wert einer *automatic variable* bezieht sich entweder auf die *prerequisites* oder das *target* einer *rule*
- *automatic variables* können nicht in der *target list* einer *rule* verwendet werden, sie haben dort keinen Wert und werden durch die leere Zeichenfolge substituiert

Automatic Variables

- *automatic variables* besitzen nur innerhalb von *recipes* einen Wert
- Der Wert einer *automatic variable* bezieht sich entweder auf die *prerequisites* oder das *target* einer *rule*
- *automatic variables* können nicht in der *target list* einer *rule* verwendet werden, sie haben dort keinen Wert und werden durch die leere Zeichenfolge substituiert
  - `$<` wird durch das erste *prerequisite* der *rule* ersetzt

Automatic Variables

- *automatic variables* besitzen nur innerhalb von *recipes* einen Wert
- Der Wert einer *automatic variable* bezieht sich entweder auf die *prerequisites* oder das *target* einer *rule*
- *automatic variables* können nicht in der *target list* einer *rule* verwendet werden, sie haben dort keinen Wert und werden durch die leere Zeichenfolge substituiert
  - `$<` wird durch das erste *prerequisite* der *rule* ersetzt
  - `$^` wird durch eine Liste aller *prerequisites* der *rule* ersetzt

Automatic Variables

- *automatic variables* besitzen nur innerhalb von *recipes* einen Wert
- Der Wert einer *automatic variable* bezieht sich entweder auf die *prerequisites* oder das *target* einer *rule*
- *automatic variables* können nicht in der *target list* einer *rule* verwendet werden, sie haben dort keinen Wert und werden durch die leere Zeichenfolge substituiert
  - `$<` wird durch das erste *prerequisite* der *rule* ersetzt
  - `$^` wird durch eine Liste aller *prerequisites* der *rule* ersetzt
  - `$@` wird durch den Namen des *targets* ersetzt

Direktive - Include - Beispiel

```
include filename filename ...
```

Direktive - Include

- `include` weist Make an, das Lesen des aktuellen Makefiles auszusetzen und ein oder mehrere andere Makefiles zu lesen

Direktive - Include

- `include` weist Make an, das Lesen des aktuellen Makefiles auszusetzen und ein oder mehrere andere Makefiles zu lesen

  Nach dem Lesen der Makefiles wird mit dem Lesen des ursprünglichen Makefiles fortgefahren

Direktive - Include

- `include` weist Make an, das Lesen des aktuellen Makefiles auszusetzen und ein oder mehrere andere Makefiles zu lesen

  Nach dem Lesen der Makefiles wird mit dem Lesen des ursprünglichen Makefiles fortgefahren

- Alle durch `include` eingelesen Makefile Inhalte stehen anschließend zur Verfügung

Direktive - Include

- `include` weist Make an, das Lesen des aktuellen Makefiles auszusetzen und ein oder mehrere andere Makefiles zu lesen

  Nach dem Lesen der Makefiles wird mit dem Lesen des ursprünglichen Makefiles fortgefahren

- Alle durch `include` eingelesen Makefile Inhalte stehen anschließend zur Verfügung

  Ein Projekt, mit mehreren Makefiles, kann z.B. mithilfe von `include` auf einen gemeinsamen Variablenbestand zugreifen

Direktive - Include

- `include` weist Make an, das Lesen des aktuellen Makefiles auszusetzen und ein oder mehrere andere Makefiles zu lesen

  Nach dem Lesen der Makefiles wird mit dem Lesen des ursprünglichen Makefiles fortgefahren

- Alle durch `include` eingelesen Makefile Inhalte stehen anschließend zur Verfügung

  Ein Projekt, mit mehreren Makefiles, kann z.B. mithilfe von `include` auf einen gemeinsamen Variablenbestand zugreifen

- Variablen- oder Funktionsreferenzen im Dateinamen der einzulesenden Makefiles werden substituiert

Da *implicit rules* in diesem Vortrag nicht behandelt werden wird der Begriff *rule* als Synonym für *explicit rule* verwendet!

Rule

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

Rule - Definition

- Mithilfe einer *rule* kann Make feststellen, ob die *targets* der *rule* veraltet sind und wie diese *targets* bei Bedarf aktualisiert werden können

Rule - Definition

- Mithilfe einer *rule* kann Make feststellen, ob die *targets* der *rule* veraltet sind und wie diese *targets* bei Bedarf aktualisiert werden können
- Ob ein *target* veraltet ist, wird anhand der *prerequisites* festgestellt

Rule - Definition

- Mithilfe einer *rule* kann Make feststellen, ob die *targets* der *rule* veraltet sind und wie diese *targets* bei Bedarf aktualisiert werden können
- Ob ein *target* veraltet ist, wird anhand der *prerequisites* festgestellt
- Ein *target* ist veraltet, wenn es nicht vorhanden oder älter als eines der *prerequisites* ist

Rule - Definition

- Mithilfe einer *rule* kann Make feststellen, ob die *targets* der *rule* veraltet sind und wie diese *targets* bei Bedarf aktualisiert werden können
- Ob ein *target* veraltet ist, wird anhand der *prerequisites* festgestellt
- Ein *target* ist veraltet, wenn es nicht vorhanden oder älter als eines der *prerequisites* ist
- Das *recipe* einer *rule* beschreibt, wie das *target* (neu) abgeleitet und somit aktualisiert werden kann

Rule - Beispiel

```
# Beispiel einer simplen Regel.

main.o: main.c defs.h
    gcc -c main.c
```

Rule - Default Goal

- Die Reihenfolge der *rules* im Makefile ist nicht signifikant, außer für die Bestimmung des *default goals*

Rule - Default Goal

- Die Reihenfolge der *rules* im Makefile ist nicht signifikant, außer für die Bestimmung des *default goals*
- Falls Make ohne Parameter aufgerufen wird, versucht es dieses *default goal* abzuleiten

Rule - Default Goal

- Die Reihenfolge der *rules* im Makefile ist nicht signifikant, außer für die Bestimmung des *default goals*
- Falls Make ohne Parameter aufgerufen wird, versucht es dieses *default goal* abzuleiten
- Das *default goal* ist das *target* der ersten *rule* im ersten Makefile

Rule - Default Goal

- Die Reihenfolge der *rules* im Makefile ist nicht signifikant, außer für die Bestimmung des *default goals*
- Falls Make ohne Parameter aufgerufen wird, versucht es dieses *default goal* abzuleiten
- Das *default goal* ist das *target* der ersten *rule* im ersten Makefile
- Wenn die erste *rule* mehrere *targets* hat, wird nur das erste *target* als *defaul goal* verwendet

Rule - Recipe

```
target ... : prerequisites ... ; recipe
    recipe
    ...
    ...
```

Rule - Recipe

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

Rule - Recipe

- In einem Makefile existieren normalerweise zwei Syntaxen

Rule - Recipe

- In einem Makefile existieren normalerweise zwei Syntaxen
- Alle *non-recipe lines* werden in Make Syntax verfasst, wo hingegen *recipe lines* in der Shell Syntax geschrieben sind

Rule - Recipe

- In einem Makefile existieren normalerweise zwei Syntaxen
- Alle *non-recipe lines* werden in Make Syntax verfasst, wo hingegen *recipe lines* in der Shell Syntax geschrieben sind
- Das *recipe* einer *rule* besteht aus einer oder mehreren *recipe lines*

Rule - Recipe

- In einem Makefile existieren normalerweise zwei Syntaxen
- Alle *non-recipe lines* werden in Make Syntax verfasst, wo hingegen *recipe lines* in der Shell Syntax geschrieben sind
- Das *recipe* einer *rule* besteht aus einer oder mehreren *recipe lines*
- Die erste *recipe line* steht entweder in der selben Zeile wie die *prerequisites*, durch ein Semikolon von diesen getrennt oder in der Zeile nach den *prerequisites*

Rule - Recipe

- In einem Makefile existieren normalerweise zwei Syntaxen
- Alle *non-recipe lines* werden in Make Syntax verfasst, wo hingegen *recipe lines* in der Shell Syntax geschrieben sind
- Das *recipe* einer *rule* besteht aus einer oder mehreren *recipe lines*
- Die erste *recipe line* steht entweder in der selben Zeile wie die *prerequisites*, durch ein Semikolon von diesen getrennt oder in der Zeile nach den *prerequisites*
- Letztere *recipe lines* beginnen immer mit einem Tabulator!

Rule - Recipe

- Make gibt *recipe line* für *recipe line* in der aufgeführten Reihenfolge an die Shell weiter

Rule - Recipe

- Make gibt *recipe line* für *recipe line* in der aufgeführten Reihenfolge an die Shell weiter
- Variablensubstitution ist eines der wenigen Modifikationen, welche Make an *recipe lines* vornimmt, bevor es diese an die Shell übergibt

Rule - Recipe

- Make gibt *recipe line* für *recipe line* in der aufgeführten Reihenfolge an die Shell weiter
- Variablensubstitution ist eines der wenigen Modifikationen, welche Make an *recipe lines* vornimmt, bevor es diese an die Shell übergibt
- Jede *recipe line* wird in einer neuen Shell Instanz ausgeführt

Rule - Recipe

- Make gibt *recipe line* für *recipe line* in der aufgeführten Reihenfolge an die Shell weiter
- Variablensubstitution ist eines der wenigen Modifikationen, welche Make an *recipe lines* vornimmt, bevor es diese an die Shell übergibt
- Jede *recipe line* wird in einer neuen Shell Instanz ausgeführt
- *recipes* in Makefiles werden immer von `/bin/sh` interpretiert, sofern im Makefile nichts anderes spezifiziert ist

Rule - Prerequisite

```
target ... : prerequisites ...
    recipe
    ...
    ...
```

Rule - Prerequisite

- Ein *prerequisite* ist eine Datei welche zum erzeugen des *targets* benötigt wird

Rule - Prerequisite

- Ein *prerequisite* ist eine Datei welche zum erzeugen des *targets* benötigt wird
- Ein *target* benötigt oft mehrere *prerequisites*

Rule - Prerequisite

```
target ... : prerequisites ...
     recipe
     ...
     ...
```

Rule - Target

Ein *target* ist normalerweise der Name einer Datei, welche durch das *recipe* abgeleitet wird

Rule - Independent Target

- Falls eine *rule* mehrere *targets* angibt und diese von den *prerequisites* mit dem Separator `:` trennt, so handelt es sich um *independent targets*

Rule - Independent Target

- Falls eine *rule* mehrere *targets* angibt und diese von den *prerequisites* mit dem Separator `:` trennt, so handelt es sich um *independent targets*
- Diese Schreibweise entspricht dem mehrfachen Schreiben der *rule* für jedes *target* mit gleichen *recipe* und *prerequisites*

Rule - Independent Target - Beispiel

```
bigoutput littleoutput: text.g
    generate text.g -$(subst output,,$@) >$@

# Die obere Regel ist äquivalent zu
# der untern Regel.

bigoutput: text.g
    generate text.g -big >bigoutput
littleoutput: text.g
    generate text.g -little >littleoutput
```

Rule - Grouped Targets

- Falls eine Regel mehrere *targets* angibt und diese von den *prerequisites* mit dem Separator `&:` trennt, so handelt es sich um *grouped targets*

Rule - Grouped Targets

- Falls eine Regel mehrere *targets* angibt und diese von den *prerequisites* mit dem Separator `&:` trennt, so handelt es sich um *grouped targets*
- Ist eines der *grouped targets* veraltet bzw. nicht existent, so werden alle *grouped targets* aktualisiert bzw. abgeleitet

Rule- Grouped Targets - Beispiel

```
# Falls eines der drei Ziele
# veraltet ist, so werden alle
# Ziel aktualisiert.

win gar dium &: levi ohhh sa
	echo $^ >win
	echo $^ >gar
	echo $^ >dium
```

Rule - Phony Target

- Ein *target* muss nicht der Name einer Datei, sonder kann auch der Name einer auszuführenden Aktion sein

Rule - Phony Target

- Ein *target* muss nicht der Name einer Datei, sonder kann auch der Name einer auszuführenden Aktion sein
- Ein *phony target* ist der Name für ein *recipe*, das bei expliziter Anfrage ausgeführt wird

Rule - Phony Target - Beispiel

```
# Ein flasches Ziel ohne
# Voraussetzungen.

clean:
    rm *.o temp
```

Rule - Phony Target

- Falls eine Datei mit dem Namen `clean` im gleichen Verzeichnis wie das Makefile existiert, würden die *recipes* der *rule* mit dem Ziel `clean` nie ausgeführt werden, da `clean` immer als aktuell angesehen wird

Rule - Phony Target

- Falls eine Datei mit dem Namen `clean` im gleichen Verzeichnis wie das Makefile existiert, würden die *recipes* der *rule* mit dem Ziel `clean` nie ausgeführt werden, da `clean` immer als aktuell angesehen wird
- Dieses Verhalten kann umgangen werden, indem ein *phony target* als *prerequisite* des speziellen *target* `.PHONY` aufgeführt wird

Rule - Phony Target

- Falls eine Datei mit dem Namen `clean` im gleichen Verzeichnis wie das Makefile existiert, würden die *recipes* der *rule* mit dem Ziel `clean` nie ausgeführt werden, da `clean` immer als aktuell angesehen wird
- Dieses Verhalten kann umgangen werden, indem ein *phony target* als *prerequisite* des speziellen *target* `.PHONY` aufgeführt wird
- Ein *phony target* wird mit `.PHONY` explizit als solches gekennzeichnet

Rule - Phony Target - Beispiel

```
# Das flasche Ziel clean wird
# explizit als solches gekenn-
# zeichnet.

.PHONY: clean
clean:
    rm *.o temp
```

Rule - Phony Target

Falls sich in einem Verzeichnis mehrere Programme befinden, ist es oft angenehm diese mit nur einem Befehl abzuleiten. Die einfachste Lösung hierfür ist eine *rule* mit einem *phony target* zu Beginn des Makefiles (*default goal*) einzufügen. Dieses *phony target* wird oft `all` genannt. Als *prerequisites* müssen die Dateinamen aller abzuleitenden Programme angegeben werden.

