Bob Documentation

Release 0.17.0rc4.dev5+g8ff7923

The BobBuildTool Contributors

May 29, 2020

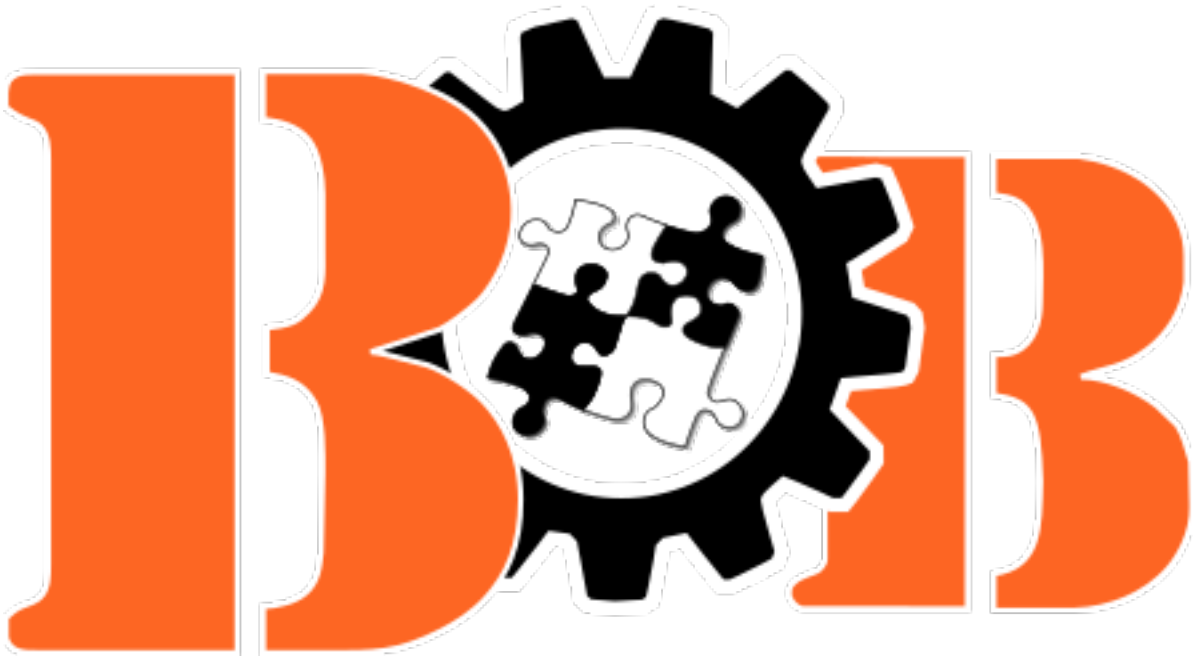
Contents

1 Contents **3**
- 1.1 Installation . . . 3
- 1.2 Bob Tutorial . . . 6
- 1.3 Bob User Manual . . . 14
- 1.4 Man Pages . . . 69
- 1.5 Bob Release Notes . . . 96

2 Copyright **115**

3 Indices and tables **117**

Index **119**



To get a glimpse on Bob start with the *Tutorial* or have a look at the website. For a comprehensive reference on Bob's usage and features have a look at the *User manual*.

CHAPTER 1

Contents

1.1 Installation

1.1.1 Dependencies

Bob is built with Python3 (>=3.5). Some additional Python packages are required. They are installed automatically as dependencies.

Apart from the Python dependencies additional run time dependencies could arise, e.g.:

- GNU `bash` >= 4.x
- Microsoft PowerShell
- GNU coreutils (`cp`, `ln`, `sha1sum`, ...)
- GNU `tar`
- `hexdump`
- `curl` as the default URL SCM downloader
- source code management handlers as used (`curl`, `cvs`, `git`, `svn`)
- extractors based on the supported extensions (`7z`, GNU `tar`, `gunzip`, `unxz`, `unzip`)
- `azure-storage-blob` Python library if the `azure` archive backend is used. Either install via pip (`python3 -m pip install azure-storage-blob`) or download from GitHub.

The actually needed dependencies depend on the used features and the operating system.

1.1.2 Install

There are several options how to install Bob on your system. If in doubt stick to the standard `pip` method.

If you are unfamiliar with the installation of Python packages make sure to read Installing Packages from the Python Packaging User Guide. The instructions below assume that you have installed Python and that it is available on the command line.

Supported Platforms

- Linux
- Windows 10
- MSYS2 (Windows 10)
- Other POSIX platforms should work but are not actively tested

See below for platform specific installation notes.

PyPI release versions

To get the latest released version just use `pip` to download the package and its depedencies from PyPI:

```
$ python3 -m pip install BobBuildTool [--user]
```

Release versions are supposed to be stable and keep backwards compatibility.

Install latest development version

If you want to test pre-release versions you can instruct `pip` to fetch and build the package directly from git:

```
$ python3 -m pip install --user git+https://github.com/BobBuildTool/bob
```

Note that during development minor breakages can occur.

Hacking on Bob

For the basic hacking there is no installation needed. Just clone the repository:

```
$ git clone https://github.com/BobBuildTool/bob.git
$ cd bob
```

and add this directory to your `$PATH` or set a symlink to `bob` from a directory that is already in `$PATH`. You will have to manually install all required dependencies and the bash completion, though.

Attention: The `pip install -e .` resp. `python3 setup.py develop` commands do *not* work for Bob. The problem is that these installtion variants are only really working for pure python projects. In contrast to that Bob comes with manpages and C helper applets that are not built by these commands.

The following additional packages and Python modules that are not part of the standard library and need to be installed:

- PyYAML. Either install via pip (`python3 -m pip install PyYAML`) or the package that comes with your distribution (e.g. python3-yaml on Debian).
- schema. Either install via pip (`python3 -m pip install schema`) or the package that comes with your distribution (e.g. python3-schema on Debian).

- python-magic. Either install via pip (`python3 -m pip install python-magic`) or the package that comes with your distribution (e.g. python3-magic on Debian).
- pyparsing. Either install via pip (`python3 -m pip install pyparsing`) or the package that comes with your distribution (e.g. python3-pyparsing on Debian).

To fully run Bob you need the following tools:

- `gcc`
- python3-sphinx

The compiler is only required on Linux.

1.1.3 Linux/POSIX platform notes

Shell completion

Bob comes with a bash completion script. If you installed Bob the completion should already be available (given that `$(DESTDIR)/share/bash-completion/completions` exists on your system). Otherwise simply source the script contrib/bash-completion/bob from your ~/.bashrc file. Optionally you can copy the script to some global directory that is picked up automatically (e.g. `cp contrib/bash-completion/bob /etc/bash_completion.d/bob` on Debian).

Zsh is able to understand the completion script too. Enable it with the following steps:

```
zsh$ autoload bashcompinit
zsh$ bashcompinit
zsh$ source contrib/bash-completion/bob
```

Sandbox capabilities

You might have to tweak your kernel settings in order to use the sandbox feature. Bob uses Linux's user namespaces to run the build in a clean environment. Check if

```
$ cat /proc/sys/kernel/unprivileged_userns_clone
1
```

yields "1". If the file exists and the setting is 0 you will get an "operation not permitted" error when building. Add the line

```
kernel.unprivileged_userns_clone = 1
```

to your `/etc/sysctl.conf` (or wherever your distro stores that).

1.1.4 Windows platform notes

Bob can be used in two flavours on Windows: as native application or in a MSYS2 POSIX environment. Unless your recipes need Unix tools the native installation is recommended.

Native usage

Python comes with extensive documentation about how to install it on Windows. Only the full installer has been tested but the other methods should probably work as well.

Make sure to add the Python interpreter to `%PATH%`. If your recipes use Bash you must additionally install MSYS2 and add the path to `bash.exe` *after* the native Python interpreter. Otherwise the MSYS2 Python interpreter might be invoked which does not work.

Note: Windows path lengths have historically been limited to 260 characters. Starting with Windows 10 the administrator can activate the "Enable Win32 long paths" group policy or you may set the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem@LongPathsEnabled` registry key to `1`. Either option is sufficient to remove the path length limitation.

MSYS2

Follow the standard MSYS2 installation. Then install `python3` and `python-pip` and use one of the install methods above.

1.2 Bob Tutorial

Contents:

1.2.1 Build a demo project

Prequisites

You should have Bob somewhere in your `$PATH`. As the first step clone the tutorial projects:

```
$ git clone https://github.com/BobBuildTool/bob-tutorials.git
$ cd bob-tutorials/sandbox
```

This tutorial will use the sandbox example that should work basically everywhere. The next steps are all executed in the same directory.

Build release

To see the packages that you can build type:

```
$ bob ls -r
vexpress
    sandbox::debian-8.2-x86
    toolchain::x86
    toolchain::arm-linux-gnueabihf
    initramfs
        busybox
            toolchain::make
    linux-image
        toolchain::make
```

As you can see there is a single top-level package called `vexpress`. This demo project builds a QEMU image for the ARM Versatile Express machine. To build the project simply do the following:

```
$ bob build vexpress
```

This will fetch the toolchains and sources and will build the image locally in the `work` directory. Grab a coffee and after some time the build should finish:

```
...
>> vexpress
   BUILD     work/vexpress/build/1/workspace
   PACKAGE   work/vexpress/dist/1/workspace
Build result is in work/vexpress/dist/1/workspace
```

Now you can run the example if you have QEMU installed:

```
$ ./work/vexpress/dist/1/workspace/run.sh
```

Some words about the directory layout. The general layout is `work/<package>/{src,build,dist}/#/` where:

- `<package>` is the name of the package. In case of namespaces that might be several subdirectories, e.g. toolchain::arm-linux-gnueabihf will be built in `work/toolchain/arm-linux-gnueabihf/...`.
- `{src,build,dist}` corresponds to the result of the step, e.g. `src` is where the checkout step is run.
- `#` is a sequential number starting from 1 that is increased for every variant of the package. New variants can emerge as recipes are updated.

Under the directory of a package you can find the following files and directories:

- `workspace/`: This is the directory where the step is executed and that holds the result.
- `{checkout,build,package}.sh`: A wrapper script that executes this specific step. Running the script will execute this particular step again. If you call the script with `shell` as first argument a new shell is spawned with exactly the same environment as the step script would find.
- `script`: The actual script that was computed from the recipe and the inherited classes. This is not directly executable because it expects the right environment and arguments.
- `log.txt`: Logs of all runs.

Development build

When executing `bob build` you build the requested packages in *release mode*. This mode is intended for reproducible builds. The example already employs a sandbox so that no host tools or paths are used. Though this is great for binary reproducible builds it is inconvenient to debug and incrementally change parts of the project.

For this purpose you can build the project in *development mode*. This mode basically does the same things but with some important differences:

- Sandboxes are not used by default. The host should have the required development tools installed, for example make, gcc, perl and so on. You may still build inside a sandbox by adding the `--sandbox` option.
- Different directory layout to group sources, build and results of all packages: `dev/{src,build,dist}/<package>/#/`.
- Incremental builds.
- Stable directory tree with incremental updates to the workspace.

In this mode it is possible to build packages, make some changes and rebuild. If your environment has the right tools installed you should get the same result as the release mode. But because sandboxes are not used it is still possible to debug the created binaries. So let's build the kernel in development mode:

```
$ bob dev vexpress/linux-image
>> vexpress/linux-image
>> vexpress/toolchain::arm-linux-gnueabihf
   CHECKOUT  dev/src/toolchain/arm-linux-gnueabihf/1/workspace
...
>> vexpress/linux-image
   CHECKOUT  dev/src/linux-image/1/workspace
   BUILD     dev/build/linux-image/1/workspace
   PACKAGE   dev/dist/linux-image/1/workspace
Build result is in dev/dist/linux-image/1/workspace
```

Notice that the development mode builds in a separate directory: `dev`. The numbering beneath the package name directory is kept stable. The numbers represent only the currently possible variants of the package from the recipes. If the `checkoutSCM` in the recipe is changed the old checkout will be moved aside instead of using a new directory like in the release mode.

Suppose we want to make a patch to the kernel. This is as simple as to go to `dev/src/linux-image/1/workspace`, edit some files and call Bob again to rebuild:

```
$ vi dev/src/linux-image/1/workspace/linux-4.3.3/...
$ bob dev vexpress/linux-image
```

Bob will detect that there are changes in the sources of the kernel and make an incremental build. For the sake of simplicity we might rebuild the top-level package to test the full build:

```
$ bob dev vexpress
$ ./dev/dist/vexpress/1/workspace/run.sh
```

Note: Touching (`touch ...`) source files will not have any effect. Bob detects changes purely by its content and not by looking on the file meta data.

Now that we have a kernel we might want to change the kernel configuration and rebuild the kernel with the new one. From the output you can see that the kernel was built in `dev/build/linux-image/1/workspace`. We might edit the `.config` there directly but using `make menuconfig` is much more convenient:

```
$ ./dev/build/linux-image/1/build.sh shell -E
$ make menuconfig
```

Now make and save your changes. Then rebuild the kernel:

```
...
  HOSTLD  scripts/kconfig/mconf
scripts/kconfig/mconf  Kconfig
configuration written to .config

*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.

$ make -j $(nproc) bzImage
```

If you know how grab the kernel image directly out of the build tree and test it. Alternatively you can rebuild the top-level package

```
$ bob dev vexpress
```

and test the whole QEMU image. The choice is yours.

Warning: Making changes to the build step tree is only detected by Bob in development mode. These changes should be properly saved in the sources or the recipe before moving on. Otherwise you risk that your changes are wiped out if Bob determines that a clean build is needed (e.g. due to recipe changes).

Query SCM status

After you have developed a great new feature you may want to know which sources you have touched to commit them to a SCM. Bob offers `bob status <options> <package>` to show a list of SCM which are unclean. SCMs are unclean in case they have modified files, unpushed commits, switched URLs or non matching tags or commit ids.

The output looks like the following line:

```
STATUS <status code> <scm path>
```

Status codes:

- `U` : Unpushed commits (Git only)
- `u` : unpushed commits on local branch (Git only)
- `M` : Modified sources.
- `S` : Switched. Could be different tag, commitId, branch or URL.
- `O` : Overridden. This Scm is overridden (*scmOverrides*). Depends on *–show-overrides*.

Firing up a Jenkins

You might let Bob configure a Jenkins server for you to build a project. Bob requires that the following plugins are available:

- Conditional BuildStep Plugin: used to efficiently support shared packages
- Copy Artifact plugin: used to carry results between the different jobs
- Git plugin: to clone git repositores
- Multiple SCMs plugin: used to support recipes that have multiple checkouts
- Subversion plugin: to checkout SVN modules
- Workspace Cleanup Plugin: to make clean builds if requested

Additionally some of the Bob helper tools must be installed on the Jenkins server and be available in the PATH. The `bob-hash-engine` and `bob-audit-engine` scripts are always needed. If you're using the sandbox feature `bob-namespace-sandbox` must be available too. To keep the setup simple it is recommended to install Bob entirely on the server.

Suppose you have a suitable Jenkins server located at http://jenkins.intranet.local:8080. Go to the project root directory and tell Bob about your server and what you want to build there (substitute `<user>` and `<pass>` with your actual credentials):

```
$ bob jenkins add intranet http://<user>:<pass>@jenkins.intranet.local:8080 -p␣
↪sandbox- -r vexpress
```

This adds a synonym ("intranet") for your Jenkins server. The `-p` adds the `sandbox-` prefix to every job. At least one `-r` option must be given to specify what should be built. To view the settings type:

```
$ bob jenkins ls -vv
intranet
 URL: http://<user>:<pass>@jenkins.intranet.local:8080/
 Prefix: sandbox-
 Upload: disabled
 Sandbox: disabled
 Roots: vexpress
 Jobs:
```

As you can see there is no job configured yet on the server. This is done by

```
$ bob jenkins push intranet
```

which pushes the local state of the recipes as Jenkins jobs to the server. Note that Bob does not need to be available on the server. The content of the recipes is inserted as shell steps into the jobs with special prologues to accommodate for the special environment.

If all required tools and plugins have been installed on Jenkins the build should succeed. Go into the "sandbox-vexpress" job, download the archived artifacts and run them locally.

Using IDEs with Bob

You may want to use a IDE with Bob. At the moment QTCreator and Eclipse are supported. You can add more IDE's using *Generators* extension. To generate project files the basic call is:

```
$ bob project <genericArgs> <generator> <package> <specificArgs>
```

with `genericArgs`:

- `-n`: Do not build. Usually bob project builds the given package first to be able to collect binaries and add them to the IDEs run/debug targets.
- `-D -c -e -E`: These arguments will be passed to bob dev and will also be used when compiling from IDE.

with `generator`:

- `eclipseCdt`: Generate project files for eclipse. Tested with eclipse MARS.
- `qt-creator`: Generate project files for QtCreator. Tested with 4.0 and 4.1.

and `package` which is the name of a package to generate the project for. Usually all dependencies for this package will be visible in the IDE. The `specificArgs` arguments are used by the generator itself. They differ from generator to generator (see below).

QTCreator

QtCreator specific Arguments:

- `--destination`: destination directory for the project files. Default is <workingDir>/projects/package_stack.
- `--name`: name of the project. Default is packageName.
- `-I`: additional include directories. They will only be added for indexer and will not change the buildresult.
- `-f`: additional files. Normally only c[pp] and h[pp] files will be added. You can add more files using a regex.
- `--kit`: kit to use for this project. You may want to use a different sysroot for includes and buildin preprocessor settings from your compiler. To tell QtCreator which toolchain to use you need to specify a kit. There are at least two options to create a kit: using the GUI or the sdkTools.

The following example shows how to create a cross compiling project for the sandbox-tutorial and the included arm-toolchain:

```
$ sdktool addTC \
    --id "ProjectExplorer.ToolChain.Gcc:arm" \
    --name "ARM-Linux-Gnueabihf" \
    --path "<toolchain-dist>/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin/arm-
↪linux-gnueabihf-g++" \
    --abi arm-linux-generic-elf-32bit
$ sdktool addDebugger \
    --id "gdb:ARM32" \
    --name "ARM-gdb" \
    --binary <toolchain-dist>/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/bin/
↪arm-linux-gnueabihf-gdb
$ sdktool addKit \
    --id "ARM_Linux" \
    --name "ARM Linux Gnueabi" \
    --devicetype Desktop \
    --toolchain "ProjectExplorer.ToolChain.Gcc:arm" \
    --sysroot <toolchain-dist>/gcc-linaro-arm-linux-gnueabihf-4.9-2014.09_linux/arm-
↪linux-gnueabihf/libc/ \
    --debuggerid "gdb:ARM32"
$ bob project qtcreator vexpress --kit ARM_LINUX
```

EclipseCdt

Eclipse specificArgs:

- `--destination`: destination directory for the project files. Default is <workingDir>/projects/package_stack.
- `--exclude`: eclipse indexer sometimes runs OutOfMemory on large sourcetrees. You can specify package names (or use a regular expression) to define packages excluded from build. This will stop indexer from indexing these packages.
- `--name`: name of the project. Default is packageName.
- `-I`: additional include directories. They will only be added for indexer and will not change the buildresult.

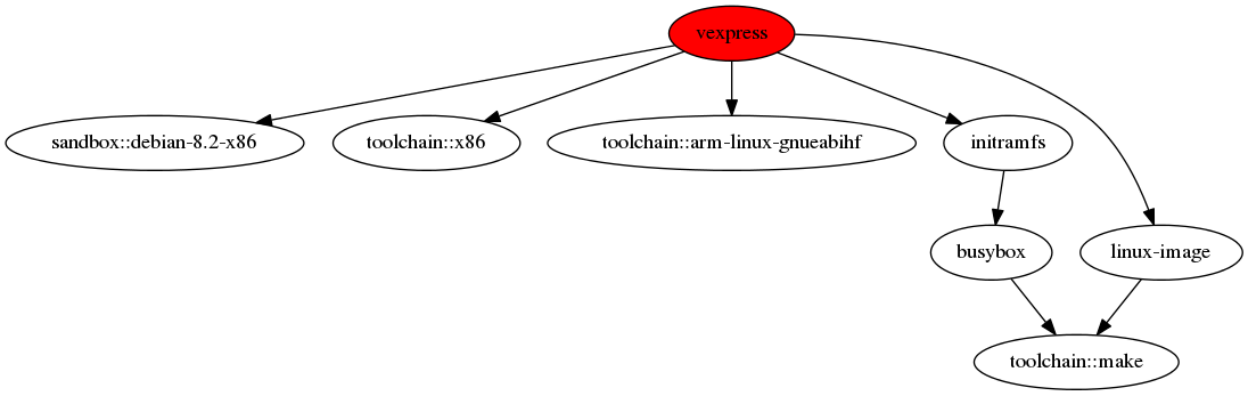
Visualizing dependencies

A dependency graph visualizes your package dependencies. Bob supports two different graph types: ‘dot’ and ‘d3’.

For the sandbox-totorial the output of

```
$ bob graph vexpress -t dot
$ dot -Tpng -osandbox_graph.png graph/vexpress.dot
```

gives you the following image:



For more complex projects 'dot' graphs doesn't scale well. Therefore you can make interactive graphs using the d3 javascript library:

Using `bob graph` with the basement-project, enable node dragging and highlighting the zlib packages:

```
$ bob graph sandbox -H zlib.* -o d3.dragNodes=True
```

1.2.2 Create a simple cross compiling project

Prequisites

1.2.3 How to properly tell Bob about host dependencies

Bob closely tracks the input of all packages. This includes all checked out sources and the dependencies to other packages. If something is changed Bob can accurately determine which packages have to be rebuilt. This information is also used to find matching binary artifacts. If a recipe depends on resources that are outside of the declared recipes the situation changes, though. Bob cannot infer what external resources are actually used and how these influence the build result.

To make these external resources visible to Bob a `fingerprintScript` must be used. The script is executed and the output is taken as fingerprint for the external resource. This way Bob can detect if the external resource has been changed and if a binary artifact is suitable on other machines. See *Host dependency fingerprinting* for more details.

Generally speaking fingerprint scripts should only be evaluated in case of a host-build. For cross-compiling the resources are usually provided by other recipes. The exception would be a recipe that uses some host resources during cross compilation, e.g. an IDL compiler that ships a target library too.

Fingerprinting the C/C++ compiler

The most common fingerprinting application is the host compiler. Usually a project should define a stub host compiler recipe that just represents the host compiler. The following example is stripped down for clarity:

```
provideTools:
   toolchain:
      path: .
      environment:
         CC: cc
         CXX: c++
      fingerprintIf: true
      fingerprintScript: |
```

(continues on next page)

(continued from previous page)

```
        bob-libc-version
        bob-libstdc++-version
  host-toolchain:
     path: .
```

By convention the tool name for the C/C++-Toolchain is just called `toolchain`. The `fingerprintIf` of it will unconditionally enable fingerprinting of whatever package is using this tool. The `fingerprintScript` will be added to these packages. In these scripts the libc and libstdc++ versions are checked via the built-in helpers.

Note that there is a separate `host-toolchain` tool that is basically the same as `toolchain` but without the `fingerprintScript` and `fingerprintIf`. This special tool is used in recipes that always need the host compiler even if they are cross-compiled with a different toolchain. The Linux kernel is a notable example. The `fingerprintScript` is not needed there because the result of such packages do not depend on the actual host compiler.

Using external libraries

Using a library from the host adds another dependency that must be declared to Bob. The example below assumes that the compiler is already fingerprinted as described in the previous chapter. Generally speaking the fingerprint script should properly detect the version of the library on the host.

Using pkg-config libraries

If your recipe uses an external library that ships a proper `.pc` file it is usually as simple as calling `pkg-config` in the `fingerprintScript`:

```
fingerprintScript: |
   pkg-config --modversion <external-dependency>
```

Note the absence of a `fingerprintIf` in the example. This is left out deliberately because the recipe should typically not know if it is compiled on the host or cross-compiled. The toolchain is supposed to enable fingerprinting if the recipe is compiled for the host. For cross-builds the used toolchain/SDK is normally provided by another recipe and thus fully known to Bob. In contrast to that the host toolchain is just a stub recipe but it will enable the fingerprinting if you follow the suggestion of the first section.

Using a library without meta information

If the library you are using does not provide any form of meta information it must be assumed that it is already in the search path of the linker. Bob provides a small helper that links a dummy executable to find the actual library that the linker was using:

```
fingerprintScript: |
   bob-hash-libraries ffi
```

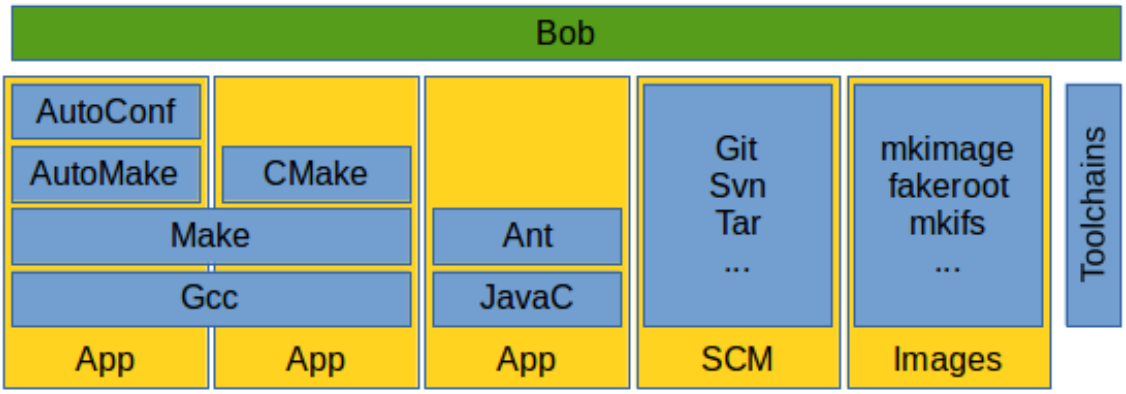
This will let the linker search the library. The result of the fingerprint is the hash sum of all used libraries, including transitive dependencies. While this may be more pessimistic than using a version number it is on the other hand guaranteed to detect different host configurations regarding this library.

1.3 Bob User Manual

Contents:

1.3.1 Introduction

Bob is a build automation tool inspired by bitbake and portage. It's main purpose is to build software packages, very much like packages in a Linux distribution. It typically works on coarse entities i.e. not on individual source files.



In contrast to similar tools, Bob tries to focus on the following requirements that are special when building complex embedded systems:

- Holistic approach: Bob can be used to describe and build the whole software stack of a project. At the same time, Bob can be used to build, change and test arbitrary parts of the project by involved developers.
- Cross compilation with multiple tool chains: Some of these tool chains may have to be built during the build process. Bob can also be used to verify the build environment, override specific host tools or abort the process if some prerequisites are not met.
- Reproducible builds: Bob aims to provide a framework which enables reproducible and even bit identical builds. To do so, each package declares its required environment, tools and dependencies. With this information Bob executes the build steps in a controlled environment.
- Continuous integration: building from live branches and not just fixed tarballs is fully supported. All packages are described in a declarative way. Using this information, the packages can be built locally but also as separate jobs on a build server (e.g. Jenkins). Bob can track all dependencies between the packages, and commits can trigger rebuilds of all affected packages.
- Variant management: because all packages declare their input environment explicitly, Bob can compute if a package must be built differently or can be reused from another build.

All in all Bob is just a framework for the controlled execution of shell scripts. To maximize reproducibility, Bob tracks the environment and the input of these scripts. If in doubt, Bob will rebuild the (supposedly) changed package.

What sets Bob apart from other systems is the functional approach. Bob takes the input for each package and processes the instructions to build the result, very much like a (imperfect) mathematical function. Every package is kept separately and only declared dependencies are available to the package build scripts.

In contrast to that, typical other package build systems describe dependencies that must be satisfied in a shared root file system. This ensures that required files are present at the known locations but it is perfectly ok that more is there. Bob on the other hand has no concept of "installation". Packages are computed with their scripts and from the declared input.

