CiaoDevel Reference Manual

Edited by:
The Ciao Development Team

Table of Contents

1 Introduction

2 Reporting Bugs

Author(s): The Ciao Development Team.

When identifying a potential bug please report to the corresponding mailing list (`ciao-bugs@cliplab.org`) and specify enough information to reproduce the problem:

- affected code (e.g., bundle) and your configuration;
- specify your machine, operating system, compiler version, etc.;
- try to generate (if possible) a minimal working example (MWE);
- check (if possible) that it is not a duplicate;

Bugs are preferably stored at each bundle with the following structure:

```
BUNDLENAME/bugs/
 NODISTRIBUTE
 Fixed
 ...more specific directories...
 Unclassified
```

For example, the `bugs` directory of the `core` bundle looks like:

```
CIAOROOT/core/bugs/
 Fixed
 NODISTRIBUTE
 Pending
 Performance
 Unclassified
```

For internal development, bugs may also be filed at our Phabricator site (as tasks). Bugs may also be filed as github issues, although we recommend the other channels mentioned above.

Once fixed, bugs are moved to the `bugs/Fixed/` sub-directory.

3 Coding Style

Author(s): The Ciao Development Team.

3.1 Coding style for Ciao code

- Keep line width less than 80 characters (if possible).
- Do not leave lines ending with spaces or tabs (trailing whitespace).
- Any file should be ended by a new line.
- Use spaces instead of tabulations.
- Avoid predicates with the same name but different arity.
- Use descriptive module names, specially in libraries (to avoid clash with user-defined modules, since module names are global).
- Similarly use descriptive names for multifile predicates (they are global).
- When possible enumerate imported predicates explicitly.
- When contributing bug reports, they are typically stored at `<bundle>/bugs/Pending`. When fixed, bug entries are moved to `<bundle>/bugs/Fixed`. Place at `<bundle>/bugs/Unclassified` potential bugs whose status may be unknown.
- Make sure that you make proper usage of clause indexing (in Ciao it is done by default on the first argument of predicates) and cuts (`!/0`), for the expected instantiation modes for each predicate. Doing unnecessary backtracking and leaving choice points can introduce nasty performance bugs that can be hard to debug. Assertions can help in this task.
- Understand the algorithmic cost of operations (e.g., assert/consult is typically linear in the size of terms, var-nonvar unification is typically constant time, etc.)

3.2 Coding style for C Code

- In general, avoid `inline` functions (see Linux Kernel Coding Style (`https://www.kernel.org/doc/Documentation/CodingStyle`)) – unless you can collect evidence that they produce optimal code (size and speed) with the state of the art C compilers.
- Use macros (even big ones) when you need to inline code. A macro is much better than duplicated code.
- Do not use comma at the statement level (e.g. `int a, b;` or `foo(), bar();`) since that confuses the CPP macro expansion (when that code is wrapped inside `{}` and passed as argument).
- `goto` is not harmful for low level code (specially for automatically generated code).

3.3 Some helper Emacs tools

To highlight trailing whitespaces you can either:

- use the `highlight-chars` emacs package (available at MELPA) and add the following lines in your `.emacs`:

```
;; ------------------------------------------------------------------------
;; highlight tabs and trailing whitespace
(require 'highlight-chars)
(add-hook 'font-lock-mode-hook 'hc-highlight-tabs)
(add-hook 'font-lock-mode-hook 'hc-highlight-trailing-whitespace)
```

- toggle the `Show Trailing Whitespace` option on, which can be found at `Options > Customize Emacs > Browse Customization Groups` below:

```
[-]-\ Group Emacs
   [-]-\ Group Editing
    | [-]-\ Group Editing Basics
    |     `--- Option Show Trailing Whitespace
```

It is not adviced to remove trailing whitespace automatically since this can mess up the diff of your changes (and potentially lead to merge conflicts later on).

4 Git Workflow

Author(s): The Ciao Development Team.

4.1 General Advice

4.1.1 Q: How to undo the modifications I made on one file?

Sometimes one may want to reverse the modifications done on one file. This may appear because the conflict on this file has been marked as solved (e.g., one only resolves with mergetool some of the conflicts in a file and then `git` assumes that all conflicts have been solved) or that the assumptions made to solve the conflicts were wrong.

