View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Biblio at Institute of Formal and Applied Linguistics

The Prague Bulletin of Mathematical Linguistics
NUMBER 99 APRIL 2013 39–58

# The Design of Eman, an Experiment Manager

Ondřej Bojar, Aleš Tamchyna

Charles University in Prague, Faculty of Mathematics and Physics, Institute of Formal and Applied Linguistics

**Abstract**

We present `eman`, a tool for managing large numbers of computational experiments. Over the years of our research in machine translation (MT), we have collected a couple of ideas for efficient experimenting. We believe these ideas are generally applicable in (computational) research of any field. We incorporated them into `eman` in order to make them available in a command-line Unix environment.

The aim of this article is to highlight the core of the many ideas. We hope the text can serve as a collection of experiment management tips and tricks for anyone, regardless their field of study or computer platform they use. The specific examples we provide in `eman`'s current syntax are less important but they allow us to use concrete terms. The article thus also fills the gap in `eman` documentation by providing some high-level overview.

## 1. Introduction

Computational sciences including computational linguistics and computer science require broad experimenting to support theories and evaluate various techniques or methods. Very often, even the authors of some novel idea cannot guess the best possible method parameters and some form of search for them is desirable. This becomes more apparent if the method combines several independent modules or processing steps, each of which may or may not have been evaluated independently of the overall goal.

Another common aspect of natural sciences is the overarching strive for reproducibility. A novel method is never completely trusted until validated by a few independent laboratories, a program has to be tested and evaluated on a range of inputs and so on.

We pinpoint these two aspects of science by noting that: research = **re**producible **search**.

© 2013 PBML. All rights reserved.
Corresponding author: bojar@ufal.mff.cuni.cz
Cite as: Ondřej Bojar, Aleš Tamchyna. The Design of Eman, an Experiment Manager. The Prague Bulletin of Mathematical Linguistics No. 99, 2013, pp. 39–58. doi: 10.2478/pralin-2013-0003.

In this article, we describe a very general tool that facilitates both reproducibility and search for the best configuration and parameters of complex experimental pipelines. Our eman also supports the collaboration of several people on the experiment.

Eman is open-source software, freely available for both non-commercial and commercial use.[1] The most recent version of the tool as well as other documentation is accessible at:

http://ufal.mff.cuni.cz/eman

The article is structured as follows: in Section 2, we explain what we perceive as the state-of-the-art techniques in efficient experimenting, highlighting the design goals of our new tool. Section 3 introduces our terminology and the basic building blocks of experiments in eman's terms. Section 4 summarizes the first area of eman's utility: navigation in the space of steps and experiments. Section 5 is devoted to the idea of cloning experiments and Section 6 describes the third key contribution: a general technique for collecting and interpreting the results. We conclude by introducing eman's support for teamwork (Section 7), related tools (Section 8) and our future plans (Section 9).

While we show some calls of eman commands in their exact syntax, the main goal of this article is to describe the underlying general ideas, not to serve as a reference guide for the tool. For this, the user is advised to the manual page of eman which can be obtained by running:

```
eman --man
```

## 2. Design Objectives

The design of eman builds on our experience that the following features of experimentation environment are essential:

**Reuse of results.** In order to save both computation time and disk space, we need to reuse as many intermediate results as possible.

**Encapsulation.** Scientific experiments usually consist of complex sequences of processing steps, each carried out using a different tool that itself often needs some analysis, debugging, tweaking or optimization. To simplify switching and keeping focus, eman promotes encapsulation of each logical step into a separate directory. This directory should be as self-contained as reasonable, so when the researcher later inspects it, all the inputs and outputs are in one place.

**Detailed records.** Detailed logging of program outputs as well as of commands issued is essential for ensuring reproducibility, debugging and analysis of errors and comparison of results. We extend this to recording also the exact versions of (third-party) tools used in the experiment and also the procedure needed to

[1] Eman is licensed under the Creative Commons Attribution-ShareAlike License 3.0 (CC-BY-SA).

obtain and install the tools. This is achieved by treating the (source) code of the tools as input data of the experiment and including the compilation of the tools in the pipeline of the experiment. The reuse of intermediate results ensures the code is compiled only once.

**Immutability.** To simplify the record keeping, we opt for immutability of all data that is created in the experiment. Whenever some intermediate result is created based on some settings, `eman` never changes it. Modifications of the run are of course possible, but they always obtain a new identifier and reside in a new directory.

**Hacking welcome.** Admittedly, research prototype software is often quickly patched and far from anything that could be called a stable release. Furthermore, and this is a more important issue, research software does not always fit the purpose in new experiments. It is thus common that the tools have to be adapted or that a manual intervention is necessary after a random unexpected failure. `Eman` introduces a great deal of flexibility of experiment design – experiments are composed of individual steps which are further split into several lifetime stages – to allow for such an intervention.

**Cloning.** Research partially comprises of examining a range of minor modifications of a setup. In `eman`'s view, as it will be described below, experiments are defined by arbitrary variables and such setup modifications usually amount to setting these variables differently. Section 5 provides examples of one-line commands that take an existing experiment and apply a given set of modifications to it (such as setting a parameter differently or reversing the source and target language in an MT experiment). Finally, the necessary minimum of new processing steps are created and launched, reusing the steps common to both setups.