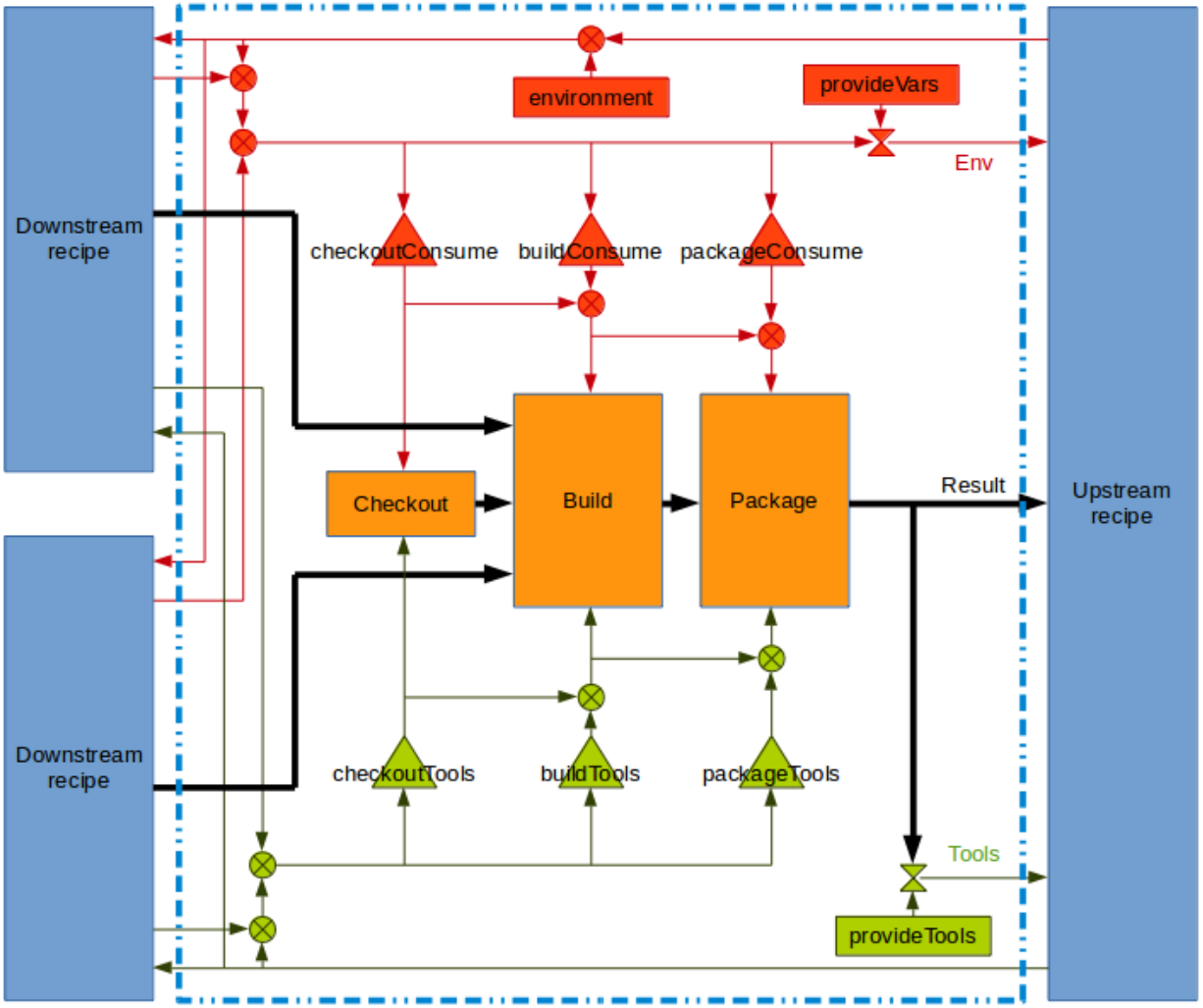
1.3.2 Concepts

Bob is, at its core, just another package build system. Conceptually it is divided into two parts: a front end that reads a domain specific language that describes how to build packages, and several backends that can build these packages.

Recipes, Classes and Packages

All build information for Bob is declared in so called *recipes*. Effectively they are blueprints for what should be built. The mathematical term for Bob's recipes would be that of a "function". A *package* is the result of the function, that is when the recipe was "computed". To keep things simple, common parts of multiple recipes may be stored in *classes*.

Some parts of the recipe may depend on additional information that is provided by other recipes. The computed result of a recipe, where all inputs are resolved, is called a package. Each package is created from a single recipe but there might be multiple packages that are created from a particular recipe. Inside each recipe there are always three steps: checkout, build and package. The following picture shows the processing inside a recipe and the interaction with upstream and downstream recipes:



There are four kinds of objects that are exchanged between recipes: results, dependencies, environment variables and tools. Results (shown as black arrows) are always propagated upstream. These are the actual build artifacts that are created. Dependencies (downstream recipes) may be propagated upwards which is not shown in the picture. Environment variables are key-value-pairs of strings. They are passed as shell variables to the individual build steps. Tools are scripts or executables that are needed to produce the build result, e.g. compilers, image generator or post processing scripts. Consuming recipes of tools get the directory of the required tool added to their $PATH so that they are available for the build steps. By explicitly defining the dependencies and required environment variables/tools of a recipe, Bob can track changes that influence the build result with a high degree of certainty.

The actual processing during build time is done in the orange steps. They are scripts that are executed with (and only with) the declared environment and tools. Bob assumes that the result of the steps depends only on the scripts themselves, their environment and tools. Additionally, Bob assumes that the "build" and "package" steps are deterministic in the sense that they produce equivalent results for the same input. Equivalent results may not necessarily be bit identical but must have the same function and must thus be interchangeable. This property is required to reuse binary build results from previous build runs or from external build servers.

The flow of environment variables is depicted in red while the flow of tools is shown in green. By default only build results and dependencies are exchanged. A recipe may declare that it consumes certain environment variables ({checkout,build,package}Consume) and tools ({checkout,build,pac kage}Tools). On the other hand a recipe may also declare to provide a dependency, tool or variable, providing the necessary input for upstream recipes. If a tool or environment variable is used without declaring its usage, Bob will stop processing. This will either happen when

parsing the recipes (if detectable) or during execution of the build. All executed scripts are configured to fail if an undefined variable is used or any command returns a failure status.

Implicit versioning

A key concept of Bob is that recipes do not have an explicit version. Instead, Bob constructs an implicit version that is derived from the recipe and the input to this recipe when building a particular package. This is called the Package-Id. The recipe language is static in the sense that the Package-Id of a package can be calculated in advance without executing any build steps. This enables Bob to determine exactly when a package has to be built from scratch (e.g. the build script changed) or if Bob has to build several packages from the same recipe due to varying input parameters.

Technically, the Package-Id is the Variant-Id of the package step. The Variant-Id of each step (checkout, build and package) is calculated as follows:

$$Id_{variant}(step) = H_{sha1}(script_{utf8}||\{Id_{variant}(t)||RelPath_{utf8}(t)||LibPaths_{utf8}(t) : t \in tools\}||env||\{Id_{variant}(i) : i \in input\}$$

where

- *script* is the script of the step,
- *tools* is the sorted list of tools that are consumed by the step,
- *env* is the sorted list of the environment key-value-pairs and
- *input* is the list of all results that are passed to the step (i.e. previous step, dependencies).

To keep the Variant-Id stable in the long run, the scripts of SCMs in the checkout step are replaced by a symbolic representation.

There exists also a second implicit version, called Build-Id, which identifies the build result in advance. The Build-Id can be used to grab matching build artifacts from another build server instead of building them locally. The Build-Id is derived from the actual sources created by the checkout step, the build/package Scripts, the environment and all Build-Ids of the recipe dependencies:

$$Id_{build}(step) = \begin{cases} H_{sha1}(script_{utf8}||\{Id_{build}(t)||RelPath_{utf8}(t)||LibPaths_{utf8}(t) : t \in tools\}||env||\{Id_{build}(i) : i \in input\}) \\ H_{sha1}(src) \end{cases}$$

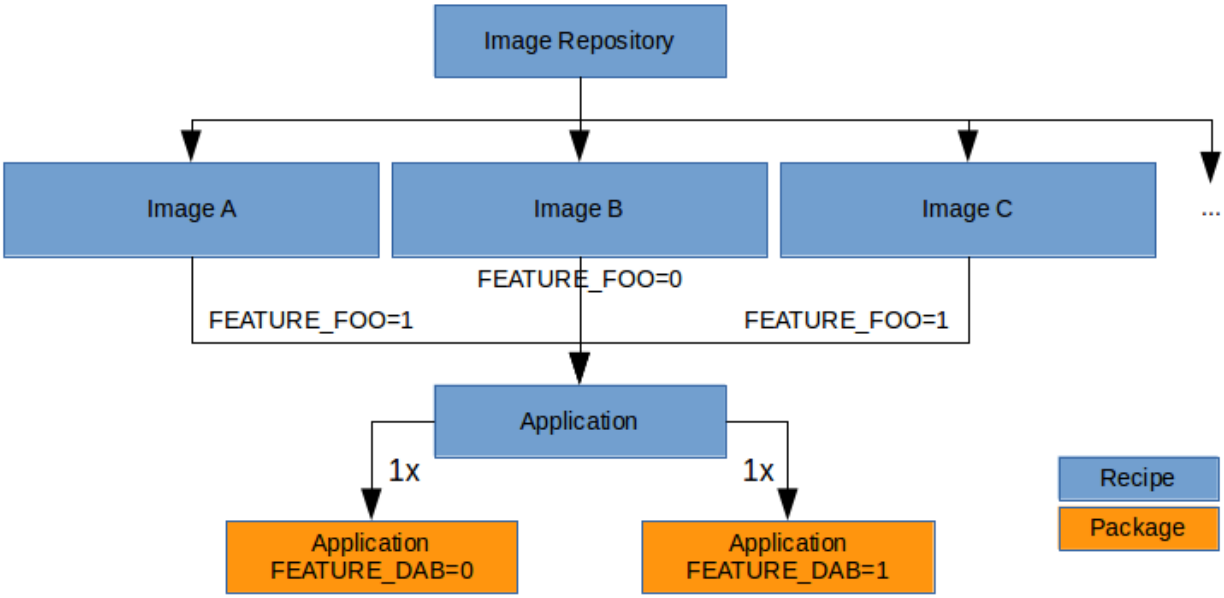
where

- *script* is the symbolic script of the step,
- *tools* is the sorted list of tools that are consumed by the step,
- *env* is the sorted list of the environment key-value-pairs and
- *input* is the list of all results that are passed to the step (i.e. previous step, dependencies).
- *src* are the actual sources created by the checkout step

The special property of the Build-Id is that it represents the expected result. To calculate it, all involved checkout steps have to be executed and the results of the checkouts have to get hashed.

Variant management

Variant management is handled entirely by environment variables that are passed to the recipes. Through implicit versioning, Bob can determine if multiple packages have to be built from the same recipe due to varying environment variables.



Variant management will typically be done by defining a dedicated environment variable for each feature, e.g. FEATURE_FOO which is either disabled ("0") or enabled ("1"). A recipe declares that it depends on this variable in the build step by listing FEATURE_FOO in the buildVars clause. Through this declaration, Bob can selectively set (only) the needed environment variables in each step and can track their dependency on them. When building the whole software, Bob can calculate how many variants of the recipe have to be built by resolving all dependent variables.

Re-usage of build artifacts

When building packages, Bob will use a separate directory for each Step-Id. Future executions of a particular step will use the same directory unless the step is changed and gets a new Step-Id. By using the Step-Id as discriminator, a safe incremental build is possible. The previous directory will be reused as long as the Step-Id is stable. If anything is changed that might influence the build result (step itself or any dependency), it will result in a new Step-Id and Bob will use a new directory. Likewise, if the changes are reverted, the Step-Id will get the previous value and Bob will restart using the previous directory.

In local builds, the build results are shared directly with upstream packages by passing the path to the upstream steps. On the Jenkins build server the build results are copied between the different work spaces.

Based on the Build-Id, it is possible to fetch build results of a build server from an artifact repository instead of building it locally. To compute the Build-Id, Bob requires that the checkout step of the recipe and all its dependencies must be deterministic. Then Bob will look up the package result from the artifact repository based on the Build-Id. If the artifact is found it will be downloaded and the build and package steps are skipped. Otherwise the package is built as always. Additionally, Bob requires the following properties from a recipe:

- The build and package steps have to be deterministic. Given the same script with the same input it has to produce the same result, functionality-wise. It is not required to be bit-identical, though.
- The build result must be relocatable. The build server will very likely have used a different directory than the local build. The result must still work in the local directory.
- The build result must not contain references to the build machine or any dependency. Sometimes the build result contains symlinks that might not be valid on other machines.

Under the above assumptions Bob is able to reliably reuse build results from other build servers.

1.3.3 Configuration

When building packages, Bob executes the instructions defined by the recipes. All recipes are located relative to the project root directory in the `recipes` subdirectory. Recipes are YAML files with a defined structure. The name of the recipe and the resulting package(s) is derived from the file name by removing the trailing '.yaml'. To aid further organization of the recipes, they may be put into subdirectories under `recipes`. The directory name becomes part of the package name with a `::`-separator.

To minimize repetition of common functionality, there is also an optional `classes` subdirectory. Classes have the same structure as recipes and can be included from recipes and other classes to factor out common stuff. Files that do not have the '.yaml' extension are ignored when parsing the recipes and classes directories.

There are two additional configuration files: `config.yaml` and `default.yaml`. The former contains static configuration options while the latter holds some options that can have a default value and might be overridden on the command line. Putting that all together, a typical recipe tree looks like the following:

```
.
├── classes
│   └── make.yaml
├── config.yaml
├── default.yaml
└── recipes
    ├── busybox.yaml
    ├── initramfs
    │   ├── inittab
    │   └── rcS
    ├── initramfs.yaml
    ├── linux.yaml
    ├── toolchain
    │   ├── arm-linux-gnueabihf.yaml
    │   ├── make.yaml
    │   └── x86.yaml
    └── vexpress.yaml
```

Such a recipe and configuration tree is meant to be handled by an SCM (Source Code Manager). As you can see in the above tree there is a `toolchain` subdirectory in the recipes. The packages in this directory will be named `toolchain::arm-linux-gnueabihf`, `toolchain::make` and `toolchain::x86`. You can also see that there are other files (initramfs/...) that are included by recipes but are otherwise ignored by Bob.

In addition to the single project tree configuration, Bob supports layers. A layer has the same structure as shown above but is merged with the other layers during parsing. This structure is recursive (a layer may contain another layer) but is flattened during parsing. A typical structure might look like the following:

```
.
├── classes
│   └── ...
├── config.yaml
├── default.yaml
├── layers
│   ├── bsp
│   │   ├── default.yaml
│   │   └── recipes
│   │       ├── linux.yaml
│   │       └── toolchain
│   │           └── arm-linux-gnueabihf.yaml
│   └── myapp
│       └── recipes
│           └── fancy-app.yaml
```

(continues on next page)

(continued from previous page)

```
└── recipes
    ├── busybox.yaml
    ├── initramfs
    │   ├── inittab
    │   └── rcS
    ├── initramfs.yaml
    ├── toolchain
    │   ├── make.yaml
    │   └── x86.yaml
    └── vexpress.yaml
```

Conceptually the recipes (and classes) from the `bsp` and `myapp` layers are parsed together at the same level with the recipes of the project root. The recipes may reference each other freely. The only restriction is that a particular recipe or class must only be defined by one layer. It is not allowed to "overwrite" a recipe in a layer above.

Layers must be configured in `config.yaml` to be picked up. This is just a simple list of layer names that are then expected in the `layers` directory:

```
layers:
    - myapp
    - bsp
```

The order of layers is important with respect to settings made by `default.yaml` in the various layers. The project root has the highest precedence. The layers in `config.yaml` are named from highest to lowest precedence. A layer with a higher precedence can override settings from layers of lower precedence.

Layers allow you to structure your projects into larger entities that can be reused in other projects. This modularity helps to separate different aspects of bigger projects like the used toolchain, the board support package and the applications integration.

Principle operation

All packages are built by traversing the recipe tree starting from one or more root recipes. These are recipes that have the `root` attribute set to `True`. There must be at least one root recipe in a project. The tree of recipes is traversed depth first. While following the dependencies, Bob keeps a local state that consists of the following information:

Environment Bob always keeps the full set of variables but only a subset is visible when executing the scripts. Initially, only the variables defined in `default.yaml` in the `environment` section are available. Environment variables can be set at various points that are described below in more detail.

Tools Tools are aliases for paths to executables. Initially there are no tools. They are defined by `provideTools` and must be explicitly imported by upstream recipes by listing `tools` in the `use` attribute. Like environment variables, the tools are kept as key value pairs where the key is a string and the value is the executable and library paths that are imported when using a tool.

Sandbox This defines the root file system and paths that are used to build the package. Unless a sandbox is consumed by listing `sandbox` in the `use` attribute of a dependency, the normal host executables are used. Sandboxed builds are described in a separate section below.

All of this information is carried as local state when traversing the dependency tree. Each recipe gets a local copy that is propagated downstream. Any updates to upstream recipes must be done by explicitly offering the information with one of the `provide*` keywords and the upstream recipe must consume it by adding the relevant item to the `use` attribute of the dependency.

Step execution

The actual work when building a package is done in the following three steps. They are scripts that are executed with (and only with) the declared environment and tools.

Checkout The checkout step is there to fetch the source code or any external input of the package. Despite the script defined by `checkoutScript`, Bob supports a number of source code management systems natively. They can be listed in `checkoutSCM` and are fetched/updated before the `checkoutScript` is run.

Build This is the step where most of the work should be done to build the package. The `buildScript` receives the result of the checkout step as argument `$1`, and any further dependency whose result is consumed is passed in order starting with `$2`. If no checkout step was provided, `$1` will point to some invalid path.

Package Typically, the build step will produce a lot of intermediate files (e.g. object files). The package step has the responsibility to distill a clean result of the package. The `packageScript` will receive a single argument with the path to the build step.

Each step of a recipe is executed separately and always in the above order. The scripts' working directory is already where the result is expected. The scripts should make no assumption about the absolute path or the relative path to other steps. Only the working directory might be modified.

Script languages

Bob itself is written in the Python scripting language but actually independent of the scripting language that is used during step execution (see above). Currently Bob supports two scripting languages: bash and PowerShell. Classes and recipes may define their scripts in one or both scripting languages. The actually used language at build time is determined by the *scriptLanguage* key or, if nothing was specified, by the project *scriptLanguage* setting. The other language scripts are ignored.

Environment handling

The variables listed in *environment* of `default.yaml` with their configured value are mangled through *String substitution* by the current OS environment and are then taken over into the initial environment. The user might additionally override or set certain variables from the command line. Such variables are always taken over verbatim. The so calculated set of variables is the starting point for each root recipe.

Note: Depending on the *cleanEnvironment* policy the initial environment may first be populated with the whitelisted variables named by *whitelist* from the current OS environment. The new behaviour (i.e. enabled policy) is to start with a clean environment.

The next steps are repeated for each recipe as the dependency tree is traversed. A copy of the environment is inherited from the upstream recipe.

1. Any variable defined in `environment` is set to the given value.
2. Make a copy of the local environment that is subsequently passed to each dependency (named "forwarded environment" thereafter).
3. For each dependency do the following:
    a. Make a dedicated copy of the environment for the dependency.
    b. Set variables given in the `environment` attribute of the dependency in this copy.
    c. Descend to the dependency recipe with that environment.

   d. Merge all variables of the `provideVars` section of the dependency into the local environment if `environment` is listed in the `use` attribute of the dependency.

   e. If the `forward` attribute of the dependency is `True` then any merged variable of the previous step is updated in the forwarded environment too.

After all dependencies have been processed, the environment variables of tools (see *provideTools*) that are used in the recipe are merged into the local environment. Finally, variables defined in *privateEnvironment* and *metaEnvironment* are merged too.

A subset of the resulting local environment can be passed to the three execution steps. The variables available to the scripts are defined by *{checkout,build,package}Vars* and *{checkout,build,package}VarsWeak*. The former property defines variables that are considered to influence the build while the latter names variables that are expected to *not* influence the outcome of the build.

A variable that is consumed in one step is also set in the following. This means a variable consumed through checkoutVars is also set during the build and package steps. Likewise, a variable consumed by buildVars is set in the package step too. The rationale is that all three steps form a small pipeline. If a step depends on a certain variable then the result of the following step is already indirectly dependent on this variable. Thus it can be set during the following step anyway.

A recipe might optionally offer some variables to the upstream recipe with a `provideVars` section. The values of these variables might use variable substitution where the substituted values are coming from the local environment. The upstream recipe must explicitly consume these provided variables by adding `environment` to the `use` attribute of the dependency.

Tool handling

Tools are handled very similar to environment variables when being passed in the recipe dependency tree. Tools are aliases for a package together with a relative path to the executable(s) and optionally some library paths for shared libraries. Another recipe using a tool gets the path to the executable(s) added to its `$PATH`.

Starting at the root recipe, there are no tools. The next steps are repeated for each recipe as the dependency tree is traversed. A copy of the tool aliases is inherited from the upstream recipe.

1. Make a copy of the local tool aliases that is subsequently passed to each dependency (named "forwarded tools" thereafter).
2. For each dependency do the following:

   a. Descend to the dependency recipe with the forwarded tools

   b. Merge all tools of the `provideTools` section of the dependency into the local tools if `tools` is listed in the `use` attribute of the dependency.

   c. If the `forward` attribute of the dependency is `True` then any merged tools of the previous step are updated in the forwarded tools too.

While the full set of tools is carried through the dependency tree, only a specified subset of these tools is available when executing the steps of a recipe. The available tools are defined by {checkout,build,package}Tools. A tool that is consumed in one step is also set in the following. This means a tool consumed through checkoutTools is also available during the build and package steps. Likewise, a tool consumed by buildTools is available in the package step too.

To define one or more tools, a recipe must include a `provideTools` section that defines the relative execution path and library paths of one or more tool aliases. These aliases may be picked up by the upstream recipe by having `tools` in the `use` attribute of the dependency.

Sandbox operation

Unless a sandbox is configured for a recipe, the steps are executed directly on the host. Bob adds any consumed tools to the front of `$PATH` and controls the available environment variables. Apart from this, the build result is pretty much dependent on the installed applications of the host.

By utilizing user namespaces on Linux, Bob is able to execute the package steps in a tightly controlled and reproducible environment. This is key to enable binary reproducible builds. The sandbox image itself is also represented by a recipe in the project.

Initially no sandbox is defined. A downstream recipe might offer its built package as sandbox through `provideSandbox`. The upstream recipe must define `sandbox` in the `use` attribute of this dependency to pick it up as sandbox. This sandbox is effective only for the current recipe. If `forward` is additionally set to `True` the following dependencies will inherit this sandbox for their execution.

Inside the sandbox, the result of the consumed or inherited sandbox image is used as root file system. Only direct inputs of the executed step are visible. Everything except the working directory and `/tmp` is mounted read only to restrict side effects. The only component used from the host is the Linux kernel and indirectly Python because Bob is written in this language. The sandbox image must provide everything to execute the steps. In particular, the following things must be provided by the sandbox image:

- There must be an `etc/passwd` file containing the "nobody" user with uid 65534.
- There must *not* be a `home` directory. Bob creates this directory on demand and will fail if it already exists.
- There must *not* be a `tmp` directory for the same reason.
- The interpreter of the used script language must be available (`bash` or `pwsh`) and it must be in `$PATH`. When using bash (the default) at least version 4 must be installed. Bob uses associative arrays that are not available in earlier versions.

String substitution

At most places where strings are handled in keywords, it is possible to use variable substitution. These substitutions might be simple variables, but a variety of string processing functions is also available that can optionally be extended by plugins. The following syntax is supported:

- **Variable substitution**
  - `${var}`: The value of `var` is substituted. The variable has to be defined or an error will be raised. The braces can be omitted if the variable name consists only of letters, numbers and _ and when the name is followed by a character that is not interpreted as part of its name.
  - `${var:-default}`: If variable `var` is unset or null, the expansion of `default` is substituted. Otherwise the value of `var` is substituted. Omitting the colon results in a test only for `var` being unset.
  - `${var:+alternate}`: If variable `var` is unset or null, nothing is substituted. Otherwise the expansion of `alternate` is substituted. Omitting the colon results in a test only for `var` being unset.
- `$(fun,arg1,...)`: Substitutes the result of calling `fun` with the given arguments. Unlike unix shells, which employ word splitting at whitespaces, the function arguments are separated by commas. Any white spaces are kept and belong to the arguments. To put a comma or closing parenthesis into an argument it has to be escaped by a backslash or double/single quotes.
- **Quoting**
  - `"..."`: Double quotes begin a new substitution context that runs until the matching closing double quote. All substituions are still recognized.

  - `'...'`: Enclosing characters in single quotes preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.
  - `\.`: A backslash preserves the literal meaning of the following character. The only exception is within single quotes where backslash is not recognized as meta character.

The following built in string functions are supported:

- `$(eq,left,right)`: Returns `true` if the expansions of `left` and `right` are equal, `false` otherwise.
- `$(match,string,pattern[,flags])`: Returns `true` if `pattern` is found in `string`, `false` otherwise. Quoting the pattern is recommended. Flags are optional. The only currently supported flag is `i` to ignore case while searching.
- `$(if-then-else,condition,then,else)`: The expansion of `condition` is interpreted as a boolean value. If the condition is true the expansion of `then` is returned. Otherwise `else` is returned.
- `$(is-sandbox-enabled)`: Return `true` if a sandbox is enabled in the current context, `false` otherwise.
- `$(is-tool-defined,name)`: If `name` is a defined tool in the current context the function will return `true`. Otherwise `false` is returned.
- `$(ne,left,right)`: Returns `true` if the expansions of `left` and `right` differ, otherwise `false` is returned.
- `$(not,condition)`: Interpret the expansion of `condition` as boolean value and return the opposite.
- `$(or,condition1,condition2,...)`: Expand each condition and then interpret each condition as boolean. Return `false` when all conditions are false, otherwise `true`.
- `$(and,condition1,condition2,...)`: Expand each condition and the interpret each condition as booelan. Return `true` when all conditions are true, otherwise `false`.
- `$(strip,text)`: Remove leading and trailing whitespaces from the expansion of `text`.
- `$(subst,from,to,text)`: Replace every occurence of `from` with `to` in `text`.

The following built in string functions are additionally supported in *package path queries*. They cannot be used in recipes as they work on packages:

- `$(matchScm,property,pattern)`: Return `true` if there is at least one *checkoutSCM* in the package that has a `property` that matches the `pattern`. Otherwise returns `false`. Shell globbing patterns may be used as `pattern`.

Plugins may provide additional functions as described in *String functions*. If a string is interpreted as a boolean then the empty string, "0" (zero) and "false" (case insensitive) are considered as logical "false". Any other value is considered as "true".

Host dependency fingerprinting

Bob closely tracks the input of all packages. This includes all checked out sources and the dependencies to other packages. If something is changed, Bob can accurately determine which packages have to be rebuilt. This information is also used to find matching binary artifacts. If a recipe depends on resources that are outside of the declared recipes, the situation changes though. Bob cannot infer what external resources are actually used and how these influence the build result.

A common host dependency that "taints" the build result is the host compiler. While the host compiler typically does not change, it limits the portability across machines in the form of binary artifacts. The dependency on the host architecture is obvious, but also the libc has to be considered. This can be extended to other libraries that might be used by the recipe.

To let Bob know about the usage and state of an external host resource, a fingerprint script can be used in the recipe. The output of the fingerprint script is used to "tag" the created package. If the fingerprint changes, the package is rebuilt. The fingerprint is also attached to the binary artifact. To download a binary artifact of a package, the fingerprint has to match.

The fingerprint does not apply to the *checkoutScript*, though. If the result of your *checkoutScript* depends on the host that it runs on, you have to set *checkoutDeterministic* to *False*. The fingerprint serves only as a virtual input to the build and package steps to declare to Bob what part of the host is used by the recipe.

The impact of the host that is declared by a fingerprint script applies only to the result of a recipe. Specifically, it does not apply to the implied *behaviour* of any provided tools. This means that when using a tool from another recipe that is directly or indirectly affected by a fingerprint, the using recipe is not affected. The rationale for this exception of transitivity is that it typically does not matter *where* a tool is built but how it *behaves*.

See *fingerprintScript[{Bash,Pwsh}]* and *provideTools* for information where fingerprint scripts can be configured.

Recipe and class keywords

{checkout,build,package}Script[{Bash,Pwsh}]

Type: String

This is the script that is executed by Bob at the respective stage when building the Packet. It is strongly recommended to write the script as a newline preserving block literal. See the following example (note the pipe symbol on the end of the first line):

```
buildScript: |
    $1/configure
    make
```

The suffix of the keyword determines the language of the script. Using the `Bash` suffix (e.g. `buildScriptBash`) defines a script that is interpreted with `bash`. Likewise, the `Pwsh` suffix (e.g. `buildScriptPwsh`) defines a PowerShell script. Which language is used at build time is determined by the *scriptLanguage* key or, if nothing was specified, by the project *scriptLanguage* setting. A keyword without a suffix (e.g. `buildScript`) is interpreted in whatever language is finally used at build time. If both the keyword with the build time language suffix and without a suffix are present then the keyword with the build language suffix takes precedence.