The command to revert the change made in a file `<file>` when resolving conflict is:

```
$ git checkout -- <filepath>
```

4.1.2 Q: What is an automatic merge commit and how to avoid those?

Automatic merge commits mess up with the changes history. Assume that while you were working on some feature in a separate branch, someone pushed a number of commits into the `master` and currently the repo looks like:

```
                          HEAD
                           |
1 -> 2 -> 3 -> 4  -> 5 -> 6
          \
           `-> 4b -> 5b
```

After a `git pull` you will have all commits in your repo, and in such case git usually creates an automatic merge that creates a new commit, such that 6->7 and 5b->7 (two parents).

```
                                HEAD
                                 |
1 -> 2 -> 3 -> 4  -> 5 -> 6 -> 7
          \                    /
           `-> 4b -> 5b -----`
```

Whereas with `git pull --rebase` the merge commits could be avoided and the commit history would loks like:

```
1 -> 2 -> 3 -> 4  -> 5 -> 6         HEAD
                          \          |
                           `-> 4b -> 5b
```

4.2 Merging Branches Manually

This section describes some useful tips when merging branches in git.

4.2.1 Q: How can I reduce the number of merge conflicts? How can I ease future merges?

There is no magic. To avoid conlicts, reduce conflict zone size and ease conflict resolution one has to make their code easy to merge, even if it works in `master`. To do so one should keep in mind that merge process works line by line (i.e. a line that has diverged in both branches is considered as conflict).

As merging branches is always more complicated than expected, consider again if a branch `mybranch` is really needed (e.g., if it can be replaced or combined with conditional compilation in C and Prolog to minimize the conflicts).

4.2.2 Q: How do I merge two branches?

To merge into a branch, your `HEAD` should be at the branch you want to merge into. If you want to merge into `master` a branch of yours with a lot of changes, you should first merge `master` into your branch (resolve the potential conflicts) and then merge your branch in `master` (it should have no conflict). When merging you should have no uncommitted modifications in your `HEAD`, in general the cleaner your `HEAD` is the smoother the merge will be.

E.g. To merge `master` into `<mybranch>` run:

```
$ get checkout <mybranch>
$ get merge master
```

4.2.3 Q: How to abort an uncommitted merge?

```
$ git reset --hard HEAD
```

4.2.4 Q: What is the best tool to resolve merge conflict?

When resolving conflicts it is highly recommended to use a tool that can show simultaneously all the different versions of the files involved in the merge:

- The older common ancestor (identified by `BASE`)
- The file in your `HEAD` before the merge (identified by `LOCAL`)
- The file in the other branch (identified by `REMOTE`)
- The file resulting from the merging the two branches (identified by the fact that it has the unmodified filename)

Some Ciao developers recommend `meld` (which is available in most Linux distributions and can be installed using `macports` or `homebrew` in macOS).

To configure `git` to use `meld` for resolving conflicts add the following in your `.gitconfig`:

```
[merge]
tool = mymeld
[mergetool "mymeld"]
cmd = meld --diff $BASE $LOCAL --diff $BASE $REMOTE --diff $LOCAL $MERGED $REMOTE
```

Then, to resolve all the conflicts with `meld` just run:

```
$ git mergetool
```

You get a window with a lot of tabs.

- The two first tabs allow you to have a view of the history of the file. They show the pair `LOCAL-BASE` and the pair `REMOTE-BASE`. This allows you to view how the file evaluated in the two branches.
- The third tab is the most important one. It shows the result of the merge (in the center) together with the `LOCAL` and the `REMOTE` in left and the right hand side. You have to modified (and finally save) the result of the merge (in the center column) of the third tabs. For simple conflict, it should be enough to consider the third tab only, but for complex conflict it is often necessary to go to the historic tabs (the two first ones) to understand really how the file diverges in the two branch and how it should be fix.

Warning: once you have save the file and quit `meld`, `git` considers the conflict has been resolved in this files, even if there is still mark in conflict in the head.

4.2.5 Q: What do I do after resolving a merge conflict?

Before committing your merge (even if there is no conflict), you should verify that everything is going well. You should compile the project from scratch. For instance, one library/predicate used by specific code of your branch may have changed name/location in the master. Also run the tests.

If you do something else that just resolving syntactical conflicts, write details about those changes in the merge commit message.