Cloning is in fact such a powerful idea that the relatively simple implementation of it in `eman` (regular expressions applied to experiment configuration files) allowed to create the tool Prospector, an automatic researcher (Tamchyna and Bojar, 2013). Prospector automatically searches the "space of possible MT systems" by evaluating various settings specified by its configuration file. The search can be guided by any metric, e.g. the well-known BLEU (Papineni et al., 2002) as calculated in the final evaluation step. Several search algorithms are implemented (greedy, exhaustive, genetic, random).

Prospector allows researches to avoid the tedious work of e.g. finding optimal parameters or meta-parameters for the MT decoder (beam size etc.) or any other experimental settings. It is freely available and distributed along with `eman`.

**Parallelism.** The parallelizations common in contemporary computer science (multiple processor cores, clusters of computers) allow for parallel execution of experiments. This is highly desirable because each individual experiment often takes a long time. Carrying out experiments in a strictly serial order would waste researchers' time and not fully exploit the available computational resources.

On the other hand, the researcher can easily lose track and focus when running many experiments in parallel.
Eman naturally allows to submit individual processing steps to a computer cluster, but more importantly, eman is designed to simplify the orientation in the large number of experiments already performed or in execution (see Section 4) and to some extent also the foreseen ones (see Section 6.3). The design also allows to derive (clone) new experiments from old ones even before the old ones complete.

**Collaboration.** The most recent feature of eman is the support for distributed experimenting. Currently we require a common filesystem (such as NFS), but that is reasonably easy to set up even across large distances. Individual processing steps can be launched by different researchers at different sites. The simple command "eman add-remote" issued once allows to include all the partial results of a remote site in the local environment. Circular inclusion is permitted allowing multiple researchers to "work at a common desk", reusing other people's processing steps (not just the programs but also the outputs of their particular runs), or to reinterpret their results (e.g. by creating new tabular views).
The same mechanism can be beneficial even for a single researcher as it allows to strictly separate some core source data (such as multiple training sets that nevertheless needed some preparation) from different branches of experiments.

**Succinct notation.** Shortcuts and abbreviations are very useful for improving the efficiency of the operating researcher. Eman provides shortcuts at several occasions, which is very useful e.g. for checking the status of the experiments over SSH in the cell phone.

## 3. Seeds, Steps, Experiments

Each *experiment* consists of atomic tasks called *steps*. In the context of MT, steps correspond e.g. to training a language model, translating a test set or running tuning. The individual steps depend on each other – the experiment is then a DAG (directed acyclic graph) of steps.

Each step has a type such as tm (translation model) or translate. The code which is executed when the step is run is generated by the corresponding *seed*. In the terminology of object-oriented programming (OOP), seeds can be viewed as classes and steps as their instances. Unlike in OOP, eman's positive stance to hacking allows different steps (instances) of the same type (class) run code customized arbitrarily, not just using proper subclassing.

In our particular implementation of eman, each step is simply a directory named using the pattern "s.steptype.abcHASH.20121215-1234" where the date and the hash value make the name unique. Seeds are then simply programs (in any language of the researcher's choice) which interpret some Unix environment variables and generate

```
+- s.compress.370c2483.20121108-1216
|  | CMD=bzip2
|  | CMDARGS=
|  | DATASTEP=s.data.aaf8c8b1.20121108-1149
|  +- s.data.aaf8c8b1.20121108-1149
|  |  | CMD="cat ../binary.test"
|  |  | SIZE=10000
|  |  | TYPE=binary
```

*Figure 1. Example of an eman traceback.*

executable code (again, in any language). The code is stored in the step directory and later run (once all predecessors are ready and the step is started).

Eman is used in a directory called *playground* – all steps are created there, based on seeds in the subdirectory eman.seeds. The "eman add-remote" allows to link remote playgrounds to the current one. By adding a remote playground, the directory structure is not changed but eman suddenly knows about steps coming from the remote playgrounds, it can show their properties and include them in local experiments.

### 3.1. A Sample Experiment

For illustration, we implemented a "compression playground" which provides an environment for evaluating compression algorithms. This sample playground contains two seeds:

**data** Imports data into the playground – the data can be generated by any command (specified by the variable CMD) and the user can limit the amount of data using the variable SIZE.

**compress** Given some data, compress it using the command given in the variable CMD with some optional CMDARGS and calculate the compression rate.

Figure 1 shows an example of an experiment in this playground in eman's format. This *traceback* is a full definition of the experiment. The seeds were "instantiated" to steps with some variable values (e.g. the compression command is bzip2) and connected to form a DAG – note that the dependency is explicitly captured in the variable DATASTEP.

### 3.2. Lifetime of a Step

Figure 2 depicts the lifetime of a step.

