COMP0104 Software Development Practice: Building programs with MAKE

Jens Krinke

Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

Building programs with MAKE
Overview

- Building programs out of components
- MAKE

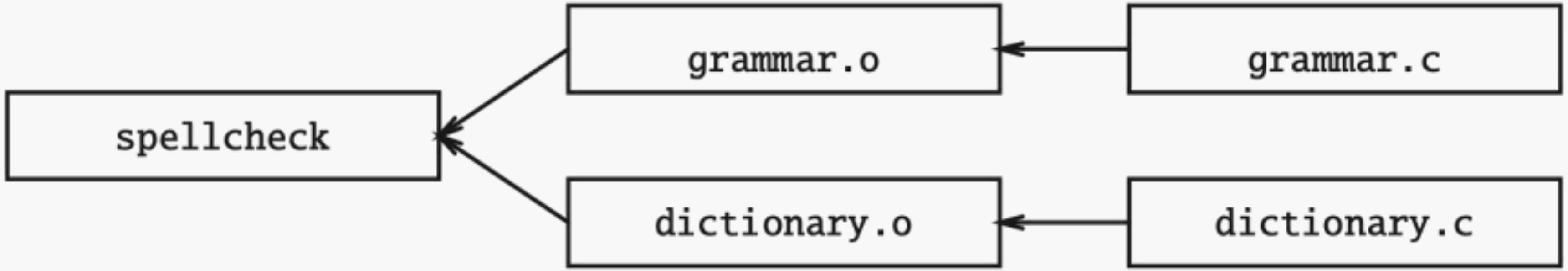
Building programs out of components

- How do I build the system?
- How do I build the system after a change?

Components in a system

- Components that have been created manually are **source components**.
- Components that can be created automatically are **derived components**.
- Component ***A*** is derived from a component ***B***, if ***A*** refers to ***B*** in some way.
- If ***A*** is derived from ***B***, then ***A*** depends on ***B***.

A dependency graph



A script to build the system

- `cc -c -o grammar.o grammar.c`
- `cc -c -o dictionary.o dictionary.c`
- `cc -o spellcheck grammar.o dictionary.o`

Problem:
all construction steps are always executed

A change in component *A* can only impact the components that are derived from *A.*

Incremental build

Let A be a derived component,
dependent on n other components $A_1, A_2, \ldots, A_n$.

The component A must be rebuilt, if

1. Component A does not exist, or
2. at least one A_i from $A_1, A_2, \ldots, A_n$ has changed, or
3. at least one A_i from $A_1, A_2, \ldots, A_n$ must be rebuilt.

MAKE

- A tool to build systems incrementally
- Developed 1975 by Stuart Feldman (Bell Labs)
- One of the most influential software tools

Incremental construction: System Model

A system model is a description
of the software product that

- lists the individual **components**
- together with their **dependencies**
- and the steps required for their construction

The system model of MAKE is a **Makefile**.

Rules in a Makefile

- A **rule** indicates on which components one or more components are dependent,
- as well as the commands (the recipe) that are necessary to build the component(s).

target$_1$ … target$_n$:   source$_1$ … source$_m$

        command$_1$

        command$_2$

        command$_3$

        …

Makefile (Example)

```
spellcheck: grammar.o dictionary.o
	cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
	cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
	cc -c -o dictionary.o dictionary.c
```

How MAKE works

- MAKE takes a Makefile as input, together with a target A_0 of this Makefile.
- MAKE calculates the dependency graph and begins its work with the component A_0.
- The basic algorithm is a depth-first search into the dependency graph, in which the construction steps are executed after every return, if necessary.

MAKE algorithm

- Suppose A is the current target component.
  - From the dependency graph, determine the components A_1, A_2, . . . ,A_n on which A depends.
  - If A does not occur as a target in the Makefile, $n = 0$.
- Call the algorithm recursively for each A_i from $A_1,A_2,...,A_n$.
- If one of the components $A_1,...,A_n$ has changed, or if A does not exist, A must be rebuilt. For this purpose, the commands (the recipe) associated with A as a target are executed.