4.2.6 Q: How do I rebase locally before a merge?

`branch` is the branch to be rebased and `target` is the target branch (usually `master`).

> **Warning:** This process rewrites history, so **do not run it** unless you are sure no one else is using your branch and there are no branches starting at your branch. `git fetch origin target; git merge origin/target` is a less preferred alternative to this process.

1. Make sure the current branch corresponds to the branch you want to rebase: `git checkout branch`.
2. Fetch upstream changes: `git fetch origin target`.
3. As we squash commits on merge anyway, we can do so when rebasing onto `target` too to reduce the number of conflicts to handle: `git rebase --autostash -i origin/target` (`pick` first and `fixup` rest).
4. In order to allow the repository administrators to modify the commit message in GitLab, add an empty commit to the branch: `git commit --allow-empty`.
5. Push changes: `git push -v --force-with-lease origin branch`

5 Testing your changes

Author(s): The Ciao Development Team.

5.1 Testing your changes

After you make any changes please check thoroughly that nothing has stopped working. Specifically,

- Make sure that the documentation is up-to-date and correctly generated.
- Verify that new features have their associated tests.
- Run automated tests (`ciao test`).
- Run other tests in the `testsuite` bundle.
- Try out, to the extent possible, examples and code without automated tests, as well as tests from third-party code that depends on Ciao.

5.2 Testing documentation

The documentation should be as complete and correct as possible. Please check that every change you do can be documented (introducing syntactic errors is easy). In order to generate the documentation:

- Make sure that there are no (new) error/warnings in the output of `ciao build --docs`
- Verify that the documentation has been generated correctly. It should appear under `<CIAODIR>/build/doc` in `.info`, `.pdf`, and HTML formats. Have a look at it, verify that the links (in info and HTML formats) are not dead.
- Check that every library in the distribution has a well-formed manual page. A component should appear here iff it goes into the distribution; otherwise the documentation will be out of sync with respect to the distribution.

5.3 Generating and testing distributions

See the `ciao-distro-tools` bundle to learn how we produce *multirepostory* source distributions from the development *monorepository*.

5.3.1 Generating a distribution

`ciao` (super-command) offers some commands to generate packaged bundles for several platforms with the command `gen_pbundle --kind=Kind`, where `Kind` is one of:

- `win32`: Creates a Windows installer.
- `rpm [--option=value...]`: Creates an `RPM` package. For more details please read the documentation for the `gen_pbundle --kind=rpm` module. One or more options may be added; main ones are:

  `vendor_independent=yes`
  : Creates packages that should work in all mainstream `RPM` Linux distributions. This is the default. If disabled, packages are only guaranteed to work in the same distribution (vendor) they were generated for.

- `bin`: Creates the binary packages (using `gz` and `bzip2`).
- `bin_tgz`: Creates the binary package compressed using `gz`.
- `bin_tbz`: Creates the binary package compressed using `bzip2`.

- `src`: Creates the `tar` packages in `gz` and `bzip2` compressed formats.
- `tgz`: Creates the `tar` packages in `gz` compressed format.
- `tbz`: Creates the `tar` packages in `bzip2` compressed format.

Note: distribution generation may ignore some files under the source tree based on some marks (`NOCOMPILE`, `NODISTRIBUTE`, `NOINSTALL`, etc.). See a more complete description of at `library(source_tree)`.

5.3.2 Testing a distribution

Distributions should be installed and tested in clean environments. Ideally, this means using a machine with no Ciao installed. Otherwise, you should clear your environment variables of any trace of paths and environment variables pointing to Ciao stuff. 'which ciao' will point to executables – remove from `PATH` the corresponding paths. Any environment variable starting by `CIAO` (`env | grep CIAO`) should also be removed prior to compilation.

We **must** compile and test in all supported architectures.

Perform the installation **exactly** as it is stated in the `INSTALLATION` file.

For global installations, compile and install under a user and test with another user, in order to make sure that everything gets compiled and needed. You can make the installation as root and the testing as yourself. Make sure every step of the installation works.

