Guix and Org mode, a powerful association for building a reproducible research study

Hands-on session

Marek Felšöci[*]

June 13, 2022

Contents

1 Foreword

Reproducibility of a research study in computer science has always been a complex matter. On the one hand, building exactly the same software environment on multiple computing platforms may be long, tedious and sometimes virtually impossible to be done manually. On the other hand, while the experimental method is usually explained in research studies, the instructions required to reproduce the latter from A to Z are often missing or incomplete.

In this edition of Tuto Techno, we will introduce the GNU Guix transactional package manager [4] together with the principles of literate programming [15] through Org mode [9, 13] and learn how we can take advantage of the association of these tools to build a reproducible research study. We will use the Guix package manager to ensure the reproducibility of software environments and Org mode to ensure an exhaustive, clear and accessible description of experiments, source code and procedures involved in the construction of experimental software environments, execution of benchmarks as well as gathering and post-processing of results.

The present document is organized as follows. In Section 2, we expose the goals and the motivations of this work. In Section 3, we describe the workspace which will be used during the

[*]marek.felsoci@inria.fr

hands-on session the instructions for which are given in Section 4. Finally, we provide some additional useful pointers in Section 5.

2 Goals and motivations

Our starting point will be an existing research study, relying on the `test_FEMBEM` [12] linear system solver test suite, in which we do not use Guix nor literate programming paradigm to ensure reproducibility. In the rest of the document, we refer to this version of the study as to the *Baseline setup*.

Unfortunately, the installation, module loading and build instructions provided in the repository have only been composed for a personal computer running *Debian GNU/Linux 11* and for the PlaFRIM high-performance computing platform [10]. They may vary on another machine or system configuration which may provide different package versions or not provide the required software at all. In an attempt to ease the reproduciblity of the software environment of the study, its author provides a *Debian GNU/Linux 11*-based Singularity image containing the necessary packages.

On the one hand, the container solution does not alleviate some of the most important concerns. For example, if we want to use a different version of one or more packages, we either have to modify the container interactively, which would make it even less reproducible, or re-build it from scratch, which can take lot of time if performed regularly. Moreover, because the container is based on an existing Linux distribution, we are limited to the versions of various core packages (compilers, MPI [1], BLAS [2], ...) provided by that particular distribution in that particular release unless we want to re-build and re-configure a good chunk of the software environment.

On the other hand, when it comes to reproducing the experiments, we have to settle for comments in source code and to a `README.md` file to know how to use the scripts dedicated to run experiments and post-process results.

To cope with these limitations, the idea is to manage the software environment with Guix and rely on the literate programming paradigm in an attempt to make the research study better reproducible from the point of view of both the experimental software environment and the experiments themselves. Using Guix, we can provide a self-contained, executable description of the whole software environment used to run our experiment. This precision is a crucial building block for reproducible scientific workflows, yet it is something READMEs and containers do not even approximate.

The goal of this session is thus to build a standalone git repository containing the same research study made entirely reproducible thanks to Guix and the literate description of the experimental environment, source code and methods in Org mode. In the rest of this document, we refer to this version of the study as to the *Advanced setup*.

Finally, we will learn how to make use of the Software Heritage project [11] to guarantee the availability of our work even in case of migration of the git repository or shutdown of the hosting platform, e.g. Inria Forge.

[1]Message Passing Interface - a message-passing library interface specification addressing primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each proces.

[2]Basic Linear Algebra Subprograms - routines that provide standard building blocks for performing basic vector and matrix operations.

3 Workspace

A dedicated project group has been created on the GitLab of Inria for the needs of this session, Tuto Techno Guix HPC, with the following structure:

- test_FEMBEM
  - *Baseline setup*
  - *Advanced setup*
- Guidelines
- Slides

The `test_FEMBEM` subgroup contains two versions of the same resarch study:

1. *Baseline setup*, which does not rely on Guix and Org mode for reproducibility and
2. *Advanced setup*, which does rely on Guix and Org mode.

The `Guidelines` repository contains the Org sources of the present document and `Slides` the Org sources of the introductory presentation together with Guix channel and manifest files allowing for a quick setup of the associated software environment. We do not report further on these repositories.

