The Prague Bulletin of Mathematical Linguistics
NUMBER 100 OCTOBER 2013 9–18

Makefiles for Moses

Ulrich Germann

University of Edinburgh

Abstract

Building MT systems with the *Moses* toolkit is a task so complex that it is rarely done manually. Over the years, several frameworks for building, running, and evaluating *Moses* systems have been developed, most notably the *Experiment Management System* (EMS). While EMS works well for standard experimental set-ups and offers good web integration, designing new experimental set-ups within EMS is not trivial, especially when the new processing pipeline differs considerably from the kind EMS is intended for. In this paper, I present M4M (*Makefiles for Moses*), a framework for building and evaluating *Moses* MT systems with the *GNU Make* utility. I illustrate the capabilities by a simple set-up that builds and compares two different systems with common resources. This set-up requires little more than putting training, tuning and evaluation data into the right directories and running *Make*.[1] The purpose of this paper is twofold: to guide first-time users of *Moses* through the process of building baseline MT systems, and to discuss some lesser-known features of the *Make* utility that enable the MT practitioner to set up complex experimental scenarios efficiently. M4M is part of the *Moses* distribution.

1. Introduction

The past fifteen years have seen the publication of numerous open source toolkits for statistical machine translation (SMT), from word alignment of parallel text to decoding, parameter tuning and evaluation (Och and Ney, 2003; Koehn et al., 2007; Li et al., 2009; Gao and Vogel, 2008; Dyer et al., 2010, and others). While all these tools greatly facilitate SMT research, building actual systems remains a tedious and complex task. Training, development and testing data have to be preprocessed, cleaned

[1] For the sake of convenience, I use *Make* to refer to *GNU Make* in this paper. *GNU Make* provides a number of extensions not available in the original *Make* utility.

© 2013 PBML. All rights reserved.
Corresponding author: ugermann@inf.ed.ac.uk
Cite as: Ulrich Germann. Makefiles for Moses. The Prague Bulletin of Mathematical Linguistics No. 100, 2013, pp. 9–18. doi: 10.2478/pralin-2013-0007.

up and word-aligned. Language and translation models have to be built, and system parameters have to be tuned for optimal performance. Some of these tasks can be performed in parallel. Some can be parallelized internally by a split-and-merge approach. Others need to be executed in sequence, as some build steps depend on the output of others.

There are generally three approaches to automating the build process. The first approach is to use **shell scripts** that produce a standard system setup. This is the approach taken in *Moses for Mere Mortals*.[2] This approach works well in a production scenario where there is little variation in the setup, and where systems are usually built only once. In a research scenario, where it is typical to pit numerous systems variations against one another, this approach suffers from the following drawbacks.

- Many of the steps in building SMT systems are computationally very expensive. Word alignment, phrase table construction and parameter tuning can each easily take hours, if not days, especially when run without parallelization. It is therefore highly desirable *not* to recreate resources unnecessarily. Building such checks into regular shell scripts is possible but tedious and error-prone.
- When the build process fails, it can be hard to determine the exact point of failure.
- Parallelization, if desired, has to be hand-coded.

The second approach is to write **a dedicated build system**, such as the *Experiment Management System* (EMS) for *Moses* (Koehn, 2010), or *Experiment Manager* (*Eman*), a more general framework for designing, running, and documenting scientific experiments (Bojar and Tamchyna, 2013).

EMS was designed specifically for *Moses*. It is capable of automatically scheduling independent tasks in parallel and includes checks to ensure that resources are only (re)created when necessary. EMS works particularly well for setting up a standard baseline system and then tweaking its configuration manually, while EMS keeps track of the changes and records the effect that each tweak has on overall system performance. In its job scheduling capabilities, EMS is reminiscent of generic build systems such as *Make*. In fact, the development of EMS is partly due to perceived shortcomings of *Make* (P. Koehn, personal communication), some of which we will address later on.