Rule - Phony Target - Beispiel

```
# Mit der Anweisung make prog1
# lässt sich z.B. das Programm
# prog1 unabhängig von den anderen
# ableiten.

.PHONY: all
all: prog1 prog2 prog3

prog1: prog1.o utils.o
    gcc -o prog1 prog1.o utils.o

prog2: prog2.o
    gcc -o prog2 prog2.o

prog3: prog3.o sort.o utils.o
    gcc -o prog3 prog3.o sort.o utils.o
```

Makefile - Beispiel

```
# Das Makefile im Verzeichnis extra enthält
# eine Definition der Variable 'var'.
include extra/Makefile

objects = main.o command.o display.o \
          insert.o search.o files.o
remove := rm

# Leite die ausführbare Datei 'edit' aus allen in
# objects' aufgelisteten Objektdateien ab.

edit : $(objects)
    gcc -o edit $(objects)

main.o : main.c defs.h
    gcc -c main.c
command.o : command.c defs.h command.h
    gcc -c command.c
display.o : display.c defs.h buffer.h
    gcc -c display.c
insert.o : insert.c defs.h buffer.h
    gcc -c insert.c
search.o : search.c defs.h buffer.h
    gcc -c search.c
files.o : files.c defs.h buffer.h command.h
    gcc -c files.c

# Gebe den Inhalt aller Varialben aus.
.PHONY : echo
echo :
    @echo $(objects) $(remove) $(var)

# Lösche alle in diesem Makefile beschriebenen Ziel.
.PHONY : clean
clean :
    $(remove) edit $(objects)
```

Autotools

Motivation

- Der Software-Build-Prozesses wird weitestgehend durch Make und Makefiles automatisiert. Wenn aber ein Softwarepaket auf einer anderen Plattform als der, auf der es entwickelt wurde, erstellt werden soll, muss sein Makefile normalerweise angepasst werden. Beispielsweise kann der Compiler einen anderen Namen haben oder mehr Optionen erfordern.

Motivation

- Der Software-Build-Prozesses wird weitestgehend durch Make und Makefiles automatisiert. Wenn aber ein Softwarepaket auf einer anderen Plattform als der, auf der es entwickelt wurde, erstellt werden soll, muss sein Makefile normalerweise angepasst werden. Beispielsweise kann der Compiler einen anderen Namen haben oder mehr Optionen erfordern.

- 1991 hatte David J. MacKenzie es satt, Makefile für die 20 Plattformen anzupassen, mit denen er sich befassen musste. Stattdessen erstellte er ein kleines Shell-Skript namens `configure`, um das Makefile automatisch anzupassen. Das Ausführen von `./configure && make` war nun ausreichend um den Software-Build-Prozess automatisiert auf verschiedenen System durchzuführen.

GNU-Build-System

- Heutzutage ist dieser Prozess im GNU-Projekt standardisiert

GNU-Build-System

- Heutzutage ist dieser Prozess im GNU-Projekt standardisiert
- Die GNU-Coding-Standards beschreiben Minimalanforderungen an das Konfigurationsskript und Konventionen, welchen das Makefile folgen sollte

GNU-Build-System

- Heutzutage ist dieser Prozess im GNU-Projekt standardisiert
- Die GNU-Coding-Standards beschreiben Minimalanforderungen an das Konfigurationsskript und Konventionen, welchen das Makefile folgen sollte
- Die Standardisierung hat zu einem einheitlichen Build-System dem GNU-Build-System geführt, welche die Installation von GNU-Projekt konformen Softwarepaketen auf verschiedenen Plattformen mit der simplen Anweisung `./configure && make && make install` ermöglicht

Autotools

- Autotools sind Werkzeuge, mit denen ein GNU-Build-System für ein Softwarepaket erstellt werden kann

Autotools

- Autotools sind Werkzeuge, mit denen ein GNU-Build-System für ein Softwarepaket erstellt werden kann
- Unter den Autotools versteht man die Softwarepakete Autoconf, Automake und Libtool, welche sich unter anderem in folgende Programme untergliedern:

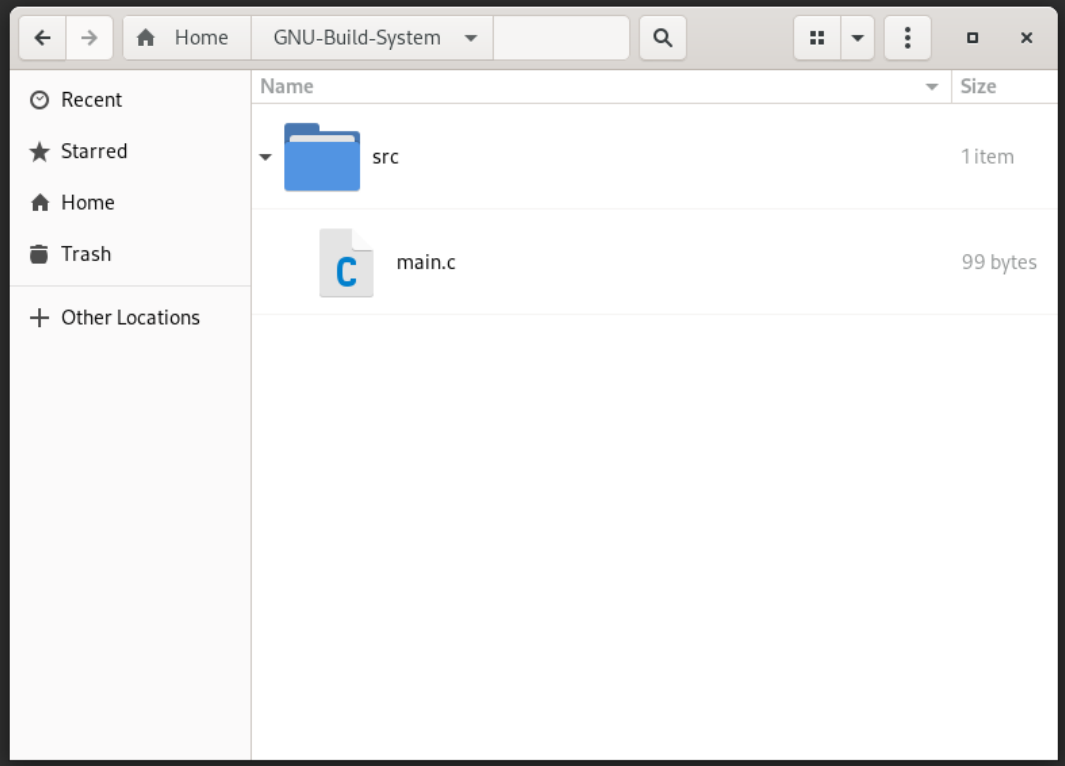
Autotools

- Autotools sind Werkzeuge, mit denen ein GNU-Build-System für ein Softwarepaket erstellt werden kann
- Unter den Autotools versteht man die Softwarepakete Autoconf, Automake und Libtool, welche sich unter anderem in folgende Programme untergliedern:
  - Autoconf: `autoconf`, `autoreconf`, `autoheader`, `autoscan`
  - Automake: `aclocal`, `automake`
  - Libtool: `libtoolize`

Autotools

- Autotools sind Werkzeuge, mit denen ein GNU-Build-System für ein Softwarepaket erstellt werden kann
- Unter den Autotools versteht man die Softwarepakete Autoconf, Automake und Libtool, welche sich unter anderem in folgende Programme untergliedern:
  - Autoconf: `autoconf`, `autoreconf`, `autoheader`, `autoscan`
  - Automake: `aclocal`, `automake`
  - Libtool: `libtoolize`
- Autoconf konzentriert sich hauptsächlich auf das Konfigurationsskript, wohingegen Automake sich auf die Makefiles konzentriert

Projektstruktur



main.c

```c
#include <config.h>
#include <stdio.h>

int main(void)
{
    printf("GNU-Build-System");
    return 0;
}
```

Autotools Konfigurationsdateien - configure.ac

```
dnl Initialisiere Autoconf, spezifiziere den Paketname und dessen Versionsnummer
dnl sowie eine E-mail Adresse für Fehlermeldungen.
AC_INIT([GNUBuildSystem], [1.0], [benedikt.deike@studium.uni-hamburg.de])

dnl Initialisiere Automake, spezifiziere dieses Paket als fremdes Paket und
dnl aktiviere alle Automake Warnungen und Melde diese als Fehler.
AM_INIT_AUTOMAKE([foreign -Wall -Werror])

dnl Suche nach einem C-Compiler
AC_PROG_CC

dnl Deklariere config.h als Ausgabe header.
AC_CONFIG_HEADERS([config.h])

dnl Deklariere Makefile und src/Makefile als Ausgabe Dateien.
AC_CONFIG_FILES([Makefile src/Makefile])

dnl Gib alle als Ausgabe deklarierten Dateien aus.
AC_OUTPUT
```

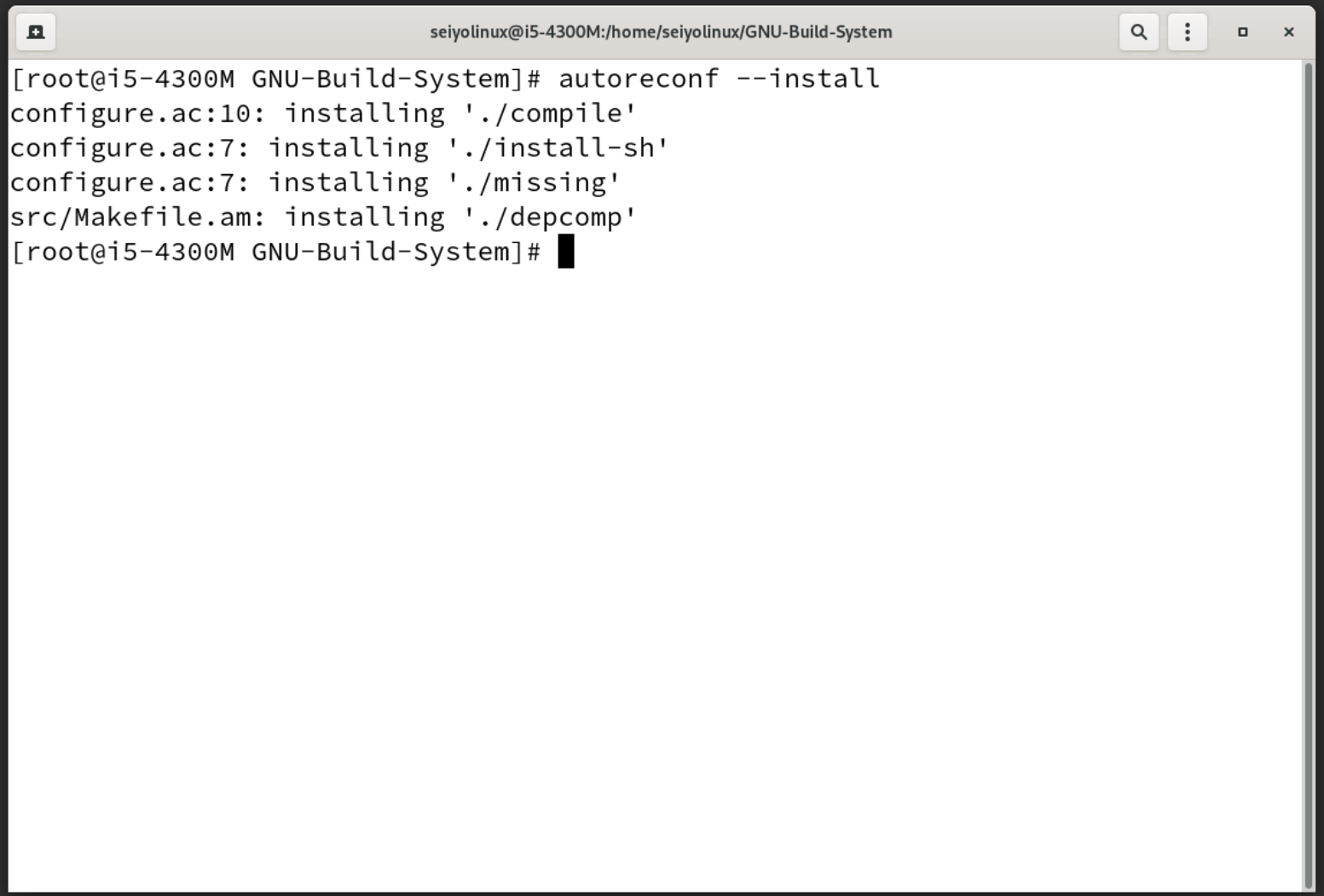
Autotools Konfigurationsdateien - Makefile.am

```
# Baue rekursiv in src/.
SUBDIRS = src
```