The script is subject to file inclusion with the `$<<path>>` and `$<'path'>` syntax. The files are included relative to the current recipe. The given `path` might be a shell globbing pattern. If multiple files are matched by `path` the files are sorted by name and then concatenated. The `$<<path>>` syntax imports the file(s) as is and replaces the escape pattern with a (possibly temporary) file name which has the same content. Similar to that, the `$<'path'>` syntax includes the file(s) inline as a quoted string. In any case, the strings are fully quoted and *not* subject to any parameter substitution.

Note: When including files as quoted strings (`$<'path'>` syntax), they have to be UTF-8 encoded.

The scripts of any classes that are inherited which define a script for the same step are joined in front of this script in the order the inheritance is specified. The inheritance graph is traversed depth first and every class is included exactly once.

During execution of the script only the environment variables SHELL, USER, TERM, HOME and anything that was declared via {checkout,build,package}Vars are set. The PATH is reset to "/usr/local/bin:/bin:/usr/bin" or whatever was declared in config.yaml. Any tools that are consumed by a {checkout,build,package}Tools declaration are added to the front of PATH. The same holds for `$LD_LIBRARY_PATH` with the difference of starting completely empty.

Additionally, the following (environment) variables are populated automatically:

- `BOB_CWD`: Environment variable holding the working directory of the current script as absolute path.
- `BOB_ALL_PATHS`: An associative array that holds the paths to the results of all dependencies indexed by the package name. This also includes indirect dependencies such as consumed tools or the sandbox.
- `BOB_DEP_PATHS`: An associative array of all direct dependencies. This array comes in handy if you want to refer to a dependency by name (e.g. `${BOB_DEP_PATHS[libfoo-dev]}`) instead of the position (e.g. `$2`).
- `BOB_TOOL_PATHS`: An associative array that holds the execution paths to consumed tools indexed by the package name. All these paths are in `$PATH` resp. `%PATH%`.

The associative arrays are no regular environment variables. Hence they are not inherited by other processes that are invoked by the executed scripts. In bash scripts they are associative arrays. See Bash Arrays for more information. In PowerShell scripts they are defined as Hash Tables.

For PowerShell scripts a utility function called `Check-Command` is available. It has two arguments: the first one (`ScriptBlock`) expects a script block that is executed. The optional second argument (`ErrorAction`) lets you override the error action. After the script block was executed the `Check-Command` function will check the last exit status and invoke the error action if it is not zero. Example:

```
Check-Command { cmake --build . }
```

By default it will halt the script execution. This helper is needed because there is no possibility to configure PowerShell to stop execution when an external command fails. Make sure to wrap calls to external tools with `Check-Command` or check `$lastexitcode` yourself. Otherwise the build will not detect errors involving external commands!

{checkout,build,package}Tools

Type: List of strings

This is a list of tools that should be added to `$PATH` during the execution of the respective checkout/build/package script. A tool denotes a folder in an (indirect) dependency. A tool might declare some library paths that are then added to `$LD_LIBRARY_PATH`. The order of tools in `$PATH` and `$LD_LIBRARY_PATH` is unspecified. It is assumed that each tool provides a separate set of executables so that the order of their inclusion does not matter.

A tool that is consumed in one step is also set in the following. This means a tool consumed through checkoutTools is also available during the build and package steps. Likewise a tool consumed by buildTools is available in the package step too. The rationale is that all three steps form a small pipeline. If a step depends on a certain tool then the result of the following step is already indirectly dependent on this tool. Thus it can be available during the following step anyway.

{checkout,build,package}Vars

Type: List of strings

This is a list of environment variables that should be set during the execution of the checkout/build/package script. This declares the dependency of the respective step to the named variables.

It is not an error if a variable listed here is unset. This is especially useful for classes or to implement default behaviour that can be overridden by the user from the command line. If you expect a variable to be unset, it is your responsibility to handle that case in the script. Every reference to such a variable should be guarded with `${VAR-somthing}` or `${VAR+something}`.

A variable that is consumed in one step is also set in the following. This means a variable consumed through checkoutVars is also set during the build and package steps. Likewise, a variable consumed by buildVars is set in the package step too. The rationale is that all three steps form a small pipeline. If a step depends on a certain variable then the

result of the following step is already indirectly dependent on this variable. Thus it can be set during the following step anyway.

The following variables are populated internally by Bob and might be added to the variable list:

- `BOB_HOST_PLATFORM` - the platform identifier where Bob is running on. The following values are defined:
  - `linux`: Linux
  - `msys`: Windows/MSYS2
  - `win32`: Windows
  - `darwin`: Mac OS X
- `BOB_RECIPE_NAME` - name of the recipe that defined the package
- `BOB_PACKAGE_NAME` - name of the actual package. Might be different from the recipe name if `multiPackage` is used.

Note that you should keep the usage of these variables to a minimum because they may force separate builds of packages that are otherwise identical. For example using `BOB_PACKAGE_NAME` in `buildVars` will force separate builds of all involved `multiPackage` keys even if they have a common `buildScript` because `BOB_PACKAGE_NAME` will be unique for each `multiPackage` entry.

{checkout,build,package}VarsWeak

Type: List of strings

This is a list of environment variables that should be set during the execution of the checkout/build/package script. These variables are not considered to influence the result, very much like the variables listed in *whitelist*.

Warning: Bob expects that the content of these variables is irrelevant for the actual build result. They neither contribute to variant management nor will they trigger a rebuild of a package if they change.

For example, a typical usage of `buildVarsWeak` is to specify the number of parallel make jobs. While it changes the behaviour of the job (the number of parallel compiler processes) it will not change the actual build result. The weak inclusion of a variable has no effect if it is also referenced by *{checkout,build,package}Vars*. In this case the variable will always be considered significant for the build result.

It is not an error that a variable listed here is unset. This is especially useful for classes or to implement default behaviour that can be overridden by the user from the command line. If you expect a variable to be unset it is your responsibility to handle that case in the script. Every reference to such a variable should be guarded with `${VAR-somthing}` or `${VAR+something}`.

A variable that is consumed in one step is also set in the following. This means a variable consumed through checkoutVarsWeak is also set during the build and package steps. Likewise, a variable consumed by buildVarsWeak is set in the package step too. The rationale is that all three steps form a small pipeline. If a step depends on a certain variable then the result of the following step is already indirectly dependent on this variable. Thus it can be set during the following step anyway.

{build,package}NetAccess

Type: Boolean

By default the external network is not accessible during build or package steps when building inside a sandbox. Checkout steps always have network access. If such access is still needed a recipe may set the `buildNetAccess` or the `packageNetAccess` to `True`.

Warning: Bob assumes that build and package steps are deterministic. Do not rely on external state that changes the behavior of the build. Unless the input of a package changes (sources, dependencies) Bob will not re-build a package.

Note: Before Bob 0.14 (see *offlineBuild* policy) the network access was always possible. The policy will determine the default value of this property.

To configure the network access based on the actually used tools by a recipe you can set the `netAccess` property in *provideTools*. The `{build,package}NetAccess` should only be set if the script in the recipe itself requires the network access during build or package steps.

checkoutAssert

Type: List of checkout assertions

Using `checkoutAssert` you can make a build fail if a file content has been changed. This is especially useful to detect modifications in license files and copyright notices in source files.

The following properties are supported:

Property	Description
`file`	The file in the workspace to check. Must be a relative path.
`digestSHA1`	Digest of the file / part (lower case). Either pre calculate it using `sha1sum` command or take the output of the first (failing) run.
`start`	First line of the file that is checked. Optional integer number. Defaults to 1 (first line of file).
`end`	Last line of file that is checked. Optional integer number. Defaults to last line of file.

Line numbers start at 1 and are inclusive. The `start` line is always taken into account even if the `end` line is equal or smaller. The line terminator is always `\n` (ASCII "LF", 0x0a) regardless of the host operating system.

Example:

```
checkoutAssert:
    - file: LICENSE
      digestSHA1: "2f7285314f4c057c75dbc0e5fad403b2d0691628"
    - file: src/namespace-sandbox/namespace-sandbox.c
      digestSHA1: "5ee22fb054c92560ec17202dec67202563e0d145"
      start: 3
      end: 13
```

checkoutDeterministic

Type: Boolean

By default any checkoutScript is considered indeterministic. The rationale is that extra care must be taken for a script to fetch always the same sources. If you are sure that the result of the checkout script is always the same you may

set this to `True`. All checkoutSCMs on the other hand are capable of determining automatically whether they are determinstic.

If the checkout is deemed deterministic it enables Bob to apply various optimizations. It is also the basis for binary artifacts.

checkoutSCM

Type: SCM-Dictionary or List of SCM-Dictionaries

Bob understands several source code management systems natively. On one hand it enables the usage of dedicated plugins on a Jenkins server. On the other hand Bob can manage the checkout step workspace much better in the development build mode.

All SCMs are fetched/updated before the checkoutScript of the package are run. The checkoutScript should not move or modify the checkoutSCM directories, though.

If the package consists of a single git module you can specify the SCM directly:

```
checkoutSCM:
    scm: git
    url: git://git.kernel.org/pub/scm/network/ethtool/ethtool.git
```

If the package is built from multiple modules you can give a list of SCMs:

```
checkoutSCM:
    -
        scm: git
        url: git://...
        dir: src/foo
    -
        scm: svn
        url: https://...
        dir: src/bar
```

There are three common (string) attributes in all SCM specifications: `scm`, `dir` and `if`. By default the SCMs check out to the root of the workspace. You may specify any relative path in `dir` to checkout to this directory.

By using `if` you can selectively enable or disable a particular SCM using either a string or a expression. In case a string is given to the `if`-keyword it is substituted according to *String substitution* and the final string is interpreted as a boolean value (everything except the empty string, `0` and `false` is considered true). In case you're using the expression syntax you can use *String literals* and *String function calls* to express a condition. The SCM will only be considered if the condition passes.

Currently the following `scm` values are supported:

scm	Description	Additional attributes
cvs	CVS repository	`cvsroot`: repository location (":ext:...", path name, etc.) `module`: module name `rev`: revision, branch, or tag name (optional)
git	Git project	`url`: URL of remote repository `branch`: Branch to check out (optional, default: master) `tag`: Checkout this tag (optional, overrides branch attribute) `commit`: SHA1 commit Id to check out (optional, overrides branch or tag attribute) `rev`: Canonical git-rev-parse revision specification (optional, see below) `remote-*`: additional remote repositories (optional, see below) `sslVerify`: Whether to verify the SSL certificate when fetching (optional) `shallow`: Number of commits or cutoff date that should be fetched (optional) `singleBranch`: Fetch only single branch instead of all (optional)
import	Import directory from project	`url`: Directory path relative to project root. `prune`: Delete destination directory before importing files.
svn	Svn repository	`url`: URL of SVN module `revision`: Optional revision number (optional) `sslVerify`: Whether to verify the SSL certificate when fetching (optional)
url	While not a real SCM it allows to download (and extract) files/archives.	`url`: File that should be downloaded `digestSHA1`: Expected SHA1 digest of the file (optional) `digestSHA256`: Expected SHA256 digest of the file (optional) `extract`: Extract directive

Most SCMs support the `sslVerify` attribute. This is a boolean that controls whether to verify the SSL certificate when fetching. It defaults to `True` with the notable exception of `git` before Bob 0.15 which was rectified by the introduction of the *secureSSL* policy. If at all possible, fixing a certificate problem is preferable to using this option.

cvs The CVS SCM requires a `cvsroot`, which is what you would normally put in your CVSROOT environment variable or pass to CVS using `-d`. If you specify a revision, branch, or tag name, Bob will check out that instead of the HEAD. Unfortunately, because Bob cannot know beforehand whether the `rev` you gave it points to a branch or tag, it must consider this SCM nondeterministic. To check out using ssh, you can use the syntax `:ssh:user@host:/path`, which will be translated into an appropriate `CVS_RSH` assignment by Bob. Alternatively, you can use a normal `:ext:` CVSROOT and manually pass the `CVS_RSH` value into the recipe using `checkoutVars`.

git The `git` SCM requires at least an `url` attribute. The URL might be any valid Git URL. To checkout a branch other than *master* add a `branch` attribute with the branch name. To checkout a tag instead of a branch specify it with `tag`. You may specify the commit id directly with a `commit` attribute too.

Note: The default branch of the remote repository is not used. Bob will always checkout "master" unless `branch`, `tag` or `commit` is given.

The `rev` property of the `git` SCM unifies the specification of the desired branch/tag/commit into one single property. If present it will be evaluated first. Any other `branch`, `tag` or `commit` property is evalued after it and may override a precious setting made by `rev`. The branch/tag/commit precedence is still respected, though. Following the patterns described in git-rev-parse(1) the following formats are currently supported:

- <sha1>, e.g. dae86e1950b1277e545cee180551750029cfe735. The full SHA-1 object name (40-byte hexadecimal string).
- refs/tags/<tagname>, e.g. refs/tags/v1.0. The symbolic name of a tag.
- refs/heads/<branchname>, e.g. refs/heads/master. The name of a branch.

The `remote-*` property allows adding extra remotes whereas the part after `remote-` corresponds to the remote name and the value given corresponds to the remote URL. For example `remote-my_name` set to `some/url.git` will result in an additional remote named `my_name` and the URL set to `some/url.git`.

To reduce the amount of data that is fetched from the remote repository the optional `shallow` attribute can be set. If it is an integer then only this number of commits are fetched from the tip of the remote branches (`--depth` clone parameter). It can also be a string that should be a date understood by git (passed as `--shallow-since=` to git). Either option will imply `singleBranch` to be true. This further restricts the fetching of remote branches to the configured branch only. Set `singleBranch` either to `False` to explicitly fetch all remote branches or to `True` to fetch only the current branch, regardless of the `shallow` setting.

Tip: You can set the `shallow` and `singleBranch` properties with *scmOverrides* too. This can be used to improve the build times of existing projects or to fetch the whole history if `shallow` is used in the recipes.

---

import The `import` SCM copies the directory specified in `url` to the workspace. By default the destination is not overwritten unless the source file was changed more recently than the exiting destination in the workspace. Set `prune` to `True` to always delete the destination directory before importing the files.

> **Attention:** Do not import large source trees when working with Jenkins builds. The content is included in the job configuration that will get too large otherwise.

svn The Svn SCM, like git, requires the `url` attribute too. If you specify a numeric `revision` Bob considers the SCM as deterministic.

url The `url` SCM naturally needs an `url` attribute. If a SHA digest is given with `digestSHA1` and/or `digestSHA256` the downloaded file will be checked for a matching hash sum. This also makes the URL deterministic for Bob. Otherwise the URL will be checked in each build for updates. Based on the file name ending Bob will try to extract the downloaded file. You may prevent this by setting the `extract` attribute to `no` or `False`. If the heuristic fails the extraction tool may be specified as `tar`, `gzip`, `xz`, `7z` or `zip` directly. For `tar` files it is possible to strip a configurable number of leading components from file names on extraction by the `stripComponents` attribute.

Note: Starting with Bob 0.14 (see *tidyUrlScm* policy) the whole directory where the file is downloaded is claimed by the SCM. It is not possible to fetch multiple files in the same directory. This is done to separate possibly extracted files safely from other checkouts.

---

depends

Type: List of Strings or Dependency-Dictionaries

Declares a list of other recipes that this recipe depends on. Each list entry might either be a single string with the recipe name or a dictionary with more fine grained settings. Such entries might either name another recipe directly (`name`) or a list of further dependencies (`depends`) that inherit the settings from the current entry. See the following example for both formats:

```
depends:
    - foo
    - bar
    -
        name: toolchain
        use: [tools, environment]
        forward: True
    -
        if: "${FOOBAR}"
        depends:
            - baz
            - qux
```

In the first and second case only the package is named, meaning the build result of recipe *foo* resp. *bar* is fed as `$2` and `$3` to the build script. Any provided dependencies of these packages (*provideDeps*) will be implicitly added to the dependency list too.

In the third case a recipe named *toolchain* is required but instead of using its result the recipe imports any declared tools and environment variables from *toolchain*. Additionally, because of the `forward` attribute, these imported tools and variables are not only imported into the current recipe but also forwarded to the following recipes (*baz* and *qux*).

The 4th case is a recursive definition where the simple dependencies *baz* and *qux* are guarded by a common condition. These dependencies will only be considered if the variable `FOOBAR` expands to a value that is evaluated as boolean true. If the condition passes these dependencies will be available as `$4` and `$5` to the build script. Recursive definitions might be nested freely and they might override any setting mentioned in the table below. All `if` properties on each nesting level must evaluate to true for an entry to take effect.

Detailed entries must either contain a `name` property or a `depends` list. The following settings are supported:

Name	Type	Description
name	String	The name of the required recipe.
depends	List of Dependencies	A list of dependencies inheriting the settings of this entry.
use	List of strings	List of the results that are used from the package. The following values are allowed: • `deps`: provided dependencies of the recipe. These dependencies will be added at the end of the dependency list unless the dependency is already on the list. • `environment`: exported environment variables of the recipe. • `result`: build result of the recipe. • `tools`: declared build tools of the recipe. • `sandbox`: declared sandbox of the recipe. Default: Use the result and dependencies (`[deps, result]`).
forward	Boolean	If true, the imported environment, tools and sandbox will be forwarded to the dependencies following this one. Otherwise these variables, tools and/or sandbox will only be accessible in the current recipe. Default: False.

environment	Dictionary (String -> String)	This clause allows to define or override environment variables for the dependencies. Example:
		`environment:`
		`  FOO: value`
		`  BAR: baz`

if	String	The string (subject to substitution) is interpreted as boolean value. The dependency is only considered if the string is considered as true. See *String substitution*. Default: true

if: !expr	String	The string is parsed as expression using the same *String literals* and *String function calls* as available for bobpaths. The dependency is only considered if the expression is considered as true.
		Default: true Example:
		`if: !expr |`
		`  "${FOO}" == "bar" ↪\|\| "${BAZ}"`

environment

Type: Dictionary (String -> String)

Defines environment variables in the scope of the current recipe. Any inherited variables of the upstream recipe with the same name are overwritten. All variables are passed to downstream recipes.

Example:

```
environment:
   PKG_VERSION: "1.2.3"
```

The environment of the recipe and inherited classes are merged together. The exact way of merging is subject to the *mergeEnvironment* policy.

See also *privateEnvironment*.

filter

Type: Dictionary (“environment” | “sandbox” | “tools” -> List of Strings)

The filter keyword allows to restrict the environment variables, tools and sandboxes inherited from upstream recipes. This way a recipe can effectively restrict the number of package variants.

The filters specifications may use shell globbing patterns. As a special extension there is also a negative match if the pattern starts with a “!”. Such patterns will filter out entries that have been otherwise included by previous patterns in the list (e.g. by inherited classes).

Example:

```
filter:
    environment: [ "*_MIRROR" ]
    tools: [ "*toolchain*", "!host-toolchain" ]
    sandbox: [ "*" ]
```

In the above example the recipe would inherit only environment variables that end with “_MIRROR”. All other variables are unset. Likewise all tools that have “toolchain” in their name are inherited, except the “host-toolchain”. Anything is accepted as sandbox which would also be the default if left out.

Warning: The filter keyword is still experimental and may change in the future or might be removed completely.

fingerprintScript[{Bash,Pwsh}]

Type: String

The fingerprint script is executed before a package is built or downloaded. The script is supposed to gather information about whatever external resource is used in the recipe and output that in a stable format. The actual output is irrelevant to Bob as long as it detects all relevant external influences of the build result and that subsequent executions of the script generate the same output if the external components have not changed.

Note: Defining a `fingerprintScript` does not enable fingerprinting yet. At least one inherited class, used tool or the recipe itself must enable it by setting *fingerprintIf* accordingly.

Bob will incrementally rebuild the package whenever the fingerprint script output changes. The output of the script is also used to tag binary artifacts. An artifacts will only be downloaded if the fingerprint script generated the same output. This enables Bob to prevent false sharing of binary artifacts across otherwise incompatible machines.

The fingerprint script is executed in an empty temporary directory. It does not have access to any dependencies of the recipe nor to the checked out sources. A subset of environment variables of the package (see *{checkout,build,package}Vars*) as defined by *fingerprintVars* is set. The usual bash options are applied (`nounset`, `errexit`, `pipefail`) too. If the script returns with a non-zero exit status it will fail the build. The output on stderr is ignored but will be displayed in the error message if the script fails. The scripts of inherited classes are concatenated (but only if their *fingerprintIf* condition did not evaluate to `false`). Any fingerprint scripts that are defined by used tools (see *provideTools*) are concatenated too.

The suffix of the keyword determines the language of the script. Using the `Bash` suffix (`fingerprintScriptBash`) defines a script that is interpreted with `bash`. Likewise, the `Pwsh` suffix (`fingerprintScriptPwsh`) defines a PowerShell script. Which language is used at build time is determined by the *scriptLanguage* key or, if nothing was specified, by the project *scriptLanguage* setting. The keyword without a suffix (`fingerprintScript`) is interpreted in whatever language is finally used at build time. If both the keyword with the build time language suffix and without a suffix are present then the keyword with the build language suffix takes precedence.

For common fingerprint tasks the following built-in functions are provided by Bob:

bob-libc-version Checks the host architecture together with the type and version of the libc library. The C-compiler that is used can be configured either with the first parameter of the function or it will use the `CC` environment variable. If both are not set the `cc` command is used.

This helper should typically be used with the host compiler recipe.

bob-libstdc++-version Checks the host architecture together with the type and version of the C++ standard library. The C++-compiler that is used can be configured either with the first parameter of the function or it will use the `CXX` environment variable. If both are not set the `c++` command is used.

This helper should typically be used with the host compiler recipe.

bob-hash-libraries Takes a list of libraries as arguments that should be hashed. This will link an executable that links with the given libraries, call `ldd` and hash all used libraries.

Use this helper if no other information is available about a library / libraries except the name.

These helpers can be used in the fingerprint script. Their actual implementation and output may change in the future as more systems are supported by Bob.

fingerprintIf

Type: String | Boolean | `null` | IfExpression

By default no fingerprinting is done unless at least one inherited class, used tool or the recipe explicitly enables it. This is done by either setting `fingerprintIf` to `True` or by a boolean expression string that is evaluated to `True`. This can be used e.g. to apply a fingerprint only if the package is built for the host and not cross-compiled. The *fingerprintScript[{Bash,Pwsh}]* of the recipe is only evaluated if `fingerprintIf` is true. Otherwise the fingerprint script is ignored even if some other class enables fingerprinting. Setting `fingerprintIf` to `False` will unconditionally disable the associated `fingerprintScript`.

A `null` value has a special semantic. It does not enable fingerprinting for a package but retains the associated `fingerprint` script. If some inherited class, the recipe or a used tool does enable fingerprinting then the fingerprint script will still be evaluated. This is useful for classes to provide some fingerprinting functions but, unless an inheriting recipe defines a `fingerprint` script, does not enable fingerprinting of the recipe by itself.

Examples:

```
fingerprintIf: True                          # unconditionally enable fingerprinting
fingerprintIf: "$(eq,${TOOLCHAIN},host)"  # boolean experession
fingerprintIf: null                          # same as if unset
fingerprintIf: !expr |                       # IfExpression
                  "${TOOLCHAIN}" == "host"
```

If not given it defaults to `null`.

fingerprintVars

Type: List of strings

This declares the subset of the environment variables of the affected package that should be set during the execution of the `fingerprintScript`. Only variables that are selected by *{checkout,build,package}Vars* can be used. It is not an error that a variable listed here is unset. The variables will only be set if the corresponding `fingerprintScript` is enabled too.

Note: Before Bob 0.16 (see *fingerprintVars* policy) all environment variables of the affected package were set during the execution of the `fingerprintScript`. If the policy is set to the old behaviour then this key will be ignored and has no effect.

---

inherit

Type: List of Strings

Include classes with the given name into the current recipe. Example:

```
inherit: [cmake]
```

Classes are searched in the `classes/` directory with the given name. The syntax of classes is the same as the recipes. In particular classes can inherit other classes too. The inheritance graph is traversed depth first and every class is included exactly once.

All attributes of the class are merged with the attributes of the current recipe. If the order is important the attributes of the class are put in front of the respective attributes of the recipe. For example the scripts of the inherited class of all steps are inserted in front of the scripts of the current recipe.

metaEnvironment

Type: Dictionary (String -> String)

metaEnvironment variables behave like *privateEnvironment* variables. They overrule other environment variables and can be used in all steps, but substitution is not available. In addition all metaEnvironment variables are added to the audit no matter they are used in a step or not. This predestines metaEnvironment variables to add the license type or version of a package.

The *bob-query-meta* command can be used to retrieve metaEnvironment variables.

multiPackage

Type: Dictionary (String -> Recipe)

By utilizing the `multiPackage` keyword it is possible to unify multiple recipes into one. The final package name is derived from the current recipe name by appending the key under multiPackage separated by a "-". If an empty string is given as key the separator is not inserted. Nested multiPackages are also supported. Every level of multiPackages appends another suffix to the package name. The following example recipe foo.yaml declares four packages: foo, foo-bar-x, foo-bar-y and foo-baz:

```
multiPackage:
   "":
      ...
   bar:
      buildScript: ...
      multiPackage:
         x:
            packageScript: ...
         y:
            packageScript: ...
   baz:
      ...
```

All other keywords on the same level are treated as an anonymous base class that is inherited by the defined multiPackage's. That way you can have common parts to all multiPackage entries and keep just the distinct parts separately.

A typical use case for this feature are recipes for libraries. There are two packages that are built from a library: a `-target` packet that has the shared libraries needed during runtime and a `-dev` packet that has the header files and other needed files to link with this library.

privateEnvironment

Type: Dictionary (String -> String)

Defines environment variables just for the current recipe. Any inherited variables with the same name of the upstream recipe or others that were consumed from the dependencies are overwritten. All variables defined or replaced by this keyword are private to the current recipe.

Example:

```
privateEnvironment:
   APPLY_FOO_PATCH: "no"
```

The privateEnvironment of the recipe and inherited classes are merged together. The exact way of merging is subject to the *mergeEnvironment* policy.

See also *environment*.

provideDeps

Type: List of Patterns

The `provideDeps` keyword receives a list of dependency names. These must be dependencies of the current recipe, i.e. they must appear in the `depends` section. It is no error if the condition of such a dependency evaluates to false. In this case the entry is silently dropped. To specify multiple dependencies with a single entry shell globbing patterns may be used.

Provided dependencies are subsequently injected into the dependency list of the upstream recipe that has a dependency to this one (if `deps` is included in the `use` attribute of the dependency, which is the default). This works in a transitive fashion too, that is provided dependencies of a downstream recipe are forwarded to the upstream recipe too.

Example:

```
depends:
    - common-dev
    - communication-dev
    - config

...

provideDeps: [ "*-dev" ]
```

Bob will make sure that the forwarded dependencies are compatible in the injected recipe. That is, any duplicates through injected dependencies must result in the same package being used.

provideTools

Type: Dictionary (String -> Path | Tool-Dictionary)

The `provideTools` keyword defines an arbitrary number of build tools that may be used by other steps during the build process. In essence the definition declares a path (and optionally several library paths) under a certain name that, if consumed, are added to `$PATH` (and `$LD_LIBRARY_PATH`) of consuming recipes. Example:

```
provideTools:
   host-toolchain:
      path: bin
      libs: [ "sysroot/lib/i386-linux-gnu", "sysroot/usr/lib", "sysroot/usr/lib/i386-
↪linux-gnu" ]
      netAccess: True
      environment:
         CC: gcc
         LD: ld
      fingerprintIf: True
      fingerprintScript: |
         bob-libc-version gcc
```

The `path` attribute is always needed. The `libs` attribute, if present, must be a list of paths to needed shared libraries. Any path that is specified must be relative. If the recipe makes use of existing host binaries and wants to provide them as tool you should create symlinks to the host paths.

The `netAccess` attribute allows the tool to request network access during build/package step execution even if the recipe has not requested it (see *{build,package}NetAccess*). The network access is only granted if the tool is used. This attribute might be needed if the recipe cannot know if a particular tool actually requires network access. A prominent example are proprietary compilers that need to talk to a license server. Unless a package is built with such a compiler the network access is not needed.

The `environment` attribute provides the ability to define environment variables that are automatically picked up by the recipe where the tool is used. This allows for much more fine-grained variable provisioning than *provideVars*. If multiple tools are used in a recipe they must define distinct variables because no particular order between tools is defined. The values defined in this attribute are subject to variable substitution.

The `fingerprintScript` attribute defines a fingerprint script like in a normal recipe by *fingerprintScript[{Bash,Pwsh}]*. A fingerprint script defined by a tool is implicitly added to the fingerprint scripts of all recipes that use the particular tool. Use it to automatically apply a fingerprint to all recipes whose result will depend on the host environment by using the tool. The `fingerprintIf` and `fingerprintVars` attributes are handled the in the same way.