New steps are created using "eman init STEPTYPE". Eman creates a new directory in the current playground and copies the corresponding seed into it. Then the seed is executed – at this stage, the seed only performs basic sanity checks to determine

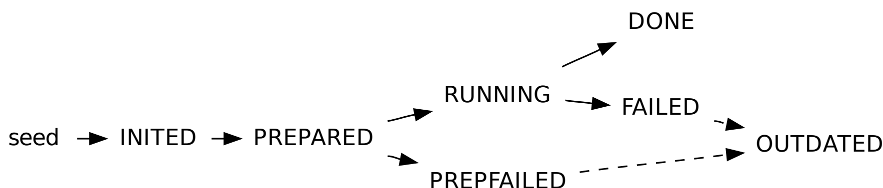


*Figure 2. The lifetime of a processing step in eman*

whether all required variables are defined etc. If everything succeeds, the step is registered in `eman` and receives the status `INITED`.

The seed is executed once more when the user runs "`eman prepare SPEC`". At this point, the seed creates an executable file `eman.command` which contains the step code. `Eman` checks whether the file was created and sets the step status to `PREPARED`.

Finally, the user runs "`eman start SPEC`" and the step is started. Its status changes to `RUNNING`. Once the step terminates, its status is either `DONE` or `FAILED`.

Steps at any stage, including the `FAILED` ones, can still serve as a basis for creating new steps with the same or similar variables, see below. For the purposes of marking that the user has already handled a failure, one more state, `OUTDATED`, was introduced. Upon request ("`--outdate`") `eman` not only creates a new instance of a failed step but also moves the failed one to the outdated state.

There are many shortcuts for user convenience – "`eman start`" on an `INITED` step automatically runs "`eman prepare`". The whole process of creating and running a step can even be done in one command (and it often is): "`eman init --start`".

The acyclicity of the lifetime diagram (no directed loops) is in line with the design objectives of immutability and detailed records. One should want to keep the logs of a failure and redo the job in a fresh instance of the step. (Indeed, this is what the command "`eman redo`" does, see Section 5.2.) In practice, a step can be very costly and fail at some late stage of execution. It would be wasteful to rerun it from scratch. The fact that each step includes its code (the file "`eman.command`", cf. the encapsulation objective) allows to manually fix this code and to jump to a recorded point shortly before the failure. After these manual changes in "`eman.command`", calling "`eman continue`" puts the failed step back to the `RUNNING` state and submits it to the cluster again (or runs it locally, depending on the user's environment).

### 3.3. Motivation for the Three Stages

What are the benefits of breaking the execution of a single processing step into the stages of initialization, preparation and the run itself?

The *initialization* is vital, it turns a blank directory into a valid `eman` step with variables defined. From this point on, the step can be incorporated into complex experiments and it can be used as a basis for cloning. There is thus no need to wait until the step finishes, we can plan ahead (and even submit for execution) other steps that will build on the future outputs.

After the initialization, the user has a chance to tweak the seed (and thus influence the actual command that will be performed). This is the point where we depart the OOP by allowing different instances run customized code.

The init phase should be very quick, it is run interactively and often repeated for many steps when cloning whole experiments.

The *preparation* phase is thus meant to get all input data in place, so that the user can check them before the actual computation, e.g. submitting the step to the cluster. In our MT experiments, the preparation phase was originally responsible for things like cutting a given subsection or subset of annotation features from the training data. As our training data grew, running these filters during the (still interactive) preparatory phase became inconvenient, so we changed our seeds and shifted even the obtaining of input data into the actual run. For `eman`, this makes no difference. The choice what happens at what phase is entirely up to the user; any of the phases can be even empty.

### 3.4. From Steps to Experiments

Steps are combined to form experiments. There is no pre-defined interface for communication between steps. Each step has access to its (direct) predecessors via variables – it can extract whatever files the previous step has created from its directory. The same step can thus serve several purposes at once, it just needs to produce outputs relevant to the respective successors.

The initial setup of experiments is somewhat tedious, the user has to run a sequence of commands like:

```
SOURCEDATASTEP=s.mydata.12345678.20121215-1234 eman init myproces-
sor
```

The user has to manually set the variable `SOURCEDATASTEP` to the name of the previously initialized `mydata` step. Once the full cascade of steps, i.e. an experiment, is set up, it is easier to derive variations of it using cloning, see Section 5.

### 3.5. Referring to Steps vs. Experiments

Note that the pointer to a step directory "`s.steptype.123`" can mean either just the single step that was carried out in the directory, or the whole experiment, i.e. the directed acyclic structure of steps that culminates with the given step. These two notions should not be confused.

It is the particular `eman` command that resolves the ambiguity between a step and an experiment. So for instance, "`eman prepare`" prepares an individual step regard-

less the status of its predecessors. Depending on what a particular step requires, the preparation may fail because the predecessors are still in the `INITED` state only and do not provide relevant data. The command “`eman start`” is more useful as it operates on the whole *experiment*. In other words, it ensures that the whole DAG of steps is first `PREPARED` and then submits all steps that were not finished yet to the cluster, introducing any necessary job dependencies.

## 4. Navigation in the Playground of Steps and Experiments

As the user creates experiments or derives clones of them (Section 5), the playground becomes quickly filled with step directories of unhelpful names like "`s.tm.1a53fg63.201`
Finding a particular step can then be difficult and time-consuming. This section briefly summarizes the four main techniques `eman` provides to ease the navigation in the playground: listing details of individual steps, finding (selecting) steps with given properties, examining the structure of experiments, i.e. how steps depend on one another, and manually tagging steps.