Example

```
spellcheck: grammar.o dictionary.o
   cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
   cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
   cc -c -o dictionary.o dictionary.c
```

- `spellcheck` has already been created once
- We now have changed `dictionary.c`.
- What happens during the invocation of MAKE with the target `spellcheck`?

Solution (1/5)

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    cc -c -o dictionary.o dictionary.c
```

- First `spellcheck` is the current target, dependent on `grammar.o` and `dictionary.o`.
- Now the target is `grammar.o`; it is dependent on `grammar.c`.
- Now the target is `grammar.c`; it is not a target in the Makefile, but it does exist.

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    cc -c -o dictionary.o dictionary.c
```

Solution (2/5)

- Back at `grammar.o`,
  since `grammar.c` has not been changed, `grammar.o` does not have to be rebuilt.
- Back at `spellcheck`,
  the next target is `dictionary.o`.

Solution (3/5)

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    cc -c -o dictionary.o dictionary.c
```

- Now the target `dictionary.o` is checked;
  it is dependent on the source file `dictionary.c`.
- Now the target is `dictionary.c`;
  it is also not a target in the Makefile,
  but it does exist.

Solution (4/5)

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    cc -c -o dictionary.o dictionary.c
```

- Back at the target `dictionary.o`, MAKE determines that `dictionary.c` has changed: `dictionary.o` has to be rebuilt.
- MAKE calls the command `cc -c -o dictionary.o dictionary.c`.

Solution (5/5)

```
spellcheck: grammar.o dictionary.o
    cc -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    cc -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    cc -c -o dictionary.o dictionary.c
```

- We are back at the target `spellcheck` again; since `dictionary.o` has now been changed, `spellcheck` also has to be recreated;
- MAKE calls the command `cc -o spellcheck grammar.o dictionary.o`.
- With this, the target `spellcheck` is recreated; MAKE is finished.

In practice:

```
$ make spellcheck
cc -c -o dictionary.o dictionary.c
cc -o spellcheck grammar.o dictionary.o
$ _
```

How does MAKE actually establish that a file has been changed?

- Every file records the time of the last change.
- If a component *C* (as a file) shows a later (more recent) date than a component that depends on it,
  C is considered to have been “changed”.

How does MAKE actually establish that a file has been changed?

- After a successful compilation, every source component is older than its derived components.
- A second call of MAKE does nothing.

```
$ make spellcheck
make: nothing to be done for ‘spellcheck’.
$
```

Variables

- Variables are defined in a Makefile using
  variable = *value*
- Each further occurrence of $(*variable*) or ${*variable*} is then replaced automatically by *value*.

Variables: Example

Variable `CC` usually contains the C compiler.

```
CC = cc

spellcheck: grammar.o dictionary.o
    $(CC) -o spellcheck grammar.o dictionary.o

grammar.o: grammar.c
    $(CC) -c -o grammar.o grammar.c

dictionary.o: dictionary.c
    $(CC) -c -o dictionary.o dictionary.c
```

Overriding variables

At the invocation of MAKE,
new values for the variables can be specified.