If no attributes except `path` are present the declaration may be abbreviated by giving the relative path directly:

```
provideTools:
   host-toolchain: bin
```

provideVars

Type: Dictionary (String -> String)

Declares arbitrary environment variables with values that should be passed to the upstream recipe. The values of the declared variables are subject to variable substitution. The substituted values are taken from the current package environment. Example:

```
provideVars:
    ARCH: "arm"
    CROSS_COMPILE: "arm-linux-${ABI}-"
```

By default these provided variables are not picked up by upstream recipes. This must be declared explicitly by a `use: [environment]` attribute in the dependency section of the upstream recipe. Only then are the provided variables merged into the upstream recipes environment.

provideSandbox

Type: Sandbox-Dictionary

The `provideSandbox` keyword offers the current recipe as sandbox for the upstream recipe. Any consuming upstream recipe (via `use: [sandbox]`) will be built in a sandbox where the root file system is the result of the current recipe. The initial `$PATH` is defined with the required `paths` keyword that should hold a list of paths. This will completely replace `$PATH` of the host for consuming recipes.

Attention: The build result is considered to be an invariant of such a sandbox (see *sandboxInvariant* policy). This implies that recipes shall produce the same result whether the sandbox is used or not.

Optionally there can be a `mount` keyword. With `mount` it is possible to specify additional paths of the host that are mounted read only in the sandbox. The paths are specified as a list of either strings or lists of two or three elements. Use a simple string when host and sandbox path are the same without any special options. To specify distinct paths use a list with two entries where the host path is the first element and the second element is the path in the sandbox.

The long format with three items additionally allows to specify a list of mount flags. The shorter formats described above have no flags set. The following flags are available:

- `nofail`: Don't fail the build if the host path is not available. Instead drop the mount silently.
- `nolocal`: Do not use this mount in local builds.
- `nojenkins`: Do not use this mount in Jenkins builds.
- `rw`: Mount as read-writable instead of read-only.

Additionally there can be an optional `environment` keyword. This works like the *provideVars* keyword and defines environment variables that are picked up by the depending recipe. In contrast to `provideVars` the variables defined here are only consumed if the sandbox is actually used (i.e. the parent recipe defined `sandbox` in the `use` section and the user builds with `--sandbox`). In this case the variables defined here have a higher precedence that the ones defined in `provideVars`.

Variable substitution is possible for the mount paths and environment variables. See *String substitution* for the available substations. The mount paths are also subject to an additional variable expansion when a step using the sandbox *is actually executed*. This can be useful e.g. to expand variables that are only available on the build server. Example:

```
provideSandbox:
    paths: ["/bin", "/usr/bin"]
    mount:
        - "/etc/resolv.conf"
        - "${MYREPO}"
        - "\\$HOME/.ssh"
        - ["\\$SSH_AUTH_SOCK", "\\SSH_AUTH_SOCK", [nofail, nojenkins]]
    environment:
        AUTOCONF_BUILD: "x86_64-linux-gnu"
```

The example assumes that the variable `MYREPO` was set somewhere in the recipes. On the other hand `$HOME` is expanded later at build time. This is quite useful on Jenkins because the home directory there is certainly different from the one where Bob runs. The last entry shows two mount option being used. This line mounts the ssh-agent socket into the sandbox if available. This won't be done on Jenkins at all and the build will proceed even if `$SSH_AUTH_SOCK` is unset or invalid. Note that such variables have to be in the *whitelist* to be available to the shell.

Note: The mount paths are considered invariants of the build. That is changing the mounts will neither automatically cause a rebuild of the sandbox (and affected packages) nor will binary artifacts be re-fetched.

The user might amend the mount and search paths in `default.yaml` by a *sandbox* entry.

relocatable

Type: Boolean

If `True` Bob can assume that the package result is independent of the actual location in the file system. Usually all packages should be relocatable as this is a fundamental assumption of Bob's working model. There might be particular tools, though, that depend on their installed location. For such tools the property should be set to `False`.

If the property is not set the default will be `True` unless the recipe defines at least one tool. In this case the default value is `False` if the *allRelocatable* policy is unset or disabled. If the policy is set the default value is always `True`. Inherited values from a class will be overwritten by the recipe or inheriting class.

root

Type: Boolean

Recipe attribute which defaults to False. If set to True the recipe is declared a root recipe and becomes a top level package. There must be at least one root package in a project.

scriptLanguage

Type: Enumeration: `bash`, `PowerShell`.

Defines the scripting language which is used to run the `{checkout,build,package,fingerprint}Script` scripts when building the package. If nothing is specified the *scriptLanguage* setting from config.yaml is used. Depending on the chosen language Bob will either invoke `bash` or `pwsh`/`powershell` as script interpreter. In either case the command must be present in `$PATH`/`%PATH%`.

shared

Type: Boolean

Marking a recipe as shared implies that the result may be shared between different projects or workspaces. Only completely deterministic packages may be marked as such. Typically large static packages (such as toolchains) are enabled as shared packages. By reusing the result the hard disk usage can be sometimes reduced drastically.

The exact behaviour depends on the build backend. Currently the setting has no influence on local builds. On Jenkins the result will be copied to a separate directory in the Jenkins installation and will be used from there. This reduces the job workspace size considerably at the expense of having artifacts outside of Jenkins's regular control.

Project configuration (config.yaml)

The file `config.yaml` holds all static configuration options that are not subject to be changed when building packages. The following sections describe the top level keys that are currently understood. The file is optional or could be empty.

bobMinimumVersion

Type: String

Defines the minimum required version of Bob that is needed to build this project. Any older version will refuse to build the project. The version number given here might be any prefix of the actual version number, e.g. "0.1" instead of the actual version number (e.g. "0.1.42"). Bob's version number is specified according to Semantic Versioning. Therefore it is usually only needed to specify the major and minor version.

The version string has to be compliant to Python PEP 440. It is allowed to specify pre-release versions (e.g. `0.16.0rc1`) and even development versions (e.g. `0.15.1.dev42`). A version without pre-release suffix is considered more recent than a version with a pre-release suffix. The development release number is only relevant if the main version and the pre-release versions are equal.

layers

Type: List of strings

The `layers` section consists of a list of layer names that are then expected in the `layers` directory relative to the `conig.yaml` referencing them:

```
layers:
    - myapp
    - bsp
```

Layers that are not named in this section but that are present in the `layers` directory are ignored. Layers that are named but that do not exist lead to a parse error. Layers can be nested, that is, a layer can itself have layers below it.

The order of layers is important with respect to settings made by `default.yaml` in the various layers. The project root has the highest precedence. The layers in `config.yaml` are named from highest to lowest precedence. A layer with a higher precedence can override settings from layers of lower precedence.

See *Configuration* for more information.

plugins

Type: List of strings

Plugins are loaded in the same order as listed here. For each name in this section there must be a .py-file in the `plugins` directory next to the recipes. For a detailed description of plugins see *Plugins*.

policies

Type: Dictionaly (Policy name -> Bool)

The policies section allows to individually set policies to their old (disabled) or new (enabled) behaviour. See *Defined policies* for a list of all policies and their rationale.

Example:

```
policies:
    relativeIncludes: False
```

This will explicitly request old behaviour for the *relativeIncludes* policy.

scriptLanguage

Type: Enumeration: `bash`, `PowerShell`.

Defines the scripting language which is used to run the `{checkout,build,package,fingerprint}Script` scripts. Defaults to `bash`. Might be overrided on a case-by-case basis in a class or recipe with *scriptLanguage*. Depending on the chosen language Bob will either invoke `bash` or `pwsh`/`powershell` as script interpreter. In either case the command must be present in `$PATH`/`%PATH%`.

User configuration (default.yaml)

The `default.yaml` file holds configuration options that may be overridden by the user. Most commands will also take an '-c' option where any number of additional configuration files with the same syntax can be specified.

Like git there are three locations where bob is looking for a configuration file. They are parsed in descending order making it possible to locally override global settings.:

```
/etc/bobdefault.yaml:
    System-wide configuration file.

$XDG_CONFIG_HOME/bob/default.yaml resp. ~/.config/bob/default.yaml:
    User-specific configuration File. If XDG_CONFIG_HOME is not set
    ~/.config/bob/default.yaml is used.

./default.yaml.
    Workspace-specific configuration file.
```

User configuration files may optionally include other configuration files. These includes are parsed *after* the current file, meaning that options of included configuration files take precedence over the current one. Included files do not need to exist and are silently ignored if missing. Includes are specified without the .yaml extension:

```
include:
    - overrides
```

Note: Depending on the *relativeIncludes* policy the base directory from where includes are resolved is different. Normally files are included relative to the currently processed file unless the *relativeIncludes* policy is disabled. In this case files included by `default.yaml` and by the command line use the project root directory as base directory.

It is possible for plugins to define additional settings. See *Plugin settings* for more information. Their meaning and typing is completely controlled by the respective plugin and Bob will just pass the data as-is without further interpretation.

User configuration files may also require specific files to be included. The `require` keyword behaves just like the `include` keyword with the exception that Bob raises a parsing error if the file to be included cannot be found:

```
require:
   - overrides
   - /path/to/some/file
```

Required include files have a lower precedence that optional include files.

environment

Type: Dictionary (String -> String)

Specifies default environment variables. Example:

```
environment:
   # Number of make jobs is determined by the number of available processors
   # (nproc).  If desired it can be set to a specific number, e.g. "2". See
   # classes/make.yaml for details.
   MAKE_JOBS: "nproc"
```

If the *cleanEnvironment* policy is enabled then these variables are subject to *String substitution* with the current OS environment. This allows to take over certain variables from the OS environment in a controlled fashion.

whitelist

Type: List of Strings

Specifies a list of environment variable keys that should be passed unchanged to all scripts during execution. The content of these variables are considered invariants of the build. It is no error if any variable specified in this list is not set. By default the following environment variables are passed to all scripts: `TERM`, `SHELL`, `USER` and `HOME`. The names given with `whitelist` are *added* to the list and does not replace the default list.

Example:

```
# Keep ssh-agent working
whitelist: ["SSH_AGENT_PID", "SSH_AUTH_SOCK"]
```

archive

Type: Dictionary or list of dictionaries

The `archive` key configures the default binary artifact server(s) that should be used. It is either directly an archive backend entry or a list of archive backends. For each entry at least the `backend` key must be specified. Optionally

there can be a `flags` key that receives a list of various flags, in particular for what operations the backend might be used. See the following list for possible flags. The default is `[download, upload]`.

download Use this archive to download artifacts. Note that you still have to explicitly enable downloads on Jenkins servers. For local builds the exact download behaviour depends on the build mode (release vs. develop).

upload Use this archive to upload artifacts. To actually upload to the archive the build must be performed with uploads enabled (`--upload`).

nofail Don't fail the build if the upload or download from this archive fails. In any case it is never an error if a download does not find the requested archive on the backend. This option additionally suppresses other errors such as unknown hosts or interrupted transfers.

nolocal Do not use this archive in local builds.

nojenkins Do not use this archive in Jenkins builds.

Depending on the backend further specific keys are available or required. See the following table for supported backends and their configuration.

Backend	Description
none	Do not use a binary repository (default).
azure	Microsoft Azure Blob storage backend. The account must be specified in the `account` key. Either a `key` or a `sasToken` may be set to authenticate, otherwise an anonymous access is used. Finally the container must be given in `container`. Requires the `azure-storage-blob` Python3 library to be installed.
file	Use a local directory as binary artifact repository. The directory is specified in the `path` key as absolute path. The optional `fileMode` and `directoryMode` keys take the desired access modes as numeric value to override the default umask derived modes.
http	Uses a HTTP server as binary artifact repository. The server has to support the HEAD, PUT and GET methods. The base URL is given in the `url` key. The optional `sslVerify` boolean key controls whether to verify the SSL certificate.
shell	This backend can be used to execute commands that do the actual up- or download. A `download` and/or `upload` key provides the commands that are executed for the respective operation. The configured commands are executed by bash and are expected to copy between the local archive (given as `$BOB_LOCAL_ARTIFACT`) and the remote one (available as `$BOB_REMOTE_ARTIFACT`). See the example below for a possible use with `scp`.

The directory layouts of the `azure`, `file`, `http` and `shell` (`$BOB_REMOTE_ARTIFACT`) backends are compatible. If multiple download backends are available they will be tried in order until a matching artifact is found. All available upload backends are used for uploading artifacts. Any failing upload will fail the whole build.

Note: The uploaded artifacts can be managed by *bob-archive*. It might be wise to use different repositories for release builds and for continous builds to keep them separated.

Example:

```
archive:
   backend: http
   url: "http://localhost:8001/upload"
```

It is also possible to use separate methods for upload and download:

```
archive:
    -
```

(continues on next page)

(continued from previous page)

```
        backend: http
        url: "http://localhost:8001/archive"
        flags: [download]
    -
        backend: shell
        upload: "scp -q ${BOB_LOCAL_ARTIFACT} localhost:archive/${BOB_REMOTE_ARTIFACT}
↪"
        download: "scp -q localhost:archive/${BOB_REMOTE_ARTIFACT} ${BOB_LOCAL_
↪ARTIFACT}"
        flags: [upload]
```

The azure backend can also be used in conjunction with the http backend in case of publicly readable containers. Given a typical configuration like this:

```
archive:
    backend: azure
    account: <account>
    container: <container name>
    key: <access key>
```

the anonymous access to the container can be used like this:

```
archive:
    backend: http
    url: https://<account>.blob.core.windows.net/<container name>
    flags: [download]
```

The `flags: [download]` makes sure that Bob does not try to upload artifacts in case other backends are configured too.

scmOverrides

Type: List of override specifications

SCM overrides allow the user to alter any attribute of SCMs (*checkoutSCM*) without touching the recipes. They are quite useful to change e.g. the server url or to override the branch of some SCMs. Overrides are applied after string substitution. The general syntax looks like the following:

```
scmOverrides:
  -
    match:
      url: "git@acme.com:/foo/repo.git"
    del: [commit, tag]
    set:
      branch: develop
    replace:
      url:
        pattern: "foo"
        replacement: "bar"
    if: !expr |
      "${BOB_RECIPE_NAME}" == "foo"
```

The `scmOverrides` key takes a list of one or more override specifications. You can select overrides using a `if` expression. If `if` condition evaluates to true the override is first matched via pattens that are in the `match` section. All entries under `match` must be matching for the override to apply. The right side of a match entry can use shell globbing patterns.

If an override is matching the actions are then applied in the following order:

- `del`: The list of attributes that are removed.
- `set`: The attributes and their values are taken, overwriting previous values.
- `replace`: Performs a substitution based on regular expressions. This section can hold any number of attributes with a `pattern` and a `replacement`. Each occurrence of `pattern` is replaced by `replacement`.

All overrides values are mangled through *String substitution*. Mangling is performed during calculation of the checkoutStep so that the full environment for this step is available for substitution.

alias

Type: Dictionary (String -> String)

Aliases allow a string to be substituted for the first step of a *relative location path*:

```
alias:
   myApp: "host/files/group/app42"
   allTests: "//*-unittest"
```

See *Alias substitution* for the rules that apply to aliases.

command

Type: Dict of command dicts

Override default command settings:

```
command:
    dev:
        [..]
    build:
        [..]
    graph:
        [..]
```

build / dev

Set default build arguments here. See *bob-dev* or *bob-build* for details.:

```
command:
    dev:
        no_logfile: True
        build_mode: "build-only"
    build:
        verbosity: 3
        download: No
```

The following table lists possible arguments and their type:

Key	Command line switch	Type
always_checkout	`--always-checkout`	List of strings (regular expression patterns)
audit	`--[no]-audit`	Boolean
build_mode	`-b|-B|--normal`	String (`normal`, `build-only` or `checkout-only`)
clean	`--clean | --incremental`	Boolean
clean_checkout	`--clean-checkout`	Boolean
destination	`--destination`	String (Path)
download	`--download`	String (`yes`, `no`, `deps`, `forced`, `forced-deps`, `forced-fallback` or `packages=<packages>`)
force	`-f`	Boolean
link_deps	`--[no-]link-deps`	Boolean
no_deps	`-n`	Boolean
no_logfiles	`--no-logfiles`	Boolean
sandbox	`--[no-]sandbox`	Boolean
upload	`--upload`	Boolean
verbosity	`-q | -v`	Integer (-2[quiet] .. 3[verbose], default 0)

graph

Set default graph arguments here. See *bob-graph* for details.:

```
command:
    graph:
        options:
            d3.dragNodes: True
        type: "d3"
        max_depth: 2
```

Supported arguments and their type:

Key	Type
options	Dictonary of String key value pairs
type	“d3” or “dot”
max_depth	Integer

hooks

Hooks are other programs or scripts that can be executed by Bob at certain points, e.g. before or after a build. Unless otherwise noted they are executed with the project root directory as working directory. Example:

```
hooks:
    postBuildHook: ./contrib/notify.sh
```

where `contrib/notify.sh` is:

```
#!/bin/bash
HEADLINE="Bob build finished"
BODY="The build in $PWD has finished: $1"
```

(continues on next page)

(continued from previous page)

```
if [[ ${XDG_CURRENT_DESKTOP:-unknown} == KDE ]] ; then
    kdialog --passivepopup "$BODY" 10 --title "$HEADLINE"
else
    notify-send -u normal -t 10000 "$HEADLINE" "$BODY"
fi
```

The currently supported hooks are described below.

preBuildHook The pre-build hook is run directly before a local build (bob dev / bob build). It receives the paths of all packages that are built as arguments.

If the hook returns with a non-zero status the build will be interrupted.

postBuildHook The post-build hook is run after a local build finished, regardless if the build succeeded or failed. It receives the status as first argument (`success` or `fail`) and the relative paths to the workspaces of the results as further arguments.

The return status of the hook is ignored.

rootFilter

Type: List of Strings

Filter root recipes. The effect of this is a faster package parsing due to the fact, that the tree is not build for filtered roots.

Works like the *filter* keyword.

sandbox

Type: Sandbox-Dictionary

The default paths and mounts of a sandbox are defined by the *provideSandbox* keyword. The `sandbox` section in the user configuration allows to specify additional mounts and additional search paths. The format of the settings is the same as in the *provideSandbox* keyword.

Example:

```
sandbox:
    mount:
        - [ "$HOME/bin", "/mnt" ]
    paths:
        - /mnt
```

The search paths from `paths` are added to `$PATH` in reverse order so that later entries have a higher precedence. In contrast to `provideSandbox` *no* variable substitution is possible for the mounts. The mount paths are still subject to shell variable expansion when a step using the sandbox *is actually executed*, though.

The example above will mount the `bin` directory of the users home directory as `/mnt` inside the sandbox. The `/mnt` directory will be in `$PATH` before any other search directory of the sandbox but still after any used tool (if any).

ui

Type: Dictionary

Specifies options of user interface.

color Color mode of console output. Can be also overridden by command line option `--color`.

never No colors in output

always Use colors in output

auto Use colors only when TTY console detected (default)

1.3.4 Policies

The Bob Policy mechanism provides backwards compatibility as a first-class feature. Most of the general documentation below is taken from CMake (CC BY 2.5) as the design goals are identical.

Motivation

Bob is an evolving project. The developers strive to support existing projects as much as possible as changes are made. Unfortunately there are some cases where it is not possible to fix bugs and preserve backwards compatibility at the same time. The Bob policies try to make the transition as smooth as possible without stalling the development of the tool due to legacy design decisions or bugs.

Design Goals

The design goals for the Bob Policy mechanism were as follows:

1. Existing projects should build with versions of Bob newer than that used by the project authors
   - Users should not need to edit code to get the projects to build
   - Warnings may be issued but the projects should build
2. Correctness of new interfaces or bugs fixed in old ones should not be inhibited by compatibility requirements
   - Any reduction in correctness of the latest interface is not fair to new projects
3. Every change to Bob that may require changes to project code should be documented
   - Each change should also have a unique identifier that can be referenced by warning and error messages
   - The new behavior is enabled only when the project has somehow indicated it is supported
4. We must be able to eventually remove code implementing compatibility with ancient Bob versions
   - Such removal is necessary to keep the code clean and allow internal refactoring
   - After such removal attempts to build projects written for ancient versions must fail with an informative message

Solution

We've introduced the notion of a policy for dealing with changes in Bob behavior. Each policy has:

- a unique name,
- a disabled (old) behavior that preserves compatibility with earlier versions of Bob,
- an enabled (new) behavior that is considered correct and preferred for use by new projects,
- documentation detailing the motivation for the change and the old and new behaviors.

Projects may configure the setting of each policy to request old (disabled) or new (enabled) behavior. When Bob encounters recipes that may be affected by a particular policy it checks to see whether the project has set the policy. If the policy has been set (enabled or disabled) then Bob follows the behavior specified. If the policy has not been set then the old behavior is used but a warning is produced telling the project author to set the policy.

Setting Policies

Policies can be set in `config.yaml`. They are either set implicitly by *bobMinimumVersion* or explicitly in the *policies* section.

In most cases a project release should simply set a policy version corresponding to the release version of Bob for which the project is written. Setting the policy version requests new behavior for all policies introduced in the corresponding version of Bob or earlier. Policies introduced in later versions are marked as not set in order to produce proper warning messages. The policy version is set using the *bobMinimumVersion* key in `config.yaml` of the project.

For example, the configuration:

```
bobMinimumVersion: "0.13"
```

will request new behavior for all policies introduced in Bob 0.13 or earlier. Of course one should replace "0.13" with a higher version as necessary.

When a new version of Bob is released that introduces new policies it will still build old projects because they do not request new behavior for any of the new policies. When starting a new project one should always specify the most recent release of Bob to be supported as the policy version level. This will make sure that the project is written to work using policies from that version of Bob and not using any old behavior.

Additionally, each policy may be set individually to help project authors incrementally convert their projects to use new behavior or silence warnings about dependence on old behavior. The *policies* section in `config.yaml` may be used to explicitly request old or new behavior for a particular policy. This overrides any defaults that were set by *bobMinimumVersion*.

Defined policies

relativeIncludes

Introduced in: 0.13

User configuration files (e.g. `default.yaml` or files passed by `-c` on the command line) can include other configuration files in the `include` section. Versions of Bob before 0.13 included these files always relative to the root of the project configuration.

Starting with Bob 0.13 it is possible to have global and user specific configuration files too. To allow inclusion of further files from these configuration files the include location was changed to "file relative" includes. That is, any included file is seared relative to the currently processed file.

Old behaviour Include further files from `default.yaml` and command line passed files relative to the project root directory. Global configuration files use the new policy in any case.

New behaviour All files are included relative to the currently processed file.

cleanEnvironment

Introduced in: 0.13

The environment variables that are consumed in recipes are fundamentally calculated from the recipes only. Bob has the notion of white listed variables that shall not influence the build result but should still be set during execution. Their value is kept unchanged from the current OS environment when building packages.

Previously the current set of environment variables during package calculation started with the ones named by *whitelist* in `default.yaml`. This made these variables bound to the value that was set during package calculation. Especially on Jenkins setups this is wrong as the machine that configures the Jenkins may have a different OS environment than the Jenkins executors/slaves. Also using such variables in the recipes made the calculated packages dependent on the state of the local machine.

Old behavior Environment computation in root recipes starts with white listed variables of the current OS environment.

New behavior Package computation starts with a clean environment. The default environment variables (*environment*) may reference OS environment variables and are taken as initial environment for package computation. White listed variables are only available while building packages and are taken verbatim from the current OS execution environment.

tidyUrlScm

Introduced in: 0.14

Historically the URL SCM was not tracking the checkout directory but the individual files that are downloaded by the SCM. This has the advantage that it is possible to download more than one file into the same directory. There are a couple of major disadvantages, though:

1. When extracting multiple archives in the same directory it might be possible that some files are overwritten.
2. Any extracted files are not tracked by Bob and will be left untouched in develop mode when the recipe is updated. This leads to stale files in the src-directory and will typically prevent that matching binary artifacts are found.
3. Trying to reliably apply patches across SCM updates is tricky because files are only overwritten and never garbage collected.

Starting with 0.14 Bob will manage the whole checkout directory. This unifies the behaviour with the other SCMs and solves the above disadvantages. This change might break existing projects because with the new behaviour it is not possible to put multiple URL SCMs into the same directory.

Old behavior Bob tracks only the downloaded file across recipe updates. Upon changes only the involved file is moved away and the new one is downloaded. Extracted files from archives stay in workspace.

New behavior The whole directory where the URL SCM is checked out is tracked by Bob. Changing the recipe will move away the whole checkout directory, including any possibly extracted files.

allRelocatable

Introduced in: 0.14

When up- or downloading binary artifacts Bob has to make sure that the artifact is independent of the actual location in the file system. This is not always the case for tools that are executed on the build host. Historically Bob assumed that all packages that were created from recipes that define at least one tool are not relocatable. Such packages were not up- or downloaded except when building in a sandbox because the sandbox virtualises the paths and makes them deterministic everywhere.

Starting with Bob 0.14 the *relocatable* property allows to specify this more fine grained. To not break existing recipes the `relocatable` property has a default value compatible to the old behaviour described above. Because this

heuristic is quite pessimistic and almost always wrong the `allRelocatable` policy switches the default to *always relocatable*.

Old behavior The default value of the *relocatable* property is `True` unless the recipe defines at least one tool. In this case the default value is `False`.

New behavior The default value of the *relocatable* property is always `True`.

Starting with Bob 0.15 the new behavior will also enable fingerprinting if a fingerprint script has been defined. In case of a non-relocatable package the fingerprint will additionally encode the workspace path. This enables safe artifact exchange even outside of a sandbox.

offlineBuild

Introduced in: 0.14

Bob assumes that build and package steps are always deterministic. It is therefore usually not a good idea to access the network other than in the checkout step where the external source code is fetched. Bob has the ability to isolate the network when building a package in a sandbox. If the network must still be accessible during build and/or package steps the recipe might set the respective properties (see *{build,package}NetAccess*).

Old behavior External network access is always possible.

New behavior During checkout steps the external network is always accessible. When building inside a sandbox the network will be isolated during build and package steps by default. A recipe might override this to still allow network access if required.

sandboxInvariant

Introduced in: 0.14

Traditionally the impact of a sandbox to the build has not been handled consistently. On one hand the actual usage of a sandbox was not relevant for binary artifacts. As such, an artifact that was built inside a sandbox was also used when building without the sandbox (and vice versa). On the other hand Bob did rebuild everything from scratch when switching between sandbox/non-sandbox builds. This inconsistent behavior is rectified by the `sandboxInvariant` policy that consistently declares builds as invariant of the sandbox.

Old behavior The sandbox is handled inconsistently. Bob will use binary artifacts across sandbox/non-sandbox builds but will rebuild clean if doing so. Changing the sandbox recipe will invalidate binary artifacts even when not using the sandbox.

New behavior The build result is always an invariant of the sandbox, that is the sandbox content and its usage makes no difference for Bob. This means that binary artifacts are used across sandbox/non-sandbox builds. Moving between sandbox/non-sandbox builds just triggers incremental builds of the affected packages. Changing the sandbox content will also trigger just incremental builds of affected packages.

In any case a recipe shall produce the same result regardless of the fact that a sandbox is used or not. This is and has always been a fundamental assumption of Bob with respect to binary artifacts. If the result of a recipe depends on the host environment then an appropriate environment variable defined by the sandbox should be used to let Bob detect this.

uniqueDependency

Introduced in: 0.14

Traditionally it was allowed to name a dependency more than once in a recipe. On the other hand the semantics were not well defined. The result was picked up only once. Due to the multiple references different variants of the

dependency could be created, though. This was detected only if the result of the dependencies was used. Otherwise this created unaddressable packages that cannot be built individually. It is also possible that, even if the packages themself are of the same variant, they might provide different dependencies or variables upwards. This is handled but not easily detectable by the user.

Old behavior Listing a dependency more than once in a recipe is tolerated. The result is only picked up once, though. Anything else (environment, tools, ...) is picked up at each instance again, possibly replacing previous definitions.

