Continuous Delivery With Ick

Lars Wirzenius

Abstract

Ick is a command line tool written by and for its author to build, test, and deliver his own projects as Debian packages. The release versions of the packages get uploaded to the author's public APT repository and/or to Debian itself.

The design of Ick has been greatly influenced by the author's personal needs, computing resources, and general outlook on life.

Contents

1 Introduction **3**

2 Installation **5**

3 Setup **6**

4 A simple project **8**

5 Building Debian packages: a discussion **10**
5.1 Assumptions 10
5.2 The assumed workflow 10
5.3 Pipeline overview 11
5.4 Differences from Debian 12
5.5 CI versus release builds 12
5.6 Version numbers 12
5.7 The build environments 13
5.8 The git repository 13
5.9 Git branches and Debian packaging files 14
5.10 Release tar archive 14
5.11 Debian source package 14
5.12 Debian binary packages 15
5.13 Uploading to an APT repository 16

6 The `cleanly` helper tool **17**
6.1 Extract project metadata 17
6.2 Create upstream release tar archive 18
6.3 Create Debian source package 18
6.4 Build a Debian binary package 19

7 Building Debian packages with Ick **21**
7.1 Single repository and branch (“a la liw”) 21
7.2 Separate repositories for upstream code and package (“a la Daniel”) 23

8 Yarn scenario step implementations **25**
8.1 Checking that the target address is set 25
8.2 Git repository setup 25
8.3 Git tagging 26
8.4 Ick file handling 26
8.5 Running Ick 26
8.6 Creating a git repository 28
8.7 Running “cleanly” 31
8.8 Checking “cleanly” output 31
8.9 Checking if a file exists 31
8.10 Checking for package files in the APT repository 32

Chapter 1

Introduction

Ick is a continuous integration tool written for its author. It might be useful for others, and if so, good. Ick's purpose in life is to allow its author to run builds and build-time tests, and cleanly build Debian packages, in a number of environments. The environments are different releases of Debian, running on different hardware architectures. A constraint on Ick is that its author only has a laptop, no server, and doesn't want Ick to start running things at unexpected times.

Ick in summary:

- a command line line tool, invoked by its user at opportune moments
- reads a YAML file that specifies the target environments and the projects to build
- supports only git as a version control system
- connects to targets with ssh only
- runs one build at a time to avoid overloading the author's laptop
- aims to be simple and small to avoid being a maintenance burden

Ick does not:

- manage environments
- configuration management and orchestration tools should be used with ick
- have a web interface
- provide an HTTP API
- run as a daemon in any way
- have a lot of configurability

- try to replace popular CI tools

Ick was written because its author had to suffer his Jenkins installation breaking one time too many, and not finding anything else that was trying to be as simple as his needs are.

All the things Ick is not, Ick can become, but not if the author has to make that happen. If you want to use Ick and you need Ick to add features to do so, you will need to write the code yourself.

Chapter 2

Installation

The author provides Debian packages at http://liw.fi/code. Follow instructions provided there.

Alternatively, you can check out the Ick source and run it directly from there (`./ick`). You'll need the PyYaml (http://pyyaml.org/) and cliapp (http://liw.fi/cliapp) Python libraries installed.

The author only supports Debian, but is willing to consider clean patches for portability. This includes patches for expanding this chapter to document installation on other systems.

Chapter 3

Setup

Ick has a number of command line options. Run `ick --help` to see them all. See the **cliapp**(5) manual page for information about using configuration files.

An "ick file" specifies the target environments to use, the projects to build, and the location where to store state between runs. It is in YAML and looks like this:

```
EXAMPLE
state: /home/liw/ick/state
targets:
    sid_amd64:
        address: ick@debian-sid-amd64
        pbuilder-ci-tgz: /var/cache/pbuilder/ci.tgz
    wheezy_i386:
        address: ick@debian-wheezy-i386
        pbuilder-ci-tgz: /var/cache/pbuilder/ci.tgz
projects:
    obnam:
        git: /home/liw/obnam/obnam
        branch: master
        pipelines:
        - shell
        - debian-ci
        - debian-release
        commands:
```

```
- ./check -yu
```

Ick will retrieve the source for each project, from the given git repository and branch, copy it to each target, and run the specified commands.