As a specialized tool that implements a specific way of running *Moses* experiments, EMS has a few drawbacks, too. Experimental setups that stray from the beaten path can be difficult to specify in EMS. In addition, the point of failure is not always easy to find when the system build process crashes, especially when the build failure is due to errors in the EMS configuration file.

[2] `http://en.wikipedia.org/wiki/Moses_for_Mere_Mortals`, `https://code.google.com/p/moses-for-mere-mortals`

Eman (Bojar and Tamchyna, 2013) also has its roots in SMT research but is designed as a general framework for running scientific experiments. Its primary objectives are to avoid unnecessary recreation of intermediate results, and to ensure that all experiments are replicable by preserving and thoroughly documenting all experimental parameters and intermediate results. To achieve this, *Eman* has a policy of never overwriting or re-creating existing files. Instead, *Eman* clones and branches whenever an experiment is re-run. Due to its roots, *Eman* comes with a framework for running standard SMT experiments.

The third approach is to rely on **established generic build systems**, such as the *Make* utility. *Make* has the reputation of being arcane and lacking basic features such as easy iteration over a range of integers, and much of this criticism language is indeed justified — *Make* is not for the faint-of-heart. On the other hand, it is a tried-and-tested power tool for complex build processes, and with the help of some of the lesser-known language features, it can be extremely useful also in the hands of the MT practitioner.

This article is foremost and above all a tutorial on how to use *Make* for building and experimenting with *Moses* MT systems. It comes with a library of *Makefile* snippets that have been included in the standard *Moses* distribution.[3]

2. Makefile Basics

While inconveniently constrained in some respects, the *Make* system is very versatile and powerful in others. In this section I present the features of *Make* that are the most relevant for using *Make* for building *Moses* systems.

2.1. Targets, Prerequisites, Rules, and Recipes

Makefile rules consist of a *target*, usually a file that we want to create, *prerequisites* (other files necessary to create the target), and a *recipe*: the sequence of shell commands that need to be run to create the target. The target is (re-)created when a file of that name does not exist, or if any of the prerequisites is missing or younger than the target itself. Prior to checking the target, *Make* recursively checks all prerequisites. The relation between target and prerequisite is called a *dependency*.

Makefile rules are written as follows.

```
target: prerequisite(s)
    commands to produce target from prerequisite(s)
```

Note that each line of the recipe must be indented by a single tab. Within the recipe, the special variables $@, $<, $^, and $| can be used to refer to the target, the first normal prerequisite, the entire list of normal prerequisites, and the entire list of *order-only* prerequisites, respectively.

[3] `https://github.com/moses-smt/mosesdecoder`; *Makefiles for Moses* is located under `contrib/m4m`

In addition to regular prerequisites, prerequisites can also be specified as *order-only* prerequisites. Order-only prerequisites only determine the order in which rules are applied, but the respective target is not updated when the prerequisite is younger than the target. Order-only dependencies are specified as follows (notice the bar after the colon).

```
target: | prerequisite(s)
    commands to produce target from prerequisite(s)
```

Makefiles for Moses uses order-only dependencies extensively; it is a safe-guard against expensive resource recreation should a file time stamp be changed accidentally, e.g. by transferring files to a different location without preservation of the respective time stamps.

A number of special built-in targets, all starting with a period, carry special meanings. Files listed as prerequisites of these targets are treated differently from normal files. In the context of this work, the following are important.

.INTERMEDIATE: Intermediate files are files necessary only to create other targets but not important for the final system. If an intermediate file listed as the prerequisite of other targets does not exist, it is created only if the target needs to be (re)created. Declaring files as intermediate allows us to remove files that are no longer needed without triggering the recreation of dependent targets when *Make* is run again.

.SECONDARY: *Make* usually deletes intermediate files when they are no longer required. Files declared as secondary, on the other hand, are never deleted automatically by *Make*. Especially in a research setting we may want to keep certain intermediate files for future use, without having to recreate them when they are needed again. The combination of .INTERMEDIATE and .SECONDARY give us control over (albeit also the burden of management of) if and when intermediate files are deleted.