New behavior A dependency must only be named once. This is enforced *after* evaluating the `if` condition of the dependencies. It is therefore still possible to have multiple references to the same package given that only one reference is active. Everything else will result in a parsing error.

mergeEnvironment

Introduced in: 0.15

The *environment* and *privateEnvironment* sections of the recipes and classes it inherits from are merged when the packages are calculated. Traditionally this was done on a key-by-key basis without variable substitution. Keys from the recipe or an inherited class would simply shadow keys from later inherited classes. This had the effect that the definitions of later inherited classes were lost. It was also not possible to pick them up via variable substitution. Suppose the following simple recipe/class structure:

```
recipes/foo.yaml:
    inherit: [asan, werror]
    privateEnvironment:
        CFLAGS: "${CFLAGS:-} -DFOO=1"

classes/asan.yaml:
    privateEnvironment:
        CFLAGS: "${CFLAGS:-} -fsanitize=address"

classes/werror.yaml:
    privateEnvironment:
        CFLAGS: "${CFLAGS:-} -Werror"
```

Previously the definition of `CFLAGS` in the recipe would completely shadow the ones of the inherited classes. So the `CFLAGS` variable would only ever be amended with `-DFOO=1`. In contrast to this unintuitive result the new behavior is to take all classes into account and merge their values by applying the usual variable substitution.

Old behavior Environment keys in the recipe or earlier inherited classes shadow any later inherited classes. Variable substitution is done only with the first definition of the key. Any shadowed deviations are not examined. Given the above example the resulting `CFLAGS` would be `${CFLAGS:-} -DFOO=1`.

New behavior All environment keys are eligible to variable substitution. The definitions of the recipe has the highest precedence (i.e. it is substituted last). Declarations of classes are substituted in their inheritance order, that is, the last inherited class has the highest precedence. Given the above example the resulting `CFLAGS` would be `${CFLAGS:-} -fsanitize=address -Werror -DFOO=1`

secureSSL

Introduced in: 0.15

Due to historical reasons Bob did not check for SSL certificate errors everywhere. While most parts were already secure the git SCM and HTTPS archive backend were still insecure by default.

Old behavior The git SCM and the HTTPS archive backend do not check for certificate errors by default. May still be enabled by setting the corresponding `sslVerify` option to `True`.

New behavior Whenever a secure connection is used the certificate is checked. May be disabled selectively by setting the corresponding `sslVerify` option to `False`.

sandboxFingerprints

Introduced in: 0.16

When *Host dependency fingerprinting* was introduced, Bob initially used a shortcut and did not execute fingerprint scripts in the sandbox. This saved a bit of complexity and also relieved the build logic from the need to build the sandbox just to execute the fingerprint script. While the old approach was not producing wrong results it was overly pessimistic. It prevents sharing of any fingerprinted artifacts between sandbox and non-sandbox builds even if the fingerprint is the same.

Old behavior Fingerprint scripts are not executed in sandbox builds. Instead the sandbox image as a whole is used as fingerprint. This prevents the exchange of fingerprinted artifacts between sandbox- and non-sandbox-builds.

New behaviour Bob will execute fingerprint scripts in the sandbox too. Fingerprinted artifacts will be shared between sandbox- and non-sandbox-builds given the *fingerprintScript[{Bash,Pwsh}]* yields the same result. Fingerprint results for sandbox builds are cached in the binary artifact cache if available. This reduces the need to build the sandbox just to calculate the fingerprint.

Old artifacts that were built in a sandbox will not be found anymore in the artifact cache. They will have to be built again. Non-sandbox build artifacts are not affected.

fingerprintVars

Introduced in: 0.16

When then *fingerprintScript[{Bash,Pwsh}]* mechanism was introduced in Bob 0.15 there was no dedicated environment variable handling implemented for them. The simple policy was to pass all environment variables of the affected package to the `fingerprintScript`. Unfortunately this results in the repeated execution of identical scripts if the variables change between packages, even if they are not used by the `fingerprintScript`.

This policy adds the support for the new *fingerprintVars* key in the recipes. This key specifies a list of variables that the `fingerprintScript` uses.

Old behavior All variables of the fingerprinted package are passed to the `fingerprintScript`. The *fingerprintVars* settings are ignored. This might lead to unnecessary executions of identical `fingerprintScript` with different variable values.

New behavior Only the subset of environment variables, defined by *fingerprintVars* of the fingerprinted package is passed to the `fingerprintScript`. Other environment variables are unset but whitelisted variables (see *whitelist*) are still available.

1.3.5 Extending Bob

Bob may be extended through plugins. Right now the functionaly that can be tweaked through plugins is quite limited. If you can make a case what should be added to the plugin interface open an issue at GitHub or write to the mailing list.

See `contrib/plugins` in the Bob repository for an example.

Another extension are generators. See *Generators*

Plugins

Plugins can be put into a `plugins` directory as .py files. A plugin is only loaded when it is listed in `config.yaml` in the *plugins* section. Each plugin must provide a 'manifest' dict that must have at least an 'apiVersion' entry. The apiVersion is compared to the Bob version and must not be greater for the plugin to load. At minimum this looks like this:

```
manifest = {
    'apiVersion' : "0.2"
}
```

A plugin may define any number of properties and state trackers.

A property describes a key in a recipe that is parsed. The class handling the property is responsible to validate the data in the recipe and store the value. It must be derived from *bob.input.PluginProperty* The property class handles to inheritance between recipes and classes too.

A state tracker is a class that is invoked on every step when walking the dependency tree to instantiate the packages. The state tracker thus has the responsibility to calculate the final values associated with the properties for every package. Like properties there can be more than one state tracker. Any state tracker provided by a plugin must be derived from *bob.input.PluginState*-

Class documentation

Plugins might only access the following classes with the members documented in this manual. All other parts of the bob Python package namespace are considered internal and might change without notice.

class bob.input.**PluginProperty**(*present*, *value*)
Base class for plugin property handlers.

A plugin should sub-class this class to parse custom properties in a recipe. For each recipe an object of that class is created then. The default constructor just stores the *present* and *value* parameters as attributes in the object.

Parameters

- **present** (*bool*) – True if property is present in recipe
- **value** – Unmodified value of property from recipe or None if not present.

getValue()
Get (parsed) value of the property.

inherit(*cls*)
Inherit from a class.

The default implementation will use the value from the class if the property was not present. Otherwise the class value will be ignored.

isPresent()
Return True if the property was present in the recipe.

static validate(*data*)
Validate type of property.

Ususally the plugin will reimplement this static method and return True only if *data* has the expected type. The default implementation will always return True.

Parameters **data** – Parsed property data from the recipe

Returns True if data has expected type, otherwise False.

class bob.input.**PluginSetting**(*settings*)
Base class for plugin settings.

Plugins can be configured in the user configuration of a project. The plugin must derive from this class, create an object with the default value and assign it to 'settings' in the plugin manifest. The default constructor will just store the passed value in the `settings` member.

Parameters **settings** – The default settings

getSettings()
Getter for settings data.

merge(*other*)
Merge other settings into current ones.

This method is called when other configuration files with a higher precedence have been parsed. The settings in these files are first validated by invoking the `validate` static method. Then this method is called that should update the current object with the value of *other*.

The default implementation implements the following policy:

- Dictionaries are merged recursively on a key-by-key basis
- Lists are appended to each other
- Everything else in *other* reuucplaces the current settings

It is assumed that the actual settings are stored in the `settings` member variable.

Parameters **other** – Other settings with higher precedence

static validate(*data*)
Validate type of settings.

Ususally the plugin will reimplement this method and return True only if *data* has the expected type. The default implementation will always return True.

Parameters **data** – Parsed settings data from user configuration

Returns True if data has expected type, otherwise False.

class bob.input.**PluginState**
Base class for plugin state trackers.

State trackers are used by plugins to compute the value of one or more properties as the dependency tree of all recipes is traversed.

Attention: Objects of this class are tested for equivalence. The default implementation compares all members of the involved objects. If custom types are stored in the object you have to provide a suitable `__eq__` and `__ne__` implementation because Python falls back to object identity which might not be correct. If these operators are not working correctly then Bob may slow down considerably.

copy()
Return a copy of the object.

The default implementation uses copy.deepcopy() which should usually be enough. If the plugin uses a sophisticated state tracker, especially when holding references to created packages, it might be usefull to provide a specialized implementation.

onEnter(*env*, *properties*)
Begin creation of a package.

The state tracker is about to witness the creation of a package. The passed environment, tools and (custom) properties are in their initial state that was inherited from the parent recipe.

Parameters

- **env** (*Mapping[str, str]*) – Complete environment
- **properties** (Mapping[str, *bob.input.PluginProperty*]) – All custom properties

onFinish(*env*, *properties*)
Finish creation of a package.

The package was computed. The passed *env* and *properties* have their final state after all downstream dependencies have been resolved.

Parameters

- **env** (*Mapping[str, str]*) – Complete environment
- **properties** (Mapping[str, *bob.input.PluginProperty*]) – All custom properties

onUse(*downstream*)
Use provided state of downstream package.

This method is called if the user added the name of the state tracker to the `use` clause in the recipe. A state tracker supporting this notion should somehow pick up and merge the state of the downstream package.

The default implementation does nothing.

Parameters **downstream** (`bob.input.PluginState`) – State of downstream package

class bob.input.**Recipe**
Representation of a single recipe

Multiple instaces of this class will be created if the recipe used the `multiPackage` keyword. In this case the getName() method will return the name of the original recipe but the getPackageName() method will return it with some addition suffix. Without a `multiPackage` keyword there will only be one Recipe instance.

getName()
Get plain recipe name.

In case of a `multiPackage` multiple packages may be derived from the same recipe. This method returns the plain recipe name.

getPackageName()
Get the name of the package that is drived from this recipe.

Usually the package name is the same as the recipe name. But in case of a `multiPackage` the package name has an additional suffix.

isRoot()
Returns True if this is a root recipe.

class bob.input.**Package**
Representation of a package that was created from a recipe.

Usually multiple packages will be created from a single recipe. This is either due to multiple upstream recipes or different variants of the same package. This does not preclude the possibility that multiple Package objects describe exactly the same package (read: same Variant-Id). It is the responsibility of the build backend to detect this and build only one package.

getAllDepSteps(*forceSandbox=False*)
Return list of all dependencies of the package.

This list includes all direct and indirect dependencies. Additionally the used sandbox and tools are included too.

getBuildStep()
Return the build step of this package.

getCheckoutStep()
Return the checkout step of this package.

getDirectDepSteps()
Return list to the package steps of the direct dependencies.

Direct dependencies are the ones that are named explicitly in the `depends` section of the recipe. The order of the items is preserved from the recipe.

getIndirectDepSteps()
Return list of indirect dependencies of the package.

Indirect dependencies are dependencies that were provided by downstream recipes. They are not directly named in the recipe.

getMetaEnv()
meta variables of package

getName()
Name of the package

getPackageStep()
Return the package step of this package.

getRecipe()
Return Recipe object that was the template for this package.

getStack()
Returns the recipe processing stack leading to this package.

The method returns a list of package names. The first entry is a root recipe and the last entry is this package.

isRelocatable()
Returns True if the packages is relocatable.

class bob.input.**Step**
Represents the smallest unit of execution of a package.

A step is what gets actually executed when building packages.

Steps can be compared and sorted. This is done based on the Variant-Id of the step. See *bob.input.Step.getVariantId()* for details.

doesProvideTools()
Return True if this step provides at least one tool.

getAllDepSteps(*forceSandbox=False*)
Get all dependent steps of this Step.

This includes the direct input to the Step as well as indirect inputs such as the used tools or the sandbox.

getArguments()
Get list of all inputs for this Step.

The arguments are passed as absolute paths to the script starting from $1.

getDigestScript()
Return a long term stable script.

The digest script will not be executed but is the basis to calculate if the step has changed. In case of the checkout step the involved SCMs will return a stable representation of _what_ is checked out and not the real script of _how_ this is done.

getEnv()
Return dict of environment variables.

getExecPath(*referrer=None*)
Return the execution path of the step.

The execution path is where the step is actually run. It may be distinct from the workspace path if the build is performed in a sandbox. The `referrer` is an optional parameter that represents a step that refers to this step while building.

getJenkinsScript()
Return the relevant parts as shell script that have no Jenkins plugin.

getLabel()
Return path label for step.

This is currently defined as "src", "build" and "dist" for the respective steps.

getLibraryPaths()
Get sorted list of library paths of used tools.

The returned list is intended to be passed as LD_LIBRARY_PATH environment variable. The paths are first sorted by tool name. The order of paths of a single tool is kept.

getPackage()
Get Package object that is the parent of this Step.

getPaths()
Get sorted list of execution paths to used tools.

The returned list is intended to be passed as PATH environment variable. The paths are sorted by name.

getSandbox(*forceSandbox=False*)
Return Sandbox used in this Step.

Returns a Sandbox object or None if this Step is built without one.

getScript()
Return a single big script of the whole step.

Besides considerations of special backends (such as Jenkins) this script is what should be executed to build this step.

getTools()
Get dictionary of tools.

The dict maps the tool name to a *bob.input.Tool*.

getVariantId()
Return Variant-Id of this Step.

The Variant-Id is used to distinguish different packages or multiple variants of a package. Each Variant-Id need only be built once but successive builds might yield different results (e.g. when building from branches).

getWorkspacePath()
Return the workspace path of the step.

The workspace path represents the location of the step in the users workspace. When building in a sandbox this path is not passed to the script but the one from getExecPath() instead.

isBuildStep()
Return True if this is a build step.

isCheckoutStep()
Return True if this is a checkout step.

isDeterministic()
Return whether the step is deterministic.

Checkout steps that have a script are considered indeterministic unless the recipe declares it otherwise (checkoutDeterministic). Then the SCMs are checked if they all consider themselves deterministic. Build and package steps are always deterministic.

The determinism is defined recursively for all arguments, tools and the sandbox of the step too. That is, the step is only deterministic if all its dependencies and this step itself is deterministic.

isPackageStep()
Return True if this is a package step.

isRelocatable()
Returns True if the step is relocatable.

isShared()
Returns True if the result of the Step should be shared globally.

The exact behaviour of a shared step/package depends on the build backend. In general a shared package means that the result is put into some shared location where it is likely that the same result is needed again.

isValid()
Returns True if this step is valid, False otherwise.

class bob.input.**Sandbox**
Represents a sandbox that is used when executing a step.

getEnvironment()
Get environment variables.

Returns the dictionary of environment variables that are defined by the sandbox.

getMounts()
Get custom mounts.

This returns a list of tuples where each tuple has the format (hostPath, sandboxPath, options).

getPaths()
Return list of global search paths.

This is the base $PATH in the sandbox.

getStep()
Get the package step that yields the content of the sandbox image.

isEnabled()
Return True if the sandbox is used in the current build configuration.

class bob.input.**Tool**
Representation of a tool.

A tool is made of the result of a package, a relative path into this result and some optional relative library paths.

getEnvironment()
Get environment variables.

Returns the dictionary of environment variables that are defined by the tool.

getLibs()
Get list of relative library paths into the result.

Returns List[str]

getNetAccess()
Does tool require network access?

This reflects the *netAccess* tool property.

Returns bool

getPath()
Get relative path into the result.

getStep()
Return package step that produces the result holding the tool binaries/scripts.

Returns *bob.input.Step*

Hooks

Path formatters

There are three plugin hooks that override the path name calculation:

- releaseNameFormatter: local build in release mode
- developNameFormatter: local build in development mode
- jenkinsNameFormatter: Jenkins builds

All hooks must be a function that return the relative path for the workspace. The function gets two parameters: the step for which the path should be returned and a dictionary to the state trackers of the package that is processed. The default implementation in Bob looks like this:

```
def releaseNameFormatter(step, properties):
    return os.path.join("work", step.getPackage().getName().replace('::', os.sep),
                        step.getLabel())

def developNameFormatter(step, properties):
    return os.path.join("dev", step.getLabel(),
                        step.getPackage().getName().replace('::', os.sep))

def jenkinsNameFormatter(step, props):
    return step.getPackage().getName().replace('::', "/") + "/" + step.getLabel()

manifest = {
    'apiVersion' : "0.1",
    'hooks' : {
        'releaseNameFormatter' : releaseFormatter,
        'developNameFormatter' : developFormatter,
        'jenkinsNameFormatter' : jenkinsNameFormatter
    }
}
```

Additionally there is a special hook (`developNamePersister`) that is responsible to create a surjective mapping between steps and workspace paths with the restriction that different variant ids must not be mapped to the same

directory. The hook function is taking the configured develop name formatter (see above) and is expected to return a callable name formatter too. The `developNamePersister` must handle two cases in the following way:

- The passed name formatter returns different paths for steps that have the same variant id. In this case the `developNamePersister` should only return one such path for the same variant id.
- The name formatter returns the same path for different variant ids. In this case the `developNamePersister` must disambiguate the path (e.g. by adding a unique suffix) to return different paths for the different variants of the step(s).

Even though it is not strictly required by Bob it is highly recommended to map all steps with the same variant id to a single directory. The hook is currently only available for the develop mode. The default implementation in Bob is to append an incrementing number starting by one for each variant to the path returned by the configured name formatter:

```
def developNamePersister(nameFormatter):
    dirs = {}

    def fmt(step, props):
        baseDir = nameFormatter(step, props)
        digest = step.getVariantId()
        if digest in dirs:
            res = dirs[digest]
        else:
            num = dirs.setdefault(baseDir, 0) + 1
            res = os.path.join(baseDir, str(num))
            dirs[baseDir] = num
            dirs[digest] = res
        return res

    return fmt

manifest = {
    'apiVersion' : "0.1",
    'hooks' : {
        'developNamePersister' : developNamePersister
    }
}
```

If your name formatter generates unique names for each variant of the steps you may want to override the persister to change this behavior, e.g. to not add a number for the first variant.

String functions

String functions can be invoked from any place where string substitution as described in *String substitution* is allowed. These functions are called with at least one positional parameter for the arguments that were specified when invoking the string function. They are expected to return a string and shall have no side effects. The function has to accept any number of additionaly keyword arguments. Currently the following additionaly kwargs are passed:

- `env`: dict of all available environment variables at the current context
- `recipe`: the current *`bob.input.Recipe`*
- `sandbox`: `True` if a sandbox is used. `False` if no sandbox was configured or if it is disabled (e.g. `--no-sandbox` option was specified).

In the future additional keyword args may be added without notice. Such string functions should therefore have a catch-all `**kwargs` parameter. A sample implementation could look like this:

```
def echo(args, **kwargs):
    return " ".join(args)

manifest = {
    'apiVersion' : "0.2",
    'stringFunctions' : {
        "echo" : echo
    }
}
```

Jenkins job mangling

Jenkins jobs that are created by Bob are very simple and contain only information that was taken from the recipes. It might be necessary to enable additional plugins, add build steps or alter the job configuration in special ways. For such use cases the following hooks are available:

- `jenkinsJobCreate`: initial creation of a job
- `jenkinsJobPreUpdate`: called before updating a job config
- `jenkinsJobPostUpdate`: called after updating a job config

All hooks take a single mandatory positional parameter: the job config XML as string. The hook is expected to return the altered config XML as string too. The function has to accept any number of additionaly keyword arguments. Currently the following additionaly kwargs are passed:

- `alias`: alias name used for jenkins
- `buildSteps`: list of all build steps (*`bob.input.Step`*) used in the job
- `checkoutSteps`: list of all checkout steps (*`bob.input.Step`*) used in the job
- `name`: name of Jenkins job
- `nodes`: The nodes where the job should run
- `packageSteps`: list of all package steps (*`bob.input.Step`*) used in the job
- `prefix`: Prefix of all job names
- `sandbox`: Boolean wether sandbox should be used
- `url`: URL of Jenkins instane
- `windows`: True if Jenkins runs on Windows

See the jenkins-cobertura plugin in the "contrib" directory for an example. The default implementation in Bob looks like this:

```
def jenkinsJobCreate(config, **info):
    return config

def jenkinsJobPreUpdate(config, **info):
    return config

def jenkinsJobPostUpdate(config, **info):
    return config

manifest = {
    'apiVersion' : "0.4",
```

(continues on next page)

(continued from previous page)

```
    'hooks' : {
        'jenkinsJobCreate' : jenkinsJobCreate,
        'jenkinsJobPreUpdate' : jenkinsJobPreUpdate,
        'jenkinsJobPostUpdate' : jenkinsJobPostUpdate
    }
}
```

Generators

The main purpose of a generator is to generate project files for one or more IDEs. There is a build-in generator for QtCreator project files.

A generator is called with at least 3 arguments:

- `package`: the *`bob.input.Package`* to build the project for.
- `argv`: Arguments not consumed by `bob project`.
- `extra`: Extra arguments to be passed back to `bob dev` when called from the IDE. These are the generic arguments that `bob project` parses for all generators.

Starting with Bob 0.17 an additional 4th argument is passed to the generator function:

- `bob`: The fully qualified path name to the Bob executable that runs the generator. This may be used to generate project files that work even if Bob is not in $PATH.

The presence of the 4th parameter is determined by the `apiVersion` of the manifest.

A simple generator may look like:

```
def nullGenerator(package, argv, extra, bob):
    return 0

manifest = {
    'apiVersion' : "0.17",
    'projectGenerators' : {
        'nullGenerator' : nullGenerator,
    }
}
```

Plugin settings

Sometimes plugin behaviour needs to be configurable by the user. On the other hand Bob expects plugins to be deterministic. To have a common interface for such settings it is possible for a plugin to define additional keywords in the *User configuration (default.yaml)*. This provides Bob with the information to validate the settings and detect changes in a reliable manner.

To define such settings the plugin must derive from *`bob.input.PluginSetting`*, create an instance of that class and store it in the manifest under `settings`. A minimal example looks like the following:

```
from bob.input import PluginSetting

class MySettings(PluginSetting):
    @staticmethod
    def validate(data):
        return isinstance(data, str)
```

(continues on next page)

(continued from previous page)

```
mySettings = MySettings("")

manifest = {
    'apiVersion' : "0.14",
    'settings' : {
        'MySettings' : mySettings
    }
}
```

This will define a new, optional "MySettings" keyword for the user configuration that will accept any string. The default, if nothing is configured in `default.yaml`, is an empty string.

Attention: Do not configure your plugins by any other means. Bob will not detect changes and, due to aggressive caching, might not call the plugin again to process the new settings. So reading external files or using environment variables results in undefined behavior.

It is not possible to re-define already existing setting keywords. This applies both to Bob built-in settings as well as settings defined by other plugins. Because Bob is expected to define new settings in the future a plugin defined setting must not start with a lower case letter. These names are reserved for Bob.

1.3.6 Audit trail

For every built artifact Bob records all involved sources and build steps that lead to a particular artifact. The recorded information contains at least, but not limited to, the following records:

- state of the recipes,
- recipe name, package path,
- build host/time,
- environment,
- dependencies (with their respective audit trail),
- state of SCMs (e.g. commit id, dirty, ...).

The information in the audit trail records may be extended with additional information in the future. An application that parses the audit trail should ignore unknown fields for future compatibility.

Storage format

Audit trails are stored as gzip compressed JSON documents. For local builds the audit trail is stored as `audit.json.gz` file next to the workspace. Jenkins builds only rely on binary artifacts where the same file is stored in the compressed tar file in a `meta/` directory. The general structure of an audit trail looks like the following:

```
{
    "artifact" : {
        // audit record
    },
    "references" : [
        {
            // audit record
```

(continues on next page)

(continued from previous page)

```
        },
        ...
    ]
}
```

The audit information about the involved artifact is stored under the `artifact` key. Any audit records about the dependencies that were used to create the artifact are included in the list under the `references` key. This includes all transitive records too. A correct audit trail must include the full transitive information to be accepted by Bob.

Records

The following sections describe the various keys and their semantics that can be found in an audit record.

Example of a single audit record:

```
{
  "artifact-id" : "c1cd9616caa783fcd8ef9b170cd968ccb306a727",
  "variant-id" : "f5aa3695a2d6f0e70af2ffaf43bb461a428e6fba",
  "build-id" : "a95ce8e3e30b7535751cefea942941c31a8ad1aa",
  "result-hash" : "7710368d8165f1c780fb9f33b34415ab76a618c0",
  "env" : "declare -- BASH=\"/bin/bash\"\ndeclare -r BASHOPTS=...",
  "metaEnv" : {
     "VERSION" : "1.2.3",
     "LICENSE" : "GPLv2"
  },
  "scms" : [],
  "dependencies" : {
     "args" : [
        "c5b2a8231156f43728af34f3a2dcb731ade2f76a"
     ]
  },
  "meta" : {
     "language" : "bash",
     "recipe" : "root",
     "step" : "dist",
     "bob" : "0.12.1",
     "package" : "root"
  },
  "build" : {
     "date" : "2019-12-02T13:19:34.193136+00:00",
     "machine" : "x86_64",
     "nodename" : "kloetzke",
     "os-release" : "PRETTY_NAME=\"Debian GNU/Linux 10 (buster)\"\nNAME=...",
     "release" : "4.19.0-6-amd64",
     "sysname" : "Linux",
     "version" : "#1 SMP Debian 4.19.67-2+deb10u2 (2019-11-11)"
  }
}
```

Basic information

`artifact-id` Hexadecimal number that identifies a particular artifact. This is also the primary key for audit records.

`variant-id` The Variant-Id as described in *Implicit versioning*.

build-id The Build-Id as described in *Implicit versioning*.

result-hash A hash sum across the content of the workspace after the artifact was built.

env Dump of the bash environment as created by `declare -p`. See bash declare. For PowerShell recipes it is a JSON string that contain all internal variables and environment variables as dictionaries. Use the `meta.language` key to determine the used scripting language.

metaEnv This is a dictionary of all *metaEnvironment* variables of the package. They are included in the audit trail regardless of their actual usage.

Recipes

If Bob recognizes that the recipes are managed in a supported SCM (currently git or svn) there will be a `recipes` key in the audit record. The format of the object under this key is described in *SCMs*.

Dependencies

Each step can have any number of dependencies. They will be recorded under a `dependencies` key. The other step is referenced by the Artifact-Id and their audit record will be found in the `references` list of the audit trail. There are three types of dependencies to other steps that each have their different representation in audit record:

arguments Ordered list of all dependencies whose result was input to this step. They correspond to the `$1` to `$n` arguments of the script that was executed.

tools Object that maps all available tools by their name to the Artifact-Id.

sandbox Used sandbox during execution.

Example:

```
"dependencies" : {
    "args" : [
        "b0a6632c6e7677220e46e4ae9c528efb949137c6"
    ],
    "tools" : {
        "toolchain" : "0b1c5e3489bed347ccf8e0e1e12dc70c92b09472"
    },
    "sandbox" : "3473b28df3891046618420428b530418ce006ad9"
}
```

SCMs

All SCMs are recorded after the checkout step was run. The audit record will contain a list of objects under the `scms` key. Each object has at least a `type` key that identifies the kind ob SCM and a `dir` key for the relative directory (or file) that was managed by the SCM in the workspace.

See the following list for the additional information that each SCM adds to the record:

git The git SCM records all remotes, the current commit that HEAD points to and if the tree is dirty. The output of `git describe` is also recorded.

Example:

```
{
    "commit": "6e986014563b70ecd867fb6a6e1adeb408f63dd6",
    "description": "v0.11.0-59-g6e98601-dirty",
    "dir": ".",
    "dirty": true
    "remotes": {
        "origin": "git@github.com:BobBuildTool/bob.git"
    },
    "type": "git",
}
```

svn Example:

```
{
    "dir" : ".",
    "dirty" : false,
    "repository" : {
        "root" : "http://svn.haiku-os.org/oldhaiku",
        "uuid" : "a95241bf-73f2-0310-859d-f6bbb57e9c96",
    },
    "revision" : 43238,
    "type" : "svn",
    "url" : "http://svn.haiku-os.org/oldhaiku/haiku/",
}
```

url Example:

```
{
    "digest" : {
        "algorithm" : "sha1",
        "value" : "697b7c87c73eb53bf80e19b65a4ac245214d530c"
    },
    "dir" : "author.txt",
    "type" : "url",
    "url" : "https://example.test/author.txt",
}
```

Meta data

There can be any number of key-value meta data pairs. They will be contained under the `meta` key and typically hold at least the following information:

bob Bob version string.

language The scripting language that was used to create the artifact. Can be `bash` or `PowerShell`. If missing it must be interpreted as `bash`. Use this to correctly parse the `env` string.

package Package path of the artifact that was built. Note that there might be multiple packages that produce the same result. Only one will be built by Bob without recording all possible package paths here.

recipe Name of the recipe that declared the package.

step The executed step for this audit record. Can be `src`, `build` or `dist`.

Example:

```
"meta" : {
    "bob" : "0.11.0-56-g9b3d2c6-dirty",
    "package" : "root/lib"
    "recipe" : "lib",
    "step" : "src",
},
```

Build data

The build data describes when and where the artifact has been built. It can be found under the `build` key and contains the following fields:

`date` The date and time of the build. This is stored as UTC time and formatted in ISO 8601 format with full precision.

`machine` The hardware identifier as returned by the uname system call. This is typically the processor architecture of the host.

`nodename` The host name.