You need to set up each target environment yourself. This includes making sure Ick can access it over ssh, and that all the necessary tools are installed on the target, as well as all the build-dependencies of each projects. Ick does not do this for you: there are many system administration tools for configuration management to take care of this problem, or you can do it manually. (FIXME: Some day, when the author has mastered ansible or something like it, add a link to a git repo with the Ick related configuration for the configuration management tool.)

Ick requires the following to be installed on the target:

- a Unix-like operating system (probably Debian)
- an ssh server
- rsync
- Python 2.6 or 2.7
- python-cliapp (see http://liw.fi/cliapp)

For building Debian packages, the target must be Debian and have the following installed:

- pbuilder

Things will fail otherwise.

Chapter 4

A simple project

This chapter shows how to use Ick to test a simple project on one target. It does this using the yarn scenario testing language.

```
SCENARIO build simple project
```

For running these tests, the caller must set the `ICK_UNSTABLE_TEST_TARGET` and `ICK_JESSIE_TEST_TARGET` environment variables to addresses that can be reached by ssh and where `sudo pbuilder` can be run without interactive password entry.

```
GIVEN the ICK_UNSTABLE_TEST_TARGET variable is set
AND   the ICK_JESSIE_TEST_TARGET variable is set
```

First of all, we need to have a project we build. We call the project FOO. For this example, it can be empty, since we don't actually need to build anything.

```
GIVEN a minimal project repository in FOO.git
```

We also need an ick file that specifies the target machine and the project. We use `localhost` as the target: we can't assume the existence of any other machine. Obviously, `localhost` needs to have been set up correctly; most importantly, the user running Ick must have ssh access to it, preferably without having to type a key passphrase or a login password all the time.

```
GIVEN an ick file KITTEN.ick containing
... {
...     "state": "KITTEN.state",
```

```
...     "targets": {
...         "test": {
...             "address": "$ICK_UNSTABLE_TEST_TARGET",
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/ci.tgz"
...         }
...     },
...     "projects": {
...         "FOO": {
...             "git": "FOO.git",
...             "branch": "master",
...             "pipelines": ["shell"],
...             "shell": "echo BUILDING FOO\ntest -d .git\n"
...         }
...     }
... }
```

The `test` command is there because we want to verify that the source repository has been cloned.

Note that we use a very silly command to build the project. The only thing we are showing is that Ick runs the build command, not that actual building works. If Ick can run one shell command, it can run any of them, so actual build commands would also work.

Then we run Ick and check that the output contains the output of the build command.

```
WHEN user runs ick KITTEN.ick
THEN ick build log for FOO in KITTEN.state contains "BUILDING FOO"
AND there is 1 build log for FOO in KITTEN.state
```

If we now run Ick again, it shouldn't do anything, since nothing in the git repository has changed.

```
WHEN user runs ick KITTEN.ick
THEN there is 1 build log for FOO in KITTEN.state
```

There we go. That's all you should need to know to run builds and build-time tests for a simple project. This doesn't produce Debian packages, but for simple cases that's not needed.

Chapter 5

Building Debian packages: a discussion

Ick projects may be marked as type `debian`, which means that instead of running arbitrary shell commands, Ick runs a built-in sequence of commands to build Debian source and binary packages. This sequence, or pipeline, is meant to be similar to what any free software project would do if they build Debian packages themselves. However, given the many variations in how projects are run, this pipeline is not guaranteed to work for more than the simple projects the author develops.

5.1 Assumptions

The Debian pipeline in Ick only supports non-native packages using the "3.0 (quilt)" source package format.

5.2 The assumed workflow

Ick is being written for its author, and to support its author's development workflow. This section outlines that workflow. The author may change the workflow in the

future, but to do that, a working CI system is a necessity, so it won't happen until Ick works for him.