3.1 *Baseline setup*

This repository contains an experimental study relying on the open-source version of the `test_FEMBEM` solver test suite with the following structure:

- benchmarks
  - `definitions.csv`
  - `run.sh`
- figures
  - `short-pipe.png`
- public
- `.gitignore`
- `.gitlab-ci.yml`
- `README.md`
- `container.def`
- `plot.R`
- `references.bib`
- `study.tex`

In this case, we do not rely on Guix and literate programming using Org mode to ensure reproducibility of the research study. To build a software environment clode enough to the original for redoing the experiments, we can use either the combination of native system package manager and manual builds, as detailed in `README.md`, or the accompanying pre-built Singularity container defined in `container.def`.

As of redoing the experiments defined in `definitions.csv` using the dedicated `run.sh` shell script, we must settle for the instructions and explanations in the `README.md` file and comments in associated source code files.

The `plot.R` R script allows for generating figures based on experimental results into the `figures` folder. Finally, `study.tex` and `references.bib` represent the LaTeX source of the study manuscript and the referenced bibliography, respectively.

Note that the `public` folder is used by the continuous integration engine for publishing repository's static webpage hosted on GitLab pages.

3.2 *Advanced setup*

This repository contains the same research study as in *Baseline setup* repository. However, here we do rely on Guix and literate programming using Org mode to ensure reproducibility of the study. See the structure of the repository below:

- benchmarks
  - `definitions.org`
  - `run.org`
- figures
  - `short-pipe.png`
- public
- styles/RR
- `.gitignore`
- `.gitlab-ci.yml`
- `README.md`
- `channels.org`
- `macros.org`
- `manifests.org`
- `plot.org`
- `publish.el`
- `readtheorginria.setup`
- `references.bib`
- `reproducing-guidelines.org`

- `study-article.org`
- `study-research-report.org`
- `study.org`

The first thing we can observe is that all the source code files are written in Org this time. The Org syntax allow us to combine formatted text with blocks of source code. The latter can then be extracted (or tangled in the Org terminology) [7] into corresponding source files, e.g. `run.org` can be tangled into `run.sh` and so on. It is also possible to evaluate code blocks directly from within Org files [5]. Note that an Org file can also be easily exported [6] to various output formats such as PDF (through LaTeX), HTML, ODF, plain text, etc. We further detail the Org format later in Section 4.4.

We added some more files to the repository too. `channels.org` and `manifests.org` represent the specification of the Guix software environment of the study (see Section 4.3). `study.org` contains the study manuscript. `study-article.org` and `study-research-report.org` are wrappers for `study.org` representing an article version (as in *Baseline setup*) and an Inria research report version of the study manuscript. `reproducing-guidelines.org` combines the literate descriptions of all the sources in the repository and thus provides an exhaustive documentation of experiments, source code and procedures involved in the construction of related software environments, execution of benchmarks as well as gathering and post-processing of results.

`readtheorginria.setup` and the files in `styles` contain the HTML and LaTeX templates used for publishing the study manuscripts and the reproducing guidelines. We do not report further on these.

Finally, `publish.el` is an Emacs Lisp script for exporting the Org files into different output formats and publishing them into the `public` folder. As it is not associated with the core subject, we will consider this script as a black box during the hands-on session. Do not hesitate to ask more details about it though.

Note that `macros.org` provides Org macros [8] reused throughout the Org documents in the repository. This is similar to custom LaTeX commands.

The hands-on session will be based on this repository. The `master` branch contains the complete configuration we should have built by the end of the session. The `level0` branch represents the starting point for the participants to be completed during the session. For the participants joining us later or wanting to skip one or more phases, there are the other `levelX` branches corresponding to different levels of completion of the hands-on session.

4 Hands-on session

In the first place, we will put the *Advanced setup* study repository aside and familiarize ourselves with Guix and Org on simple examples.

4.1 Installing Guix

Here, we assume that we are running a third-party Linux distribution such as *Debian GNU/Linux* or *Ubuntu*. We can install the Guix package manager on top of that distribution without interferring with our primary package manager. To do so, we use an installing shell script that needs to be run with superuser privileges.

```
cd /tmp
wget https://git.savannah.gnu.org/cgit/guix.git/plain/etc/guix-install.sh
chmod +x guix-install.sh
sudo ./guix-install.sh
```

Then, we just need to follow on-screen instructions.