Some (OS-specific) additional tests:

- Compile and run the tests, and check that everything works.
- All commands (`ciao`, `ciaosh`, `ciaoc`, etc.) must be reachable and they must belong to the distribution we have installed.
- Windows: from windows file explorer: check left-button menus
  - `.pl` files: load `foo.pl` into top-level shell and make executable (must generate `foo.bat` and `foo.cpx` files)
  - `.bat` and `.cpx` generated files: double-click on them and check that foo program works
- Unix-like systems:
  - The `ciao-env` command should be patched with valid paths (it is used from the shell initialization files).
  - Check that the info file works well. This can be done executing `info` and verifying that an info entry for Ciao has been placed, and also running `info ciao`. It should also be visible with `M-x info` from emacs.
  - Execute `man ciao` to check that the man page has been generated and configured successfully.
  - `ciao doc BUNDLE` should open the HTML manuals for BUNDLE
  - `.pdf` files must also be in place.

Other tests specific to the Emacs Mode:

- Verify (`C-h i`) that the Ciao info files *for the version you installed* are visible.
- Load `<CIAODIR>/examples/hw.pl`, check that buffer coloring works (substitute the Ciao path by the directory where the sources you dowloaded are).
- Check that the Ciao menu options and tool bar appear on top of Emacs window.
- `C-c l` to load the file into Ciao top-level and call `main/1` to check basic top-level functionality.
- Put the cursor over `write/1` and hit `C-c tab` to go to the Ciao documentation (in info format).

- Load `hw.pl` in debug mode (`C-c d`), run `main/1`, go through the execution until its end.
- Move the cursor to last line of `main/0`, and do `C-c S b` to check breakpoints. Run `main/1` again, and press `l` to check that the execution stops at the correct program point.

6 Code Reviews

Author(s): The Ciao Development Team.

This describes the Ciao code review workflow using **Phabricator** and **Arcanist**.

6.1 How to Submit Code for Review

6.1.1 Step 1: Prepare your working directory

By default Arcanist assumes the *One Idea, One Commit* approach for each task. In practice this means that your working directory should not contain any local changes with respect to the `HEAD` commit when you start working on a new task. Should you have any changes you would like to keep, use '`git stash:`'

```
[git stash] # optional step, only if you have changes to tracked files
git checkout master
git pull origin master
```

6.1.2 Step 2: Create a local branch you will be working on

To begin, prepare your working directory by creating a branch with the name of the task that you will be working on. Assuming that your default branch is '`master`' and the task is T42, do:

```
arc branch T42 # alternatively: arc feature T42
```

The '`arc branch`' command creates a local git task branch and switches to it. You should name your task branch the same as the Task you are working on. If you do so, Arcanist will automatically link your Diff and the Task together.

Working with multiple branches: the task branches are set up to track the local branch you are currently working on. If you accidentally create a task branch from a wrong local branch, simply delete the task branch with '`git branch -d ...`', check out the correct local branch and repeat the '`arc branch`' command. It also seems to be possible in some cases to change the local branch manually (`https://stackoverflow.com/q/23920380`) or '`arc diff`' against a particular local branch (`https://stackoverflow.com/q/19819252`).

6.1.3 Step 3: Create a code revision (AKA a `Diff`)

6.1.3.1 Revisions associated with a specific task

This is the default workflow.

After you finish working on the task on the Task branch, you commit the changes and start a code review:

```
git commit -a "commit message"
# or add the files individually and commit them as:
#   git add FILE(s)
#   git commit -m "commit message"
arc diff
```

The commit message should be a one-liner (preferable no more than 80 characters) summarizing the changes. It must begin with a tag that specifies which part of the system was changed (e.g. (tests), (core), (rtchecks), fix(lpdoc), etc.).

The 'arc diff' command will open a text editor (provided in the EDITOR environment variable) with an extended commit message template. The first line(s) should correspond to the one-liner summary entered earlier. Fill the rest of the lines with the detailed information about the changes.

The Summary and Test fields are mandatory for filling (in some cases N/A can be put into Test field iff the changes do not affect any executable code). The Summary should include a more detailed explanation in several sentences or bullet points about the new changes/fixes in the Diff. The rest of the fields (Reviewers, Subscribers, Projects) can be filled later through the web interface.

When you save the commit message in the editor and exit it, Arcanist submits the changes to the Phabricator server for review and updates your local commit to include the URL of the revision on the server.

Note: the detailed message entered at this point will be used also in the web interface of Phabricator. You can manually introduce changes to it there (e.g., add/remove reviewers, associate projects, edit/update the text, etc.). This version will also be used when you commit your final changes to the master branch, so do not worry if you entered something wrong. Just make sure that the commit message in the web interface is the one you want before the final commit.