- The author is both upstream and Debian packager of several pieces of software. Some of the packages get uploaded to Debian. All of them get uploaded to the author's own APT repository.
- The upstream code and the Debian packaging code are kept in the same branch. They are maintained more or less in sync. The author is currently happy with having the upstream tarball contain the `debian` directory in the source tree.
- The author writes programs that are simple to build and package, and do not need a `make dist` type of operation for release tarball generation. The files to go into the release tarball are kept in git as is, and none of them are generated files.

5.3 Pipeline overview

The build process starts from a git commit with all the source files are to be built.

1. A release tar archive is built by exporting the files from git.
2. A Debian source package is built (`.dsc`, `.orig.tar.xz`, `.debian.tar.xz`).
3. Debian binary packages (`.deb`) are built for each target (CPU architecture, Debian release).
4. The source and binary packages are uploaded to an APT repository specific for the ick

The rest of this chapter covers things in more detail.

Note that this pipeline does not cover running unit tests, or other build-time tests. The Debian packaging for the software being built may or may not run such tests (it probably should), but this pipeline does not run such tests outside of the Debian package build.

5.4 Differences from Debian

Debian builds each version of software once, and copies the built packages to different stages of their releases: uploads to go "unstable", and are copied first to "testing" and then to "stable". This means that the same exact build gets used at each stage. This works fine for Debian, but it's not appropriate for outsiders. Outsiders usually want to release each version of their software for several Debian releases, and that requires building it separately for each.

As an example, the Ick author also develops Obnam, and when making a release, wants users of the current Debian stable release to be able to use that version of Obnam. In addition, he wants to support the previous Debian release ("oldstable") and the unstable version. Further, he wants to do this not just for releases, but for continuous integration, so he learns as soon as possible if he make changes that might, for example, not work in the Debian oldstable release.

Apart from that, the build tools and policy compliance should match those of Debian.

5.5 CI versus release builds

The build process for CI builds versus release builds should be identical, except for the choice of version numbers. A CI build should happen in exactly the same way as a release build. Otherwise there's a chance that the release build won't work. There's also no need for them to be different.

However, it's important that CI and release builds to not mix. It's important that users don't use a CI build by mistake, or that a release build has a CI build in its build environment.

5.6 Version numbers

The pipeline needs to know the version number of the software. This is what Debian calls the upstream version. In addition, the pipeline needs to know the

Debian packaging version, and it will need to amend the full Debian version automatically builds.

A CI build should probably not assume that the upstream version or the Debian packaging version gets maintained manually in the git repository. It is just a fiddly detail that is easy to forget. Release builds must, of course, be able to rely on that version.

For example, the master branch (used for CI builds) might have the upstream version as that of the previous release, assuming that releases are prepared in the master branch and tagged there. Alternatively, the release might be prepared in a different branch, and master has no version number specified.

FIXME: What's the best way? I only need one for my own projects, and everything else can be handled later.

5.7 The build environments

The target environments must be clean. We'll use the **pbuilder** tool to do the actual builds, which provides clean environments, assuming the separation of CI and release builds. However, maintaining pbuilder "basetgz" tar archives is outside the scope of Ick.

5.8 The git repository

The source code is extracted from a git repository. The build pipeline starts from a commit, which may be indicated with a commit id, a tag. or as the tip of a branch.

For release builds, the commit should be indicated using a signed, annotated tag. The tag is created by the developer or release manager, not by the build software. Indeed, creating a suitably named tag should be what triggers a release build.

For CI builds, the tip of a branch is the way to go.

5.9 Git branches and Debian packaging files

FIXME: It is unclear to me whether it's best for me and in general to keep the Debian packaging in one branch (master) or separate branches. If the latter, things may need to be merged (manually or automatically), which is riskier and/or more work. But it may be ugly to litter master with packating for one distro (but it's what I do now). Needs thinking.

5.10 Release tar archive

The first step of the build pipeline is to create a release tar archive. It is the opinion of the author that despite current fashion release tar archives are still necessary. For many reasons it is important to be able to archive the exact source code that is used for a build, and a tar archive is a good way to do that. A signed git tag in a git repository is much more complicated to manage, and requires more special tooling to trust. A single tar archive per release is very easy to keep track of and distribute.