4.2 Running Guix for the first time

After the installation, we proceed with a short sequence of commands to ensure a smooth user experience with Guix onward. At the beginning, we install our first package using Guix, i.e. `glibc-locales` to allow the latter to switch locales.

```
guix install glibc-locales
```

Then, to be able to acquire new versions of installed packages, we will need to pull new version of Guix first. The following command can take a while to execute, especially when run for the first time.

```
guix pull
```

Once the process finishes, we need to follow the hint the command gives us and add the following lines to our `.bash_profile` or `.bashrc` to always get access to the most recent Guix generated by `guix pull`.

```
GUIX_PROFILE="$HOME/.config/guix/current"
. "$GUIX_PROFILE/etc/profile"
```

We also have to tell to our shell to use this new Guix.

```
hash guix
```

Finally, we can update our installed packages.

```
guix upgrade
```

To get information on the generation (version in Guix terminology) of Guix being used, we can use:

```
guix describe
```

4.3 Familiarization with Guix

Let us enter our first Guix environment containing four packages, `bash`, `cowsay`, `emacs` and `emacs-org`, using the `guix shell` command and launch a shell inside of that environment. We should not forget the `--pure` switch which prevents existing environment variables from pulluting the target Guix environment. Note that the `--norc` option to `bash` prevents the shell from loading our `.bashrc` file which could pollute the final environment despite passing the `--pure` option to `guix shell`.

```
guix shell --pure bash cowsay emacs emacs-org -- bash --norc
```

We can simply type `exit` to get back to our original shell. Also, we do not have to run an interactive shell inside of the environment. We can directly execute a given command like for example:

```
guix shell --pure bash cowsay emacs emacs-org -- cowsay "Hello world!"
```

The above should give us the following output:

```
 ______________
< Hello world! >
 --------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

4.3.1 Manifests

The `guix shell` command seems very convenient. However, let us imagine that we do not need only four but 26 packages in our environment. The command line would become quiet long, right? The good news is that we can instead put our list of packages into a file, referred to as manifest, then use the `-m` option to pass the manifest to our `guix shell` command line.

Manifest files [2] use the Scheme language [14] syntax which can be intimidating in the beginning. Fortunately, `guix shell` has recently got the `--export-manifest` option allowing one to automatically generate the manifest file corresponding to the environment specified on the command line. Let us thus create the manifest corresponding to our four package environment and save it to a Scheme file named `my-manifest.scm`.

```
guix shell --pure --export-manifest \
     bash cowsay emacs emacs-org > my-manifest.scm
```

Our manifest should look like this.

```
(specifications->manifest
  (list "bash" "cowsay" "emacs" "emacs-org"))
```

Finally, we can enter the target environment using the manifest file and retry a `cowsay` command.

```
guix shell --pure -m my-manifest.scm -- cowsay "Hello from the manifest!"
```

4.3.2 Channels

Note that our final environment will contain the latest versions of the specified packages available in the current revision of Guix. Well, maybe we are fine with that at this point but what happens if we want to enter the exact same environment a couple of weeks, months or years later? Maybe the packages will not be even available anymore.

Software packages in Guix are provided through dedicated git repositories called *channels* [3]. The official Guix channel `guix`, automatically set up in our Guix installation, currently provides 20,406 packages. However, many other channels are available, e.g. for specialized HPC software and so on. We will discuss the usage of multiple channels later. For the moment, let us concentrate on the default `guix` channel.

To ensure the same revision of Guix providing the same packages in the same versions, we can accompany our manifest with a channel file, also written in Scheme. In the latter we can specify the channel or channels to use together with the desired revision number, i.e. commit. We can obtain the currently used commit of the `guix` channel (and other channels, if any) by typing `guix describe`. The output should look like follows.

```
Génération 21   18 mai 2022 15:55:44    (actuelle)
  guix 69ec5ba
    URL du dépôt : https://git.savannah.gnu.org/git/guix.git
    branche : master
    commit : 69ec5baaf7aa6ed3ce5ecaad9bb66d122f91c3ec
```

Using the above information, we can create the corresponding channel file `my-channels.scm`.