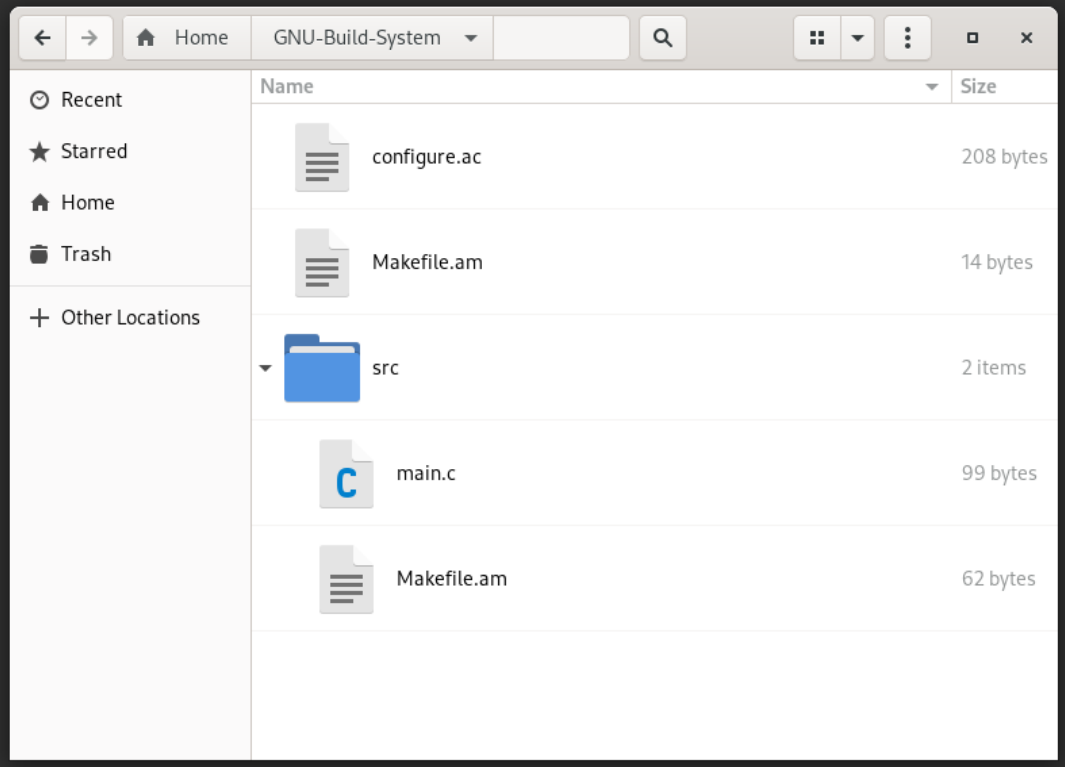
Autotools Konfigurationsdateien - src/Makefile.am

```
# Baue ein Programm namens GNUBuildSystem und installiere es in bindir.
# Der Standardwert der Variable bindir ist /usr/local/bin.
bin_PROGRAMS = GNUBuildSystem

# Leite das Programm GNUBuildSystem ab, durch kompilieren von main.c.
GNUBuildSystem_SOURCES = main.c
```

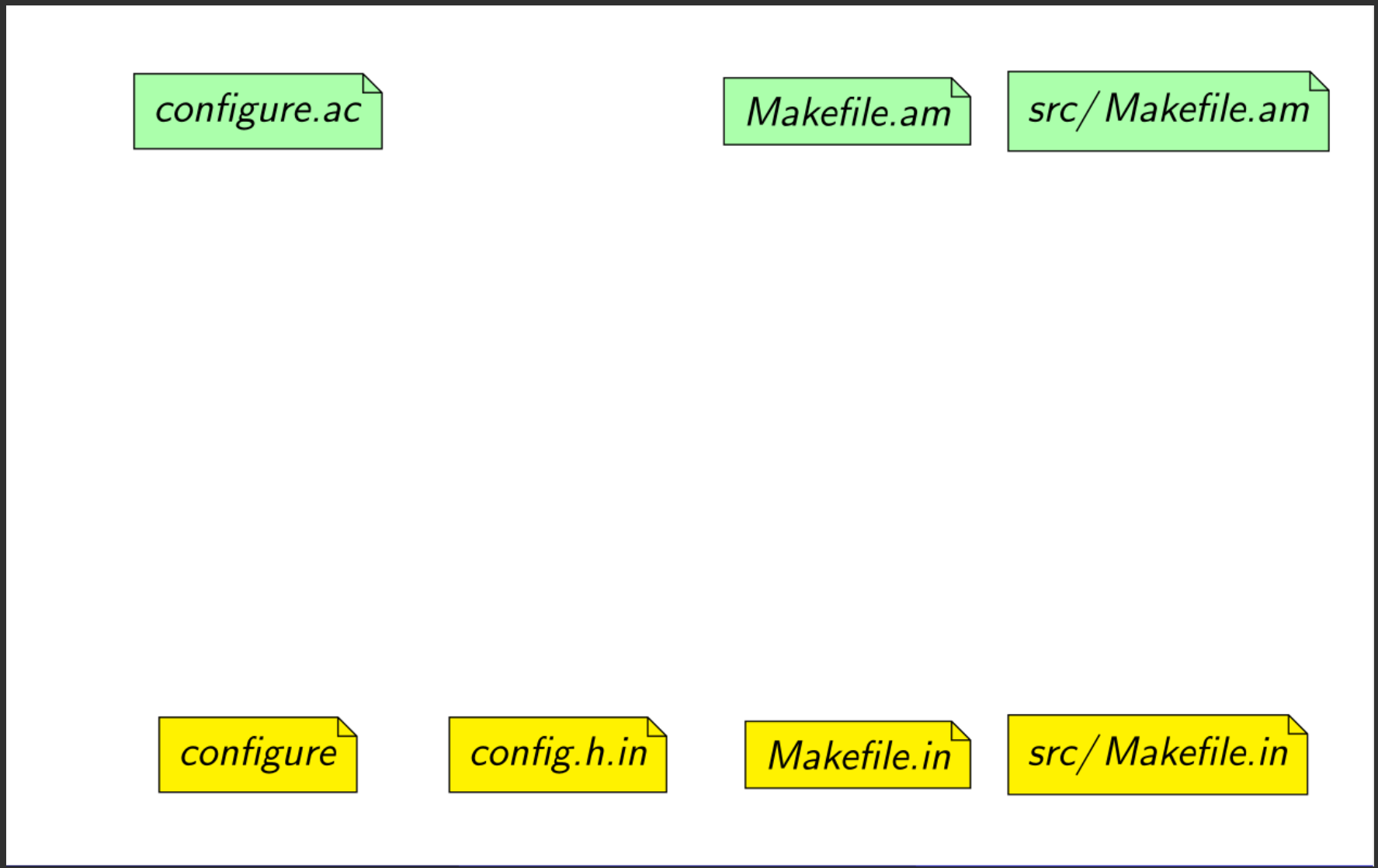
autoreconf


seiyolinux@i5-4300M:/home/seiyolinux/GNU-Build-System
[root@i5-4300M GNU-Build-System]# autoreconf --install
configure.ac:10: installing './compile'
configure.ac:7: installing './install-sh'
configure.ac:7: installing './missing'
src/Makefile.am: installing './depcomp'
[root@i5-4300M GNU-Build-System]#

autoreconf

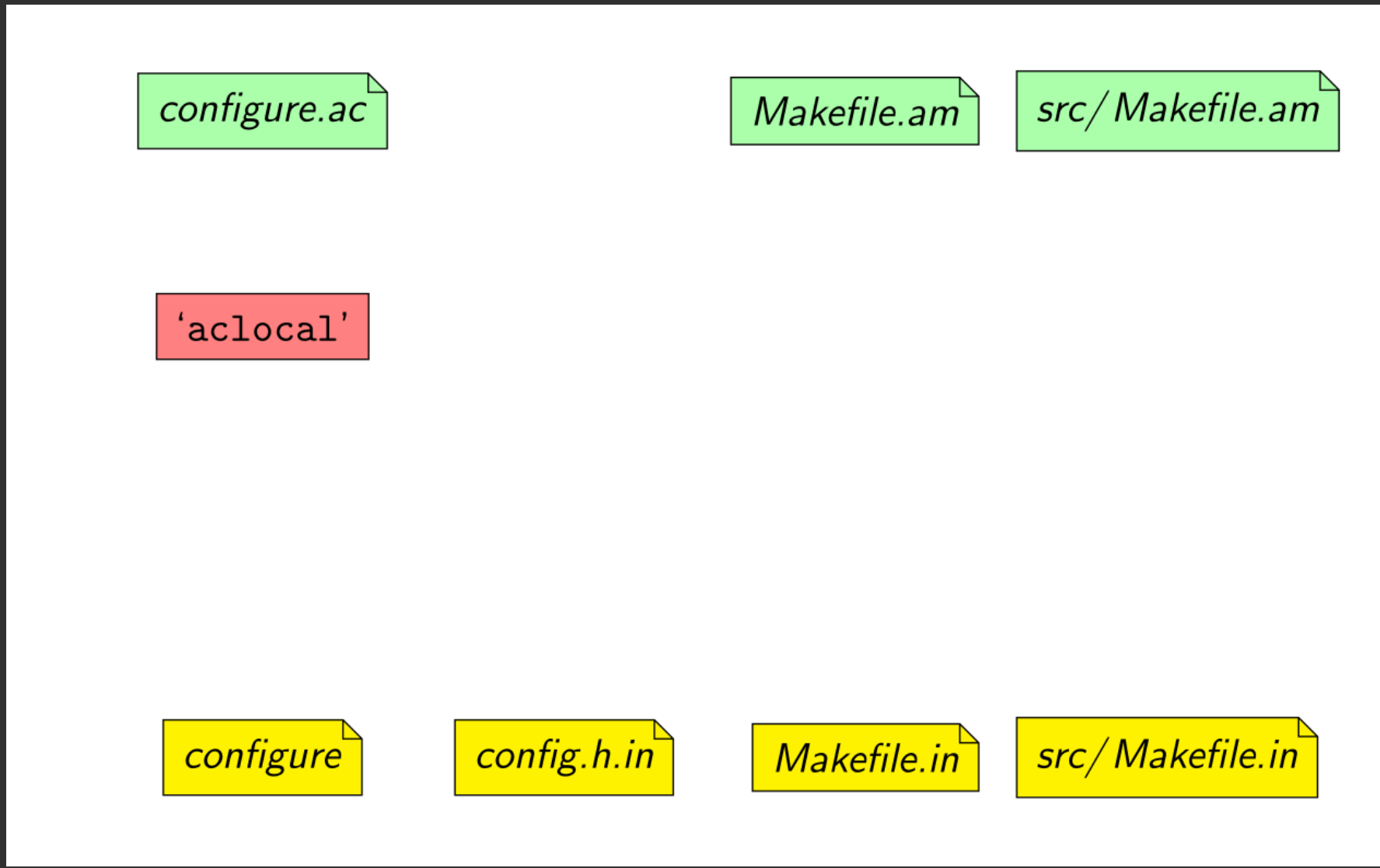

Home
GNU-Build-System
Recent
Starred
Home
Trash
Other Locations
Name
Size
configure.ac
208 bytes
Makefile.am
14 bytes
src
2 items
main.c
99 bytes
Makefile.am
62 bytes