Example:

```
$ make CC=gcc spellcheck
```

Variables can be used anywhere

The variable OBJECTS defines a list of objects, so that there is only one list.

```
OBJECTS = grammar.o dictionary.o

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

…

Suffix Rules

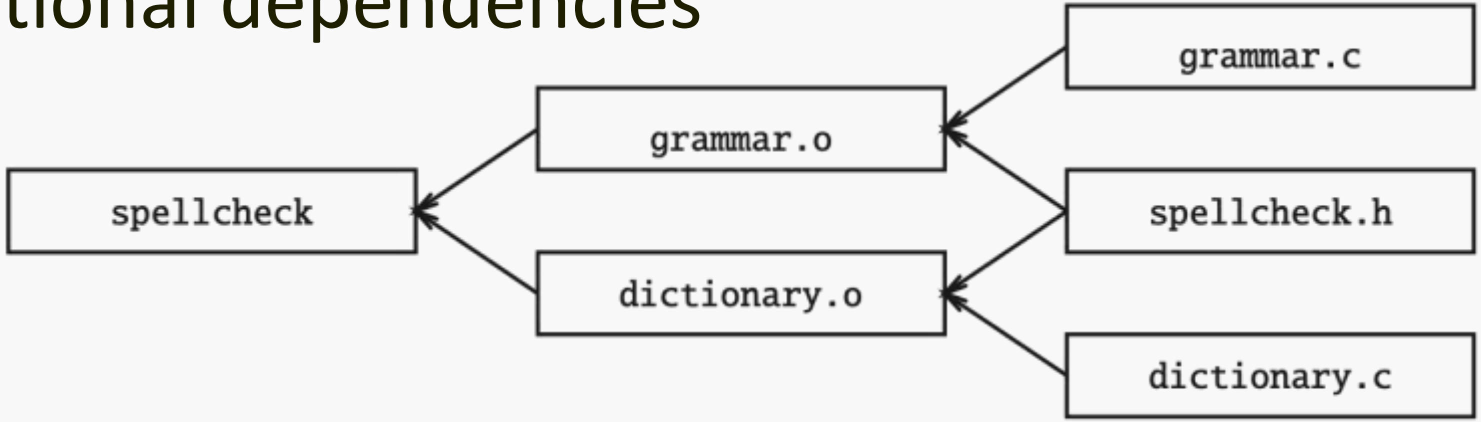
- In order to combine similar rules, MAKE offers **suffix rules**.
- A suffix rule has the form
  `.`*suffix*$_1$`.`*suffix*$_2$`:`
  command
- Typical example:

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

Makefile with suffix rules (example)

```
CC = cc

OBJECTS = grammar.o dictionary.o

.SUFFIXES: .c .o

spellcheck: $(OBJECTS)
    $(CC) -o $@ $(OBJECTS)

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

Additional dependencies



Additional dependencies can not be added to suffix rules and must be stated explicitly:

```
grammar.o: grammar.c spellcheck.h
dictionary.o: dictionary.c spellcheck.h
```

Makefile with suffix rules and additional dependencies

```
CC = cc

OBJECTS = grammar.o dictionary.o

.SUFFIXES: .c .o

spellcheck: $(OBJECTS)
	$(CC) -o $@ $(OBJECTS)

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

grammar.o: spellcheck.h
dictionary.o: spellcheck.h
```

MAKE has predefined rules and variables (implicit rules and implicit variables)

```
CC = cc

OBJECTS = grammar.o dictionary.o

.SUFFIXES: .c .o

spellcheck: $(OBJECTS)
   $(CC) -o $@ $(OBJECTS)

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

grammar.o: spellcheck.h
dictionary.o: spellcheck.h
```

MAKE has predefined rules and variables (implicit rules and implicit variables)

```
OBJECTS = grammar.o dictionary.o


spellcheck: $(OBJECTS)
   $(CC) -o $@ $(OBJECTS)


grammar.o: spellcheck.h
dictionary.o: spellcheck.h
```

Pattern rules in GNU MAKE

- “%” stands for an arbitrary character string.

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

- Pattern rules have the advantage that additional dependencies can be accommodated.

```
%.o: %.c defs.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Makefile with pattern rules

```
CC = cc

OBJECTS = grammar.o dictionary.o

spellcheck: $(OBJECTS)
	$(CC) -o $@ $(OBJECTS)

%.o: %.c spellcheck.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Other tasks in MAKE

In practice, MAKE is used for other tasks than building components.

Example:
Makefiles usually have a pseudo target `install` which installs the generated program.