The tar archive is done for CI builds as well as for releases.

FIXME: Debian version is not -1 or other reason to not re-gen orig.tar.xz. Get orig.tar.xz from APT repo instead of generating it?

5.11 Debian source package

The Debian source package is created separately from building the binary packages. The same source package is used for all building of binaries of the same version.

A Debian source package consists of several files. When using the "3.0 (quilt)" source package format, the files are:

- `orig.tar.xz` — the release tar archive ("the upstream tarball").
- `debian.tar.xz` — the Debian packaging changes to be added to the release files (usually the `debian/*` files, but may include changes to upstream files).
- `.dsc` — a description of the source files, in particular naming the other files and showing their checksums.

The source package is built on the first target specified in the ick file.

5.12 Debian binary packages

Given a Debian source package, the binary packages are built separately for each target. This requires copying the source package to the target, and running the build there.

Binary packages need to fiddle with version numbers, or they will all build files with the same name. Given a simple source package `foo_1.2-1.dsc`, all the binary packages for amd64 would be called `foo_1.2-1_amd64.deb`, even if they're build for different Debian releases. To avoid such collisions, the binary builds append the Debian release version to the version: `foo_1.2-1.debian7_amd64.deb` for the "wheezy" release, versus `foo_1.2-1.debian8_amd64.deb` for jessie. (FIXME: where do we get the "debian7" from?)

The builds for the Debian "unstable" release do not append anything to the version number. This is so that the author can upload those versions to Debian, if he chooses.

Debian binary packages can be architecture independent. For any given Debian release, such packages should be built only once. That one build produces a package that can be used everywhere. Building it elsewhere would be wasteful, and also result in package filename collisions. When setting up this build pipeline, we choose a target for each Debian release for building binary independent packages.

The build of a Debian binary package is thus:

1. Unpack the source package.
2. If not building for Debian unstable, add a `debian/changelog` entry with a new version number with the Debian release appended. The changelog entry should say something like "Build for $target".
3. If on the chosen target for architecture independent packages for a given Debian release, build all binary packages. Otherwise, build only architecture specific binary packages.

5.13 Uploading to an APT repository

Ick sets up an APT repository for each state directory. Once the source package and all binary packages are built, they are uploaded to the APT repository, but not anywhere else. Ick manages the repository itself, automatically.

FIXME: The repository has separate pockets for CI and release builds: `unstable` is for releases, `unstable-ci` is for CI builds. CI builds will use both pockets, but release builds only the release pocket.

FIXME: Further builds should use that APT repository.

FIXME: signing packages?

Chapter 6

The `cleanly` helper tool

The Debian package build pipeline contains a number of steps. To allow the steps to be easily combined and reordered, the are not integrated deeply into the Ick program, but provided by a separate helper program instead, called `cleanly`. This de-couples the pipeline and Ick, and allows the pipeline to be used with other CI systems as well. This section describes how `cleanly` works.

6.1 Extract project metadata

The first thing we need to do is determine, preferably automatically, what the name of the project is and what its version is.

```
SCENARIO cleanly performs pipeline steps
GIVEN a source code repository for project foo version 3.2-1

WHEN user runs, in foo,
... cleanly --ci --buildno 1 --debian-release debian7 get-name
THEN the output is "foo\n"
```

In CI mode, we want the upstream version number to be mangled, by appending the current build number. This avoids having to rely on the developers to update the version number diligently. For cleanly to do this, we need to tell it the build number.

```
WHEN user runs, in foo,
... cleanly --ci --buildno=5 --debian-release=debian8 get-upstream-ver
THEN the output is "3.2.0ci5\n"
```

In release build mode, however, we want no mangling.

```
WHEN user runs, in foo,
... cleanly --release --debian-release debian8 get-upstream-version
THEN the output is "3.2\n"
```