6.1.3.2 Revisions with no task

In some exceptional cases (e.g., for isolated changes related to a very specific task when there has been some previous agreement on the changes) it is acceptable to create a code revision without associating it to a specific task. However, to avoid complications during landing changes ('arc land') all the new changes should be implemented on a local branch, NOT on master. The workflow does not differ significantly for the developer and is the same for the reviewer.

To create such revision it is enough to execute the 'git commit' and 'arc diff' commands.

To close such revision it is enough to execute the 'arc amend' (or 'arc close-revision') and 'git push' commands.

6.1.4 Step 4: Respond to Reviews or Update a Revision

If a reviewer requests changes to the revision and you agree with those, implement them and submit a new code revision. To do this you introduce the changes as a new commit:

```
git commit -m "new changes description"
arc diff
```

Arcanist will send new changes back to the server for another round of review.

At the point of committing the final changes (see 'arc land' below), all commits in the differential revision will be flattened to just one commit with all the changes. The messages entered at this point are mostly for self-orientation, and the developer can go back easily to previous states of the local branch with the traditional 'git checkout' commands. See also the documentation for the 'arc diff --plan-changes' option below.

Note: make sure you never merge anything from the master branch (or other branches) onto your local branch while you are working on it. The SHA of the last commit on your local branch should not change until you 'arc land' changes.

If you had to roll back your commit before submitting an updated Diff and arc lost the track of it, you can still associate it with the specific revision with:

```
arc diff --update <revision>
```

Alternatively, a brand new revision can be created with:

```
arc diff --create
```

6.1.5 Step 5: Landing Changes

Once a reviewer requests no more modifications, the changes should be pushed to the repository with:

```
arc land
```

If no conflicts occur Arcanist merges the changes into master, deletes the local working branch, and pushes the changes to origin. Note that `arc land` can be executed for a Diff that was not accepted by any reviewer, but it is not advised to do so.

If conflicts occur, in most cases they are caused by unsuccessful merging of the commits that happened while you were working on the task. In most cases it should suffice to do the following:

```
#  fix conflicts
git commit
arc land
```

If this does not work, a rebase can be performed before landing:

```
git merge --abort
git fetch
git rebase -i origin/master
# ... fix merge conflicts
git add
git rebase --continue
# ... fix more conflicts should they occur, until the code is synced
arc land
```

6.2 How to Review Code

The goal of the code reviewer is to identify:

- Weird coding styles.
- Code in intermediate state (unless absolutely necessary).
- Non-English comments, identifiers, strings (unless absolutely necessary).
- Missing `NODISTRIBUTE`, `NOCOMPILE` in directories that are not ready for distribution.
- Failure in automated tests.

The following subsections describe the code reviewing actions with the Phabricator web interface and the Archanist commands.

6.2.1 Pre-commit reviews with Differential

6.2.1.1 Checking out the Phabricator branch (e.g., D24) with the changes

To try the new changes:

- Do '`git checkout`' to the Diff point (or '`git checkout master`' if you have most of the recent commits), and
- create a local branch with '`arc patch D24,`'
- compile, test, ...

(This branch can be deleted later with '`git branch -D arcpatch-D24`'.)

6.2.1.2 Suggesting modifications

- To annotate the changes with inline comments got to Phabricator and perform *click and drag* on the respective line numbers.
- To request changes fill the text area at the bottom of the page in Phabricator with instructions or extended comments, select *Request Changes* action from the drop-down list and hit the *Submit* button.

6.2.1.3 Updating the branch after changes

One cannot really update the branch with the patch: instead for each new update of a Diff the old patch branch should be deleted, and a fresh one created from the `master` branch, i.e.:

- Switch to master with '`git checkout master`'.
- Delete the (now outdated) branch with '`git branch -D arcpatch-D24`'.
- Check out the branch again (which now will contain the updates): '`arc patch D24`'.

Repeat the actions above until the changes look good.

6.2.1.4 Accepting Changes

To accept changes select in Phabricator the *Accept Revision* action from the drop-down list and hit the *Submit* button.