One more aspect of playground structure remains to be harnessed in a future version of `eman`: the history how steps were derived from other steps. Currently, `eman` only records the immediate origin of a derived step in the file “`eman.derived_from`”.

### 4.1. Listing Steps and Their Details

The command “`eman ls`” prints steps in the current playground. The user can filter the listing based on the step type and request additional information – most importantly step variables, status and tags (see Section 4.4) – using command-line options. The following example query returns all steps of the type “`align`” and prints their variables, status and disk usage:

```
eman ls align --vars --stat --dus
```

Some shortcuts are again provided by means of commands `eman vars`, `stat` and `tags` that print the required information about all steps, or a particular step:

```
eman vars s.tm.1a5
```

Note that it is not necessary to specify the full step name, any part of it (not necessarily the beginning) long enough to make it unique within the playground is sufficient.

### 4.2. Finding Existing Steps

The command “`eman select`” (optionally abbreviated to “`sel`”) provides a flexible means for finding steps with specified properties. The following few examples are just a brief demonstration of the query language.

- Steps which were created today and failed:

```
eman sel today f
```

- Last (most recent) five steps of the type "align":
```
eman sel t align l 5
```
- Language model steps (i.e. steps of the type "lm") trained on word lemmas (vre stands for "a **v**ariable matches **r**egular **e**xpression"):
```
eman sel t lm vre lemma
```
- Language models of order other than three (note the word not):
```
eman sel t lm not vre ORDER=3
```
- MERT (Och, 2003) steps with a (possibly indirect) predecessor of the type align whose variables match the expression "lemma" (presumably a word alignment step done on word lemmas; br stands for backward recursion and matches properties in preceding steps):
```
eman sel t mert br t align vre lemma
```
- Translation model ("tm") steps which were evaluated on a given test set (fr means forward recursion):
```
eman sel t tm fr vre TESTCORP=wmt12
```

The syntax is very succinct which allows to write complex queries with very little effort. Users of eman frequently log in to their cluster using cell-phone SSH and type simply "eman sel f" to see if any experiments need their attention.

### 4.3. Dependencies and Users of Steps

Eman provides commands to list predecessors and successors of steps. Direct predecessors (dependencies) can be obtained using the command "eman deps" while "eman traceback" or "eman tb" prints the full *traceback* (i.e. DAG) of steps.

Eman assumes that an experiment is defined by the structure of dependencies and the values of their variables; this implies that the command "eman tb --vars FINAL-STEP" outputs a full, unambiguous specification of the whole experiment.

An example of a traceback with variables was already given in Figure 1.

Analogously to the predecessors, direct and indirect successors can be listed using the commands "eman users" and "eman tf" (traceforward), respectively.

### 4.4. Tagging of Steps

Steps that are somehow special or often referred to, e.g. because they were manually tweaked before submission or because they represent the baseline or the current best result, can be "tagged". Tags are simple labels associated with a particular step.

Tags are assigned to steps using the command "eman add-tag":

```
eman add-tag BASELINE s.evaluator.123456
```

Later, tags can be used as step identifiers as long as they are unambiguous. So we can e.g. double check what was our baseline configuration:

```
eman tb --vars BASELINE
```

The tags are stored in the step directory in the file `eman.tags`. Upon re-tagging ("`eman retag`"), the labels are recursively propagated to the step successors (however their `eman.tags` files do not change, the propagation is done in `eman`'s internal index only). While this is useful for organizing results, see Section 6, it makes tags refer to more steps and thus no longer usable as step specifiers. In future versions of `eman`, we may thus remove or somehow restrict the tag propagation feature.

## 5. Cloning of Experiments

The previous sections described techniques for reusing intermediate results across experiments. Now we describe `eman` commands that allow to reuse the configurations of individual steps and whole experiments. The added twist is that when an existing experiment is "cloned", the variables may be arbitrarily changed.

### 5.1. Replicating Individual Steps

*Cloning* of an existing step means creating a new instance of the same step type, reusing most of the variable values. For instance, we may want to create somewhat larger test case for our compression experiment, and we already have the data step "`s.data.aaf8`" ready, as illustrated in Figure 1. The following command will create a new instance using the data seed and run it right away:

```
SIZE=500000 eman clone s.data.aaf8 --start
```

The above command works as an abbreviation of "`eman init data`" where all the variables would have to be specified:

```
SIZE=500000 CMD="cat ../binary.test" TYPE=binary eman init data
--start
```

The cloning will work even without changing any of the variables. In general, it is better to avoid multiple runs of the same configuration, but some computations are non-deterministic and running several copies allows to estimate confidence intervals of the result (Clark et al., 2011). In MT, the prototypical example is the minimum error-rate training, MERT. Creating four more replications of a MERT run is trivial:

```
for i in 2 3 4 5; do eman clone --start s.mert.123; done
```

Replicating a step with identical variables is also useful when a step fails. The command "`eman clone`" as described so far operates on individual steps, so any (failed) dependencies will not get recreated. A better option is described in the following section.

