Source Code Repository Work Process

These procedures describe the internal development process for Accellera Working Groups. It is largely based on the development and maintenance process used for the proof-of-concept implementation of SystemC. Legal and formal procedures are documented at www.accellera.org/about/policies.

1. Repository Organization

The central source code repository for Accellera working groups is privately hosted at GitHub (http://github.com). The Accellera repositories are private to the Accellera (OSCI-WG) organization and can be found at: https://github.com/OSCI-WG/

Members of Accellera can request with the necessary access rights to the Accellera (OSCI-WG) organization and can clone the repositories via SSH here:

```
git clone -o osci-wg git@github.com:OSCI-WG/  [repo name]
```

To obtain access to Accellera repositories, Accellera members can contact the chairs of the Working Group of their interest (www.accellera.org/activities/working-groups). Requestors will need to include their GitHub account name. Requestors can set up their Github account for free at www.github.com. The member company affiliation and email must be included in the Github user profile so that the user can be easily identified as an Accellera member.

Note: Using an explicit name of the 'remote' (for example, `-o osci-wg`) is recommended to allow using the default remote name `origin` for a personal fork where you can push your changes by default, see Section 2A “Basic Branch Setup” below.

Comprehensive documentation about Git, a distributed version control system, can be found in the Pro Git book available online. Since Git is 'distributed', it is a very natural choice for the distributed development process needed for the collaboratively evolving proof-of-concept implementation of SystemC.

To contribute changes to the different repositories, it is recommended to create personal (or company-based) forks of the repositories on GitHub and push the proposed changes (bugfixes, features, etc.) there. These forks are also only accessible to members of Accellera. Details of the intended work-flow are described in the next Section 2A “Basic Branch Setup”. It is convenient to add this GitHub fork as a remote to your local clone of the repository:

```
cd <repo>/
git remote add origin git@github.com:<your-account>/<repo>.git
git branch --set-upstream master origin/master
```

Any changes can then be pushed to GitHub using:

```
git push [options] [<repository>] [<refspec>...]
```

- If you omit the `<repository>`, the default destination is the remote of the current branch (or `origin`).

- The `<refspec>` basically follows the format `<local-branch>:<remote-branch>`, or just `<branch>`, if both are the same.
- Omitting the `<refspec>` pushes all branches with 'matching' remote branches to the repository.

A basic cheat sheet containing an overview of the general Git commands and workflow can be found online here.

2. Development flow

A) Basic branch setup

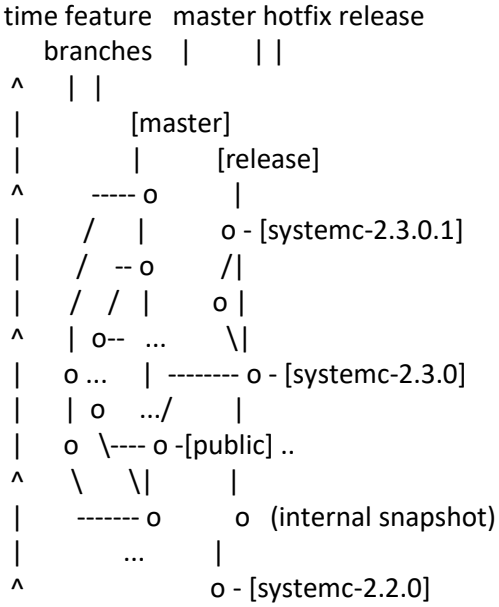
The main idea is to have two main branches, one for the development, the “Master”, and one for the releases. The “Master” is the latest and greatest `HEAD` of the material under development. This is where all the new features and fixes go. The “Release” branch is used to create the release tarballs, both internal and for public snapshots.

For “regressions”, the Release branch is more or less just a pointer to the latest revision of a snapshot (or release). It is still useful to keep a named branching point, in case of required hotfixes.

For the “core library”, the Release branch is to be different from the Master branch. The idea is to fully track the contents of the released tarball. This requires the following changes compared to the Master branch:

- The Automake generated files are added to this tree
- Accellera internal files are stripped (`.gitignore`, internal documentation, etc)

To prepare a release, the Master branch would then be merged into the Release branch, the automake files would be updated (if necessary) and the clean working tree could be used as a baseline for the tarball (e.g., via `git-archive(1)`). Details are described in the next Section 3 “Release Management”. The history of the (core library) repository could then look as shown in the following sample graph (time progresses upwards):



It is usually sufficient to keep the two branches Master and Release, and cherry-pick hotfixes for emergency releases directly on top of the Release branch. For convenience, an additional `public` branch could be used to mark the branching point for the last release.

If more sophisticated version branches are needed, a development model like the well-known A Successful GIT Branching Model" by Vincent Driessen can be deployed. Not all aspects of this model are expected to be needed for an implementation such as SystemC where only a single (i.e., the latest) public release of the Kernel is maintained.