MAKE for installation

```
# Installation program
INSTALL_PROGRAM = install

# General prefix for customer-sided
# installed programs and data
prefix = /usr/local
# Prefix for machine-specific
# programs and data
exec_prefix = $(prefix)
# Prefix for machine-specific programs
bindir = $(exec_prefix)/bin

install: spellcheck
  $(INSTALL_PROGRAM) spellcheck $(bindir)/spellcheck
```

Standard pseudo targets in MAKE

command	effect
`make all`	Constructs the whole program.
`make install`	Installs the program on a computer.
`make uninstall`	Deletes the program from the computer.
`make clean`	Deletes the program and all derived files.
`make distclean`	Like “clean”; but also reverses platform-specific adaptations of the source text.
`make dist`	Creates a source code package (distribution).
`make check`	Carries out suitable tests.
`make depend`	Recalculates dependencies.

Recursive MAKE

- Large projects are usually split into subsystems.
- The subsystems have their own Makefile.
- Example: (library in folder `readline`)

```
READLINE_TARGETS = readline/libreadline.a \
                   readline/readline.h

readline $(READLINE_TARGETS):
    cd readline && $(MAKE)
```

- If the invocation is: `$ gmake CC=gcc readline`, `$(MAKE)` expands to `gmake CC=gcc`

Recursive MAKE (2)

- A system is typically separated into modules

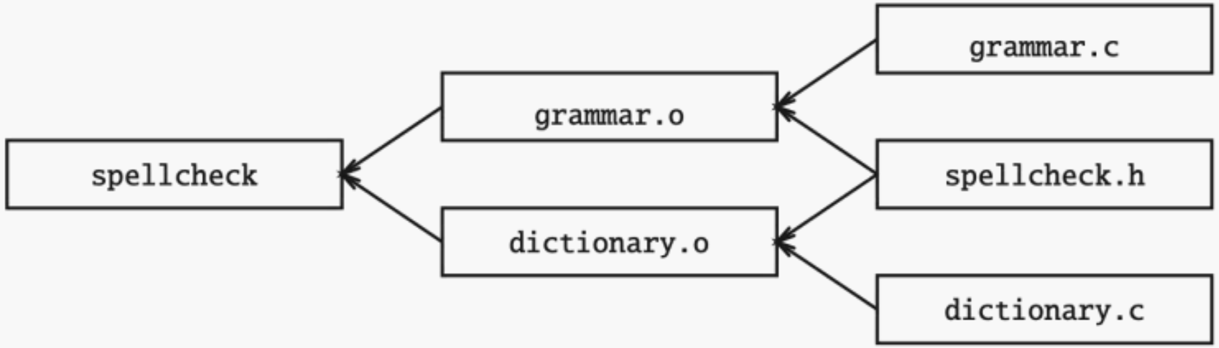
```
MODULES = termcap readline main

all install uninstall clean distclean:
    for dir in $(MODULES); do \
        (cd $$dir && $(MAKE) $@); \
    done
```

- Problem: does not allow cyclic dependencies!

Two types of dependencies

- Dependency of components used:
  Every component that is accessed
  during the construction process
  can have an effect on the derived components.
- Dependency of construction tools:
  If the compiler is changed,
  all object files have to be recompiled.

Identifying dependencies



In the Makefile:

```
grammar.o: grammar.c spellcheck.h
dictionary.o: dictionary.c spellcheck.h
```

Automatically determining dependencies

Option -M causes the C compiler
to list all dependencies:

```
$ cc -M grammar.c
grammar.o: grammar.c
grammar.o: ./spellcheck.h
$ _
```

Automatically generating dependencies

In the Makefile:

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

-include Makedeps
```

Additions and extensions

- Language-specific knowledge
- Automatically determining dependencies
- Changes in the construction
- Binary pools for derived components

Concepts (1/2)

- The aim of automatic program construction is to create all derived components from a number of source components.
- If a component has changed,
  all the components derived from it
  must be rebuilt.
- MAKE is a tool for building programs, in which existing derived files are reused as much as possible.

Concepts (2/2)

- MAKE uses a Makefile, a system description that lists the components of the system as well
  as their dependencies and construction steps.
- MAKE uses the change date of components
  in order to detect changes.
- Makefiles can be simplified and parameterised using variables and suffix or pattern rules.