```
(list
 (channel
  (name 'guix)
  (url "https://git.savannah.gnu.org/git/guix.git")
  (commit "69ec5baaf7aa6ed3ce5ecaad9bb66d122f91c3ec")))
```

Note that, we can directly obtain the list of channels currently used by the system in Scheme by typing `guix describe -f channels`. It gives us a handy starting point for building our own channel file, i.e. by changing commit numbers, branches or by adding or removing channels to or from the list.

Finally, to make our `guix shell` command execute using our channel file, we can use the `guix time-machine` command.

```
guix time-machine -C my-channels.scm -- shell --pure -m my-manifest.scm -- \
     cowsay "Great, a channel and a manifest file!"
```

Let us do some Org now!

4.4 Familiarization with Org

We choose to write the source code of scripts and various configuration files allowing us to design and automatize numerical experiments in respect of the paradigm known as literate programming [15]. The idea of this approach is to associate source code with an explanation of its purpose written in a natural language.

There are numerous software tools designed for literate programming. We rely on Org mode for the Emacs text editor [13, 1] which defines the Org markup language allowing to combine formatted text, images and figures with traditional source code. Files containing documents written in Org mode should end with the `.org` extension.

Extracting a compilable or interpretable source code from an Org document is called tangling [7]. It is also possible to evaluate a particular source code block directly from the Emacs editor [5] while editing. For example, this can be particularly useful for the visualization of experimental results.

Eventually, an Org document can be exported to various output formats [6] such as LaTeX or Beamer, HTML and so on.

Listing 1 shows an example of Org syntax. In this exceprt, there is some formatted text followed by a Python code block. The line starting with `#+PROPERTY` specifies that all of the source code blocks in that particular Org file should be tangled into a Python script file named `rss.py`. Figure 1 shows an HTML output corresponding to Listing 1.

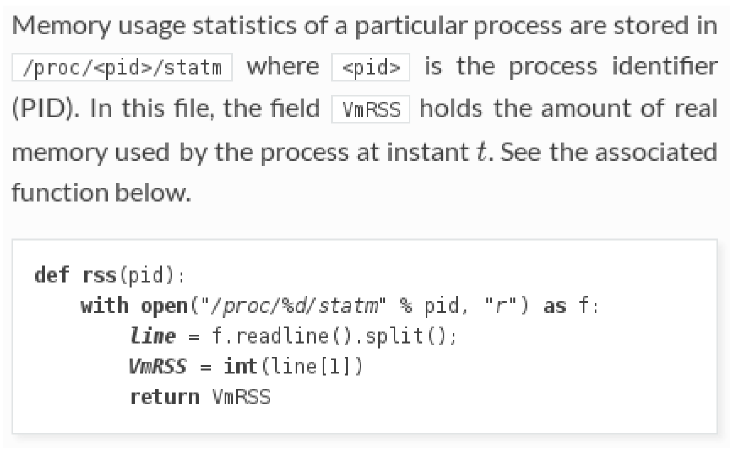
```
#+PROPERTY: header-args :tangle rss.py ...
...
Memory usage statistics of a particular process are
stored in ~/proc/<pid>/statm~ where ~<pid>~ is the
process identifier (PID). In this file, the field
=VmRSS= holds the amount of real memory used by the
process at instant $t$. See the associated function
below.

#+BEGIN_SRC python
def rss(pid):
    with open("/proc/%d/statm" % pid, "r") as f:
        line = f.readline().split();
        VmRSS = int(line[1])
        return VmRSS
#+END_SRC
...
```

Listing 1: Example of Org syntax featuring formatted text and a Python code block.

We will now compose an Org file by ourselves. Let us enter our previous Guix environment, open Emacs inside and create a file named `hello.org`.

```
guix time-machine -C my-channels.scm -- shell --pure -m my-manifest.scm -- \
     emacs --no-init-file hello.org
```

Then, we will try to describe the below short shell script in that Org file following the aforementionned example. Note that tangling of `hello.org` should produce a shell script file named


Memory usage statistics of a particular process are stored in `/proc/<pid>/statm` where `<pid>` is the process identifier (PID). In this file, the field `VmRSS` holds the amount of real memory used by the process at instant *t*. See the associated function below.