6.2.2 Post-commit reviews with Audit

For the commits that were pushed/landed without prior review it is still possible to make a code review through Phabricator's web interface:

- from the main page navigate to the `Audit` section in the tools list;
- select the commit of interest, review it, annotate it with comments if needed;
- in the *Add Action...* menu at the bottom of the web page either *Accept Commit* or *Raise Concern*.

6.3 Some Useful `arcanist` Commands

> **Note**: invocations of the `arc` utility require a working Internet connection.

- '`arc feature`': Lists the available branches and their revision status (like `git branch` but adding extra information).
- '`arc diff --plan-changes`': Updated the current Diff but does not request a review from its reviewers, useful for incomplete temporal saves (e.g., daily saves).

Also, `arc` understands some keywords in commit messages (see this web page (`https://secure.phabricator.com/T5132`) for details):

- attach a revision or a commit to a task with `Ref T42` in the summary/commit message;
- close a task when pushing a commit with `Fixes T42` in the summary/commit message;

These commands also work for regular commits (e.g., pushed via `git push`, not only during `arc land`).

7 Producing new releases

Author(s): The Ciao Development Team.

Depending on the number changes, it may be necessary to raise the version number (see `GlobalVersion`, `GlobalPath`, `version/1` at `Manifest.pl`, etc.).

Given an odd minor number version of the system (e.g., development `v1.1`), the release plan consists on selecting some new features, test and fix the system as required, compose a changelog, increment the version to an even number (e.g., stable `v1.2`), save this point in the source history, and increment again the version number to an odd number (e.g., development `v1.3`).

This is detailed as follows:

- Identify the new features and important changes for the new release (the changelog). This usually means going through the Git log and grouping all the changes in a way similar to what appears in `core/doc/CHANGELOG.pl` for previous versions.
- Iterate through: test existing, new, and improved features; check that installation from source (both local and global installs, in all supported platforms) work; check that distributions work; fix bugs if needed, or postpone/drop new features.
- When in a satisfactory state, add an updated changelog to `core/doc/CHANGELOG.pl` and increment the version number. Commit and push the changes.
- Tag the commit (omit COMMIT_ID to use HEAD):

```
git tag -a vMajor.Minor.Patch COMMIT_ID -m 'Version jumped to Major.Minor.Patch'
git push origin vMajor.Minor.Patch
```

- Make sure that distributions are up-to-date, update the website, brochure, and announce the release to `ciao-users` mailing list.
- Increment the version number again to an odd number for the new development version.

8 Adding Bibliographical References

Author(s): The Ciao Development Team.

8.1 Citations in documentation

LPdoc allows citing and referencing papers specified in one or more `.bib` files.

Please use citations for code that originated from (or was written together with) formal papers and technical reports. In such cases many details about the motivation, related work, design, technical details (and proofs) are not necessary within the source code documentation.

Nowadays it is preferable to write code that works (and that is straightforward to use for the target audience) over producing over-detailed documentation that mixes irrelevant (for the user) implementation details with usage. In cases where the technical information does not fit into a paper, please describe it in clearly identified sections or parts of the manual.

8.2 Adding new bibtex entries

If you need to add new bibtex entries please proceed as follow:

- add new entries to `clip.bib` or `general.bib` in the group's bibtex SVN repo (the same as for papers):

  `svn+ssh://cliplab.org/home/clip/SvnReps/bibtex`
- generate the documentation (you should see some unresolved citations warnings)
- use the `update-common-bib.sh` script (follow instructions) to update `common.bib`
- review the changes and commit the updated `common.bib` file

8.3 Future improvements

Note that the instructions above may change in the future (e.g., we may generate a single .bib file for each bundle or move/integrate the script somewhere else).

- Generate and use a .bib per bundle.
- Integrate the script. This is a very specific feature that may not be very useful for other Ciao users. Perhaps it fits better into the helper helper programs used to generate distributions, publish to github, etc.
- Parts of this functionality may fit into `bibutils`
- Use (our) `bibutils` instead of (third-party) `bibtool`

References

(this section is empty)

Library/Module Index

C

G

N

R

T

Predicate Index

(Index is nonexistent)

Property Index

(Index is nonexistent)

Regular Type Index

(Index is nonexistent)

Declaration Index

(Index is nonexistent)

Concept Index

(Index is nonexistent)

Author Index

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, authors, etc., are referred to in the text of the document.