6.2 Create upstream release tar archive

We create a tar archive by extracing the source code from git. For CI builds, the mangled upstream version is used. Note how the mangled upstream version gets `.0ci{buildno}` appended to it: the `.0` is to make sure the mangled version comes after `3.2` but before `3.2.1` using Debian version comparison semantics.

```
WHEN user runs, in foo,
... cleanly --ci --buildno 42 --debian-release=debian7 tarball
THEN file foo-3.2.0ci42.tar.xz exists
```

For release builds, it's the declared upstream version.

```
WHEN user runs, in foo,
... cleanly --release --debian-release debian7 tarball
THEN file foo-3.2.tar.xz exists
```

6.3 Create Debian source package

Next up is the Debian source package. We assume the upstream project provides the Debian packaging files in the same branch. Since we're doing a CI build, the upstream version number gets mangled and the Debian version is always just 1 with the target's Debian release appended.

```
WHEN user runs, in foo,
... cleanly --ci --buildno 42 --debian-release debian7
... --debian-codename unstable dsc
```

```
THEN file foo_3.2.0ci42.orig.tar.xz exists
AND file foo_3.2.0ci42-1.debian7.debian.tar.xz exists
AND file foo_3.2.0ci42-1.debian7.dsc exists
```

For a release build, once again, no version mangling. (Boring to read? Imagine how boring this is to write. I need writing lessons.)

Actually, that's a lie. The release build *will* mangle versions, by adding the upload target's release number to the Debian part of the full version number. This is so that we can build an upstream version separately for each supported Debian release (e.g., Debian 7 and 8).

```
WHEN user runs, in foo,
... cleanly --release --debian-release debian7 dsc
THEN file foo_3.2.orig.tar.xz exists
AND file foo_3.2-1.debian7.debian.tar.xz exists
AND file foo_3.2-1.debian7.dsc exists
```

However, if the release name is `unstable`, no mangling should happen. (Really.)

```
WHEN user runs, in foo,
... cleanly --release --debian-release unstable dsc
THEN file foo_3.2.orig.tar.xz exists
AND file foo_3.2-1.debian.tar.xz exists
AND file foo_3.2-1.dsc exists
```

6.4 Build a Debian binary package

This is where things get interesting. We build a Debian binary package, given a source package. (Note that for this scenario, we fake the actual build to avoid having to have `root` access.)

```
WHEN user runs, in foo,
... cleanly --test-mode --ci --buildno 42 --debian-release debian7 deb
THEN file foo_3.2.0ci42-1.debian7_all.deb exists
```

Similarly in release builds.

```
WHEN user runs, in foo,
... cleanly --test-mode --release --buildno 42 --debian-release debian
```

```
THEN file foo_3.2-1.debian7_all.deb exists

WHEN user runs, in foo,
... cleanly --test-mode --release --buildno 42 --debian-release unstab
THEN file foo_3.2-1_all.deb exists
```

Chapter 7

Building Debian packages with Ick

This chapter describes how to actually use Ick to build Debian packages. We'll use Ick to actually build a toy package. This assumes we have the necessary access to localhost.

7.1 Single repository and branch ("a la liw")

First, we'll do a simple case where the Debian packaging and the upstream source code are kept in one repository and branch.

```
SCENARIO build Debian packages from one branch
```

First of all, we need a git repository with some source code.

```
GIVEN a source code repository for project foo version 3.2-1
```

Then we need an Ick file.

```
GIVEN an ick file foo.ick containing
... {
...     "state": "foo.state",
...     "targets": {
...         "ci_unstable": {
...             "address": "$ICK_UNSTABLE_TEST_TARGET",
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/ci.tgz",
```

```
...             "debian_release": "unstable",
...             "debian_codename": "unstable"
...         },
...         "release_jessie": {
...             "address": "$ICK_JESSIE_TEST_TARGET",
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/release.tgz"
...         },
...         "release_unstable": {
...             "address": "$ICK_UNSTABLE_TEST_TARGET",
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/release.tgz",
...             "debian_release": "unstable",
...             "debian_codename": "unstable"
...         }
...     },
...     "projects": {
...         "foo": {
...             "git": "foo",
...             "branch": "master",
...             "pipelines": ["debian-ci", "debian-release"]
...         }
...     }
... }
```