`os-release` This optional field holds the content of `/etc/os-release`, if existing. If the file does not exist or cannot be read then this field will not be present.

`release` The operating system release.

`sysname` The operating system name (e.g. "Linux").

`version` The operating system version.

> **Attention:** The information of the `machine`, `release`, `sysname`, `version` and possibly `nodename` fields show the host in case of container builds, e.g. when running in a docker container. Be careful when relying on this information. The `os-release` field, if present, is more reliable in this case.

1.4 Man Pages

Contents:

1.4.1 bob-archive

Name

bob-archive - Manage binary artifacts archive

Synopsis

Generic command format:

```
bob archive [-h] subcommand ...
```

Available sub-commands:

```
bob archive clean [-h] [--dry-run] [-n] [-v] expression
bob archive scan [-h] [-v]
```

Description

The bob archive command can be used to manage local binary artifact archives. The command must be executed in the root of the archive and needs write access to create an index cache.

Artifacts are managed by the information included in their *Audit Trail*. See the Audit Trail documentation about the general included data. Currently the `bob archive` command has access to the `meta`, `build` and `metaEnv` sections of the audit trail.

Options

`--dry-run` Do not actually delete any artifacts but show what would get removed.

`-n` Don't rescan the archive for new artifacts. The command will work on the last scanned data. Useful if the scan takes a long time (e.g. big archive on network mount) and was already run recently.

`-v` Be a bit more chatty on what is done.

Commands

clean Remove unneeded artifacts from the archive.

The command takes a single argument as the retention expression. Any artifact that is matched by the expression or referenced by such other artifact is kept. If an artifact is neither matched by the given expression nor referenced by a retained artifact it is deleted.

The expression language supports the following constructs:

- Strings are written with double quotes, e.g. `"foo"`. To embed double quotes in the string itself escape them with `\`.
- Certain fields from the audit trail can be accessed by their name. Sub-fields are specified with a dot operator, e.g. `meta.package`. All fields are case sensitive and of string type.
- Strings and fields can be compared by the following operators (in decreasing precedence): <, <=, >, >=, ==, !=. They have the same semantics as in Python.
- String comparisons can be logically combined with `&&` (and) respectively `||` (or). There is also a `!` (not) logical operator.
- Parenthesis can be used to override precedence.

A typical usage of the `clean` command is to remove old artifacts from a continuous build artifact archive. Suppose the root package that is built is called `platform/app` and we want to retain only artifacts that are referenced by builds that are at most seven days old:

```
bob archive clean "meta.package == \"platform/app\" && \
                   build.date >= \"$(date -u -Idate -d-7days)\""
```

scan Scan for added artifacts.

The `archive` command keeps a cache of all indexed artifacts. To freshen this cache use this command. Even though other sub-commands will do a scan too (unless suppressed by `-n`) it might be helpful to do the scan on

a more convenient time. If the archive is located e.g. on a slow network drive it could be advantageous to scan the archive with a cron job over night.

Notes

`bob archive` only works for local binary artifact archives. If you're using a remote archive, you need shell access and a working Bob installation on the machine providing your archive in order to be able to use `bob archive`.

1.4.2 bob-build

Name

bob-build - Bob release mode build

Synopsis

```
bob build [-h] [--destination DEST] [-j [JOBS]] [-k] [-f] [-n] [-p]
          [--without-provided] [-A | --audit] [-b | -B | --normal]
          [--clean | --incremental] [--always-checkout RE] [--resume]
          [-q] [-v] [--no-logfiles] [-D DEFINES] [-c CONFIGFILE]
          [-e NAME] [-E] [--upload] [--link-deps] [--no-link-deps]
          [--download MODE] [--sandbox | --no-sandbox]
          [--clean-checkout]
          PACKAGE [PACKAGE ...]
```

Description

The *bob build* command is building packages locally in release mode. This mode is intended to provide maximum correctness at the expense of build time and disk requirements.

Default options

By default *bob build* works in the `work` subdirectory of the project root directory. The source-, build- and package-directories of packages are kept next to each other (`work/<pkg>/src`, `work/<pkg>/build` and `work/<pkg>/dist`). The `<pkg>`-subdirectories are derived from the package name. As recipes are changed Bob will always use a new, dedicated directory for each variant by adding a counting suffix to the above directories.

In contrast to *bob dev* the following options take precedence. They can be overridden individually by their inverse switches:

- `--download=yes`
- `--clean`
- `--sandbox`

Options

`--always-checkout RE` Always checkout packages that match the regular expression pattern `RE`. The option may be given more than once. In this case all patterns will be checked.

Bob may skip the checkout of packages where a correct binary artifact can be downloaded from an archive. While this can dramatically decrease the build time of large projects it can hamper actually changing and rebuilding the packages with modifications. Use this option to instruct Bob to always checkout the sources of the packages that you may want to modify.

This option will just make sure that the sources of matching packages are checked out. Bob will still try to find matching binary artifacts to skip the actual compilation of these packages. See the `--download` option to control what is built and what is downloaded.

`--audit` Generate an audit trail when building.

This is the default unless the user changed it in `default.yaml`.

`--clean` Do clean builds by clearing the build directory before executing the build commands. It will *not* clean all build results (e.g. like `make clean`) but rather make sure that no old build artifacts are in the workspace when a package is rebuilt. To actually force a rebuild (even though nothing has changed) use `-f`.

This is the default for release mode builds. See `--incremental` for the inverse option.

`--clean-checkout` Do a clean checkout if SCM state is unclean.

Bob will check all SCMs for local changes at the start of a checkout. If a SCM checkout is tainted (e.g. dirty, switched branch, unpushed commits, ...) Bob will move it into the attic and do a fresh checkout.

Use this option if you are not sure about the state of the source code. You can also use '*bob status*' to check the state without changing it.

`--destination DEST` Destination of build result (will be overwritten!)

All build results are copied recursively into the given folder. Colliding files will be overwritten but other existing files or directories are kept. Unless `--without-provided` is given using this option will implicitly enable `--with-provided` to build and copy all provided packages of the built package(s).

`--download MODE` Download from binary archive (yes, no, deps, forced, forced-deps, packages)

no build given module and it's dependencies from sources

yes download given module, if download fails - build it from sources (default for release mode)

forced like 'yes' above, but fail if any download fails

deps download dependencies of given module and build the module afterwards. If downloading of any dependency fails - build it from sources (default for develop mode)

forced-deps like 'deps' above, but fail if any download fails

forced-fallback combination of forced and forced-deps modes: if forced fails fall back to forced-deps

packages=<packages regex> download modules that match a given regular expression, build all other.

`--incremental` Reuse build directory for incremental builds.

This is the inverse option to `--clean`. Build workspaces will be reused as long as their recipes were not changed. If the recipe did change Bob will still do a clean build automatically.

`--link-deps` Create symlinks to dependencies next to workspace.

`--no-sandbox` Disable sandboxing

`--resume` Resume build where it was previously interrupted.

All packages that were built in the previous invocation of Bob are not checked again. In particular changes to the source code of these packages are not considered. Use this option to quickly resume the build if it failed and the error has been corrected in the failing package.

`--sandbox` Enable sandboxing

--upload Upload to binary archive

-A, --no-audit Do not generate an audit trail.

The generation of the audit trail is usually barely noticeable. But if a large number of repositories is checked out it can add a significant overhead nonetheless. This option suppresses the generation of the audit trail.

Note that it is not possible to upload such built artifacts to a binary archive because vital information is missing.

-B, --checkout-only Don't build, just check out sources

-D DEFINES Override default environment variable

-E Preserve whole environment.

Normally only variables configured in the whitelist are passed unchanged from the environment. With this option all environment variables that are set while invoking Bob are kept. Use with care as this might affect some packages whose recipes are not robust.

-b, --build-only Don't checkout, just build and package

If the sources of a package that needs to be built are missing then Bob will still check them out. This option just prevents updates of existing source workspaces.

-c CONFIGFILE Use additional configuration file.

The `.yaml` suffix is appended automatically and the configuration file is searched relative to the project root directory unless an absolute path is given. Bob will parse these user configuration files after *default.yaml*. They are using the same schema.

This option can be given multiple times. The files will be parsed in the order as they appeared on the command line.

-e NAME Preserve environment variable.

Unless `-E` this allows the fine grained addition of single environment variables to the whitelist.

-f, --force Force execution of all build steps.

Usually Bob decides if a build step or any of its input has changed and will skip the execution of it if this is not the case. With this option Bob not use that optimization and will execute all build steps.

-j, --jobs Specifies the number of jobs to run simultaneously.

Any checkout/build/package step that needs to be executed are counted as a job. Downloads and uploads of binary artifacts are separate jobs too. If a job fails the other currently running jobs are still finished before Bob returns. No new jobs are scheduled, though, unless the `-k` option is given (see below).

If the -j option is given without an argument, Bob will run as many jobs as there are processors on the machine.

-k, --keep-going Continue as much as possible after an error.

While the package that failed to build and all the packages that depend on it cannot be built either, the other dependencies are still processed. Normally Bob stops on the first error that is encountered.

-n, --no-deps Don't build dependencies.

Only builds the package that was given on the command line. Bob will not check if the dependencies of that package are available and if they are up-to-date.

--no-link-deps Do not create symlinks to dependencies next to workspace.

--no-logfiles Don't write a logfile. Without this bob is creating a logfile in the current workspace. Because of the pipe-usage many tools like gcc, ls, git detect they are not running on a tty and disable output coloring. Disable the logfile generation to get the colored output back.

-p, --with-provided Build provided dependencies too. In combination with `--destination` this is the default. In any other case `--without-provided` is default.

-q, --quiet Decrease verbosity (may be specified multiple times)

-v, --verbose Increase verbosity (may be specified multiple times)

--without-provided Build just the named packages without their provided dependencies. This is the default unless the `--destination` option is given too.

See also

bobpaths(7) bob-status(1)

1.4.3 bob-clean

Name

bob-clean - Delete unused workspace and attic directories

Synopsis

```
bob clean [-h] [--develop | --release | --attic] [-c CONFIGFILE]
          [-D DEFINES] [--dry-run] [-f] [-s] [--sandbox | --no-sandbox]
          [-v]
```

Description

The *bob clean* command removes workspace directories from previous *bob-dev* and *bob-build* invocations that are not referenced anymore by the current recipes. It can also remove attic directories that are wasting precious disk space.

The command has three modes of operation. By default develop mode related workspaces are garbage collected. In release mode (`--release`) workspaces that were created by *bob build* are cleaned. Lastly the `--attic` option removes attic directories that were created by *bob dev*.

The identification of the unreferenced workspace directories is based on the current recipes, user configuration files and environment definitions. You should therefore pass the same options to *bob clean* (`-c`, `-D`) that you would also pass to *bob build* resp. *bob dev*. If in doubt use `--dry-run` to see what would be deleted.

Workspaces that hold source code are never deleted by default. Add the `-s` option to consider these workspace directories too. Bob will still check each SCM in an unreferenced workspace for mdoifications. If the SCM checkout has been modified in any way (e.g. changed or untracked files, unpushed commits) then the workspace is kept. Use `-f` to also delete such workspaces too.

Options

--develop Clean develop mode (*bob dev*) directories. This is the default.

--release Clean release mode (*bob build*) directores.

--attic Remove attic directories.

-c CONFIGFILE Use additional user configuration file. May be given more than once.

The configuration files have the same syntax as `default.yaml`. Their settings have higher precedence than `default.yaml`, with the last given configuration file being the highest.

-D VAR[=VALUE] Override default environment variable. May be given more than once. If the optional `VALUE` is not supplied the variable will be defined to an empty string.

--dry-run Don't delete, just print what would be deleted.

-f, --force Remove source workspaces that have unsaved changes in their SCM(s).

> **Warning:** Using this option *will* result in data loss if there are unsaved changes in checkout workspace directories. Use with great care.

-s, --src Clean source workspaces too. By default only build and package workspaces are considered.

> **Attention:** You should double check with `--dry-run` that no unintended workspaces are actually deleted. While Bob can check SCMs that it knows it cannot detect all modifications, e.g. changes to extracted tar files.

-v, --verbose Print what is done.

1.4.4 bob-dev

Name

bob-dev - Bob develop mode build

Synopsis

```
bob dev [-h] [--destination DEST] [-j [JOBS]] [-k] [-f] [-n] [-p]
        [--without-provided] [-A | --audit] [-b | -B | --normal]
        [--clean | --incremental] [--always-checkout RE] [--resume]
        [-q] [-v] [--no-logfiles] [-D DEFINES] [-c CONFIGFILE]
        [-e NAME] [-E] [--upload] [--link-deps] [--no-link-deps]
        [--download MODE] [--sandbox | --no-sandbox] [--clean-checkout]
        PACKAGE [PACKAGE ...]
```

Description

The *bob dev* command is building packages locally in develop mode. This mode is intended to be used by developers to incrementally build the packages. Its defaults are tuned to support active development on source code by keeping a stable directory structure and minimize the edit-compile turnaround time.

Default options

By default *bob dev* works in the `dev` subdirectory of the project root directory. The source-, build- and package-directories are kept in separate hierarchies (`dev/src`, `dev/build` and `dev/dist`) to allow easy indexing of the

involved sources by IDEs. It is possible to change these paths by means of a plugin but it is advised to keep the top level structure.

In contrast to *bob build* the following options take precedence. They can be overridden individually by their inverse switches:

- `--download=deps`
- `--incremental`
- `--no-sandbox`

Source code checkout

The source workspaces are updated incrementally as good as possible even across recipe changes. This works quite well e.g. for git repositories. It could fail silently on certain some recipes, though. On URL-SCMs the downloaded file can be tracked by Bob. But if an archive is extracted Bob cannot reliably know which files were coming from the archive. If the archive changes and files vanish they will still be kept in the workspace.

If a binary archive is used Bob will try to skip the checkout of sources if possible. This will work only if matching binary artifacts are available for the current state of the recipes and their configuration. Though this will typically speed up the build it can actually make working on the source code difficult because not all involved sources are checked out.

There are a number of options to force the checkout of sources is such an environment:

- Use the `--always-checkout` option. If you're typically working on some particular packages then this option can force the checkout of these sources. It can be set in *default.yaml* so that it does not need to be specified every time again.
- Make a dedicated build of selected packages. Because of `--download=deps` the specified package will always be built from source.
- Use `--checkout-only` to fetch the sources of a package and all its dependencies.

In any case Bob will use the sources once they were checked out. Bob will also update them in subsequent builds.

Options

`--always-checkout RE` Always checkout packages that match the regular expression pattern `RE`. The option may be given more than once. In this case all patterns will be checked.

Bob may skip the checkout of packages where a correct binary artifact can be downloaded from an archive. While this can dramatically decrease the build time of large projects it can hamper actually changing and rebuilding the packages with modifications. Use this option to instruct Bob to always checkout the sources of the packages that you may want to modify.

This option will just make sure that the sources of matching packages are checked out. Bob will still try to find matching binary artifacts to skip the actual compilation of these packages. See the `--download` option to control what is built and what is downloaded.

`--audit` Generate an audit trail when building.

This is the default unless the user changed it in `default.yaml`.

`--clean` Do clean builds by clearing the build directory before executing the build commands. It will *not* clean all build results (e.g. like `make clean`) but rather make sure that no old build artifacts are in the workspace when a package is rebuilt. To actually force a rebuild (even though nothing has changed) use `-f`.

This is the default for release mode builds. See `--incremental` for the inverse option.

--clean-checkout Do a clean checkout if SCM state is unclean.

Bob will check all SCMs for local changes at the start of a checkout. If a SCM checkout is tainted (e.g. dirty, switched branch, unpushed commits, ...) Bob will move it into the attic and do a fresh checkout.

Use this option if you are not sure about the state of the source code. You can also use '*bob status*' to check the state without changing it.

--destination DEST Destination of build result (will be overwritten!)

All build results are copied recursively into the given folder. Colliding files will be overwritten but other existing files or directories are kept. Unless `--without-provided` is given using this option will implicitly enable `--with-provided` to build and copy all provided packages of the built package(s).

--download MODE Download from binary archive (yes, no, deps, forced, forced-deps, packages)

no build given module and it's dependencies from sources

yes download given module, if download fails - build it from sources (default for release mode)

forced like 'yes' above, but fail if any download fails

deps download dependencies of given module and build the module afterwards. If downloading of any dependency fails - build it from sources (default for develop mode)

forced-deps like 'deps' above, but fail if any download fails

forced-fallback combination of forced and forced-deps modes: if forced fails fall back to forced-deps

packages=<packages regex> download modules that match a given regular expression, build all other.

--incremental Reuse build directory for incremental builds.

This is the inverse option to `--clean`. Build workspaces will be reused as long as their recipes were not changed. If the recipe did change Bob will still do a clean build automatically.

--link-deps Create symlinks to dependencies next to workspace.

--no-sandbox Disable sandboxing

--resume Resume build where it was previously interrupted.

All packages that were built in the previous invocation of Bob are not checked again. In particular changes to the source code of these packages are not considered. Use this option to quickly resume the build if it failed and the error has been corrected in the failing package.

--sandbox Enable sandboxing

--upload Upload to binary archive

-A, --no-audit Do not generate an audit trail.

The generation of the audit trail is usually barely noticeable. But if a large number of repositories is checked out it can add a significant overhead nonetheless. This option suppresses the generation of the audit trail.

Note that it is not possible to upload such built artifacts to a binary archive because vital information is missing.

-B, --checkout-only Don't build, just check out sources

-D DEFINES Override default environment variable

-E Preserve whole environment.

Normally only variables configured in the whitelist are passed unchanged from the environment. With this option all environment variables that are set while invoking Bob are kept. Use with care as this might affect some packages whose recipes are not robust.

-b, --build-only Don't checkout, just build and package

If the sources of a package that needs to be built are missing then Bob will still check them out. This option just prevents updates of existing source workspaces.

-c CONFIGFILE Use additional configuration file.

The `.yaml` suffix is appended automatically and the configuration file is searched relative to the project root directory unless an absolute path is given. Bob will parse these user configuration files after *default.yaml*. They are using the same schema.

This option can be given multiple times. The files will be parsed in the order as they appeared on the command line.

-e NAME Preserve environment variable.

Unless `-E` this allows the fine grained addition of single environment variables to the whitelist.

-f, --force Force execution of all build steps.

Usually Bob decides if a build step or any of its input has changed and will skip the execution of it if this is not the case. With this option Bob not use that optimization and will execute all build steps.

-j, --jobs Specifies the number of jobs to run simultaneously.

Any checkout/build/package step that needs to be executed are counted as a job. Downloads and uploads of binary artifacts are separate jobs too. If a job fails the other currently running jobs are still finished before Bob returns. No new jobs are scheduled, though, unless the `-k` option is given (see below).

If the -j option is given without an argument, Bob will run as many jobs as there are processors on the machine.

-k, --keep-going Continue as much as possible after an error.

While the package that failed to build and all the packages that depend on it cannot be built either, the other dependencies are still processed. Normally Bob stops on the first error that is encountered.

-n, --no-deps Don't build dependencies.

Only builds the package that was given on the command line. Bob will not check if the dependencies of that package are available and if they are up-to-date.

--no-link-deps Do not create symlinks to dependencies next to workspace.

--no-logfiles Don't write a logfile. Without this bob is creating a logfile in the current workspace. Because of the pipe-usage many tools like gcc, ls, git detect they are not running on a tty and disable output coloring. Disable the logfile generation to get the colored output back.

-p, --with-provided Build provided dependencies too. In combination with `--destination` this is the default. In any other case `--without-provided` is default.

-q, --quiet Decrease verbosity (may be specified multiple times)

-v, --verbose Increase verbosity (may be specified multiple times)

--without-provided Build just the named packages without their provided dependencies. This is the default unless the `--destination` option is given too.

See also

bobpaths(7) bob-status(1)

1.4.5 bob-graph

Name

bob-graph - Generate dependency graphs

Synopsis

```
bob graph [-h] [-D DEFINES] [-c CONFIGFILE] [--sandbox | --no-sandbox]
          [--destination DEST] [-e EXCLUDES] [-f FILENAME]
          [-H HIGHLIGHTS] [-n MAX_DEPTH] [-t {d3,dot}] [-o OPTIONS]
          PACKAGE [PACKAGE ...]
```

Description

Generate a dependency graph showing the dependencies of the given `package`. If no other options are given a interactive dependency graph is generated in the `graph` subdirectory of the current working directory.

Two graph types are supported: `d3` and `dot`. The dot graph is helpfull for small projects or a very limited number of dependencies, while the D3 graph is a html page using a javascript library (www.d3js.org) to make a svg. This is interactive meaning it can be dragged and zoomed. Nodes are clickable to highlight dependencies.

Options

`-c CONFIGFILE` Use config File

`-D DEFINES` Override default environment variable

`--destination` Destination of graph output files.

`-e, --excludes` Do not show packages matching this regex. (And all it's dependencies)

`-f FILENAME, --filename FILENAME` Name of Outputfile.

`-H HIGHLIGHTS, --highlight HIGHLIGHTS` Highlight packages matching this regex.

`-n MAX_DEPTH, --max-depth MAX_DEPTH` Max depth. Show only the first `n` dependencies of `package`.

`--no-sandbox` Disable sandboxing. The graph will not have sandbox dependencies. This is the default.

`--sandbox` Enable sandboxing. Include sandbox dependencies in the graph.

`-t, --type` Set the graph type. `d3` (default) or `dot`.

`-o OPTIONS` Set extended options. (See *Extended Options* for the list of available options.

Extended Options

The following options are available. Any unrecognized options are ignored.

D3-Graph

d3.showScm Type: boolean Add the package scm's to the mouse-over box.

d3.localLib Use a local version of d3.v4.min.js. This is copied to the graph folder making it possible to use the graph offline.

d3.dragNodes Type: boolean Enable node drag functionality.

1.4.6 bob-jenkins

Name

bob-jenkins - Configure Jenkins server

Synopsis

Generic command format:

```
bob jenkins [-h] [-c NAME] subcommand ...
```

Available sub-commands:

```
bob jenkins add [-h] [-n NODES] [-o OPTIONS] [-w] [-p PREFIX] [-r ROOT]
                [-D DEFINES] [--keep] [--download] [--upload]
                [--no-sandbox] [--credentials CREDENTIALS] [--clean]
                [--shortdescription] [--longdescription]
                name url
bob jenkins export [-h] name dir
bob jenkins graph [-h] name
bob jenkins ls [-h] [-v]
bob jenkins prune [-h] [--obsolete | --intermediate] [--no-ssl-verify]
                  [-q] [-v]
                  name
bob jenkins push [-h] [-f] [--no-ssl-verify] [--no-trigger] [-q] [-v]
                 name
bob jenkins rm [-h] [-f] name
bob jenkins set-options [-h] [--reset] [-n NODES] [-o OPTIONS] [-p PREFIX]
                        [--add-root ADD_ROOT] [--del-root DEL_ROOT]
                        [-D DEFINES] [-U UNDEFINES] [--credentials CREDENTIALS]
                        [--keep | --no-keep] [--download | --no-download]
                        [--upload | --no-upload] [--sandbox | --no-sandbox]
                        [--clean | --incremental] [--autotoken AUTHTOKEN]
                        [--shortdescription]
                        name
bob jenkins set-url [-h] name url
```

Description

Options

--add-root ADD_ROOT Add new root package

--authtoken AUTHTOKEN Add a authentication token to trigger job remotely

--clean Do clean builds (clear workspace)

--credentials CREDENTIALS Credentials UUID for SCM checkouts

-D DEFINES Override default environment variable

--del-root DEL_ROOT Remove existing root package

--download Download from binary archive

-f, --force Overwrite existing jobs

--incremental Reuse workspace for incremental builds

--intermediate Delete everything except root jobs

--keep Keep obsolete jobs by disabling them

--longdescription Every path to a package will be calculated and displayed in job description

-n NODES, --nodes NODES Label for Jenkins Slave

--no-download Disable binary archive download

--no-keep Delete obsolete jobs

--no-sandbox Disable sandboxing

--no-ssl-verify Disable HTTPS certificate checking

By default only secure connections are allowed to HTTPS Jenkins servers. If this option is given then any certificate error is ignored. This was the default before Bob 0.15.

--no-trigger Do not trigger build for updated jobs

--no-upload Disable binary archive upload

-o OPTIONS Set extended Jenkins options. This option expects a `key=value` pair to set one particular extended configuration parameter. May be specified multiple times. See *Extended Options* for the list of available options. Setting an empty value deletes the option.

--obsolete Delete obsolete jobs that are currently not needed according to the recipes.

-p PREFIX, --prefix PREFIX Prefix for jobs

-q, --quiet Decrease verbosity (may be specified multiple times)

-r ROOT, --root ROOT Root package (may be specified multiple times)

--reset Reset all options to their default

--sandbox Enable sandboxing

--shortdescription Do not calculate every path for every variant. Leads to short job description: One path for each variant.

-U UNDEFINES Undefine environment variable override

--upload Upload to binary archive

-v, --verbose Show additional information

-w, --windows Jenkins is running on Windows. Produce cygwin compatible scripts.

Commands

prune Prune jobs from Jenkins server.

By default all jobs managed by the Jenkins alias will be deleted. If the 'keep' option is enabled for this alias you may use the '–obsolete' option to delete only currently disabled (obsolete) jobs. Alternatively you may delete all intermediate jobs and keep only the root jobs by using '–intermediate'. This will disable the root jobs because they cannot run anyawy without failing.

Extended Options

The following Jenkins plugin options are available. Any unrecognized options are ignored.

artifacts.copy This options selects the way of sharing archives between workspaces. Possible values are:

- **jenkins** Use copy artifacts pluing to copy result and buildId to jenkins-master. The downstream job will afterwards be configured to use copy artifact plugin again and copy the artifact into it's workspace. This is the default.
- **archive** Only copy the buildID file to to jenkins master and use the binary archive for sharing artifacts. Must be used together with `--upload` and `--download`.

jobs.isolate Regular expression that is matching package names. Any package that is matched is put into a separate job. Multiple variants of the same package are still kept in the same job, though.

This option might be used to single out specific packages into dedicated Jenkins jobs that are unrelated to other jobs in the recipe. Typical use cases are documentation and testing `multiPackage` that should not prevent other packages from building if they fail. The obvious draw back is that common checkout and build steps might be duplicated to multiple jobs, though.

jobs.policy Controls how downstream jobs are triggered and which artifacts of the upstream jobs are used. By default only stable jobs trigger further downstream builds. The following settings are available:

- **stable** Downstream jobs are triggered only if the build was stable. Likewise, only the artifacts of stable upstream builds are used. This is the default.
- **unstable** Downstream jobs are triggered on successful builds, that is stable and unstable builds. The downstream jobs will also use the last build that succeeded, even if that build was unstable.
- **always** Downstream jobs are triggered regardless of the build result, even on failed builds. The artifacts are taken from the last completed build of the upstream job which might not necessarily have published one because it failed before archiving them.

jobs.update Whenever the recipes are changed Bob has to update the individual Jenkins jobs that are affected by the change. This switch controls how the description and audit trail information is updated if only these are affected by the change. Their update may be deferred unless strictly necessary and still generate a correct build result at the expense of the freshness of this information.

- **always** Always update the description and audit trail information if they change. This is the default. Note that `bob jenkins push` will always update the description because the date and time of the update is part of the job description.
- **description** Keep the description up-to-date but defer audit trail updates unless strictly necessary. This may provide marginal speed gains but will still update all jobs because the description contains the recipe version and update time.
- **lazy** Only update a job if it will build a different artifact than before. The description and audit trail information will be left unchanged otherwise. This will provide considerable speed improvements at the expense of an outdated description of the unchanged jobs.

scm.git.shallow Instruct the Jenkins git plugin to create shallow clones with a history truncated to the specified number of commits. If the parameter is unset or "0" the full history will be cloned.

> **Warning:** Setting this parameter too small may prevent the creation of a proper change log. Jenkins will not be able to find the reference commit of the last run if the branch advanced by more commits than were cloned.

scm.git.timeout Instruct the Jenkins git plugin to use the given timeout (minutes) for clone and fetch operations.

scm.ignore-hooks Boolean option (possible values: '0' or 'false' resp. '1' or 'true') to set the "Ignore post-commit hooks" option on all jobs. This instructs Jenkins to ignore changes notified by SCM post-commit hooks if enabled. You should probably set a sensible polling interval with the `scm.poll` option unless you want to trigger the generated jobs manually.

scm.poll Without this option the Jenkins server is dependent on external commit hooks to be notified of changes in the source code repositories. While this is the preferred solution it might be necessary to fall back to polling in some setups. Set this option to a Jenkins flavoured cron line, e.g. `H/15 * * * *`.

shared.dir Any packages that are marked as *shared* (`shared: True`) are installed upon usage on a Jenkins slave in a shared location. By default this is `${JENKINS_HOME}/bob`. To use another directory set this option to an absolute path. If you expand Jenkins environment variables make sure that they follow the syntax of the default value because the path is also expanded by the Token Macro plugin.