2.2. Pattern Rules

Pattern rules are well-known to anyone who uses *Make* for compiling code. The percent symbol serves as a place holder that matches any string in the target and at least one prerequisite. For example, the pattern rule

```
crp/trn/pll/tok/%.de.gz: | crp/trn/pll/raw/%.de.gz
    zcat $< | tokenize.perl -l de | gzip > $@
```

will match any target that matches the pattern `crp/trn/pll/tok/*.de.gz`, check for the existence of a file of the same name in the directory crp/trn/pll/raw and execute the shell command

```
zcat $< | tokenize.perl -l de | gzip > $@
```

2.3. Variables

Make knows two 'flavors' of variables. By default, variables are expanded *recursively*. Consider the following example. Unlike variables in standard Unix shells, parentheses or braces around the variable name are mandatory in *Make* when referencing a variable.[4]

```
a = 1
b = $(a)
a = 2
all:
    echo $(b)
```

In most conventional programming languages, the result of the expansion of `$(b)` in the recipe would be 1. Not so in *Make*: what is stored in the variable is actually a reference to `a`, not the value of `$(a)` at the time of assignment. It is only when the value is needed in the recipe that each variable reference is recursively replaced by its value at that (later) time.

On the other hand, *simply expanded* variables expand their value at the time of assignment. The flavor of variable is determined at the point of assignment. The operator '=' (as well as the concatenation operator '+=' when used to create a new variable) creates a recursively expanded variable; simply expanded variables are created with the assignment operator ':='.

Multi-line variables can be defined by sandwiching them between the `define` and `endef` keywords, e.g.

```
define tokenize

$(1)/tok/%.$(2).gz: | $(1)/raw/%.$(2).gz
    zcat $$< | tokenize.perl -l $(2) | gzip > $$@

endef
```

Notice the variables `$(1)` and `$(2)` as well as the escaping of the variables `$<` and `$@` by double `$$`. The use of the special variables `$(1)`, ... `$(9)` turns this variable into a user-defined function. The blank lines around the variable content are intentional to ensure that the target starts at the beginning of a new line and the recipe is terminated by a new line during the expansion by `$(eval $(call ...))` below.

The call syntax for built-in *Make* functions is as follows.

```
$(function-name arg1,arg2,...)
```

[4] Except variables with a single-character name.

User-defined functions are called via the built-in *Make* function `call`. The value of

```
$(call tokenize,crp/trn/pll,de)
```

is thus

```
crp/trn/pll/tok/%.de.gz: | crp/trn/pll/raw/%.de.gz
    zcat $< | tokenize.perl -l de | gzip > $@
```

Together with the built-in *Make* functions `foreach` (iteration over a list of space-separated tokens) and `eval` (which inserts its argument at the location where it is called in the Makefile), we can use this mechanism to programmatically generate *Make* rules on the fly and in response to the current environment. For example,

```
directories := $(shell find -L crp -type d -name raw)
$(foreach d,$(directories:%/raw=%),\
$(foreach l,de en,\
$(eval $(call tokenize,$(d),$(l)))))
```

creates tokenization rules for the languages `de` and `en` for all subdirectories in the directory `crp` that are named `raw`. The substitution reference `$(directories:%/raw=%)` removes the trailing `/raw` on each directory found by the shell call to `find`.

3. Building Systems and Running Experiments

3.1. A Simple Comparison of Two Systems

With these preliminary remarks, we are ready to show in Fig. 1 how to run a simple comparison of two phrase-based *Moses* systems, using mostly tools included in the *Moses* distribution. For details on the M4M modules used, the reader is referred to the actual code and documentation in the M4M distribution. The first system in our example relies on word alignments obtained with `fast_align`[5] (Dyer et al., 2013); the second uses *mgiza++* (Gao and Vogel, 2008). Most of the functionality is hidden in the M4M files included by the line

```
include ${MOSES_ROOT}/contrib/m4m/modules/m4m.m4m
```

The experiment specified in this Makefile builds the two systems, tunes each five times on each tuning set (with random initialization), and computes the BLEU score for each tuning run on each of the data sets in the evaluation set.