### 5.2. Redoing Experiments

When an experiment fails, “`eman redo`” can be used to re-create the necessary steps in the whole experiment pipeline. Redo will check the whole traceback of the given experiment and replicate any steps that are failed or outdated. When doing this, the correct links between dependencies are honored, so whenever a step gets redone, its successors will get redone as well.

With large-scale experiments, various technical problems often come into play, making the redo command very useful in day-to-day experimenting. A particular common reason for a failure is a full local temporary disk or memory limits set too low for the given input data, which leads to jobs being killed by the cluster. `Eman`, in cooperation with the scheduling environment, can set the requirements on available memory and disk, so the following usage pattern is quite common:

```
eman redo s.myFailedExp.123 --mem 30g --disk 80g --start --outdate
```

Note that “`eman redo`” walks only the traceback, not the traceforward of the given experiment. It is thus important to ask for a redo of the *final* steps of failed experiments.

### 5.3. Deriving Whole Experiments

By mixing the idea of modifying variables and redoing whole experiments, we arrive at the full power of experiment cloning.

We have already mentioned, that the traceback with variables (i.e. the output of “`eman tb --vars FINALSTEP`”) is the complete description of an experiment. The user can modify some variables in the textual form of the traceback and clone it:

```
eman clone < traceback.modified
```

When constructing steps from such a textual traceback, `eman` automatically discovers steps which can be re-used and only creates the parts of the experiment which are really needed. Experienced `eman` users often create the traceback, substitute some values and create the modified experiment on one line:

```
eman tb -s /oldvalue/newvalue/ | eman clone --dry-run
```

The parameter `-s` defines a substitution which is applied on the whole traceback and supports full Perl regular expressions. The “`--dry-run`” is useful for a quick check before creating the many step directories or “`--start`”ing the new experiment.

We have found cloning of experiments to be extremely useful and versatile in practice. Multiple settings of an MT system can be created and evaluated easily by defining the base experiment and cloning it several times with modified variable values.

With cloning, e.g. reversing the translation direction of an experiment is a trivial change. Similarly, one can easily repeat an experiment for multiple language pairs, change datasets, adjust language model order or modify factors for word alignment.

```
data     var /SIZE=(.*)/SIZE$1B/          /000B\$/kB/
data     var /TYPE=(.*)/TYPE$1/
compress var /CMD=(.*)/CMD$1/
compress var /CMDARGS=(.*)/ARGS$1/
compress var /CMDARGS=.*?-([0-9]).*/LEVEL$1/
```

*Figure 3. Sample* "eman.autotags" *configuration.*

## 6. Making Sense of Results

By a *result*, we mean a small token, usually a number, that was observed or measured during the run of an experiment. In eman's view, results are small bits of information available somewhere in the output files of a step.

Eman provides a set of tools for collecting and interpreting results.

### 6.1. Autotagging (Tags Based on Variables)

We have already introduced manual tags (Section 4.4) that can be later used to identify e.g. results based on a particular dataset or using a particular version of a program. In addition to tags, eman provides "autotags" that are created automatically from variables of steps using regular expressions and substitutions. The main purpose of automatic tags is to select relevant information from the variables and make it available for the interpretation of results, see below.

The user configures automatic tagging by writing rules into the file "eman.autotags". Each rule consists of the type of steps to which it applies, a regular expression that is matched against the step variables and optionally a regular-expression substitution to be applied on the match – to beautify it in a way.

The tagging rules implemented for our compression example are shown in Figure 3. Each line defines one rule – on the first line, we tell eman to match variables of data steps and look for the pattern "SIZE=.*". We extract the size and prefix it with the word "SIZE". We also perform a simple substitution to shorten the value – we replace "000B" with "kB". The data step shown in Figure 1 is assigned the tag "SIZE10kB" according to this rule.

### 6.2. Collecting Results

The first task when working with results is to collect them from all the many steps in the playground to a single place. This is achieved using the command "eman collect": all results will appear in the file "eman.results" in the playground directory.

```
s.compress.ba3c96ac.20121217-1415 DONE ratio .47770000000000000000  ARGS-5 CMDbzip2 LEVEL5 SIZE10kB TYPEhexrand
s.compress.bb9c7f1e.20121217-2145 DONE ratio .52478125000000000000  ARGS"-5 --rsyncable" CMDgzip LEVEL5 SIZE512kB TYPEhexrand
s.compress.c45a5afd.20121217-1414 DONE ratio .53680000000000000000  ARGS-9 CMDgzip LEVEL9 SIZE10kB TYPEhexrand
s.compress.99ab03f6.20121217-1436 DONE ratio .43619140625000000000  ARGS-5 CMDbzip2 LEVEL5 SIZE512kB TYPEhexrand
s.compress.02a5f93f.20121217-1436 DONE time  0.042                  ARGS-5 CMDgzip LEVEL5 SIZE512kB TYPEhexrand
s.compress.0545633f.20121217-1413 DONE time  0.002                  ARGS-4 CMDgzip LEVEL4 SIZE10kB TYPEhexrand
s.compress.07151d39.20121217-0116 DONE time  0.003                  ARGS CMDgzip SIZE10kB TYPEhexrand
s.compress.c45a5afd.20121217-1414 DONE TAG   ARGS-9 CMDgzip LEVEL9 SIZE10kB TYPEhexrand
s.compress.7694fe26.20121217-1436 DONE TAG   ARGS CMDbzip2 SIZE512kB TYPEhexrand
```