autoreconf



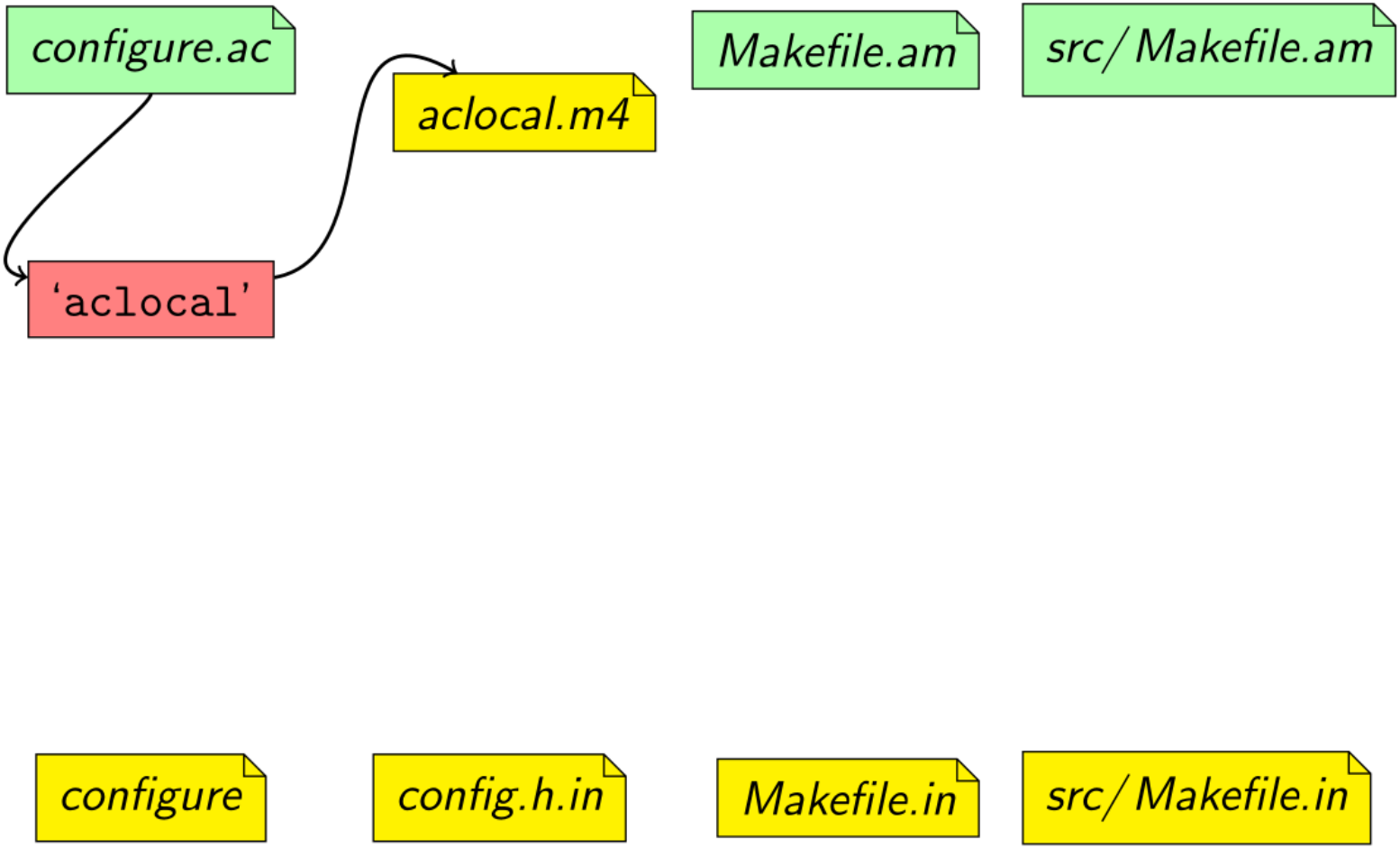
A. Duret-Lutz (2010). *Using GNU Autotools*. `https://www.lrde.epita.fr/~adl/dl/autotools.pdf`

autoreconf



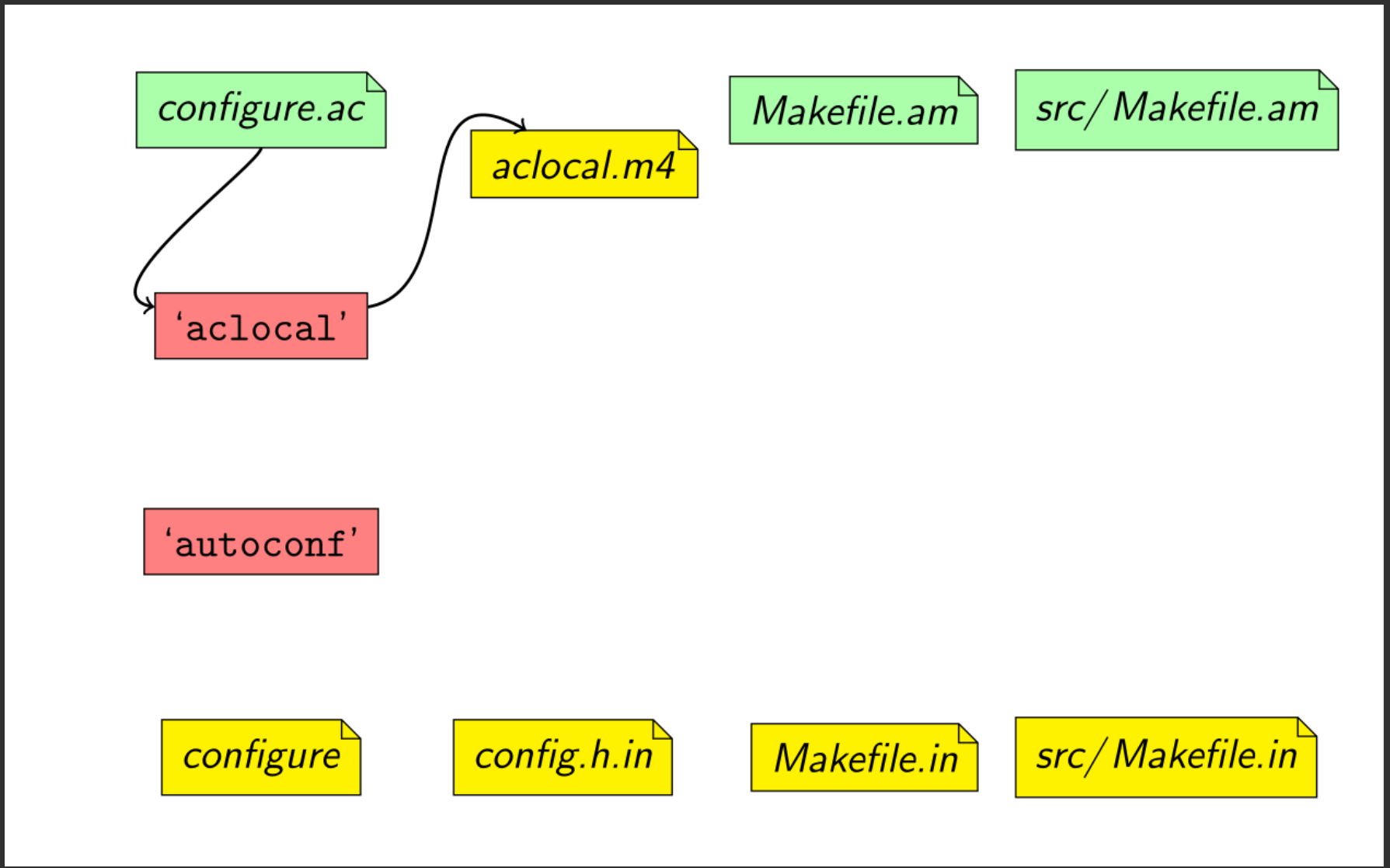
A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

autoreconf



A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

autoreconf



A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

autoconf

- `autoconf` ist ein Makroprozessor

autoconf

- `autoconf` ist ein Makroprozessor
- Es konvertiert `configure.ac`, ein Shell-Skript mit Makro Anweisungen (m4), in ein vollwertiges Shell-Skript

autoconf

- `autoconf` ist ein Makroprozessor
- Es konvertiert `configure.ac`, ein Shell-Skript mit Makro Anweisungen (m4), in ein vollwertiges Shell-Skript
- `autoconf` bietet viele Makros für das Prüfen von Konfigurationen

autoconf

- `autoconf` ist ein Makroprozessor
- Es konvertiert `configure.ac`, ein Shell-Skript mit Makro Anweisungen (m4), in ein vollwertiges Shell-Skript
- `autoconf` bietet viele Makros für das Prüfen von Konfigurationen
- Es ist nicht ungewöhnlich, dass in `configure.ac` nur Makros und keine Shell-Konstrukte vorhanden sind

autoconf

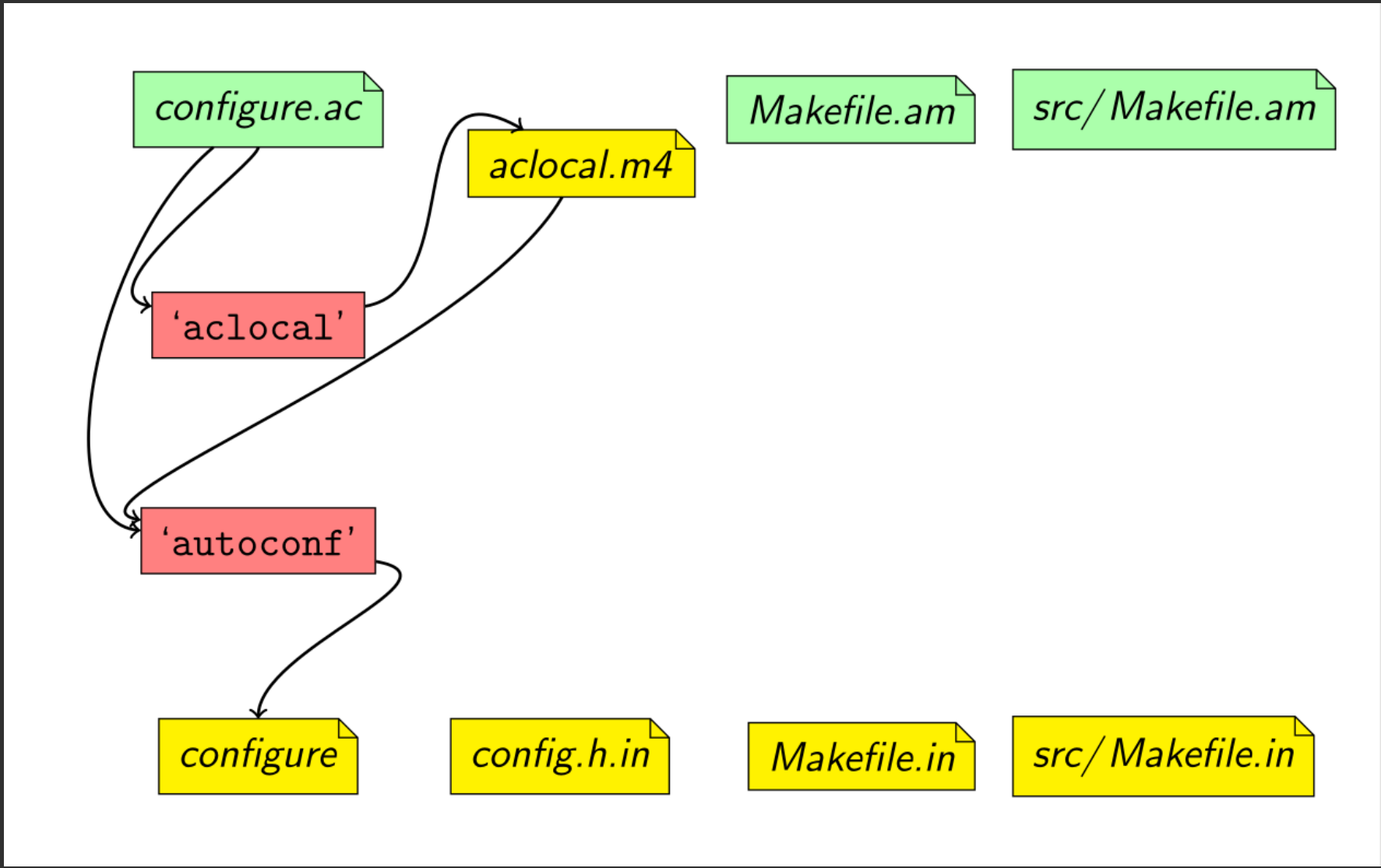
- `autoconf` ist ein Makroprozessor
- Es konvertiert `configure.ac`, ein Shell-Skript mit Makro Anweisungen (m4), in ein vollwertiges Shell-Skript
- `autoconf` bietet viele Makros für das Prüfen von Konfigurationen
- Es ist nicht ungewöhnlich, dass in `configure.ac` nur Makros und keine Shell-Konstrukte vorhanden sind
- Der eigentliche Makroprozessor ist nicht `autoconf`, sondern GNU M4

autoconf

- `autoconf` ist ein Makroprozessor
- Es konvertiert `configure.ac`, ein Shell-Skript mit Makro Anweisungen (m4), in ein vollwertiges Shell-Skript
- `autoconf` bietet viele Makros für das Prüfen von Konfigurationen
- Es ist nicht ungewöhnlich, dass in `configure.ac` nur Makros und keine Shell-Konstrukte vorhanden sind
- Der eigentliche Makroprozessor ist nicht `autoconf`, sondern GNU M4