Then we do the build.

```
WHEN user runs ick foo.ick
```

As a result, there's several resulting packages that should exist. First of all, a CI build for unstable:

```
THEN the APT repository for foo.ick contains
... foo_3.2.0ci1-1.unstable.dsc
THEN the APT repository for foo.ick contains
... foo_3.2.0ci1-1.unstable_all.deb
```

We tag a release and build it. Because our test package is architecture independent, there aren't versions for each architecture separately.

```
GIVEN a git tag foo-3.2 on tip of master in foo
WHEN user runs ick foo.ick
THEN the APT repository for foo.ick contains foo_3.2-1.dsc
```

```
THEN the APT repository for foo.ick contains foo_3.2-1_all.deb
THEN the APT repository for foo.ick contains foo_3.2-1.debian8*.dsc
THEN the APT repository for foo.ick contains foo_3.2-1.debian8*_all.de
```

7.2 Separate repositories for upstream code and package ("a la Daniel")

Some people prefer to keep Debian packaging and upstream code entirely separate. In this case, the repository with packaging contains only the packaging, and it contains the contents of the `debian` directory, not the directory itself. In other words, the repository contains `changelog` and `rules`, not `debian/changelog` and `debian/rules`.

To accomodate this, Ick allows a project to specify any number of git repositories and branches that are stitched together to form a source tree.

```
SCENARIO build Debian packages from separate repositories
```

First of all, we need a git repository with some upstream source code, but no packaging, and a separate repository with the corresponding packaging.

```
GIVEN an upstream source repository for project foo
AND a packaging repository called deb for project foo version 3.2-1
```

Then we need an Ick file.

```
GIVEN an ick file foo.ick containing
... {
...     "state": "foo.state",
...     "targets": {
...         "ci_unstable": {
...             "address": "$ICK_UNSTABLE_TEST_TARGET",
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/ci.tgz",
...             "debian_release": "unstable",
...             "debian_codename": "unstable"
...         },
...         "release_jessie": {
...             "address": "$ICK_JESSIE_TEST_TARGET",
```

```
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/release.tgz"
...         },
...         "release_unstable": {
...             "address": "$ICK_UNSTABLE_TEST_TARGET",
...             "pbuilder-ci-tgz": "/var/cache/pbuilder/release.tgz",
...             "debian_release": "unstable",
...             "debian_codename": "unstable"
...         }
...     },
...     "projects": {
...         "foo": {
...             "gits": [
...                 { "git": "foo", "branch": "master", "root": "." },
...                 { "git": "deb", "branch": "master", "root": "debia
...             ],
...             "pipelines": ["debian-ci", "debian-release"]
...         }
...     }
... }
```

Then we do the build.

```
WHEN user runs ick foo.ick
```

As a result, there's several resulting packages that should exist. First of all, a CI build for unstable:

```
THEN the APT repository for foo.ick contains
... foo_3.2.0ci1-1.unstable.dsc
THEN the APT repository for foo.ick contains
... foo_3.2.0ci1-1.unstable_all.deb
```

We tag a release and build it. Because our test package is architecture independent, there aren't versions for each architecture separately.

```
GIVEN a git tag foo-3.2 on tip of master in foo
WHEN user runs ick foo.ick
THEN the APT repository for foo.ick contains foo_3.2-1.dsc
THEN the APT repository for foo.ick contains foo_3.2-1_all.deb
THEN the APT repository for foo.ick contains foo_3.2-1.debian8*.dsc
THEN the APT repository for foo.ick contains foo_3.2-1.debian8*_all.de
```

Chapter 8

Yarn scenario step implementations

This chapter defines the various scenario steps for Yarn so they can be actually executed. Surprise! You didn't know you were reading a test suite! If you're only interested in this document as a manual, you can skip this chapter.