*Figure 4. A few random sample lines from the file "*`eman.results`*". Each line contains the step name, its status, the name of the result and its value and finally all the tags and autotags assigned to this step.*

The specification, what exactly should `eman` extract from a step directory, is provided by the user in the file "`eman.results.conf`". For instance, the configuration line:

`ratio` $\rightarrow$ `s.compress.*/ratio` $\rightarrow$ `CMD: cat`

specifies that steps of the type "`compress`" measure a particular property, namely the compression ratio that they achieved on some give data. The value can appear anywhere in a file in the step directory as long as a Unix one-line command can extract it. Here, the file "`ratio`" contains just the value of interest, so simply `cat`ting it does the job.

The possibility to run a custom "result extractor" makes collecting of results very flexible: anything can be made important. One can easily introduce new properties to observe at any later time, as long as they were recorded somewhere. Together with remote playgrounds (Section 7), one can re-interpret other people's experiments.

The file "`eman.results`" is useful on its own already. For instance, the user can quickly check if the top-scoring setup is still the same, e.g.:

```
grep ratio eman.results | sort -rn -k4 | head -n 1
```

For the purposes of the following section, we provide a snippet of the results file in Figure 4.

### 6.3. Tabulation of Results

Lonesome numbers do not have any meaning. In order to be able to interpret the observations and discuss them, individual results have to be compared and contrasted to other results. One practical issue is that a set of results can be dissected and contrasted in an endless number of ways.

`Eman` provides a succinct but extremely powerful tool for "putting relevant numbers next to each other". The technique is based on the "`eman.results`" file and one more user configuration file, "`eman.tabulate`". Running "`eman tabulate`" reorganizes the results based on the configuration and produces "`eman.niceresults`".

```
=== Compression ratios of different algorithms ===
(512k of random hex data)

TABLE
required: compress ratio
required: SIZE512
forbidden: OUTDATED LEVEL[2-46-8]
cols: CMD([^\s]*)
rows: LEVEL([0-9]) rsyncable
rowsort: CMDgzip
ENDTABLE
```

*Figure 5. Sample* “eman.tabulate” *configuration.*

#### 6.3.1. Prose with Automatic Tables

The file “eman.tabulate” is a regular text file. Any comments, observations or discussion can be simply written there. Eman copies everything verbatim, except for sections surrounded by lines saying “TABLE” and “ENDTABLE”. These sections will get expanded to tables of results. The number of tables in the file is not limited and each table can provide a different view of the results.

One can in principle use “eman.tabulate” as the LaTeX source of a scientific paper where tables are constructed automatically from the available results.

In the following sections, we describe how eman processes the configuration given in Figure 5 to obtain the table in Figure 6.

#### 6.3.2. Selecting Results to Show

The first stage of tabulation is the selection of lines from “eman.results” that should be listed in the table. This filtering allows to provide different views on the playground.

The filtering is achieved by two sets of regular expressions. Only the lines that contain all the “required” expressions and do not contain any of the “forbidden” expressions make it to the table.

Technically, the regular expressions are delimited by space in the “eman.tabulate” config, so a single “required:” line can specify several requirements. To match a space, one can use e.g. “\s”.

As each line of the results file contains a lot of details (see Figure 4), the filtering is quite powerful: we can even match e.g. the date in the step name to require steps inited during a particular day. In our example in Figure 5, we are interested in the “ratio” results of any “s.compression.*” step. The autotags provide the information

```
=== Compression ratios of different algorithms ===
(512k of random hex data)

Common properties: compress ratio SIZE512
Forbidden properties: OUTDATED LEVEL[2-46-8]

                CMDbzip2             CMDgzip
LEVEL1          .46033593750000000000 .55463281250000000000
LEVEL5 rsyncable                    - .52478125000000000000
LEVEL5          .43619140625000000000 .51907031250000000000
                .43503125000000000000 .51825585937500000000
LEVEL9                              - .51824414062500000000
```

*Figure 6. Sample results of the tabulation.*

about the file size that was used in the experiments and we require the 512kB tests. Finally, we avoid all steps in the `OUTDATED` state and we also exclude some compression levels (“`forbidden: LEVEL[2-46-8]`”) to make the table shorter.

#### 6.3.3. Constructing Row and Column Label for a Result

Each result (i.e. a single line from the results file) that survives the filtering is examined in order to construct its “`row`” and “`col`”umn label. The same regexp mechanism as above is used here, except now the successfully matched regexp is appended to the respective label.

In our example, the columns are simply the compression algorithms – these are found in the autotag starting with “`CMD`”. Rows are a little bit more interesting. In general, we want to see the compression level (“`LEVEL([0-9])`”), but in a few experiments, we also used the gzip flag “`--rsyncable`”, so we need to distinguish these runs. Adding a second regex “`rsyncable`” that may or may not be found in the result line makes the distinction.