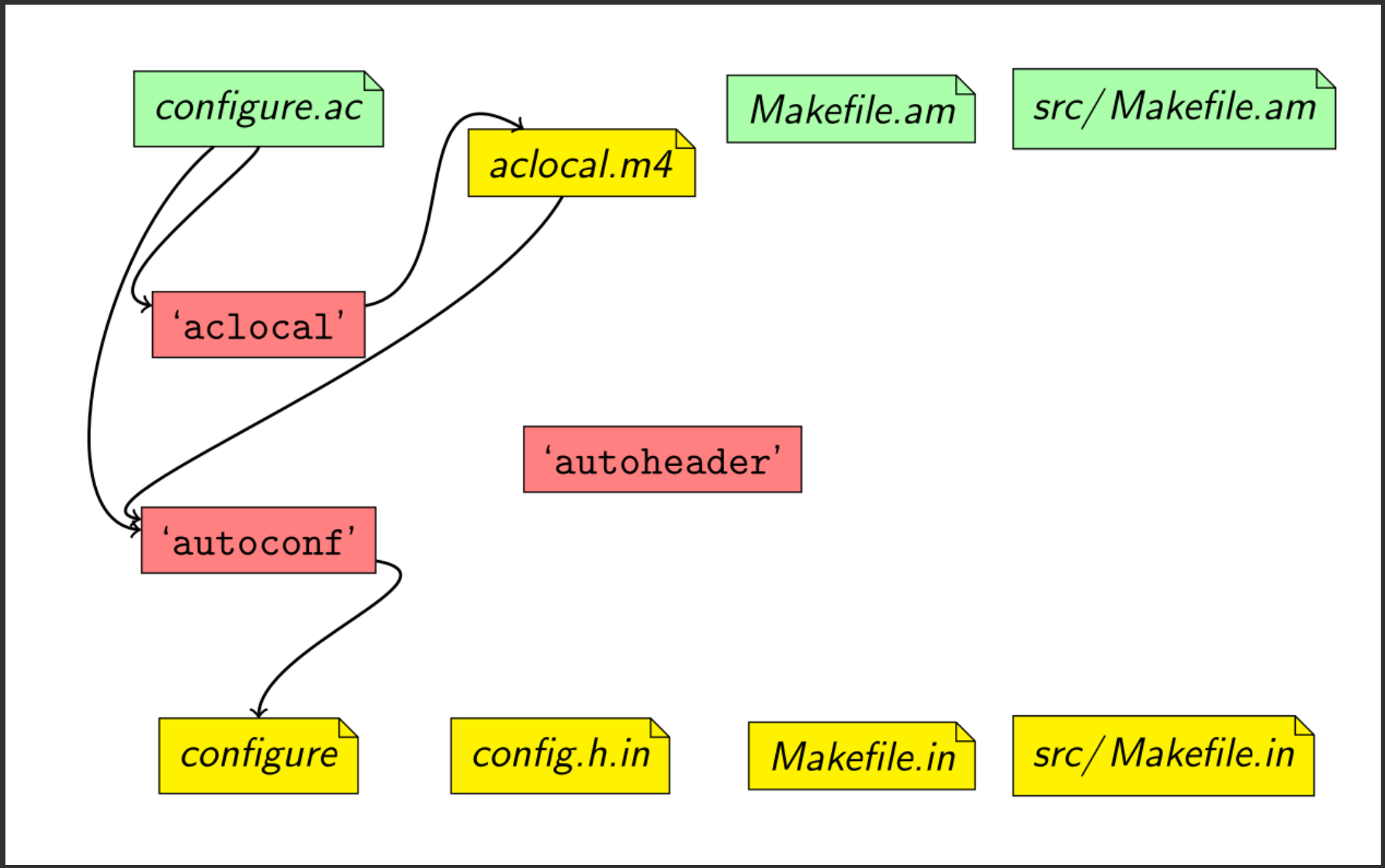
  `autoconf` bietet aufbauend auf M4 ein wenig Infrastruktur, sowie einen Pool an Makros

autoreconf



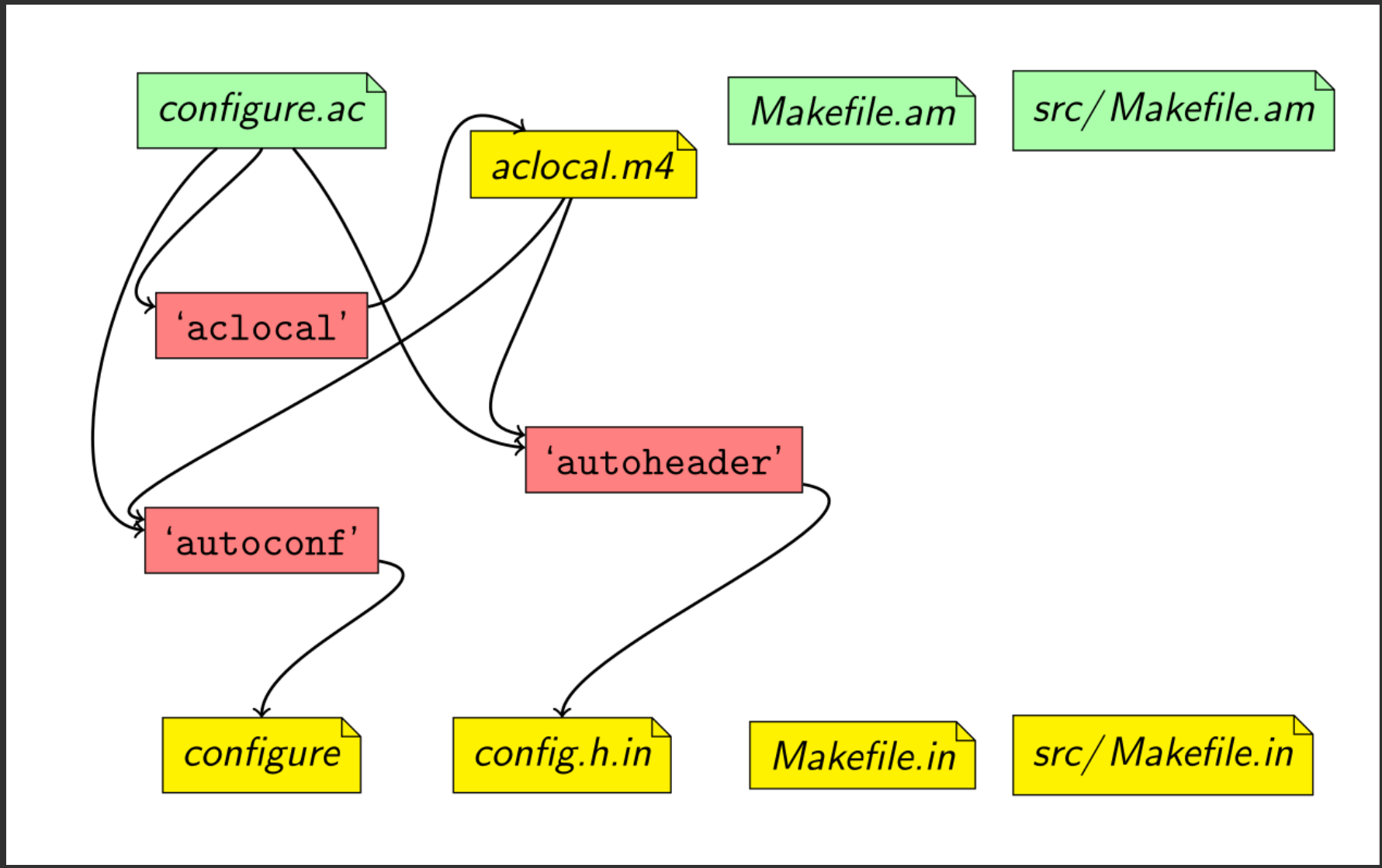
A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

autoreconf



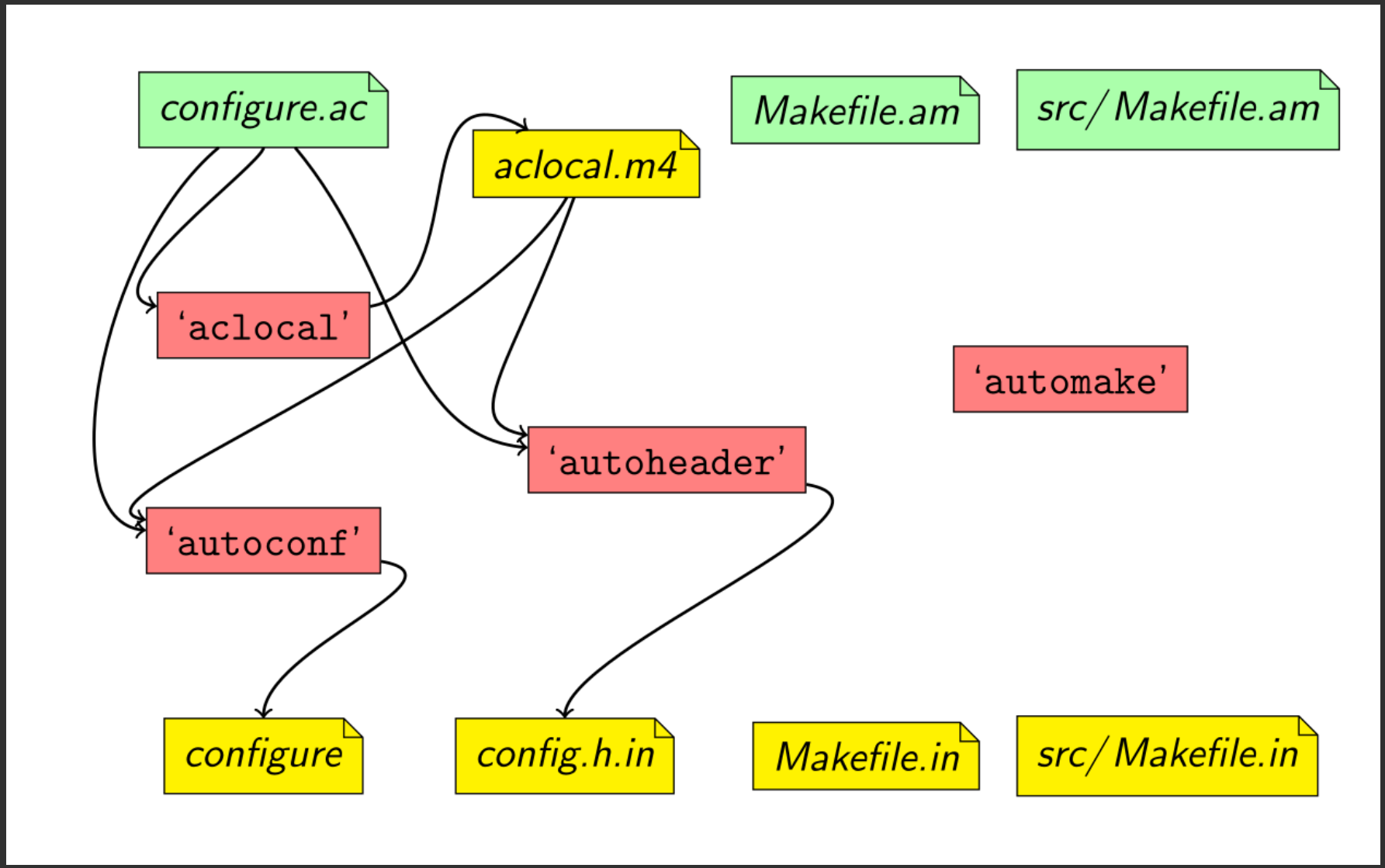
A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

autoreconf



A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

autoreconf



A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

automake

- `automake` hilft bei der Erstellung von portablen und GNU-Standard konformen Makefiles

automake

- `automake` hilft bei der Erstellung von portablen und GNU-Standard konformen Makefiles
- `automake` erstellt komplexe `Makefile.in` aus einfachen `Makefile.am` Dateien

automake

- `automake` hilft bei der Erstellung von portablen und GNU-Standard konformen Makefiles
- `automake` erstellt komplexe `Makefile.in` aus einfachen `Makefile.am` Dateien
- Die Syntax von `Makefile.am` ist der Syntax von Makefiles sehr ähnlich