1.4.7 bob-ls

Name

bob-ls - List package hierarchy

Synopsis

```
bob ls [-h] [-a] [-o] [-r] [-u] [-p | -d] [-D DEFINES] [-c CONFIGFILE]
       [--sandbox | --no-sandbox]
       [package]
```

Description

List package dependencies. The optional `package` argument specifies what package(s) should be listed. If no package is specified the virtual root package is used, thus printing all top level packages and aliases. The `/` package path selects the virtual root package too but does not list aliases as it is a absolute location path. See *bobpaths(7)* for how to specify packages and how aliases are handled.

By default only the direct dependencies of the package are displayed. By adding `-a` the indirect dependencies (i.e. dependencies collected from *provideDeps*) are displayed too. To see the relative path from where the indirect dependencies were inherited add `-o`.

Without any further options only the first level of dependencies is listed. Adding `-r` shows a graphical tree of all transitive dependencies too. To get a list of all transitive dependencies instead, specify `-p`. This will print each package on a separate line with the full package path. The aliases listed below the virtual root package are not recursively traversed as they can involve arbitrarily complex queries. If you want to recursively list the dependencies of an alias you have to specify it explicitly as `package` argument.

Listing the dependencies of the selected package(s) is not always desired. To see the selected packages of a complex query directly add `-d`. This will print the path of all *unique* packages that were selected by the query. This cannot be used in conjunction with the `-p` option and ignores further `-a`, `-o` and `-r` options.

Options

`-a, --all` Show indirect dependencies too. By default only direct dependencies (i.e. dependencies explicitly specified in the recipe) are displayed.

`-c CONFIGFILE` Use config File

-d, --direct List packages themselves, not their contents. This comes in handy if the actual result of a query shall be displayed instead of the dependencies of the selected package(s). Cannot be used at the same time as `-p`. The `-a`, `-o` and `-r` options will have no effect if `-d` is specified.

-D DEFINES Override default environment variable

--no-sandbox Disable sandboxing

-o, --origin Show origin of indirect dependencies. This is printed as relative path to the current package.

-p, --prefixed Prints the full path prefix for each package. Without this option a graphical tree of the dependencies is displayed.

-r, --recursive Recursively display dependencies

--sandbox Enable sandboxing

-u, --unsorted Show the packages in the order they were named in the recipe. By default they are sorted by name for better readability.

See also

bobpaths(7)

1.4.8 bobpaths

Name

bobpaths - Specifying paths to Bob packages

Description

Most Bob commands are working on sets of packages. They can be specified by a query language that loosely resembles Unix paths for the common case and XPath for more advanced features. In contrast to these, Bob path queries are working on general directed acyclic graphs instead of trees. Additionally *alias substitution* is supported to abbreviate often used paths.

Examples:

- `/foo/bar` selects the `bar` package under the `foo` top level package
- `//*-unittests` selects all packages that end with `-unittests`
- `/image//*[ "${LICENSE}" == "GPL" ]` selects all GPL licensed packages that are descendants of the `image` top level package

When Bob parses the recipes he builds an internal package graph. The general dependency structure is derived from the recipes. Depending on the actual content one or more packages are generated from a recipe. The Bob queries are working on the package graph.

The primary constructs of Bob paths are the location path and predicate expressions. Both are evaluated with respect to a context which consists of:

- a package,
- a set of environment variables from the context package,
- and a string function library.

The environment variables in the context are derived from the context package. Only variables that are explicitly consumed by the recipe (via *{checkout,build,package}Vars*) and *metaEnvironment* variables are available. References to unset variables will result in an empty string.

The string function library is populated by all built-in string functions and additional ones defined by *plugins*. The string functions, if evaluated during the query, will get an additional `package` parameter which holds the current context package. The string function library stays constant throughout the whole evaluation.

A path is parsed by first dividing the character string into tokens and then parsing the resulting sequence of tokens. Whitespaces are ignored between tokens and may be freely injected. Some tokes (e.g. `*`, `[` or `]`) collide with special characters of the shell. Care should be taken to correctly quote or escape these characters when invoking Bob from the command line.

Location path

Just like in Unix, location paths can be expressed using a straightforward syntax. Similar to XPath this is actually a syntactic abbreviation of the more verbose syntax which will be explained later. A location path selects a set of packages relative to the context package. The result of evaluating a location step is a set of contexts with packages that matched the axis, name test and predicate of the location step. Following steps in the location path are then recursively applied to the generated contexts.

Before diving into a formal definition here are some simple location path examples:

- `foo` selects the `foo` child package of the context package
- `f*` selects all children of the context package starting with `f`
- `/` selects the virtual root package that is parent to all top level packages (i.e. packages of recipes where `root` is true)
- `/foo` selects the `foo` top level package
- `/foo/bar` selects the `bar` child of the `foo` top level package

All examples above are abbreviations of the verbose syntax. See the following examples for the full syntax:

- `child@foo` selects the `foo` child package of the context package
- `chils@f*` selects all children of the context package starting with `f`
- `/child@foo/child@bar` selects the `bar` child of the `foo` top level package
- `descendant@foo` selects the `foo` descendants of the context package
- `descendant-or-self@foo` selects the `foo` descendants of the context package and, if the context package is named `foo`, the context package as well
- `self@foo` selects the context package if it is named `foo`, and otherwise selects nothing
- `child@foo/descendant@bar` selects the `bar` descendants of the `foo` child of the context package
- `child@*/child@foo` selects all `foo` grandchildren of the context package
- `child@*["${LICENSE}" == "GPLv2"]` selects all children of the context package that are licensed as GPLv2
- `child@lib*[child@libc]` selects the children starting with `lib` of the context package that have a `libc` child (i.e. that have a dependency to `libc`)
- `descendant-or-self@lib*["${LICENSE}" == "GPLv2" && child@libc]` selects the context package or any of it descendants that start with `lib` which are licensed as GPLv2 and have a direct dependency to `libc`

There are two kinds of location path: relative location paths and absolute location paths.

A relative location path consists of a sequence of one or more location steps separated by /. The steps in a relative location path are composed together from left to right. Each step in turn selects a set of packages relative to a context package. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of packages relative to a context package. Each package in that set is used as a context package for the following step. The sets of packages identified by that step are unioned together. The set of packages identified by the composition of the steps is this union.

An absolute location path consists of / optionally followed by a relative location path. A / by itself selects the virtual root package as context package. If it is followed by a relative location path, then the location path resolution starts with the virtual root package as input for the initial step.

Location steps A location step has three parts:

- an axis, which specifies the graph relationship between the context package and the packages selected by the location step
- a package name test, which filters the packages selected by the axis by their name
- an optional predicate, which uses an arbitrary expression to further refine the set of packages that passed the package name test

The syntax for a location step is `axis@name[predicate]`.

Axis specifier The following axis are available:

- the `self` axis contains just the context package itself,
- the `child` axis contains all children of the context package,
- the `direct-child` axis contains the direct children of the context package (i.e. without provided dependencies),
- the `descendant` axis contains all descendants of the context package; a descendant is a child or a child of a child and so on,
- the `direct-descendant` axis contains the direct descendants of the context package; a direct descendant is a direct child or a direct child of a direct child and so on,
- the `descendant-or-self` axis contains the context package and the descendants of the context package
- the `direct-descendant-or-self` axis contains the context package and the direct descendants of the context package.

Package name test For every package that is reachable by the axis the package name is matched with the package name test. Names must match exactly as given in the test. The special `*` wildcard character matches zero or more characters.

Predicates The predicate expression further filters the package set that was generated by the axis and passed the package name test. For each package in the package-set to be filtered, the expression is evaluated with that package as the context package. If the expression evaluates to true for that package, the package is included in the new package-set; otherwise, it is not included.

If the result of the expression is string, the result will be converted to a boolean. The empty string, `0` and `false` (case insensitive) are treated as false. Any other string is converted to true.

Abbreviated Syntax The following abbreviations are available:

- the `child` axis is implicitly assumed if no axis is specified. I.e. `foo` is equivalent to `child@foo`.
- `.` is a short-hand for `self@*`

- `//` is short for `/descendant-or-self@*/`. For example, `//foo` is short for `/descendant-or-self@*/child@foo` and so will select any `foo` package in the package graph; `foo//bar` is short for `child@foo/descendant-or-self@*/child@bar` and so will select all `bar` descendants of `foo` children.
- the above two short-cuts can be combined as `.//foo` which is equivalent to `descendant@foo`

Predicate expressions

Predicate expressions are evaluated as boolean functions that yield either true or false. The expression is executed for a context package. If the expressions yields true the package is kept as result of the associated location path, otherwise the package is filtered.

An expression may combine the following primitives to arbitrarily complex expressions. Several operators are available. Their associativity may be overruled by using parenthesis. Each primitive may be of only one of the following two types: string or boolean. Depending on the context a (partial) expression of string type may be implicitly converted to a boolean value. The empty string, `0` and `false` (case insensitive) are treated as false. Any other string is converted to true.

Location paths

Relative location paths are evaluated with respect to the context package of the predicate expression. Absolute location paths are evaluated independent of that. If the location path yields an empty set of packages the boolean result is false. If one or more packages are matched by the location path the result is treated as true.

Semantically this represents an *exists* predicate. As the location path is evaluated with respect to the current context of the expression the location path means "there exists a path from the current context package matched by the location path". By this primitive arbitrary graph reachability relations may be expressed.

String literals

Strings consist of a sequence of zero or more characters enclosed in double quotes (`"`). Strings are subject to the same *string substitution* as in the recipes. Unset variables are expanded to empty strings and are not treated as errors. The available variables are defined by the context of the whole expression.

To include double quotes as character into the string it has to be preceded by a backslash (`\`). To include a backslash itself use `\\`. The backslash escaping is done during parsing of the expression. Any string substitution is then performed for each context independently. As such, escape backslashes intended to preserve literal meanings of other characters during variable substitution must be written as `\\`.

Alternatively strings may be enclosed by single quotes (`'`). Such strings span from the first single quote until the next. Any character in between is taken verbatim and is not subject to any string substitution.

Examples:

```
"foo"
"${ENABLED}"
"$(match,${LICENSE},GPL)"
```

String function calls

String functions may be called directly without relying on string substitution. The general syntax is the funcion name, an opening parenthesis, zero or more arguments separated by comma and a closing parenthesis.

The following two lines are semantically equivalent:

```
"$(match,${LICENSE},GPL)"
match("${LICENSE}", "GPL")
```

The primitives can be combined with a number of operators. The following table lists all operators sorted by decreasing precedence. Operator precedence may be overruled by using parenthesis. The result of all operators is always a boolean. String comparison is done character by character, based on the Unicode code point. If the end of string is reached the string lengths are compared.

Operator	Associativity	Operand type	Meaning
`!`	Right	String or boolean	Logical NOT.
`<`	Left	String	Strictly less than.
`<=`	Left	String	Less than or equal.
`>`	Left	String	Strictly greater than.
`>=`	Left	String	Greater than or equal.
`==`	Left	String	Equal.
`!=`	Left	String	Not equal.
`&&`	Left	String or boolean	Logic AND.
`||`	Left	String or boolean	Logic OR.

See the following examples for some complex expressions:

- `"${FOO}" == "bar"` selects packages which use variable `FOO` an where the value is `bar`
- `!match("${LICENSE}", "GPL") && *[ match("${LICENSE}", "GPL") ]` selects packages that are *not* GPL-licensed and depend on a GPL-licensed package

Alias substitution

Aliases allow a string to be substituted for the first step of a *relative location path*. Absolute location paths (e.g. `/foo`) and relative location paths in predicates (e.g. `*[ foo ]`) are not not subject to alias substitution. Aliases are only substituted once. It is therefore not possible to reference an alias from another alias definition.

Example definitions:

```
alias:
   myApp: "host/files/group/app42"
   allTests: "//*-unittest"
   myAppDeps: "myApp/*"
```

Given the definitions above the following substations will be performed:

Query	Substituted query
myApp	host/files/group/app42
/myApp	/myApp
myAppDeps	myApp/*
foo/myApp	foo/myApp
myApp/lib	host/files/group/app42/lib
allTests/*[myAppDeps]	//*-unittest/*[myAppDeps]

1.4.9 bob-project

Name

bob-project - Create IDE project files

Synopsis

```
bob project [-h] [--list] [-D DEFINES] [-c CONFIGFILE] [-e NAME] [-E]
            [--download MODE] [--resume] [-n] [-b] [-j [JOBS]]
            [--sandbox | --no-sandbox]
            [projectGenerator] [package] ...
```

Description

Generate Project Files.

Options

-c CONFIGFILE Use config File

--download MODE Download from binary archive (yes, no, deps, packages)

See *bob-dev(1)* for details.

-D DEFINES Override default environment variable

-e NAME Preserve environment variable

-E Preserve whole environment

-j, --jobs Specifies the number of jobs to run simultaneously.

--list List available Generators

-n Do not build (bob dev) before generate project Files. RunTargets may not work

--no-sandbox Disable sandboxing

-b Do build only (bob dev -b) before generate project Files. No checkout

--resume Resume build where it was previously interrupted

--sandbox Enable sandboxing

Eclipse CDT project generator

```
bob project eclipseCdt <package> [-h] [-u] [--buildCfg BUILDCFG] [--overwrite]
                         [--destination DEST] [--name NAME]
                         [--exclude EXCLUDES] [-I ADDITIONAL_INCLUDES]
```

The Eclipse CDT generator has the following specific options. They have to be passed on the command line *after* the package name.

--buildCfg BUILDCFG Adds a new buildconfiguration. Format: <Name>::<flags>. Flags are passed to bob dev. See bob dev for a list of availabe flags.

--destination DEST Destination of project files.

--exclude EXCLUDES Packages will be marked as 'exclude from build' in eclipse. Usefull if indexer runs OOM.

-I ADDITIONAL_INCLUDES Additional include directories. (added recursive starting from this directory)

--name NAME Name of project. Default is complete_path_to_package

--overwrite Remove destination folder before generating.

-u, --update Update project files (.project).

QtCreator project generator

```
bob project qt-project <package> [-h] [-u] [--buildCfg BUILDCFG] [--overwrite]
                       [--destination DEST] [--name NAME]
                       [-I ADDITIONAL_INCLUDES] [-f Filter]
                       [--exclude Excludes] [--include Includes] [--kit KIT]
                       [-S START_INCLUDES] [-C CONFIG_DEF]
```

This generator also supports generation of project files for native Windows QtCreator by using MSYS2. The prerequisite is, that MSYS2 must be started by msys2_shell.cmd script.

The QtCreator project generator has the following specific options. They have to be passed on the command line *after* the package name.

--buildCfg BUILDCFG Adds a new buildconfiguration. Format: <Name>::<flags>

--destination DEST Destination of project files

-f Filter, --filter Filter File filter. A regex for matching additional files.

--exclude Excludes Package filter. A regex for excluding packages in QTCreator.

--include Includes Include package filter. A regex for including only the specified packages in QTCreator. Use single quotes to specify your regex. For exmaple: –include 'foobar-.*' You can also mix the Includes with the Excludes. In this case always use the Includes option beforehand. For example: –include 'foobar-.*' –exclude 'foobar-baz' This will ensure you only include packages wtih foobar-* but excludes the foobar-baz package.

-I ADDITIONAL_INCLUDES Additional include directories. (added recursive starting from this directory)

--kit KIT Kit to use for this project

--name NAME Name of project. Default is complete_path_to_package

--overwrite Remove destination folder before generating.

-u, --update Update project files (.files, .includes, .config)

-S START_INCLUDES Additional include directories, will be placed at the beginning of the include list.

-C CONFIG_DEF Add line to .config file. Can be used to specify preprocessor defines used by the QTCreator.

1.4.10 bob-query-meta

Name

bob-query-meta - Query metaEnvironment variables

Synopsis

::

bob query-meta [-h] [-D DEFINES] [-c CONFIGFILE] [-r] [–sandbox | –no-sandbox] packages [packages ...]

Description

This command lists variables from the metaEnvironment section of the recipe.

Options

-c CONFIGFILE Use config File

-D DEFINES Override default environment variable

--no-sandbox Disable sandboxing

-r Also list metaEnvironment variables for all dependencies.

--sandbox Enable sandboxing

1.4.11 bob-query-path

Name

bob-query-path - Query path information

Synopsis

```
bob query-path [-h] [-f FORMAT] [-D DEFINES] [-c CONFIGFILE]
               [--sandbox | --no-sandbox] [--develop | --release]
               [-q] [--fail] PACKAGE [PACKAGE ...]
```

Description

This command lists existing workspace directory names for packages given on the command line. Output is formatted with a format string that can contain placeholders

{name}	package name
{src}	checkout directory
{build}	build directory
{dist}	package directory

The default format is '{name}<tab>{dist}'.

If a directory does not exist for a step (because that step has never been executed or does not exist) or if one or more of the given packages does not exist, a error message is printed unless the `-q` option is provided.

Options

`-c CONFIGFILE` Use config File

`-D DEFINES` Override default environment variable

`--develop` Use developer mode

`-f FORMAT` Output format string

`-q` Be quiet in case of errors

`--fail` Return a non-zero exit code in case of errors

`--no-sandbox` Disable sandboxing

`--release` Use release mode

`--sandbox` Enable sandboxing

1.4.12 bob-query-recipe

Name

bob-query-recipe - Query package sources

Synopsis

```
bob query-recipe [-h] [-D DEFINES] [-c CONFIGFILE]
                 [--sandbox | --no-sandbox]
                 package
```

Description

Query recipe and class files of package.

Options

`-c CONFIGFILE` Use config File

`-D DEFINES` Override default environment variable

`--no-sandbox` Disable sandboxing

`--sandbox` Enable sandboxing

1.4.13 bob-query-scm

Name

bob-query-scm - Query SCM information

Synopsis

::

bob query-scm [-h] [-D DEFINES] [-c CONFIGFILE] [-f FORMATS] [–default DEFAULT] [-r] [–sandbox | –no-sandbox] packages [packages . . .]

Description

Query SCM configuration of packages.

By default this command will print one line for each SCM in the given package. The output format may be overridden by '-f'. By default the following formats are used:

- git="git {package} {dir} {url} {branch}"
- svn="svn {package} {dir} {url} {revision}"
- cvs="cvs {package} {dir} {cvsroot} {module}"
- url="url {package} {dir}/{fileName} {url}"

Options

-c CONFIGFILE Use config File

-D DEFINES Override default environment variable

--default DEFAULT Default for missing attributes (default: "")

-f FORMATS Output format for scm (syntax: scm=format). Can be specified multiple times.

--no-sandbox Disable sandboxing

-r, --recursive Recursively display dependencies

--sandbox Enable sandboxing

1.4.14 bob-status

Name

bob-status - Show SCM status

Synopsis

```
bob status [-h] [--develop | --release] [-c CONFIGFILE] [-D DEFINES]
           [--attic] [-r] [--sandbox | --no-sandbox] [--show-clean]
           [--show-overrides] [-v]
           [packages [packages ...]]
```

Description

Show SCM status of existing workspaces.

This command is intended to list the status of all checkouts, especially modifications, in a project. It can be used in two different modes. If the command is invoked without a package then all workspaces of the project are scanned for SCM changes. The other mode, when called with at least one package, shows only the status of the given package(s). In this case the `--develop` (default) and `--release` options may be used to select the workspaces. Adding `-r` scans the dependencies of the given package(s) too.

Options

`--attic` Consider attic directories too.

Normally SCMs that were moved to the attic next to the workspace are not checked. By using this option the attic directories that belong to a workspace are scanned too.

> **Attention:** Bob versions before 0.15 did not store the locations of the attic directories in the project. If the project was initially built with an older version then the attic listing will probably be incomplete. In this case Bob will print a warning that you should not rely on the output.

`-c CONFIGFILE` Use additional user configuration file. May be given more than once.

The configuration files have the same syntax as `default.yaml`. Their settings have higher precedence than `default.yaml`, with the last given configuration file being the highest.

`-D VAR[=VALUE]` Override default environment variable. May be given more than once. If the optional `VALUE` is not supplied the variable will be defined to an empty string.

`--develop` Use developer mode. This is the default.

`--no-sandbox` Disable sandboxing

`-r, --recursive` Recursively display dependencies. Only direct dependencies are considered, i.e. packages that are named in the `depends` section of the recipe. Consumed tool- and sandbox- packages that were forwarded are thus not visited.

`--release` Use release mode.

`--sandbox` Enable sandboxing

`--show-clean` Show the status of a checkout even if unmodified. This includes `--show-overrides`. See *Verbosity* below.

`--show-overrides` Show checkouts that have active *scmOverrides* (`O`) even if the SCM is unchanged. Override information is always displayed if a checkout is shown but a `STATUS` line is normally only emitted if the SCM was modified. Adding `-v` will additionally show the detailed override status. See *Verbosity* below.

`-v, --verbose` Increase verbosity. May be specified multiple times. See *Verbosity* below.

Output

The output of bob status is one line per SCM checkout. Only existing workspaces are considered. A status line consists of one ore more status codes followed by the SCM path.

Status codes can be interpreted as follows:

- `?` = missing information. The workspace was created by an older version of Bob that did not store enough information. The state of the SCM directory cannot be determined.

  Attention: The directory can be in any state with respect to the original checkout. You should manually inspect it because it might contain unsaved changes.

- `A` = attic. The recipe was changed for this checkout or the checkout is not referenced anymore in the recipe. The SCM path will be moved to the attic the next time the package is built.
- `C` = collision. The SCM was not yet checked out but there is an existing file/directory at the checkout location. The next run of the checkout step will fail.
- `E` = error. The SCM state could not be determined. The checkout is probably messed up. Use `-v` to get additional information about the error.
- `M` = modified. Some sources have been modified and not yet committed to SCM.
- `N` = new. The SCM was not yet checked out. There might still be an old checkout at the same location. This is indicated by a simultaneous `A` flag (see above).
- `O` = overridden. This SCM is affected by a *scmOverrides*. Pass `--show-overrides` to force the output of a `STATUS` line if the SCM is otherwise unmodified.
- `S` = switched. The commit/tag/branch/URL is different from the recipe.
- `U` = unpushed commits on configured branch. Git only. Some commits were made locally to the configured branch but they have not yet been pushed to the remote.
- `u` = unpushed commits not on configured branch. Git only. There are commits on other local branches than the configured branch, on a possibly detached HEAD or in the stash that have not been pushed to a remote.

The command shows the status of the current checkout. If the recipe was changed and the next build would move a checkout to the attic then the current information still refers to the existing checkout.

Verbosity

By default modified checkouts (`M` and `U` flags) and mismatches with respect to the recipes (`A`, `N` and `S` flags) are shown. Exceptional conditions like a collision (`C` flag), missing information to determine the SCM status (`?` flag) or if there was an error while retrieving the status (`E` flag) are always shown too.

By adding one or more `-v` options the display of less important information can be enabled. The following levels are available:

- `-v` shows a detailed description of all important flags (`A`, `C`, `E`, `M`, `N`, `S`, `U` and `?`). In particular the modified files and dirty commits of a SCM are listed. This level also shows git checkouts that have only unpushed commits that are not related to the configured branch (`u` flag), but without detailed description.
- `-vv` shows the full list of commits related to the `u` flag. Additionally the status of SCMs that are not modified is shown, i.e. without flags or only the `O` flag. If you want to display this information without also enabling the detailed descriptions (see above) use `--show-clean` or `--show-overrides`.
- `-vvv` enables the detailed description of the `O` flag. Additionally skipped workspaces that do not exist are shown.

1.5 Bob Release Notes

1.5.1 Bob 0.13 Release Notes

Changes made since Bob 0.12.0 include the following.

New commands

bob graph - highlevel command for dependency graphs

`bob graph` can be used to make package dependencies visible. For now graphviz (dot) graphs and d3 graphs are available, where d3 is the default. D3 graphs are interactive (zoom, drag, clickable, with hover effects) graphs using the d3.v4 JavaScript library.

See *bob-graph* and *Visualizing dependencies* for more information.

bob archive - manage binary artifacts

The `bob archive` command can be used to manage binary artifact archives. The command works on the included audit trails of the artifacts and can be used to selectively remove unneeded artifacts from the archive.

See *bob-archive* for more information.

New features

General

- Query language for package selection

  Packages can be selected through a query language starting with Bob 0.13. The language loosely resembles XPath. Almost all Bob commands (except `bob project`) have been converted to the new syntax. See *bob-paths* for more information.

- Introduce policies in `config.yaml`

  Introduce a policy scheme where backwards compatibility can be maintained. Unless a matching version is required by *bobMinimumVersion* the policies will retain their "old" state to keep backwards compatibility for existing projects.

  If a policy was neither set implicitly by *bobMinimumVersion* nor by an explicit *policies* entry in config.yaml Bob will warn the user if the policy is used. Typically the user should forward `bobMinimumVersion` and set an explicit behaviour.

  See *Policies* for more information.

- Introduce system wide / user Bob configuration

  Beside the existing project specific `default.yaml` this enables parsing of a system wide `bobdefaults.yaml` (`/etc/bobdefault.yaml`) as well as a user wide `default.yaml` (`$XDG_CONFIG_HOME/bob/default.yaml` or if `$XDG_CONFIG_HOME` is not set `~/.config/bob/default.yaml`).

  Parse order is from system-wide to workspace, meaning you can override settings from the system wide `default.yaml` in your user default's and this can be overridden in the workspace `default.yaml`.

- git: add remotes property to gitSCM

  The `remote-*` property allows adding extra remotes whereas the part after `remote-` corresponds to the remote name and the value given corresponds to the remote URL. For example `remote-my_name` set to `some/url.git` will result in an additional remote named `my_name` and the URL set to `some/url.git`.

  In conjunction with *scmOverrides* this can provide a convenient development option to automatically add remotes to private repositories.

Bob build / bob dev

- Added *command* configuration section in `default.yaml` to override Bob's default build arguments.
- Add option (`--no-log`) to disable logfile generation

  Many tools like `gcc`, `ls` or `git` detect whether they are running on a tty to colorize there output. If Bob writes a logfile a pipe is used and these tools do no longer provide a colored output. With the new `--no-log` option it's possible to switch of logfile generation and get colored output back.
- Create symlinks to dependencies next to workspace

  This adds a `deps` directory next to the workspace that will hold symlinks to all dependencies. The links are sorted by category:

  - If a sandbox is used the symlink will be called "sandbox"
  - All tools are linked by their tool name in the "tools" directory
  - Arguments (i.e. classic dependencies) are linked with their position and name in the "args" directory.

  This allows a quick traversal of the dependency tree after the build.
- Add option to build provided deps

  Use `--with-provided` to build provided dependencies and `--without-provided` to suppress building of provided dependencies (see *provideDeps*). In combination with `--destination`, `--with-provided` is default, otherwise `--without-provided` is default. Together with the new path query syntax one has now quite complete control over what will land within the destination folder.
- Pre- and post-scripts

  Add two optional *hooks* to `default.yaml` et.al. that can be run before (preBuildHook) and after (postBuildHook) the build. The preBuildHook receives the packages that should be built. It will fail the build if it returns with a non-zero status. The postBuildHook will get the status (success/fail) and the paths to the results as arguments.

  An example script for a postBuildHook can be found in `contrib/notify.sh`.
- Add `forced-fallback` download mode

  If the desired is artifact is available it is downloaded, otherwise all dependencies have to be downloaded.

Bob project

- QtCreator projects learned options to add include directories (`-S`) and compile definitions (`-C`) to a project.

Bob ls

- Add option (`-d`) to print package instead of contents

  The query syntax allows to select many packages with a single line. The new `-d` option allows to see the result set directly instead of the dependencies of these packages.

Bob jenkins

- Add `scm.ignore-hooks` extended option

  Sets the "Ignore post-commit hooks" option on all jobs. This instructs Jenkins to ignore changes notified by SCM post-commit hooks if enabled.

Changed behaviour

Backwards compatible policies

Bob will retain the old behavior unless instructed otherwise. See *Policies* for more information.

- Make `default.yaml` includes relative to including yaml file. See *relativeIncludes* policy.
- Do not take white listed variables into initial environment

  Previously the current set of environment variables during package calculation started with the ones named by `whitelist` in `default.yaml`. This made these variables bound to the value that was set during package calculation. Especially on Jenkins setups this is wrong as the machine that configures the Jenkins may have a different OS environment than the Jenkins executors/slaves.

  See *cleanEnvironment* policy for more details.