The round brackets in the regexes express important parts of the match. The respective portion of the regex will be replaced by the actual string matched. So the “level” regex appears as a few distinct tokens like “`LEVEL1`”, or “`LEVEL9`” in the final table, see Figure 6. Without the round brackets (“`LEVEL[0-9]`”), we would get the same token for all lines, namely “`LEVEL[0-9]`”. This can be useful to skipping unimportant differences, i.e. specifying a “gappy pattern”.

The order of the regexes is also important, because labels are constructed left-to-right. So regexes that construct the beginning of row or column label need to appear in the configuration first.

Not all result lines match all row/column regexes. That is fine, the label is then simply shorter. As an example, we see the default run of the two compression algorithms where the row label is empty – no level was specified at all.

Not all settings are meaningful or used across all experiments. This is also fine, the cells will then contain just a dash. In our example, it is the "rsyncable" option, which is not available in bzip2, and the level-9 bzip2 experiment which we forgot to run for the purposes of Section 6.3.6.

There are a few other minor tricks for handling cases like multiple matches of the same regex, but these are beyond the scope of this article.

#### 6.3.4. Putting the Table Together, Solving Conflicts

Having established the row and column labels for each value, it is trivial to construct the table. Values sharing the column label will appear in the same column, values sharing the row label will appear in the same row. This gives us a two-dimensional view on the results.

If two or more distinct values share the same row *and* column label, `eman` reports a conflict and the user has two options. If such a conflict is not desirable then some regex (and perhaps also some tag or autotag) should be added to filter out unwanted values or put the conflicting values on different rows or columns. There are however cases where we have deliberately run the very same experiment several times and some randomness or outside condition leads to different results. In this case, one adds the following line to the table specification:

```
collectdelim:,
```

This switch instructs `eman` to indeed show all the results in a single cell, delimited by the given string (a comma in our example).

#### 6.3.5. Sorting Rows and Columns

Finally, the user can specify the full label of the column that should be used to sort the rows ("`rowsort`") and/or the full label of the row that should be used to sort the columns ("`colsort`").

Note that adding regexes that construct row and column labels can easily change the labels so sorting fails to find the given criterion.

#### 6.3.6. Back to Experimenting

`Eman` consults the file "`eman.results`" when resolving step specifiers. This neat trick allows to go directly back from the (tabulated) results to experimenting.

We can ask questions like: what exact configuration did I use to produce the compression ratio 0.5368:

```
eman tb --vars 5368
```

It is wise to double-check that the numbers we contrasted by putting them on the same line or column actually differ only in the properties we are mentioning. In bash, this amounts to inspecting the diff of the two tracebacks, e.g. in the editor vim:

```
vimdiff <(eman tb --vars 4361) <(eman tb --vars 5190)
```

It is also easy to use the cloning mechanism (Section 5.3) to start experiments whose results will fill missing cells. We pick an existing result from the given row (or column, whichever is more convenient) and apply the necessary change to it. We exemplify it by filling the level-9 compression experiment by bzip2 that was missing in Figure 6. The bzip2 run is derived from the corresponding gzip experiment:

```
eman tb 518244 -s /gzip/bzip2/ | eman clone --start
```

## 7. Team Experimenting

The command "eman add-remote" is implemented in a very light-weight fashion. The user provides the path to the remote playground and an alias – eman then simply creates a symbolic link to the directory in the local playground and registers the link in the file eman.subdirs.

Remote steps then become equivalent to steps in the local playground – they can be used in experiments, cloned and even modified (e.g. started, outdated) if the file system permissions allow it (otherwise, eman automatically switches to read-only mode).

Eman does not search the remote playground recursively (i.e. it does not explore its remote playgrounds), which makes this feature quite flexible; even circular dependencies are possible, although they do create a soft-link loop in the filesystem.

Commands such as "eman ls" or "eman select" list only local steps by default. To consider remote playgrounds, the option "--remote" has to be used. Eman can also display the playground of each step in the listing if "--dir" is given.

Finally, since step directories are no longer local subdirs of the playground, the command "eman path" is useful to get the full pathname of a step.

## 8. Related Tools

Two similar tools come from the MT environment: Ducttape and EMS. Ducttape (formerly LoonyBin; https://github.com/jhclark/ducttape; Clark et al., 2010) is functionally similar to the combination of eman and Prospector (included in eman package). The user specifies "hyperworkflows", packed sets of experiments, where a number of variables has a number of requested values. Hyperworkflows are actually more flexible than that, separate hyperworkflow branches can have different step structure. Given a hyperworkflow, Ducttape runs either the full Cartesian product of variable values or a subset of it based on some "realization plan". Implemented in Java, it originally provided only a graphical user interface but now there is also a command-line interface and a minimal web-interface available.

Experiment Management System (EMS; Koehn, 2010), is distributed with the Moses translation system (Koehn et al., 2007) and it is primarily intended for it. Its general management capabilities are again centered around distinct runs of the complete experiment. Data reuse is achieved by noticing that some partial output from a previous run is still valid. This is against our encapsulation objective.