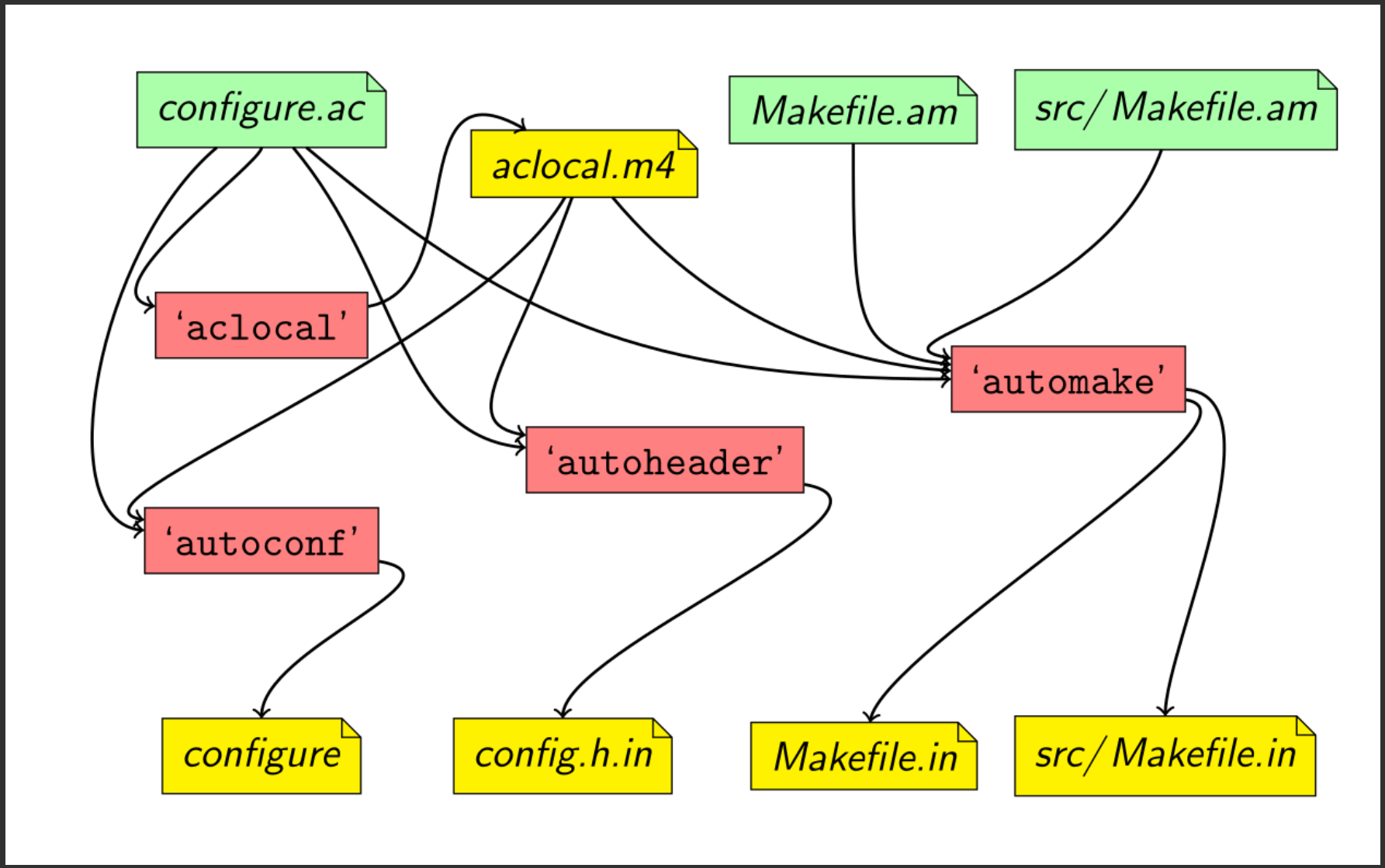
automake

- `automake` hilft bei der Erstellung von portablen und GNU-Standard konformen Makefiles
- `automake` erstellt komplexe `Makefile.in` aus einfachen `Makefile.am` Dateien
- Die Syntax von `Makefile.am` ist der Syntax von Makefiles sehr ähnlich
- `Makefile.am` enthält normalerweise nur Variablendefinitionen, welche von `automake` dann zu Ableitungsregeln konvertiert werden

automake

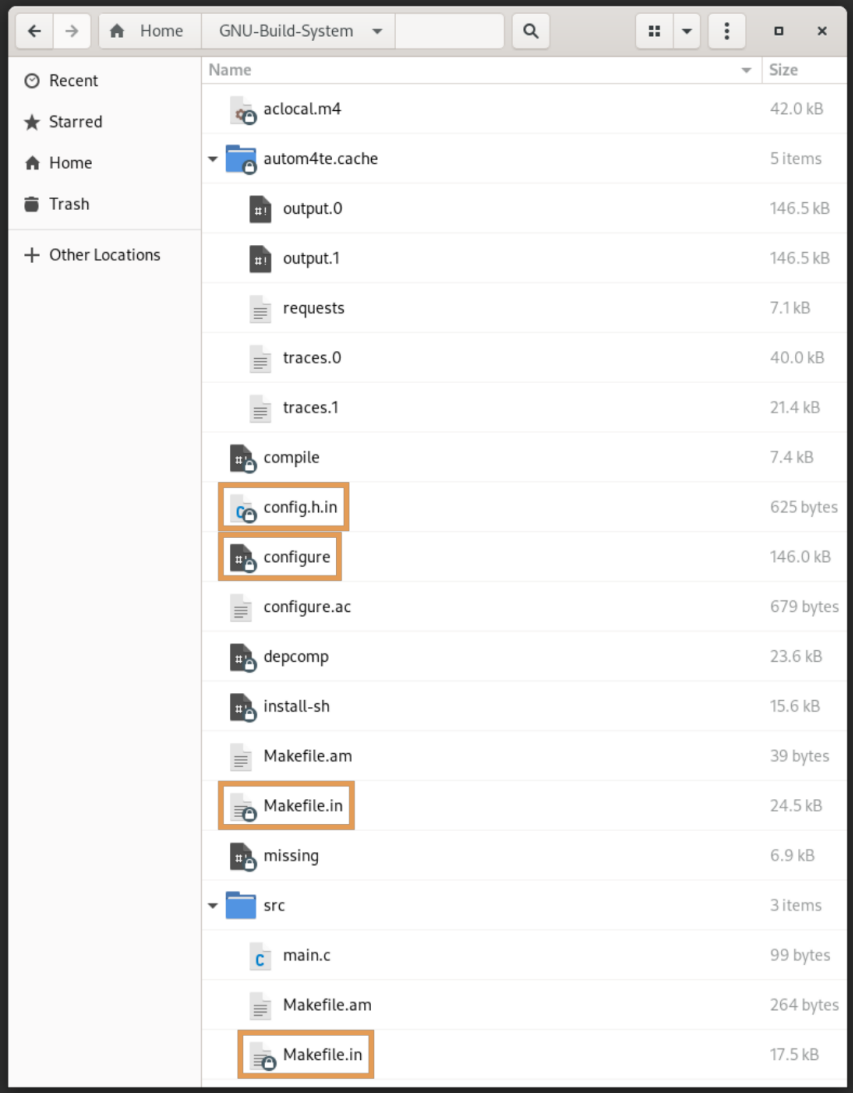
- `automake` hilft bei der Erstellung von portablen und GNU-Standard konformen Makefiles
- `automake` erstellt komplexe `Makefile.in` aus einfachen `Makefile.am` Dateien
- Die Syntax von `Makefile.am` ist der Syntax von Makefiles sehr ähnlich
- `Makefile.am` enthält normalerweise nur Variablendefinitionen, welche von `automake` dann zu Ableitungsregeln konvertiert werden
- `Makefile.am` kann explizite *rules* in Makefile Syntax enthalten, da `automake` diese in `Makefile.in` übernimmt

autoreconf



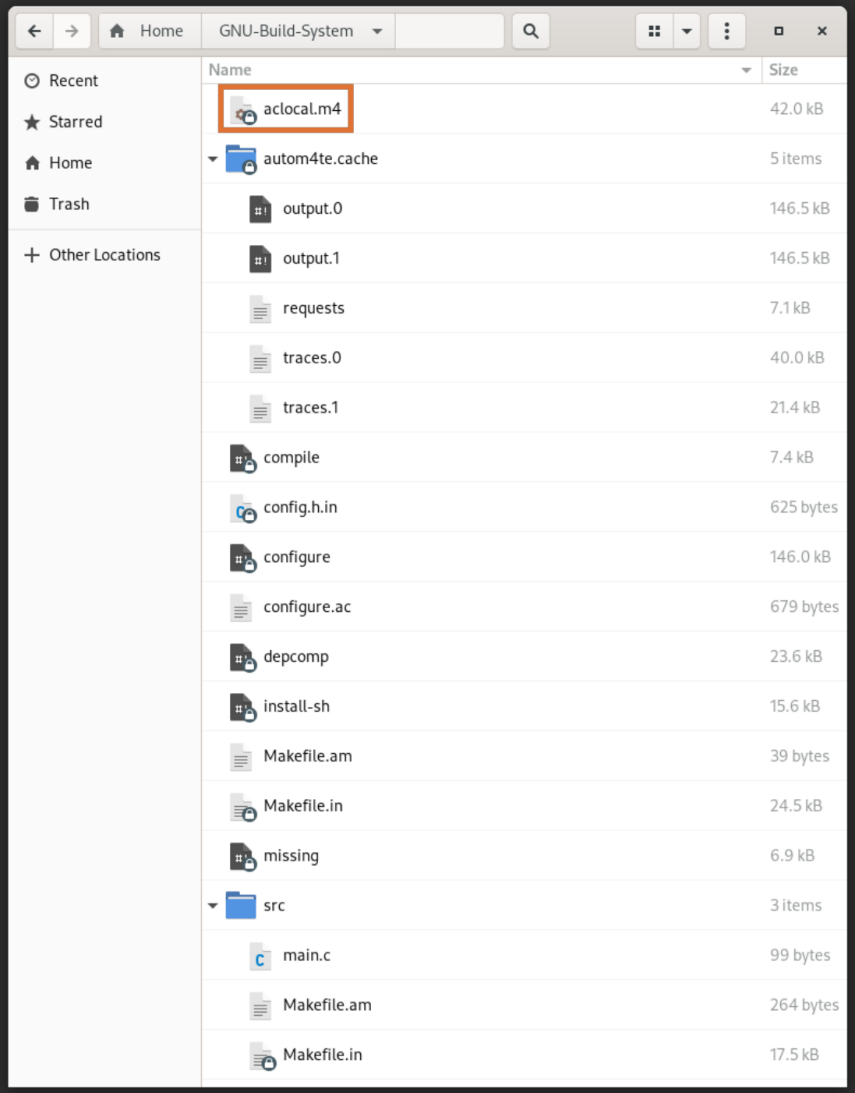
A. Duret-Lutz (2010). *Using GNU Autotools*. https://www.lrde.epita.fr/~adl/dl/autotools.pdf

Projekstruktur



Das *configure* Skript und Konfigurationsvorlagen für dieses.

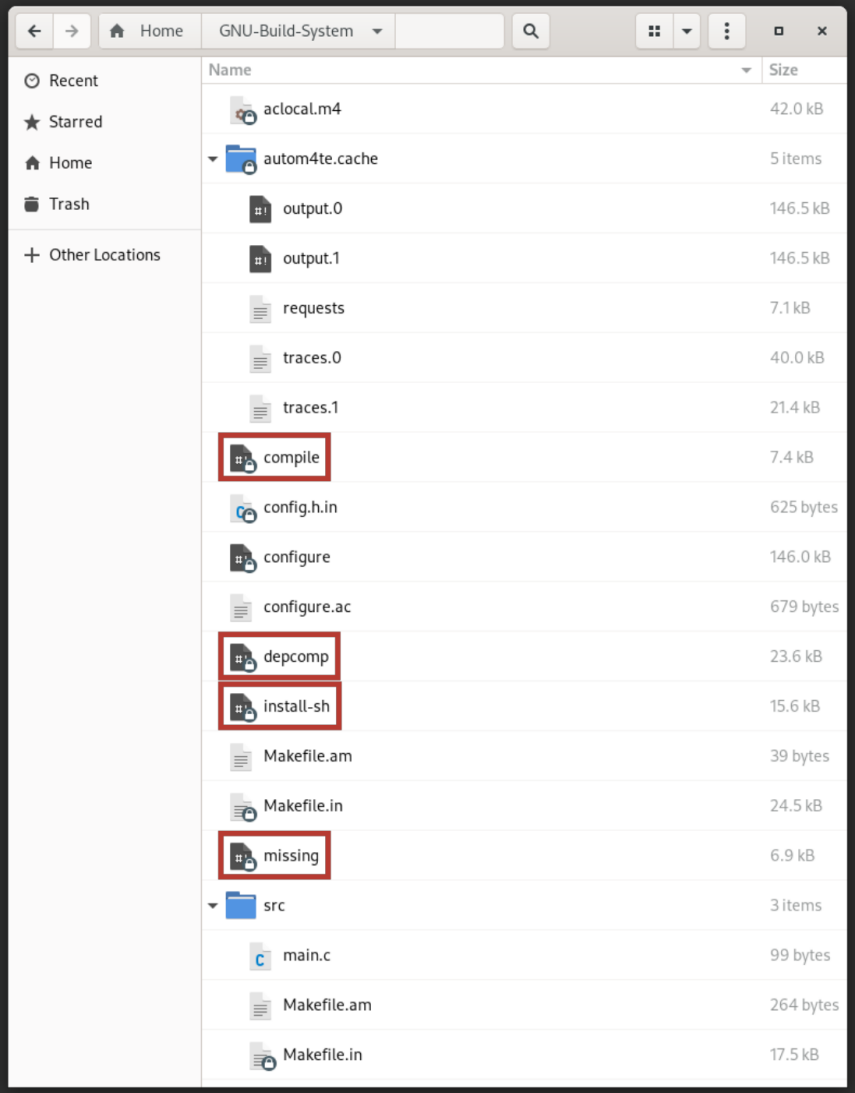
Projektstruktur



Das *configure* Skript und Konfigurationsvorlagen für dieses.