```
def rss(pid):
    with open("/proc/%d/statm" % pid, "r") as f:
        line = f.readline().split();
        VmRSS = int(line[1])
        return VmRSS
```

FIGURE 1: HTML output corresponding to the Org document excerpt in Listing 1.

`hello.sh`.

```
MESSAGE="Hello world!"

if test "$1" != "";
then
  MESSAGE="$1"
fi

cowsay "$MESSAGE"
```

If all went well, we should have something similar to this in our `hello.org` file.

```
#+PROPERTY: header-args :tangle hello.sh

This file describes a simple shell script ~hello.sh~. It begins by defining a
default greeting message.

#+BEGIN_SRC shell
MESSAGE="Hello world!"
#+END_SRC

However, if the user provides a custom message, we prefer to show this one
instead.

#+BEGIN_SRC shell
if test "$1" != "";
then
  MESSAGE="$1"
fi
#+END_SRC

Finally, let the cow say our message!

#+BEGIN_SRC shell
```

```
cowsay "$MESSAGE"
#+END_SRC
```

To save our modifications to `hello.org`, we can use `C-x C-s`. In order for the `#+PROPERTY` setting to take effect, we need to refresh the current buffer by selecting `Org > Refresh/Reload > Refresh setup current buffer` from the Emacs application menu.

Then, to tangle the shell script from our Org file, we can use this sequence of keystrokes `C-c C-v C-t`. To execute our shell script in our Guix environment, we close Emacs using `C-x C-c` and fire the following command.

```
guix time-machine -C my-channels.scm -- shell --pure -m my-manifest.scm -- \
     bash hello.sh "I can do Guix and Org now!"
```

4.5 Building a reproducible study

We are now ready for the core part of the hands-on session. We are going to make the research study from the *Baseline setup* repository better reproducible thanks to Guix and the literate programming approach in Org mode.

Before we begin, we need to connect to our Inria GitLab account, and fork the *Advanced setup* repository we are going to work with. It already contains all of the files required for it to work but some of them need to be completed.

Then, we make a local clone of our fork using the `git clone` command and navigate to the root directory of the clone. Now, depending on where in the hands-on session we want to join, we checkout the right branch to start from:

- `git checkout level0`: the channel definition (the very beginning),
- `git checkout level1`: the manifest definition,
- `git checkout level2`: the creation of a literate description of a source code file in Org,
- `git checkout level3`: the reproduction of the study in a Guix environment.

Note that whenever we need to edit an Org file during the hands-on session, we can use the following command.

```
guix shell --pure emacs emacs-org -- emacs --no-init-file <file-name>.org
```

4.5.1 Channels

The study relies on the `test_FEMBEM` solver suite which in turn depends on packages that are not available through the official `guix` channel. We will thus need two extra channels.

Fill the Scheme code block in `channels.org` with the follwing list of channels while respecting the syntax seen in Section 4.3.2. At the end, we can tangle the corresponding channel file `channels.scm` from within Emacs using `C-c C-v C-t`.

1. `guix`, the official Guix channel

- link: `https://git.savannah.gnu.org/git/guix.git`
- commit: `eb34ff16cc9038880e87e1a58a93331fca37ad92`

2. `guix-hpc`, the channel of the GuixHPC effort providing some commonly used HPC applications, e.g. solvers, runtimes, ...
   - link: `https://gitlab.inria.fr/guix-hpc/guix-hpc.git`
   - commit: `7506a50557beca54903fea496f3185b86c354e35`
3. `guix-hpc-non-free`, a companion channel to `guix-hpc` providing non-free libraries (MKL, CUDA, ...) and non-free versions of some of the packages provided by `gui-hpc` (PaStiX with MKL, ...)
   - link: `https://gitlab.inria.fr/guix-hpc/guix-hpc-non-free.git`
   - commit: `cb2c8ec608c85d0cfd61a215ca75b171a25a39ff`

4.5.2 Manifests

There are some packages dedicated exclusively to the execution of benchmarks and some other dedicated to the post-proccesing of results and the publication of manuscripts. We will thus have two manifest files in this study, `experiments.scm` and `post-processing.scm`. Both of them are described in a unique Org file named `manifests.org`. The manifest for post-processing results has already been written into the latter. Therefore, we well only concentrate on the most important, the benchmark execution software environment.

At first, we will need to compose the `guix shell` command allowing us to enter the correct environment. We begin by verifying whether a `test_FEMBEM` package is provided by one of the channels we specified earlier. For this, we can use the `guix search` command.