8.1 Checking that the target address is set

```
IMPLEMENTS GIVEN the (.+) variable is set
env | grep -F "$MATCH_1="
```

8.2 Git repository setup

Set up git repository. We add a dummy README file so that the repository has a `master` branch.

```
IMPLEMENTS GIVEN a minimal project repository in (.+)
git init "$DATADIR/$MATCH_1"
cd "$DATADIR/$MATCH_1"
touch README
git add README
git commit -m "Initial commit"
```

8.3 Git tagging

Add a tag to a repository.

```
IMPLEMENTS GIVEN a git tag (\S+) on tip of (\S+) in (\S+)
cd "$DATADIR/$MATCH_3"
git tag -a -m "A git tag" "$MATCH_1" "$MATCH_2"
```

8.4 Ick file handling

Create an ick file.

```
IMPLEMENTS GIVEN an ick file (.+).ick containing (.*)
# Expand all $foo references.
expand_env_vars()
{
    local temp="$(mktemp)"
    printf '#!/bin/sh\ncat <<EOF\n%s' "$1" > "$temp"
    sh "$temp"
    rm -f "$temp"
}

expand_env_vars "$MATCH_2" > "$DATADIR/$MATCH_1.ick"
```

8.5 Running Ick

Run ick, capturing its output for later inspection. We run ick in `$DATADIR` so that relative references in the ick file work.

```
IMPLEMENTS WHEN user runs ick (.+).ick
cd "$DATADIR"
if ! "$SRCDIR/ick" --no-default-config \
    --log "$DATADIR/ick.log" \
    --verbose \
    "$DATADIR/$MATCH_1.ick" \
```

```
    > "$DATADIR/ick.stdout" 2> "$DATADIR/ick.stderr"
then
    cat "$DATADIR/ick.stdout"
    cat "$DATADIR/ick.stderr"
    exit 1
fi
```

Inspect the captured output of the latest ick file.

```
IMPLEMENTS THEN ick build log for (.+) in (.+) contains "(.+)"
export PROJECT="$MATCH_1"
export STATE="$MATCH_2"
export PATTERN="$MATCH_3"
builds="$DATADIR/$STATE/$PROJECT/builds"
build_log="$(ls "$builds"/*/build.log | tail -n1)"
cat "$build_log"
grep "$PATTERN" "$build_log"

IMPLEMENTS THEN ick build log for (.+) in (.+) doesn't contain "(.+)"
export PROJECT="$MATCH_1"
export STATE="$MATCH_2"
export PATTERN="$MATCH_3"
builds="$DATADIR/$STATE/$PROJECT/builds"
build_log="$(ls "$builds"/*/build.log | tail -n1)"
cat "$build_log"
! grep "$PATTERN" "$build_log"

IMPLEMENTS THEN there is (\d+) build log(s?) for (.+) in (.+)
export COUNT="$MATCH_1"
export PROJECT="$MATCH_3"
export STATE="$MATCH_4"
builds="$DATADIR/$STATE/$PROJECT/builds"
export NUM_BUILDS="$(ls "$builds" | wc -l)"
if [ "$NUM_BUILDS" != "$COUNT" ]
then
    echo "Expected $COUNT builds, found $NUM_BUILDS" 1>&2
    exit 1
fi
```

8.6 Creating a git repository

We need a git repository with some source code. It should have a master branch, which includes at least one file. We'll create a README file for that. We also add basic Debian packaging, so packages can be built. Following the author's habit, the Debian packaging goes into the master branch as well.

```
IMPLEMENTS GIVEN a source code repository for project (.+) version (.+
gitdir="$DATADIR/$MATCH_1"
git init "$gitdir"
cd "$gitdir"
echo "This is $MATCH_1" > README
git add README
git commit -m "Add README"

printf '#!/bin/sh\necho hello, world\n' > hello
chmod +x hello
git add hello
git commit -m "Add hello"

mkdir debian
mkdir debian/source
echo '3.0 (quilt)' > debian/source/format
echo Public domain > debian/copyright
echo 9 > debian/compat
echo hello usr/bin > debian/install
dch --create --package "$MATCH_1" --newversion "$MATCH_2" \
    "Initial version."
dch -r ''

cat << EOF > debian/control
Source: $MATCH_1
Maintainer: John Doe <john@example.com>
Section: python
Priority: optional
Standards-Version: 3.9.6
Build-Depends: debhelper (>= 7.3.8)
```

```
Package: $MATCH_1
Architecture: all
Depends: \${misc:Depends}
Description: this is a test package
 Test package is this.

EOF

printf '#!/usr/bin/make -f\n%%:\n\tdh $@\n' > debian/rules
chmod +x debian/rules

git add debian
git commit -m "Add Debian packaging"
```