Makrodefinitionen von Drittanbietern welche in *configure.ac* verwendet werden können.

Projektstruktur

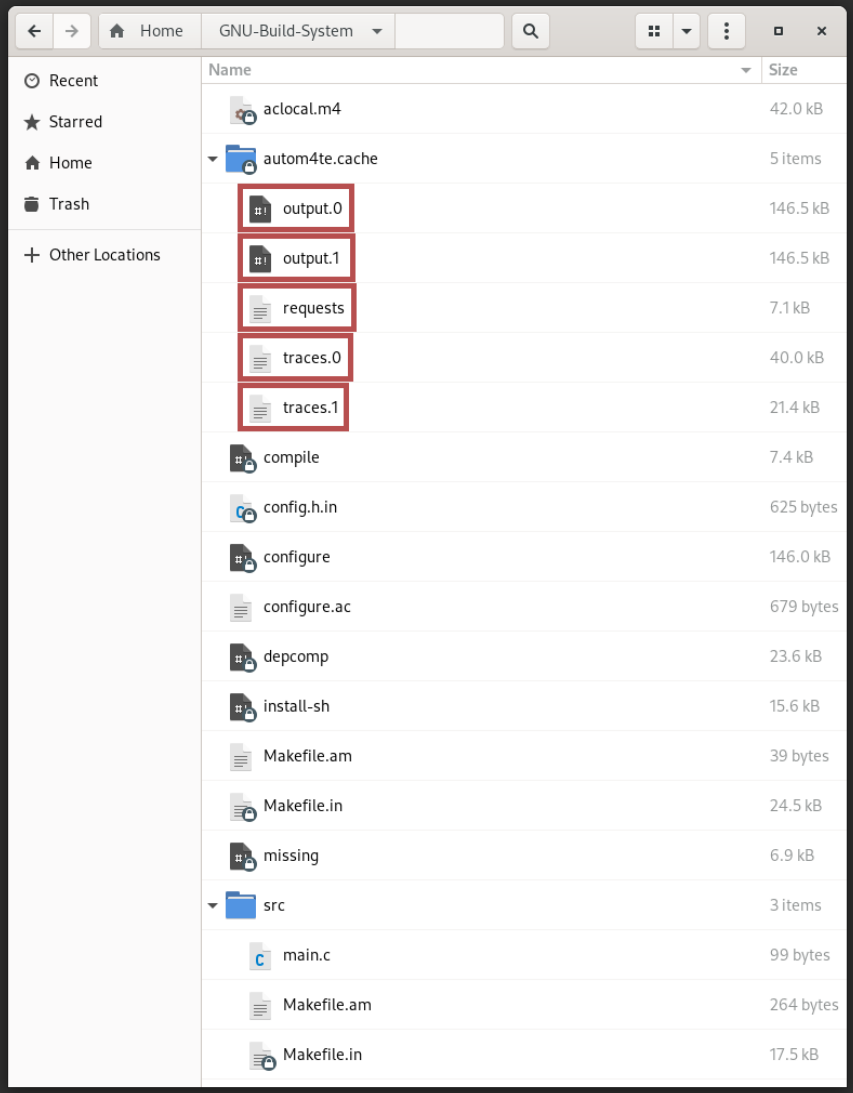


Das *configure* Skript und Konfigurationsvorlagen für dieses.

Makrodefinitionen von Drittanbietern welche in *configure.ac* verwendet werden können.

Hilfswerkzeuge die während des Builds Prozesses verwendet weden.

Projektstruktur



Das *configure* Skript und Konfigurationsvorlagen für dieses.

Makrodefinitionen von Drittanbietern welche in *configure.ac* verwendet werden können.

Hilfswerkzeuge die während des Builds Prozesses verwendet weden.

Cache Dateien von Autotools.

CMake

Beschreibung

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. [1]

Eigenschaften

- CMake generiert Build-System Dateien für viele verschiedene Build-Systeme wie z.B.:
  - Microsoft Visual Studio (Windows)
  - Xcode (macOS)
  - Ninja (Linux, macOS, Windows)
  - Make (Linux, macOS)

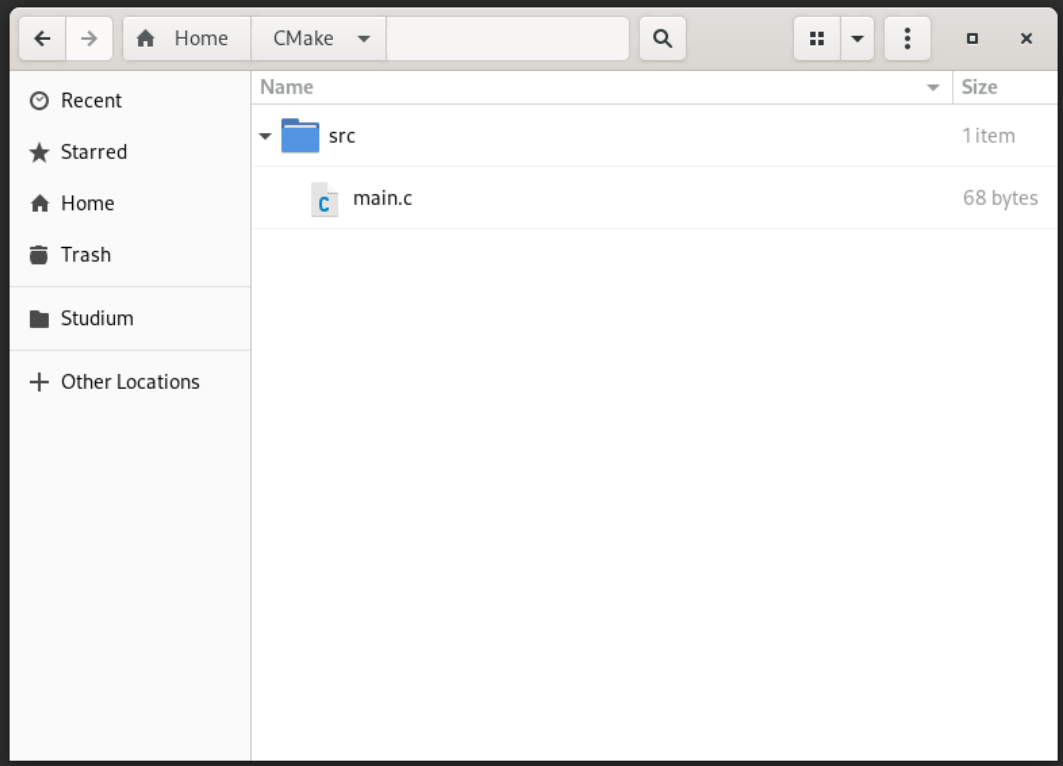
Eigenschaften

- CMake generiert Build-System Dateien für viele verschiedene Build-Systeme wie z.B.:
  - Microsoft Visual Studio (Windows)
  - Xcode (macOS)
  - Ninja (Linux, macOS, Windows)
  - Make (Linux, macOS)
- CMake verwendet Compiler unabhängige Konfigurationsdateien (`CMakeLists.txt`), welche in der CMake Sprache verfasst sind

Eigenschaften

- CMake generiert Build-System Dateien für viele verschiedene Build-Systeme wie z.B.:
  - Microsoft Visual Studio (Windows)
  - Xcode (macOS)
  - Ninja (Linux, macOS, Windows)
  - Make (Linux, macOS)
- CMake verwendet Compiler unabhängige Konfigurationsdateien (`CMakeLists.txt`), welche in der CMake Sprache verfasst sind
- CMake ermöglicht den Build-Prozess, das Testen und das Verpacken von Software

Projektstruktur



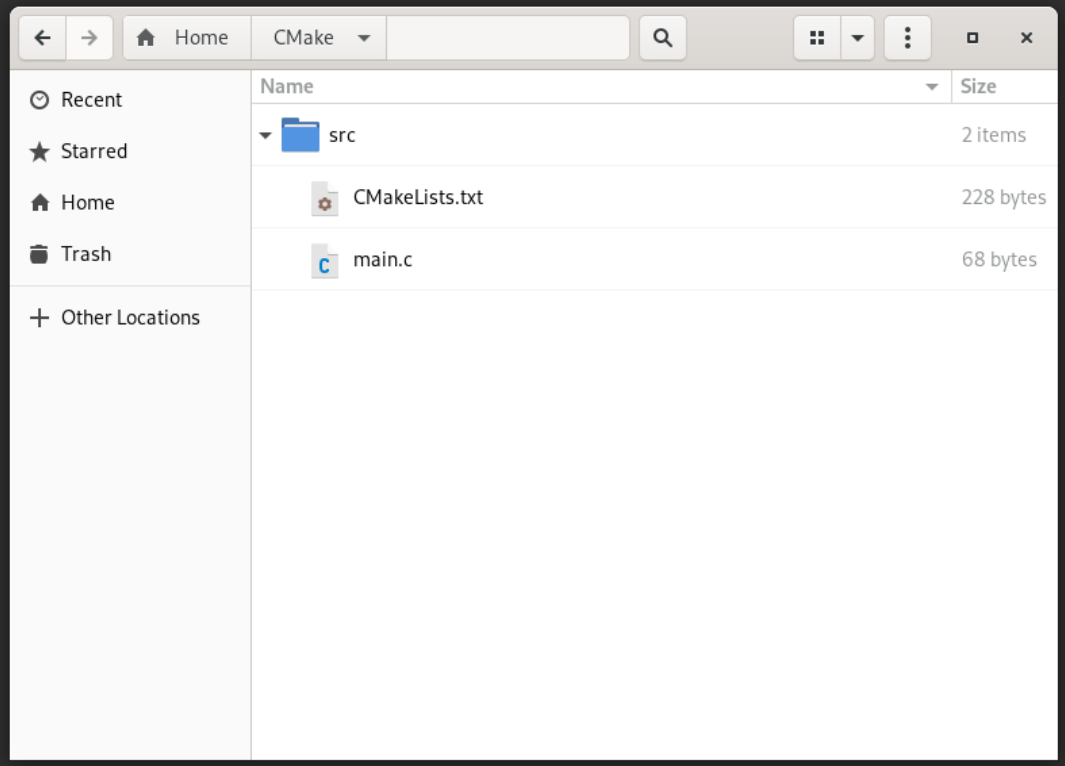
main.c

```c
#include <stdio.h>

int main(void)
{
    printf("CMake");
    return 0;
}
```

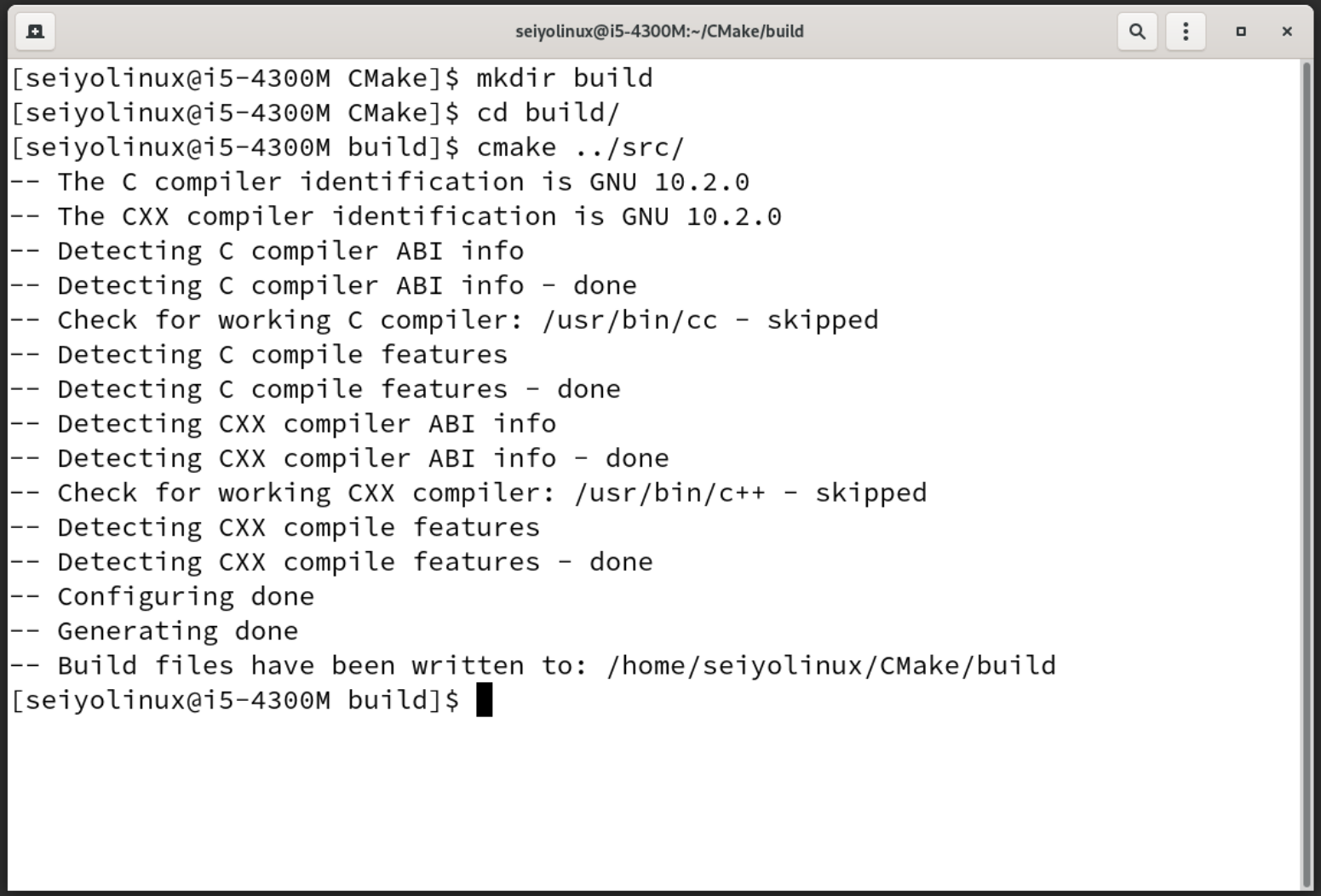
CMakeLists.txt

```
# Minimal erforderliche CMake Version.
cmake_minimum_required(VERSION 3.19.1)

# Definiere den Projektnamen und die Projektversion.
project(CMake VERSION 1.0)

# Deklariere eine ausführbare Datei.
add_executable(CMake main.c)
```