```
guix time-machine -C channels.scm -- search "test_FEMBEM"
```

In addition to this core package, we will need the following packages as well: `openmpi`, `openssh`, `sed`, `which`, `grep`, `coreutils` and `bash`. We can observe that the list of requested packages here is substantially shorter than the one in the `README.md` file in the *Baseline setup* repository of the same study. This is because we do not need to include the packages required to build `test_FEMBEM`, the package will be built in the appropriate environment by Guix automatically.

Once we have composed our `guix shell` command, we can verify whether it is working by running a quick `test_FEMBEM` test inside of the target environment like so.

```
guix time-machine -C channels.scm -- shell --pure <list-of-packages> -- \
     test_FEMBEM --fembem -nbpts 1000 -solvehmat
```

By default, all the packages in the environment that depend on a BLAS [2] library use the OpenBLAS implementation. However, for this study, we want to use Intel(R) MKL instead. To replace a dependency, or input in Guix terminology, in a package tree, it is possible to use the `--with-input` option of `guix shell`.

Modify your Guix command line like so.

```
guix time-machine -C channels.scm -- shell --pure --with-input=openblas=mkl \
     <list-of-packages> -- test_FEMBEM --fembem -nbpts 1000 -solvehmat
```

Once everything is working, we can export the corresponding manifest using:

```
guix time-machine -C channels.scm -- shell --pure --with-input=openblas=mkl \
     --export-manifest <list-of-packages>
```

Finally, we need to fill the Scheme source code block in `manifest.org` meant for the `eperiments.scm` manifest with the output of the above command and tangle our manifest files using `C-c C-v C-t`.

To enter the benchmark execution environment, we can now use:

```
guix time-machine -C channels.scm -- shell --pure -m experiments.scm -- ...
```

and to enter the post-processing and publishing environment, we can now use:

```
guix time-machine -C channels.scm -- shell --pure -m post-processing.scm -- ...
```

4.5.3 Org

In this section, we are going to create a literate description in Org of an entire source file from scratch. We can see in the *Advanced setup* repository that the `benchmarks/run.org` and `benchmarks/definitions.org` has already been completed. We are thus going to focus on `plot.org` meant for describing the `plot.R` script from the *Baseline setup* repository.

Fill in the `plot.org` file based on the contents of the original `plot.R` script. We can consider comments as surrounding formatted text and split the source code into R source code blocks. Do not hesitate to consult the pre-filled Org files to inspire you.

Once it is done, we should not forget to tangle the script into `plot.R`, within the *Advanced setup* repository this time, using `C-c C-v C-t`.

4.5.4 Reproducing the study

We are now getting to the most important challenge of the day. We are about to reproduce the study using Guix. Before going further, we have to tangle the sources from all of the Org documents in the repository. We have already done it for some of them. However, the following command line allow us to tangle all the Org documents in the repository recursively.

```
guix shell --pure git emacs emacs-org -- emacs --batch --no-init-file -l org \
     --eval '(progn (setq org-src-preserve-indentation t) (dolist (file
      ↪  (directory-files-recursively "." "\\.org$")) (org-babel-tangle-file
      ↪  file)))'
```

From this point, we can look into `README.md` in the *Baseline setup* repository and try to identify the command for running experiments and then the command for post-processing results involving the `run.sh` and the `plot.R` scripts, respectively. Then, try to run them from within the root of the *Advanced setup* repository and in the right Guix environment, i.e. using `channels.scm` and `experiments.scm` for running benchmarks and `post-processing.scm` for post-processing the results. We have seen the associated `guix` command lines at the end of Section 4.5.2.

Note that after the execution of benchmarks we should obtain a file named `results.csv` in `benchmarks/results` and after results post-processing we should obtain three scalable vector graphics *.svg figures under the figures directory, i.e. chameleon.svg, hmat-chameleon.svg and hmat-chameleon-error.svg.

Finally, to publish study manuscripts featuring our results, we can use this command.