The design goal behind the setup shown is to achieve what I call the washing machine model: put everything in the right compartment, and the machine will automatically process everything in the right order. There is a standard directory structure that determines the role of the respective data in the training process, shown in Table 1.

[5] `https://github.com/clab/fast_align`

```
MOSES_ROOT = ${HOME}/code/moses/master/mosesdecoder
MGIZA_ROOT = ${HOME}/tools/mgiza
fast_align = ${HOME}/bin/fast_align
# L1: source language; L2: target language
L1   = de
L2   = en
WDIR = $(CURDIR)

include ${MOSES_ROOT}/contrib/m4m/modules/m4m.m4m

# both systems use the same language model
L2raw := $(wildcard ${WDIR}/crp/trn/*/raw/*.${L2}.gz)
L2data := $(subst /raw/,/cased/,${L2trn})
lm.order  = 5
lm.factor = 0
lm.lazy   = 1
lm.file   = ${WDIR}/lm/${L2}.5-grams.kenlm
${lm.file}: | $(L2data)
$(eval $(call add_kenlm,${lm.file},${lm.order},${lm.factor},${lm.lazy}))
.INTERMEDIATE: ${L2data}

# for the first system, we use fast_align
word-alignment = fast
system  = ${word-alignment}-aligned
ptable  = model/tm/$(system).${L1}-${L2}
dtable  = model/tm/$(system).${L1}-${L2}
$(eval $(call add_binary_phrase_table,0,0,5,${ptable}))
$(eval $(call add_binary_reordering_table,0,0,8,\
              wbe-mslr-bidirectional-fe-allff,${dtable},${ptable}))
$(eval $(call create_moses_ini,${system}))
SYSTEMS := $(system)

# for the second system, we use mgiza
word-alignment = giza
$(eval $(clear-ptables))
$(eval $(clear-dtables))
$(eval $(call add_binary_phrase_table,0,0,5,${ptable}))
$(eval $(call add_binary_reordering_table,0,0,8,\
              wbe-mslr-bidirectional-fe-allff,${dtable},${ptable}))
$(eval $(call create_moses_ini,${system}))
SYSTEMS += $(system)
ifdef tune.runs
EVALUATIONS :=
$(eval $(tune_all_systems))
$(eval $(bleu_score_all_systems))
all: ${EVALUATIONS}
        echo EVALS ${EVALUATIONS}
else
all:
        $(foreach n,$(shell seq 1 5),${MAKE} tune.runs="$n␣$n";)
endif
```

Figure 1. Makefile for a simple baseline system. All the details for building the system are handled by M4M.

`crp/trn/pll/`	parallel training data
`crp/trn/mno/`	monolingual training data
`crp/dev/`	development data for parameter tuning
`crp/tst/`	test sets for evaluation
`model/tm`	phrase tables
`model/dm`	distortion models
`model/lm`	language models
system`/tuned/`*tset*`/`*n*`/moses.ini`	result of tuning system *system* on tuning set *tset* (n-th tuning run)
system`/eval/`*tset*`/`*n*`/`*eset*`.*`	evaluation results for test set *eset*, translated by system *system*`/tuned/`*tset*`/`*n*`/moses.ini`

Table 1. Directory structure for standard M4M setups

3.2. Writing Modules

The bulk of the system building and evaluation work is done by the various M4M modules. While an in-depth discussion of all modules is impossible within the space limitations of this paper, a few points are worth mentioning here.