B) Adding a Feature Set

The development of a new contribution in form of a feature or a complex bug fix is best done in a new feature branch, which is forked and checked out from the Accellera Master branch:

```
git checkout -b <company>-<feature-xyz> master
```

Then code up the new contribution. Please try to facilitate code review by other Accellera members by logically grouping your changes into one commit per addressed issue. For the commit messages, consider following these suggestions:

Note - Commit messages: Though not required, it is a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. Tools that turn commits into email, for example, use the first line on the "Subject:" line and the rest of the commit in the body.

During the development of the contribution, the Master branch may receive other commits. In that case, consider rebasing the commits in your feature branch onto the `HEAD` of the Master branch to keep the history clean. Once the contribution is ready for review by the working group, push the feature branch in your fork of the respective repository on GitHub:

```
git push <your-github-fork-remote-name> <company>-<feature-xyz>
```

Then, send a pull request either manually or via GitHub to initiate the code review by the working group members. The summary can be manually generated to be sent to the WG reflector by:

```
git request-pull master git@github.com/<account>/<repo>.git \
        <company-feature-xyz>
```

To review the proposed contributions, one can either browse the repository or add the remote location to a local clone of the repository

```
# add the fork to your set of "remotes"
git remote add <remote-name> git@github.com/<account>/<repo>.git
git fetch  <remote-name>

# examine differences
git diff master..<remote-name>/<company-feature-xyz>
git log <remote-name>/<company-feature-xyz>
```

After the contribution is accepted, it will be merged into the working group's Master branch by the responsible source code maintainer. This should be done with an explicit “merge commit”, to keep the individual contributions separated:

```
git merge --no-ff --log \
  <remote-name>/<company-feature-xyz>
```

Instead of fully merging the contribution, the maintainer may choose to select individual commits or to re-base the feature branch on an intermittently updated Master. He may also request additional changes to be done by the submitter. In that case, the submitter may need to merge recent changes to the Master branch into his feature branch before carrying out the requested changes.

After the contribution has been fully merged into Master, the feature branch in the local and Github fork may be deleted.

```
git branch -d <company-feature-xyz>     # delete local branch
git push  origin :<company-feature-xyz>  # delete remote branch
```

C) Tracking release tarballs

To track the release tarballs (both WG-internal and public ones), a tool named “Pristine-tar” is used. It enables tracking archives directly within the repository.

When looking at the branches (e.g., by using `gitk --all`), you may notice an additional, separate branch called `pristine-tar`. This branch contains the metadata for the pristine-tar tool. `pristine-tar` enables tracking original archives with minimal overhead, since only small binary deltas instead of the full tarballs are stored within the repository.

Note: Currently, only Linux and Mac OS X are supported platforms for `pristine-tar`. Porting to Windows (MinGW) is to be done in the future, which should be possible with reasonable effort to support the required use cases within the Accellera working groups.

Note: The use of the `pristine-tar` tool is entirely optional since the archives can be downloaded directly from the GitHub repository based on the tags: https://github.com/OSCI-WG/<repository>/releases

D) Basic workflow

```
# adding an archive (done by the maintainer)
pristine-tar [-m message] commit systemc-2.3.1.tgz release
git push origin pristine-tar

# retrieving an archive
pristine-tar checkout systemc-2.3.0.tgz
```

The maintenance of the `pristine-tar` branch (i.e., adding new archives to the `pristine-tar` branch) is done by the WG chairs during the release management (see Section __).

E) Maintaining a private set of branches

Companies may be interested in maintaining their own, in-house flow to align the internal development of a derived implementation, while being able to pick fixes from an Accellera Working Group's tree (and hopefully) contributing fixes and features back to the proof-of-concept implementation.

For this purpose, members may employ the already mentioned successful branching model by Vincent Driessen. The company can branch its own development branch, e.g., `develop-<company>` from the already tracked working group development branch Master in his clone of the WG repository. The user is then able to integrate commits on the WG development branch by merging it into his company development branch.

Bug fixes to be contributed back to the WG consist usually of one or several isolated commits. They need to be selected from the company's development branch into a new branch created from the WG development branch:

```
git checkout -b <vendor>-fix-<bug> origin/master
git cherry-pick <commit>...
```

Once the bug fix branch is ready, it should be pushed into the company's github account and a pull request created, as described in Section B “Adding a Feature Set”.

A new feature consists usually of a series of commits developed in a dedicated feature branched of the company's or WG's development branch. Only in the first case, a rebase on the top of the WG's development branch is necessary. To this end, branch first from the feature branch:

```
git checkout -b <vendor>-<new-feature> <private-feature-branch>
git rebase [-i|--interactive] --onto origin/master develop-<vendor>
```

3. Release Management

To prepare a new release tarball, the following set steps are to be performed by the maintainer.

a) Prepare the release in the Master branch.