Taverna (`http://www.taverna.org.uk/`) is a widely used complex workflow management tool. It introduces the Taverna language to describe workflows, provides a graphical user interface including an editor of workflows and various servers and clients for running workflows or providing services that can be used as processing blocks in workflows remotely. The remote processing is perhaps the biggest advantage: research institutes provide web-based services directly usable in user's workflows. Compared to `eman`'s 4k lines of Perl, Taverna's command-line tool is 151 MB. Taverna originated in bioinformatics but it is being used in many other fields of research. The only Taverna application in NLP so far are probably the PANACEA tools (`http://www.panacea-lr.eu/`) for compiling various linguistic resources from texts.

Cluster or grid computing environments, e.g. Pegasus (`http://pegasus.isi.edu/`), also have workflow managers like DAGMan (Couvares et al., 2007). These allow to express dependencies between jobs but focus on automatic recovery from job failures in an unreliable cluster environment, not on experiment variation or any interpretation of results.

## 9. Open Issues and Future Development

There are certainly limitations of the current version of `eman`. The most serious issue from the practical point of view is that the indexing of steps walks many directories and files.[2] With a larger number of steps, this becomes inconveniently slow. A principled solution would use clever incremental updates of only the bits that got invalid due to some change. Unfortunately, this is rather tricky: e.g. changing the autotag configuration would require to propagate new tags to existing steps etc. but `eman` does not get automatically called when the user edits the file "`eman.autotags`".

We have also mentioned, that some inspection and reuse of the *derivation history* for steps is desirable. This would allow further shortcuts in experimenting and new types of observations, e.g. why does the foobar switch make the baseline experiment faster but it slows down our improved setup?

Finally, `eman` has no visual output, but it would be quite easy to display the various dependencies between steps using e.g. the graphviz library (Gansner and North, 2000).

---

[2] `eman` accumulates an index of steps during regular operations. Full reindexing is required only occasionally and done upon request ("`eman reindex`" for the core index of steps and their variables, "`eman retag`" for autotag application and propagation and "`eman collect`" for the collection of results).

## 10. Conclusion

We presented eman, an open-source experiment manager for command-line Unix environment.

Hopefully, we highlighted and explained a couple of ideas that are generally useful for speed up and a better guidance of experimenting. We feel the following features are the most important ones: keeping detailed records, reusing intermediate results and reusing whole experiments by cloning new variants of them. We also provided a couple of suggestions for organizing and examining obtained results.

For readers interested in eman specifically, this article should provide a high-level overview spiced with example calls and commands.

## 11. Acknowledgment

The work on this project was partially supported by the grants P406/11/1499 of the Grant Agency of the Czech Republic, FP7-ICT-2011-7-288487 (MosesCore) of the European Union and 7E11042 of the Ministry of Education (EU project FP7-ICT-2010-6-257528).

## Bibliography

Clark, Jonathan H., Jonathan Weese, Byung Gyu Ahn, Andreas Zollmann, Qin Gao, Kenneth Heafield, and Alon Lavie. The Machine Translation Toolpack for LoonyBin: Automated Management of Experimental Machine Translation HyperWorkflows. *Prague Bulletin of Mathematical Linguistics*, 93:117–126, 2010. ISSN 0032-6585.

Clark, Jonathan H., Chris Dyer, Alon Lavie, and Noah A. Smith. Better hypothesis testing for statistical machine translation: Controlling for optimizer instability. In *Proc. of ACL/HLT 2011*, pages 176–181, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P11-2031.

Couvares, Peter, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow Management in Condor. In *Workflows for e-Science*, pages 357–375. Springer London, 2007. ISBN 978-1-84628-519-6. doi: 10.1007/978-1-84628-757-2_22. URL http://dx.doi.org/10.1007/978-1-84628-757-2_22.

Gansner, Emden R. and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11): 1203–1233, 2000.

Koehn, Philipp. An Experimental Management System. *Prague Bulletin of Mathematical Linguistics*, 94:87–96, Sept. 2010. ISSN 0032-6585.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open Source Toolkit for Statistical Machine Translation. In *Proc. of ACL 2007, Companion Volume Proceedings of the Demo and*

*Poster Sessions*, pages 177–180, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P/P07/P07-2045.

Och, Franz Josef. Minimum error rate training in statistical machine translation. In *Proc. of ACL 2003*, pages 160–167, Sapporo, Japan, 2003. ACL. URL http://aclweb.org/anthology-new/P/P03/#1000.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: A method for automatic evaluation of machine translation. In *Proc. of ACL 2002*, pages 311–318. Association for Computational Linguistics, 2002. URL http://aclweb.org/anthology-new/P/P02/.

Tamchyna, Aleš and Ondřej Bojar. No free lunch in factored phrase-based machine translation. In *Proc. of CICLing 2013*, volume 7817 of *Lecture Notes in Computer Science*, pages 210–223, Samos, Greece, 2013. Springer-Verlag.

**Address for correspondence:**
Ondřej Bojar
bojar@ufal.mff.cuni.cz
Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics,
Charles University in Prague
Malostranské náměstí 25
118 00 Praha 1, Czech Republic