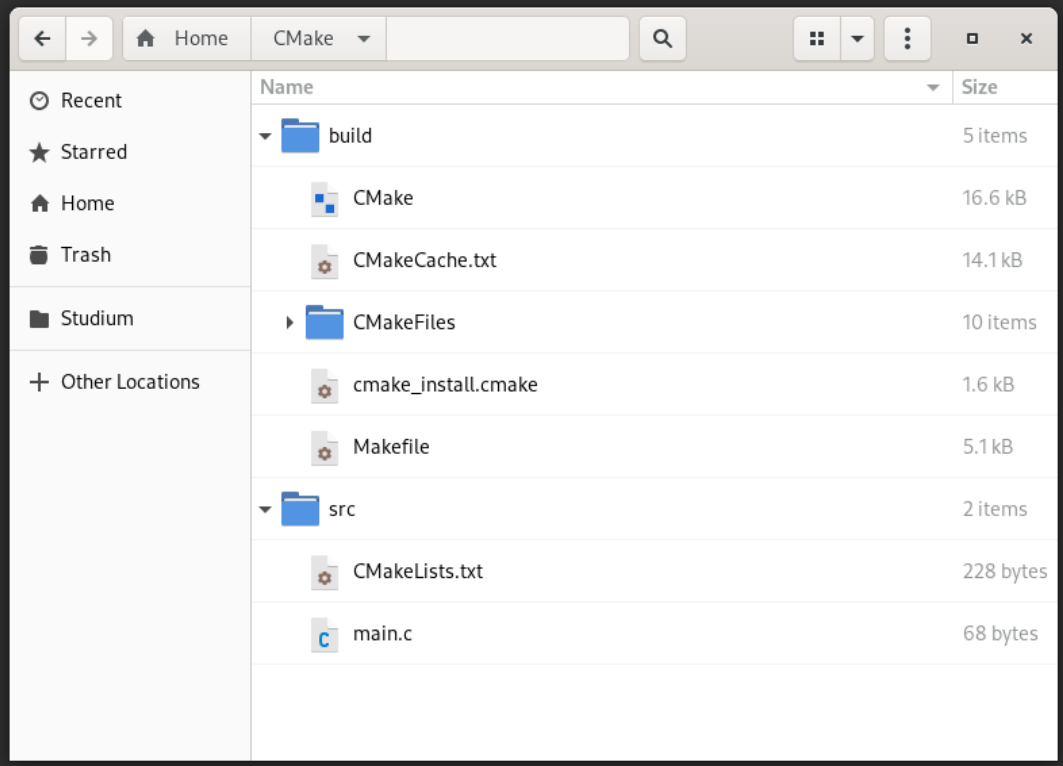
Projektstruktur



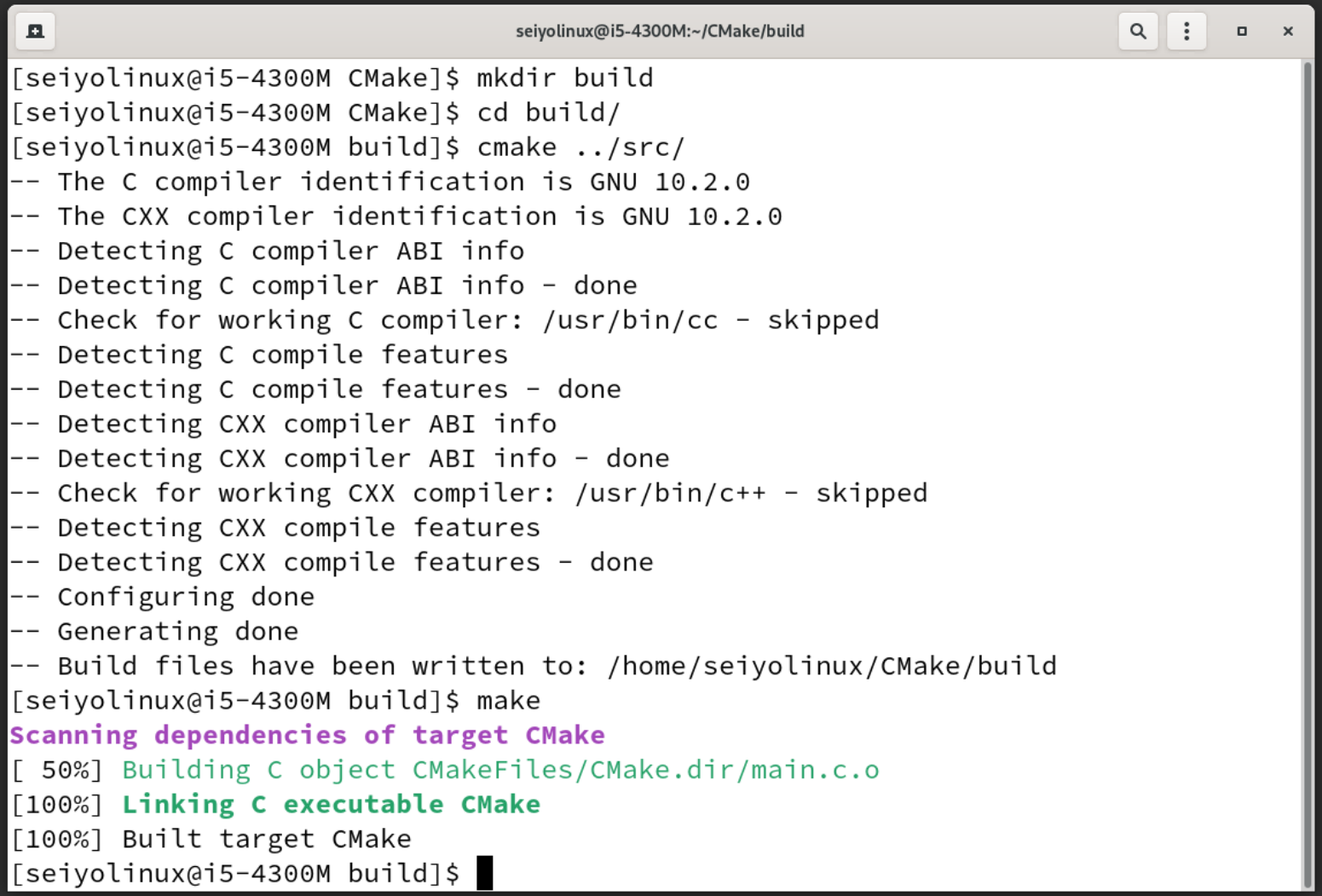
cmake


seiyolinux@i5-4300M:~/CMake/build
[seiyolinux@i5-4300M CMake]$ mkdir build
[seiyolinux@i5-4300M CMake]$ cd build/
[seiyolinux@i5-4300M build]$ cmake ../src/
-- The C compiler identification is GNU 10.2.0
-- The CXX compiler identification is GNU 10.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/seiyolinux/CMake/build
[seiyolinux@i5-4300M build]$

Projektstruktur



make


seiyolinux@i5-4300M:~/CMake/build
[seiyolinux@i5-4300M CMake]$ mkdir build
[seiyolinux@i5-4300M CMake]$ cd build/
[seiyolinux@i5-4300M build]$ cmake ../src/
-- The C compiler identification is GNU 10.2.0
-- The CXX compiler identification is GNU 10.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/seiyolinux/CMake/build
[seiyolinux@i5-4300M build]$ make
Scanning dependencies of target CMake
[50%] Building C object CMakeFiles/CMake.dir/main.c.o
[100%] Linking C executable CMake
[100%] Built target CMake
[seiyolinux@i5-4300M build]$

Zusammenfassung

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien
- Makefiles enthalten Shell als auch Makefile Syntax

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien
- Makefiles enthalten Shell als auch Makefile Syntax
- Shell-Code wird ausgeführt um (Programm)Dateien aus Sourcecode abzuleiten

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien
- Makefiles enthalten Shell als auch Makefile Syntax
- Shell-Code wird ausgeführt um (Programm)Dateien aus Sourcecode abzuleiten
- Makefiles bestehen aus 5 Komponenten: Kommentaren, Variablen, Direktiven, *explicite rules* und *implicite rules*

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien
- Makefiles enthalten Shell als auch Makefile Syntax
- Shell-Code wird ausgeführt um (Programm)Dateien aus Sourcecode abzuleiten
- Makefiles bestehen aus 5 Komponenten: Kommentaren, Variablen, Direktiven, *explicite rules* und *implicite rules*
- Die Autotools helfen bei der Generierung eine GNU-Build-Systems

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien
- Makefiles enthalten Shell als auch Makefile Syntax
- Shell-Code wird ausgeführt um (Programm)Dateien aus Sourcecode abzuleiten
- Makefiles bestehen aus 5 Komponenten: Kommentaren, Variablen, Direktiven, *explicite rules* und *implicite rules*
- Die Autotools helfen bei der Generierung eine GNU-Build-Systems
- Das `configure` Skript konfiguriert ein Makefile, abhängig von der Plattform des Benutzers

Zusammenfassung

- Make leitet Dateien aus Sourcecode mithilfe von Makefiles ab
- Makefiles definieren Ableitungsregeln und Abhängigkeiten zwischen Dateien
- Makefiles enthalten Shell als auch Makefile Syntax
- Shell-Code wird ausgeführt um (Programm)Dateien aus Sourcecode abzuleiten
- Makefiles bestehen aus 5 Komponenten: Kommentaren, Variablen, Direktiven, *explicite rules* und *implicite rules*
- Die Autotools helfen bei der Generierung eine GNU-Build-Systems
- Das `configure` Skript konfiguriert ein Makefile, abhängig von der Plattform des Benutzers
- CMake generiert Build-System Dateien für viele verschiedene Build-Systeme

Literatur Verzeichnis

Literatur Verzeichnis I

1. *CMake [Stand 11.12.2020]*, https://cmake.org/
2. *Build automation - Wikipedia [Stand 03.12.2020]*, https://en.wikipedia.org/wiki/Build_automation
3. *Make (software) - Wikipedia [Stand 03.12.2020]*, https://en.wikipedia.org/wiki/Make_(software)
4. *Make - GNU Project - Free Software Foundation [Stand 05.12.2020]*, https://www.gnu.org/software/make/
5. *GNU make [Stand 05.12.2020]*, https://www.gnu.org/software/make/manual/make.html
6. *GNU Autotools - Wikipedia [Stand 08.12.2020]*, hhttps://en.wikipedia.org/wiki/GNU_Autotools
7. A. Duret-Lutz, *Using GNU Autotools [Stand 08.12.2020]*, https://www.lrde.epita.fr/~adl/dl/autotools.pd.

Literatur Verzeichnis II

8. *Autoconf [Stand 08.12.2020]*, https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.70/autoconf.html
9. *automake [Stand 08.12.2020]*, https://www.gnu.org/savannah-checkouts/gnu/automake/manual/automake.html
10. *CMake Tutorial - CMake 3.19.2 Documentation [Stand 11.12.2020]*, https://cmake.org/cmake/help/latest/guide/tutorial/index.html
11. F. Castelli, *Florent Castelli: Introduction to CMake*, https://www.youtube.com/watch?v=jt3meXdP-QI