Before creating a release snapshot, the documentation and version information in the package should be updated within the Master branch. This includes files such as:

- `ChangeLog`, `RELEASENOTES`, `README`, `INSTALL`
- `src/sysc/kernel/sc_ver.h`,
- `src/tlm_core/tlm_version.h`

During the release preparation phase, other functional changes should not be added/merged to the Master branch.

b) Update the Release branch.

```
# switch to release branch
git checkout release
```

```
# merge master branch
git merge --no-commit master
git rm <new-internal-file...> # drop new or changed "private" files
git commit -m "merge master branch for x.x.x release"
```

c) Update the Autoconf (and other auto-generated) files.

```
autoreconf -if # or config/bootstrap
git add -u     # add changed files
git status     # check for untracked files
git add <new files to distribute>
git commit -m "update autogenerated files for x.x.x release"
```

d) Tag the release revision.

To keep track of the release snapshots, the revisions used for creating the release tarballs should be marked with an *annotated* and optionally signed Git tag.

```
# git tag -a -m "<package> <version>" <version> <refspec>
git tag -a -m "SystemC 2.3.0" 2.3.0 release
```

The tagname should contain the `<version>`, following the versioning rules of the IEEE. There are three standard formats:

- `x.x.x_beta_<isodate>` for beta/internal versions
- `x.x.x_pub_rev_<isodate>` for public review versions, and
- `x.x.x` for public release versions.

The tag should be on the Release branch, to enable the automated tarball creation in the next step.

e) Create the release tarball.

- `git archive` can then be used to create the release tarball
- `git describe` can be used to obtain the correct tarball name based on the current tag.

```
PACKAGE="`basename $(git rev-parse --show-toplevel)`" # or directly 'systemc'
VERSION="`git describe release`"
git archive -o ${PACKAGE}-${VERSION}.tgz \
        --prefix=${PACKAGE}-${VERSION}/ release
```

Note that even without a tag, a quick-shot release of the release branch can be generated this way. The resulting archive can then be added to the `pristine-tar` branch to keep track of the release history:

```
pristine-tar commit ${PACKAGE}-${VERSION}.tgz release
```

Note: The use of the `pristine-tar` tool is entirely optional since the archives can be downloaded directly from the GitHub repository based on the tags, for example:

<https://github.com/OSCI-WG/systemc/releases>
<https://github.com/OSCI-WG/systemc-regressions/releases>

f) Publish the release.

Upload the archive to the WG area for internal review and push the changes to GitHub.

```
git push osci-wg \
    master release pristine-tar \
    <version>
```

Note that the tag needs to be pushed explicitly.

4. Issue tracking

Open issues (bugs, cleanups, features) related to the proof-of-concept implementation of SystemC/TLM are tracked in GitHub's issue tracking system:

<https://github.com/OSCI-WG/systemc/issues> (core library)
<https://github.com/OSCI-WG/systemc-regressions/issues> (regression tests)

Issues are grouped (by using labels) in the following categories for different parts of the implementation:

- `core` - SystemC core language, i.e. everything in `sc_core`
- `datatypes` - SystemC datatypes, i.e. in `sc_dt`
- `tlm` - TLM-1.0, TLM-2.0
- `infrastructure` - build system(s), scripts, etc.

Additional labels are used to classify issues according to their severity (10 highest), according to the following guidelines:

10	`critical`	Show-stoppers that must be fixed, affects all (or at least most) platforms and violates fundamental specifications for most applications.
09	`serious`	At least one of the explicitly supported platforms is affected and causes significant problems for many applications
06	`medium`	Covers an area, where the standard may not be clearly specified. May require changes to external/standard API
05	`feature`	New feature proposal, beyond the current standard. Includes internal (and external, providing adoption by an IEEE WG) API changes.
04	`-errata`	Inconvenience (errata) for users of many platforms, workaround available. Solution may require internal API changes.
02	`documentation`	Documentation inconsistency or insufficiency (e.g. whitepaper unclear or misleading), no code changes.
01	`inconvenience`	Inconvenience (workaround available), for some platforms
00	`cosmetic`	Changes addressing performance or clarity of implementation, no API changes.

The discussion on issues usually starts on the LWG reflector or during the LWG meetings. After an initial consensus on the "validity" of the issue, the issue is added to the issue tracking system, a classification is done (including a target milestone), and preferably a responsible person is assigned.

5. References:

This process was first developed for the Accellera SystemC implementation v1.0, June 2013.
An HTML-rendered copy of this document can be found at https://github.com/OSCI-WG/release-flow-test/blob/master/docs/DEVELOPMENT.md.

6. Authors

The following people developed and documented these procedures on behalf of Accellera:

- Philipp A. Hartmann, Intel philipp.a.hartmann@intel.com
- Torsten Maehne, torsten.maehne@bfh.ch

Please report and updates or error to Lynn Garibaldi, Accellera, lynn@accellera.org.