Create repository with just upstream code.

```
IMPLEMENTS GIVEN an upstream source repository for project (\S+)
gitdir="$DATADIR/$MATCH_1"
git init "$gitdir"
cd "$gitdir"
echo "This is $MATCH_1" > README
git add README
git commit -m "Add README"

printf '#!/bin/sh\necho hello, world\n' > hello
chmod +x hello
git add hello
git commit -m "Add hello"
```

Create a git repository with just Debian packaging files.

```
IMPLEMENTS GIVEN a packaging repository called (\S+) for project (\S+)

gitdir="$DATADIR/$MATCH_1"
PROJECT="$MATCH_2"
VERSION="$MATCH_3"

git init "$gitdir"
cd "$gitdir"
```

```
mkdir source
echo '3.0 (quilt)' > source/format
echo Public domain > copyright
echo 9 > compat
echo hello usr/bin > install

export DEVSCRIPTS_CHECK_DIRNAME_LEVEL=0
dch --create --package "$PROJECT" --newversion "$VERSION" \
    -c changelog "Initial version."
dch -c changelog -r ''

cat << EOF > control
Source: $PROJECT
Maintainer: John Doe <john@example.com>
Section: python
Priority: optional
Standards-Version: 3.9.6
Build-Depends: debhelper (>= 7.3.8)

Package: $PROJECT
Architecture: all
Depends: \${misc:Depends}
Description: this is a test package
 Test package is this.

EOF

printf '#!/usr/bin/make -f\n%%:\n\tdh $@\n' > rules
chmod +x rules

git add .
git commit -m "Add Debian packaging"
```

8.7 Running "cleanly"

We need to run the `cleanly` command in various ways. We invoke it from the Ick source tree, and save the standard output and error, for later investigation.

```
IMPLEMENTS WHEN user runs, in (.+), cleanly (.+)
cd "$DATADIR/$MATCH_1"
# Note that we do NOT quote the second match. We let the shell
# parse it as-is. This is necessary to allow multiple arguments.
if ! "$SRCDIR/icklib/cleanly" --no-default-config \
    --log "$DATADIR/cleanly.log" \
    --results "$DATADIR" $MATCH_2 \
    > "$DATADIR/cleanly.stdout" \
    2> "$DATADIR/cleanly.stderr"
then
    echo ==== stdout ====
    cat "$DATADIR/cleanly.stdout"
    echo ==== stderr ====
    cat "$DATADIR/cleanly.stderr" 1>&2
    echo ==== end of stderr ====
    exit 1
fi
```

8.8 Checking "cleanly" output

Check what the output of the latest invocation of `cleanly` was.

```
IMPLEMENTS THEN the output is "(.*)"
printf "$MATCH_1" | diff - "$DATADIR/cleanly.stdout"
```

8.9 Checking if a file exists

Does a file exist?

```
IMPLEMENTS THEN file (.+) exists
```

```
ls -a "$DATADIR"
test -e "$DATADIR/$MATCH_1"
```

8.10 Checking for package files in the APT repository

Does a file with a given basename exist in the APT repository's pool tree?

```
IMPLEMENTS THEN the APT repository for (.+)\.ick contains (.+)
find "$DATADIR/$MATCH_1.state"
find "$DATADIR/$MATCH_1.state/debian/pool" \
    -type f -name "$MATCH_2" | grep .
```