One of the inherent risks in using build systems is that two independent concurrent build runs with overlapping targets may interfere with one another, overwriting each other's files. In deviation from the usual philosophy of build systems — recreate files when their prerequisites change — M4M adopts a general policy of only creating files when they do not exist, never recreating them. It is up to the user to first delete the files that they do want to recreate. To prevent concurrent creation of the same target, we adopt the following lock/unlock mechanism.

```
define lock
mkdir -p ${@D}
test ! -e $@
mkdir $@.lock
echo -n "Started␣at␣$(shell␣date)␣"         > $@.lock/owner
echo -n "by␣process␣$(shell␣echo␣$$PPID)␣" >> $@.lock/owner
echo    "on␣host␣$(shell␣hostname)"        >> $@.lock/owner
endef

define unlock
rm $@.lock/owner
rmdir $@.lock
endef
```

The first line of the `lock` mechanism ensures that the target's directory exists. The second line triggers an error when the target already exists. Recall that our policy is to never re-create existing files. The third line creates a semaphore (directory creation is an atomic file system operation). When invoked without the `-p` parameter, `mkdir`

will refuse to create a directory that already exists. The logging information added in the fourth and subsequent lines is helpful in error tracking. It allows us to determine easily which process created the respective lock and check if the process is still running.

Another risk is that partially created target files may falsely be interpreted as fully finished targets, either due to concurrent *Make* runs with overlapping targets, or due to a build failure in an earlier run. (Normally, *Make* deletes the affected target if the underlying recipe fails. However, we disabled this behavior by declaring all files .SECONDARY.) We can address this issue by always creating a temporary target under a different name and renaming that to the proper name upon successful creation. The pattern for a module definition thus looks as follows.

```
target: prerequisite
    $(lock)
    create-target > $@_
    mv $@_ $@
    $(unlock)
```

4. Conclusion

I have presented *Makefiles for Moses*, a framework for building and evaluating *Moses* MT system within the *GNU Make* framework. The use of the `eval` function in combination with custom functions allows us to dynamically create *Make* rules for multiple systems in the same Makefile, beyond the limitations of simple pattern rules.

A simple but effective semaphore mechanism protects us from the dangers of running multiple instances of *Make* over the same data. By using order-only dependencies and .INTERMEDIATE statements, we can specify a build system that creates resources only once, and allows for the removal of intermediate files that are no longer needed, without *Make* recreating them when run again.

Make's tried-and-tested capabilities for parallelization in the build process are fully available.

While *Makefiles for Moses* lacks the bells and whistles of EMS particularly with respect to progress monitoring and web integration of the experimental results, it offers greater flexibility in experimental design, especially with respect to scriptability of system setup.

5. Acknowledgements

The work described in this paper was performed as part of the following projects funded under the European Union's Seventh Framework Programme for Research (FP7): *Accept* (grant agreement 288769), *Matecat* (grant agreement 287688), and *Casmacat* (grant agreement 287576).

Bibliography

Bojar, Ondřej and Aleš Tamchyna. The design of Eman, an experiment manager. *Prague Bulletin of Mathematical Linguistics*, 99:39–58, April 2013.

Dyer, Chris, Adam Lopez, Juri Ganitkevitch, Johnathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, July 2010.

Dyer, Chris, Victor Chahuneau, and Noah A. Smith. A simple, fast, and effective reparameterization of IBM Model 2. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 644–648, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

Gao, Qin and Stephan Vogel. Parallel implementations of word alignment tool. In *Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 49–57, Columbus, Ohio, June 2008. Association for Computational Linguistics.

Koehn, Philipp. An experimental management system. *Prague Bulletin of Mathematical Linguistics*, 94:87–96, September 2010.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics: Demonstration Session*, Prague, Czech Republic, June 2007.

Li, Zhifei, Chris Callison-Burch, Chris Dyer, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. Joshua: An open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 135–139, Athens, Greece, March 2009. Association for Computational Linguistics.

Och, Franz Josef and Hermann Ney. A systematic comparison of various statistical alignment models. *Computational Linguistics*, 29(1):19–51, March 2003.

Address for correspondence:
Ulrich Germann
ugermann@inf.ed.ac.uk
School of Informatics
University of Edinburgh
10 Crichton Street
Edinburgh, EH8 9AB, United Kingdom