```
guix time-machine -C channels.scm -- shell --pure -m post-processing.scm -- \
     emacs --batch --no-init-file --load publish.el \
     --eval '(org-publish "manuscripts")'
```

4.5.5 Reproducing guidelines

The aim of the reproducing-guidelines.org Org document is to provide all the information necessary to reproduce our study as well as to describe and explain all of the source code and procedures involved in construction of the experimental software environment, execution of benchmarks, post-processing of results and publishing of manuscripts. The document is actually a wrapper for all the Org documents in the repository describing source code files. In addition to that, Section 2 of the document provides the instructions and commands to reproduce the study.

Complete the two empty shell code blocks in the aforementionned section of the document with the instructions for running experiments and post-processing results used in the previous step.

At this point, if we re-publish the reproducing-guidelines.org document,

```
guix time-machine -C channels.scm -- shell --pure -m post-processing.scm -- \
     emacs --batch --no-init-file --load publish.el --eval '(org-publish "RT")'
```

we obtain a complete Inria technical report on how to reproduce our study and understand the source code involved in the process.

4.5.6 Software Heritage

The last step of the hands-on session consist of archiving our study repository on Software Heritage to ensure its availability in the long term.

To do this, we open the link https://archive.softwareheritage.org/save/ in our browser and simply follow the instructions on screen.

Once the archival process completes, the archive receives a unique identifier, e.g. swh:1:snp:79f450e0f43828f56f261d81b3e86aaab18362eb. The latter can be easily integrated into a study manuscript so the readers can quickly access all the information necessary for reproducing the study.

5 Pointers

In addition to the bibliography at the end of the document, the following pointers may be of interest for those who would like to learn further on how to use Guix and Org mode for Emacs:

- https://hpc.guix.info/ (Guix-HPC, reproducible software deployment for high-performance computing: channels, packages, events, . . .)

- https://cours-mf.gitlabpages.inria.fr/is328/tuto-chameleon.html (tutorial on how to use Guix or Singularity images produced by Guix on HPC platforms such as PlaFRIM)
- https://felsoci.sk/blog/posts.html (blog of Marek Felšöci with posts on Guix and Org mode usage - for work and for home)

References

[1] *GNU Emacs: An extensible, customizable, free/libre text editor — and more.* https://www.gnu.org/software/emacs/.

[2] *GNU Guix Cookbook: Basic setup with manifests.* https://guix.gnu.org/cookbook/en/html_node/Basic-setup-with-manifests.html.

[3] *GNU Guix Reference Manual: Channels.* https://guix.gnu.org/manual/en/html_node/Channels.html.

[4] *GNU Guix software distribution and transactional package manager.* https://guix.gnu.org.

[5] *Org mode documentation (Evaluating source code).* https://orgmode.org/manual/Evaluating-Code-Blocks.html.

[6] *Org mode documentation (Exporting).* https://orgmode.org/manual/Exporting.html.

[7] *Org mode documentation (Extracting source code).* https://orgmode.org/manual/Extracting-Source-Code.html.

[8] *Org mode documentation (Macro Replacement).* https://orgmode.org/manual/Macro-Replacement.html.

[9] *Org mode for Emacs.* https://orgmode.org/.

[10] *PlaFRIM: Plateforme fédérative pour la recherche en informatique et mathématiques.* https://plafrim.fr/.

[11] *Software Heritage.* https://www.softwareheritage.org/.

[12] *test_FEMBEM, a simple application for testing dense and sparse solvers with pseudo-FEM or pseudo-BEM matrices.* https://gitlab.inria.fr/solverstack/test_fembem.

[13] C. Dominik, *The Org Mode 9.1 Reference Manual*, 12th Media Services, 2018.

[14] R. Dybvig and J. Hébert, *The Scheme Programming Language, Fourth Edition*, 2009.

[15] D. E. Knuth, *Literate Programming*, Comput. J., 27 (1984), p. 97–111.