Other behavioural changes

- Add "live-build-id" support

  Previously binary artifacts required that all involved sources are checked out first. This adds quite a bit of space and time overhead if most of the artifacts are available or if they are updated only sporadically. Now Bob can query git servers and try to download an artifact before any sources are checked out.

  While this can speed up initial builds considerably it comes at the price that the sources are sometimes not checked out at all. Bob `buid/dev` learned the new `--always-checkout` option that accepts a regex for package names whose sources are always checked out. See *bob-dev* for mroe information.

- Jenkins: 'discard builds' deactivated for 'roots'

  Previously all Jenkins jobs artifacts were discarded if a non-root package is configured as Jenkins root. With this commit the configured roots will keep their artifacts.

- git: prune stale remote tracking branches

  Always prune stale remote branches. Otherwise branch renames in git repositories may cause jobs to fail.

- cvs: prune empty directories on initial checkout

  `cvs co` does not have a `-P` option like `cvs up` has. That option removes empty (=deleted) directories. We therefore use a `cvs up` after the initial `cvs co`, to get the same behaviour for the initial and subsequent builds.

Previously, `cvs co` would have created these empty directories, causing Bob to invoke the build step even if nothing changed in the repository between initial and subsequent checkout.

1.5.2 Bob 0.14 Release Notes

Changes made since Bob 0.13.0 include the following.

New features

- Add *rootFilter* setting to `default.yaml`

  This setting allows to filter root recipes by matching then with a globbing pattern. The effect of this is a faster package parsing due to the fact, that the tree is not build for filtered roots.

- Extended colored output options

  Bob gained a `--color=` switch that can be set to `auto` (default), `always` or `never` to control the colorized output of all commands.

- Add Microsoft Azure Blob storage backend

  The azure backend uses the Microsoft cloud block blob storage. It will use the same layout as all other backends so that the container can be made public readable and be fetched by the http backend without giving out the credentials.

  See *archive* for all options. Requires the `azure-storage-blob` Python3 library to be installed.

- The `-D` switch gained the ability to take values that contain =.

  The value of an environment variable can have any character in the recipes. When passed on the command line this is now possible too. Previously values that had a = (e.g. `-DFOO=A=B` were rejected or lead to crashes.

- Allow user to amend sandbox mounts/paths

  The default sandbox mount and search paths are defined in the sandbox recipe(s). If the user wanted to mount additional directories inside the sandbox (e.g. a local source code mirror in conjunction with a `scmOverride`) he previously had to edit the recipes.

  Now it is possible to add such paths to `default.yaml`. See *sandbox*.

- Allow substitution on scmOverrides

  The values of scmOverrides are mangled through *String substitution*.

Recipes

- Add `checkoutAssert` keyword

  With *checkoutAssert* you can define a checksum for some files or parts of files. This can be useful to detect license changes.

  Example:

```
checkoutAssert:
    -
        file: 'LICENSE'
        digestSHA1: be520183980a2e06b5272f3669395782f186d6d0
    -
        file: 'main.c'
```

(continues on next page)

(continued from previous page)

```
        start: 1
        end: 20
        digestSHA1: 949d02bed248e000d23bb81dfcce55e5603e8789
```

- Add `stripComponents` attribute for URL SCM

  With `stripComponents` it is possible to strip a configurable number of leading components from file names on extraction. This works only for tar files, though.

- Define `relocatable` property

  Historically any recipe that did define at least one tool was deemed to be non-relocatable. This implied that it cannot be stored as binary artifact unless we're in a sandbox.

  This is overly pessimistic. By introducing a `relocatable` property the user is able to express if a package is really relocatable or not. For backwards compatibility the default value is `False` if the recipe defines a tool, otherwise it is `True`.

  See *relocatable*.

- Implement proper inheritance for `shared` keyword

  The `shared` keyword was always ignored in classes. Now we deduct the final value by the usual inheritance rules just like all other values.

- Add `environment` property to *provideSandbox*

  A sandbox brings its own host environment that overrides the host environment when used. This can be quite different and be even a different architecture (think of an i386 sandbox image that is used on an x86_64 host).

  Bob regards the sandbox as a build invariant, that is it must have no impact on the results of the recipes that are built inside the sandbox. On the other hand there can be situations where this still the case. To properly handle this a sandbox recipe can now define environment variables that are picked up when this sandbox is actually used. By consuming this variable in other recipes Bob can determine the impact of the sandbox, if any.

  See *provideSandbox*.

Bob build / bob dev

- Allow build workspace to symlink checkout result

  This adds support for having symlinks to files of the checkout workspace in the build workspace (e.g. linking big archives). It has always been guaranteed to have the checkout and build workspace of the package available in the package step. Thus the package step can access this link instead of having to copy the archive in the build step.

Bob clean

- Added `-c` option
- Added `-D` option

Bob graph

- Added `--(no-)sandbox` option

Bob jenkins

- Prevent running in stale workspaces

  Jenkins does not delete workspaces automatically. He also has no clue if the job that is running in an old workspace is compatible or not. Hence we have to make sure ourself that we do not collide with old data in case of incremental builds.

  To pro-actively prevent such clashes every Jenkins server in Bob now get's a 32-bit random UUID which is appended to every path. This should minimize the chance to collide with another or old instance.

  Because of the low entropy the UUID does not provide a guarantee that no collision occurs. It also does not help on existing instances where no UUID was assigned. Therefore, as a 2nd line of defence, every workspace stores a canary with the variant-id. If the canary exists it has to match. Otherwise the build is aborted because we don't build the same thing in this existing workspace. The user has to clean up the workspace manually then.

- Add `jobs.policy` job trigger policy extended option

  This extended option that makes the actual trigger threshold between jobs configurable. It defaults to "stable" which was the default before. The "unstable" setting triggers on unstable builds and takes artifacts from stable and unstable builds. The "always" setting does what it says but might not be that useful.

Bob ls

- Add `--unsorted` option

  If given, show the packages in the order they were named in the recipe. By default they are sorted by name for better readability.

Plugins

- Add plugin settings support

  Sometimes plugin behaviour needs to be configurable by the user. On the other hand Bob expects plugins to be deterministic. To have a common interface for such settings it is possible for a plugin to define additional keywords in `default.yaml`. This provides Bob with the information to validate the settings and detect changes in a reliable manner.

  See *Plugin settings* and *`bob.input.PluginSetting`* for more details.

Changed behaviour

Backwards compatible policies

Bob will retain the old behavior unless instructed otherwise. See *Policies* for more information.

- `url` SCM: track checkout directory instead of file

  Historically the URL SCM was not tracking the checkout directory but the individual files that are downloaded by the SCM. This had the advantage that it is possible to download more than one file into the same directory. There are a couple of major disadvantages, though, that are now solved by "owning" the whole directory by the SCM where the file is downloaded.

  See *tidyUrlScm* policy for more details.

- Define `allRelocatable` policy

  The *allRelocatable* policy changes the default of the *relocatable* property to `True` regardless of any defined tools. This get's rid of the old heuristic which was too pessimistic in most cases.

- Define offline build properties and policy

  Bob will prevent network access by default during build and package steps when using a sandbox. The *{build,package}NetAccess* properties can override this behavior and the *offlineBuild* policy controls the default setting.

- Define `sandboxInvariant` policy

  Traditionally the impact of a sandbox to the build has not been handled consistently. On one hand the actual usage of a sandbox was not relevant for binary artifacts. As such, an artifact that was built inside a sandbox was also used when building without the sandbox (and vice versa). On the other hand Bob did rebuild everything from scratch when switching between sandbox/non-sandbox builds. This inconsistent behavior is rectified by the *sandboxInvariant* policy that consistently declares builds as invariant of the sandbox.

- Warn/Fail on duplicate dependencies (`uniqueDependency` policy)

  Naming the same dependency multiple times in a recipe is deprecated. If encountered a warning is shown unless the *uniqueDependency* policy is set to the new behavior. In this case the parsing is stopped with an error.

  See *uniqueDependency* policy for more details.

Dev/build behavioural changes

- Clean workspace on manual invocation of build/package.sh too

  For package steps the workspace is always cleaned. This was not the case if package.sh was invoked manually, though. Fix this and also adapt the behaviour of build.sh to the last Bob invocation (clean vs. incremental).

- Check all dependencies for input changes

  Changing the source code of a tool did not trigger an incremental rebuild of packages that use this tool. Such changes are now taken into account an will trigger an incremental build (if possible) of affected packages.

Other behavioural changes

- Handle dependency correctly if it is named multiple times

  It is possible (but not useful) to name the same dependency multiple times in a recipe. If only the tools are used it was not detected when multiple variants of the same package were specified. We now do the duplicate check independent of what is actually used.

  If a package is named multiple times in the depends section of a recipe and they are the same variant then we will issue a warning. It is (and was already in the past) defined that only the first result in the list is taken. But it is most likely an error of the user if there is more than one reference to the same package. It is also possible that, even if the packages themself are of the same variant, they might provide different dependencies or variables upwards. This is handled but not easily detectable by the user.

  Setting the *uniqueDependency* policy to the new behavior will halt the parsing with an error when a dependency is detected multiple times.

- In rare circumstances it was possible that a dependency was considered more than once in a package. This bug lead to wrongly calculated variant- and build-IDs. Unfortunately fixing this bug can lead to incompatible binary artifacts with previous versions of Bob.

There is no impact on the build result by this fix because the bug only affected the internal calculation. Bob already passed the dependency only once to the build script of such a package.

Performance improvements

- Optimize internal data structures to lower memory footprint

  Depending on the recipes the memory consumption is roughly halved. This also improves package calculation time to some extent.

- Bob dev: improve startup time

  The directory layout is now cached across invocations. This saves a couple of seconds until a build starts. It will also keep the directory assignment more stable in case of recipe changes.

- Bob dev: optimize audit trail generation time
- Bob project: improve project generation speed on large projects
- Jenkins: projects with many jobs are calculated an order of magnitude faster
- archive: improve local upload speed

  The compression level was reduced to 6 which produces only marginally bigger archives but is dramatically faster on compression. Some performance numbers with a reasonably (~600MB) sized workspace of text and binary files:

```
Bob 0.13 (level 9): 220MiB, 2m10s
Bob 0.14 (level 6): 222MiB, 0m25s
```

- Plugins: fix `PluginState` class comparison.

  Any plugin that used the *bob.input.PluginState* class caused a massive performance drop on huge projects. This is fixed for most plugins but it might still be necessary to update some plugins. See *bob.input.PluginState* for the details.

1.5.3 Bob 0.15 Release Notes

Changes made since Bob 0.14.0 include the following.

Prerequisites

Bob requires Python 3.5 or later starting with 0.15.

New features

Recipes

- Added the `sslVerify` SCM attribute to git, svn and url SCMs.

  The boolean attribute controls whether to verify the SSL certificate when fetching. It defaults to `true` but might be set to `false` to ignore certificate problems. The default for git is defined by the *secureSSL* policy.

- Introduced the `netAccess` *provideTools* property.

  Starting with the introduction of the *offlineBuild* policy the network access is usually restricted during build and package steps. There might be tools that need network access and are used during these steps, though. This can

be configured with the *{build,package}NetAccess* recipe properties but if the recipe does not know the exact tool it might not want to set these properties unconditionally.

The main use case are proprietary compilers that need to talk to a license server. The recipe should not bother which compiler is used exactly.

- *provideTools* gained the `environment` property.

  Certain tools (e.g. the C-compiler) and their associated environment variables (`CC`, `LD`, ...) are related to each other and must be defined and used together. Traditionally they are provided more or less independently of each other by a tools recipe with `provideTools` and `provideVars`. This has several drawbacks:

  - Not all recipes that define the same tool (e.g. the compiler) are providing the same set of environment variables.
  - Different tools might provide the same variables.

  By defining the environment variables with the tool their value can be picked up where it is actually used. This does not pollute the environment with variables that are only used in conjunction with the tool and are not used elsewhere.

  To prevent ambiguities each variable must only be defined by one tool. The parsing will fail if two tools are used in the same recipe that define the same variable. The reason is that Bob assumes that tools are independent of each other and no particular order is defined among them.

- Host fingerprint support.

  The fingerprint feature is used to track all dependencies of the recipes to the host environment in a generic manner. This is done by defining small scripts in the recipes that are invoked when the fingerprint is required. The output of these scripts is then taken as fingerprint.

  Fingerprints may range from the host C-compiler up to some libraries that are used by certain packages from the host. Because such external dependencies limit the exchange of artifacts between hosts the fingerprint is attached to binary artifacts to prevent unintended sharing between incompatible systems.

  The fingerprints are also used to detect relevant changes of the host environment. A package is always rebuilt if the fingerprint changes. The fingerprint is thus like an external build trigger. There are two major cases why Bob wants to rebuild when the fingerprint changes:

  - The package was built inside the sandbox and now the user builds outside of the sandbox (or vice versa).
  - The package was built without sandbox and the fingerprint changes. Bob has to assume that the external dependency has changed its state and thus an incremental rebuild is needed.

  If a package is not relocatable the workspace path is also some kind of host dependency. Bob treats the installation path of these tools as an additional fingerprint. This enables safe exchange of such artifacts that were previously only up-/downloaded when building inside a sandbox. This is only done if the *allRelocatable* policy is set to the new behaviour, though.

  See *fingerprintScript[{Bash,Pwsh}]* and *provideTools* for further details.

- Allow local files with the "url" SCM.

  If the URL starts with `/`, it is treated as a local file and is just copied with `cp` instead of downloaded using `curl`. This makes it possible to easily build from tarballs in a sandbox that has no network access and no `curl` command.

- Add `BOB_RECIPE_NAME` and `BOB_PACKAGE_NAME` built-in variables

  These variables are set internally by Bob and can be consumed by a recipe or class to know which recipe/package is built. For example it might be used to name the artifact (think rpm/deb/ipkg pacakge) like the recipe.

User configuration (default.yaml)

- Added `require` keyword.

  User configuration files may also require sometimes specific files to be included. Therefore the `require` keyword is introduced. The `require` keyword throws an error for missing files. Other than that it behaves just like the existing `include` keyword.

  See *User configuration (default.yaml)*.

- Added the `sslVerify` http archive backend attribute.

  The boolean attribute controls whether to verify the SSL certificate when fetching/uploading to/from HTTPS servers. The default depends on the *secureSSL* policy. The new behavior is to default to `true` but it might be set to `false` to ignore certificate problems.

Bob build / bob dev

- Gained parallel build support.

  If requested by `-j` a number of jobs can run simultaneously. Any checkout/build/package step that needs to be executed are counted as a job. Downloads and uploads of binary artifacts are separate jobs too. If a job fails the other currently running jobs are still finished before Bob returns. No new jobs are scheduled, though, unless the `-k` option is given.

  If the `-j` option is given without an argument, Bob will run as many jobs as there are processors on the machine.

- Added the `--no-link-deps` option.

  Creating links to dependencies confuses indexers like OpenGrok. This switch is there to disable the creation of linked dependencies.

Bob clean

The `clean` command has been extend to develop mode and attic directories. It will now purge unused workspace directories from develop mode builds too. If requested, it also removes attic directories.

At the same time the default is changed to 'develop' mode. This streamlines the behaviour with the other commands in Bob that also work in 'develop' mode by default. Together with the added `--(no-)sandbox` options it's configuration is finally consistent with the other commands.

If the user removes source workspaces (`-s` or `--attic` options) Bob will now check the SCMs in these directories for unsaved changes. The directory will only be deleted if no changes are found.

Attic directories are tracked starting with Bob 0.15. Any attic directories that were created with an older version of Bob in a project are not known and will not be cleaned. Bob will print a warning in case the project was created with an older version and attic directories should be cleaned.

Bob Jenkins

- Added the `jobs.update` option

  Updating a Jenkins job is costly. On projects with hundreds of jobs the overhead can be significant when just the description or audit trail information is updated but the job is not actually scheduled. The 'jobs.update' option adds a knob to control the update behaviour and provide lazy update methods.

Bob status

- Flag directories that will move to the attic or will be created.

  If the recipe is changed for a SCM checkout it will be moved to the attic when the checkout is run the next time. Bob now flags the directory as `A` (as in "attic") so that the user knows about the stale state. Likewise the `N` flag shows checkouts that do not exist in the workspace yet but will be created on the next run.

- Print override status as yaml.

  The `scmOverrides` configuration is done in a yaml file. Print the active overrides in the same format for consistency.

- The package argument is now optional.

  If no package is given then all known checkouts are scanned for changes. This will only give meaningful results if the project was created with Bob 0.15, though, because older versions of Bob did not store enough information about the project.

- Added an `--attic` switch to display information about the attic directories.

  The switch is orthogonal to the package argument. If a package is selected then all attic directories of this package are displayed. Otherwise all known attic directories are scanned.

- For git repositories `bob status` will now check all unpushed commits.

  This does not only check local branches for unpushed commits but all refs. Therefore stash, detached HEAD, etc. are checked as well.

- Improved output verbosity and streamlined its selection.

  The `--show-clean` was added to explicitly show unchanged checkouts. These are now hidden by default. The behaviour of the `-v` and `--show-clean` options has been made more consistent. See *Verbosity* of the `bob status` manpage.

- The `--sandbox` / `--no-sandbox` options were added.

Bob query-meta, query-scm, query-recipe

- The `--sandbox` / `--no-sandbox` options were added.

Changed behaviour

Backwards compatible policies

- Added *mergeEnvironment* policy.

  The `environment` and `privateEnvironment` sections of the recipes and classes it inherits from are merged when the packages are calculated. Traditionally this was done on a key-by-key basis without variable substitution. Keys from the recipe or an inherited class would simply shadow keys from later inherited classes. This had the effect that the definitions of later inherited classes were lost.

  The new behavior is to make all environment keys eligible to variable substitution. The definitions of the recipe has the highest precedence (i.e. it is substituted last). Declarations of classes are substituted in their inheritance order, that is, the last inherited class has the highest precedence.

  See *mergeEnvironment* for more details.

- Added the *secureSSL* policy.

  Due to historical reasons Bob did not check for SSL certificate errors everywhere. While most parts were already secure the git SCM and HTTPS archive backend were still insecure by default. This is rectified by the secureSSL policy where the new behavior is to always check the certificate.

Other behavioural changes

- `bob jenkins` defaults to secure SSL connections.

  Always use secure SSL connections by default. If the user still needs to connect to insecure HTTPS servers the `--no-ssl-verify` option may be used.

- The default project directory name of the built-in `bob project` generators have been shortened.

  By default the package path was used for the project directory. If the built package was deep in the hierarchy this could lead to excessively long path names. Instead Bob now uses the project name (which defaults to the package name) as directory name.

  This could theoretically create clashes if different variants of a recipe are built simultaneously. But the user can handle that by the `--destination` option rather than creating excessively long paths by default.

- Fixed the *sandboxInvariant* policy.

  Setting the *sandboxInvariant* policy to the new behaviour had exactly the opposite effect as intended. It caused artifacts of sandbox and non-sandbox builds to be always incompatible to each other. This has been rectified.

- On git checkouts only tags that are on branches are cloned by default. Previously all tags have been cloned. This is done to prevent `bob status` from flagging a repository as having unpushed commits because orphaned tags are not referenced from remote heads. Note that git does not fetch such tags anyway on a `git pull` or `git fetch`. The user has to fetch orphaned tags explicitly either by naming them on the command line or with the `--tags` option. If the recipe requests such a tag to be checked out then it will still be fetched explicitly by Bob.

Backwards incompatible changes

- Recipes cannot define variables starting with `BOB_`

  Variable names starting with `BOB_` have historically been used by Bob all over the place. This release added even some more (`BOB_RECIPE_NAME`, `BOB_PACKAGE_NAME`). To prevent accidential breakage of recipes by future versions of Bob that define additional variables, Bob will reject recipes that define variables in this name space. This may break existing recipes but it cannot be guaranteed that they will work with future versions either.

- The default mode of the `bob clean` command was changed from release mode to develop mode. This makes its behaviour consistent with all other Bob commands.

- The *String functions* and *`bob.input.PluginState`* APIs have changed.

  Due to the redesigned package calculation it is not possible to pass the *`bob.input.Tool`* or *`bob.input.Package`* objects to the plugins anymore that use *`bob.input.PluginState`*. Plugins requiring the old API still work but the `tools` and `package` arguments are filled with empty values.

  Similarly the string functions lost the `tools` parameter. The `sandbox` parameter was converted to a plain `bool`.

  This changes the behaviour and projects relying on the removed parameters will have to be refactored. But at least the parsing should not crash. A warning is displayed for every usage of a deprecated API.

Performance improvements

- Refactored package calculation.

  The internal logic of the package calculation was optiomized. The parsing time and memory footptrint are improved by up to an order of magnitude.

- `bob graph` scales much better on large package graphs.

1.5.4 Bob 0.16 Release Notes

Changes made since Bob 0.15.0 include the following.

Installation

- Bob is now installed via `pip3`.

  Bob installation is now based on standard Python `setuptools`. If you upgrade from Bob 0.15.0 make sure to delete all files that were installed by `make install` previously. Bob 0.15.1 also used the standard Python facilities already. See *Install* for more details.

- Bob can run directly from a git worktree checkouts now.

New features

Audit trail

- Put original URL of URL-SCM into audit trail record.

  The audit trail just recorded the hash sum and file name of URL-SCMs. The URL where it was downloaded was only available through the referenced recipes. While it may be possible to restore the information from the recipes, Bob now puts the originating URL directly into the audit record.

- Added `/etc/os-release` to build information

  Basically all major distributions have adopted the systemd initiated unified OS identification via `/etc/os-release`. The content of the file is now included in the "build" section of the audit trail as "os-release". This information is especially valuable when you build in containers where the `uname` information just shows on which host kernel the build is run but not in which container image.

See *Audit trail* for more details.

Recipes

- Bob gained the support for splitting up a monolithic project into several *layers*.

  Layers allow you to structure your projects into larger entities that can be reused in other projects. This modularity helps to separate different aspects of bigger projects like the used toolchain, the board support package and the applications integration.

  See *Configuration* for an overview.

- URL SCMs use `wget` as fallback in case `curl` isn't available.

  This fallback has some limitations, though:

  - `wget` does not support as many protocols as `curl`, but for the commonly used HTTP(S) and FTP schemes it works.
  - The time-stamp mechanism (`-N`) can't be used as with `curl`, because wget doesn't support it together with own outputfile names (`-O`). Therefore `wget` cannot download updates but will only fetch the file if it does not exist yet in the workspace.

- Added support for bare variable names in strings.

  Usually the name of the variable must be enclosed in curly braces when dereferencing it, e.g. `${FOO}`. In regular POSIX shells this is not needed if the variable name consists of only letters, numbers and underscores *and* if the next character is something different to terminate the variable name, e.g. `"Hello $NAME."`. In the light of more compatibility Bob supports this notion too now.

- Added the *fingerprintVars* keyword

  This keyword controls the set of environment variables that are passed to a *fingerprintScript[{Bash,Pwsh}]* in conjunction with the *fingerprintVars* policy. This is useful to prevent the unneeded execution of identical fingerprint scripts. See the policy and the keyword for more details.

- Added support for pre-release and development version numbers in *bobMinimumVersion*.

  Projects can now reliably require pre-release or development versions and do not have to wait until the next release of Bob is published.

User configuration (default.yaml)

- Added `fileMode` and `directoryMode` options to override file modes in file backend.

  Normally the binary artifacts are created with default permissions by the file backend. It might be desirable to set some special access modes, though. An example would be to grant the group write access which is normally prohibited by the umask.

  See *archive* for more information.

Bob dev/build

- The overall progress is now shown during parallel builds.
- Added the `packages` download option.

  With `--download=packages=RE` a regular expression `RE` can be used to specify which packages should be downloaded. If the package cannot be found in any artifact cache then it will still be built.

Bob jenkins

- Added timeout option for Jenkins `git` checkouts

  By default Jenkins has a timeout of 10 minutes for `git clone` and `fetch` operations. Depending on the server and the repository size this might not be enough. By setting the new `scm.git.timeout` option it is possible to change the timeout.

Bob project

- Qt Creator generator gained support for Windows by MSYS2

The `bob project qt-creator` plugin will be able to create a Windows native Qt Creator project by using `MSYS2`. This requires that `MSYS2` must have been started by `msys2_shell.cmd` to have the `WD` environment available.

- Added Visual Studio 2019 generator.

  This generator works currently only on MSYS2. Bob and the build is run on MSYS2 while Visual Studio is running natively. The drawback is that debugging of applications does not work because they would need to be built with the MS compiler. Error messages are also not understood by VS because the paths emitted by the compier refer to MSYS names and not to native Windows paths.

Changed behaviour

Backwards compatible policies

- Introduced the *sandboxFingerprints* policy.

  When *Host dependency fingerprinting* was introduced, Bob initially used a shortcut and did not execute fingerprint scripts in the sandbox. This saved a bit of complexity and also relieved the build logic from the need to build the sandbox just to execute the fingerprint script. While the old approach was not producing wrong results it was overly pessimistic. It prevents sharing of any fingerprinted artifacts between sandbox and non-sandbox builds even if the fingerprint is the same.

  When set to the new behaviour the fingerprint scripts will be executed in the sandbox too. A caching of these results by the artifact cache is also implemented to reduce the need of fetching the sandbox image. Fingerprinted artifacts will be shared between sandbox- and non-sandbox-builds given the *fingerprintScript[{Bash,Pwsh}]* yields the same result.

- Added the *fingerprintVars* policy

  When *Host dependency fingerprinting* was introduced there was no dedicated environment variable handling implemented for them. The simple policy was to pass all environment variables of the affected package to the `fingerprintScript`. Unfortunately this results in the repeated execution of identical scripts if the variables change between packages, even if they are not used by the `fingerprintScript`.

  The newly *fingerprintVars* keyword now allows to specify the subset of variables that are used. As this defaults to an empty set it would change the behaviour of of fingerprints in existing recipes. This policy hence controls the evaluation of the added *fingerprintVars* keyword.

Other behavioural changes

- Fixed a bug where the URL of binary artifact servers was not properly quoted when building on Jenkins.

  The URL of a `archive` backend is not subject to string substitution. While this has been possible on Jenkins builds in the past it was never supported for local builds ever. On Jenkins it is now prevented, too.

- Relaxed the requirement of what must be matched by regular expressions.

  All options that take a regular expression did implicitly match the string start on Bob 0.15 and before. This is unexpected by the user and in stark contrast to tools like `grep` and `perl`. Starting with Bob 0.16 a regular expression can match anywhere in the string. If you really require to match the line start you can do this by adding the `^` meta character to the regex.

- Raised the severity of `A` and `N` flags of `bob status`.

  The handling of the `A` (attic) and `N` (new) flags was inconsistent with respect to the `S` (switched branch) flag. All these flags are shown by default now. They are all inconsistencies with respect to the recipes. They are now

treated with equal severity like direct source modifications because these are consequences of modifications in the recipes or manual changes of the checkouts.

Backwards incompatible changes

- The git SCM does not fetch commits explicitly from the server.

  If the recipe checks out a particular commit (`commit` key used) then Bob 0.15 used to fetch this commit explicitly from the server. But fetching commits explicitly is not supported by default by git servers and will typically be denied as follows:

  ```
  error: Server does not allow request for unadvertised object
  ```

  Instead Bob clones all branches and tags by default and relies on the assumption that the required commit is reachable by any of them.

Known issues

- Incorrect Jenkins shared artifacts may be used when policies are changed.

  The Jenkins logic tracks packages that are marked as *shared* by their Variant-Id. But if the Build-Id of such an artifact changes it is not updated at the shared location. While the build result will still be correct the created artifacts will not be found by other builds because they have an incorrect Build-Id. This bug can be triggered if one of the following policies are changed explicitly or implicitly by increasing the *bobMinimumVersion*:

  - *sandboxInvariant*
  - *sandboxFingerprints*
  - *fingerprintVars*

  Workaround: prune shared location on all Jenkins slaves when upgrading recipes that change one of the above policies. The default shared location is `${JENKINS_HOME}/bob` but it can be configured by the `shared.dir` *extended option*.

  See issue #287.

1.5.5 Bob 0.17 Release Notes

Changes made since Bob 0.16.0 include the following.

New features

Windows compatibility

Bob can be installed on Windows with a native Python installation. Together with the newly added PowerShell support (see below) it is possible to use Bob without any Unix Tools on Windows. It is also possible to use MSYS2 `bash.exe` from a natively installed Bob.

Recipes

- Bob gained support for multiple scripting languages.

  Traditionally Bob supported only bash scripts. Starting with this version it is also possible to use PowerShell in a recipe. The default language is still 'bash' but this may be overridden by the *scriptLanguage* setting in `config.yaml`. The default language is used to call the right interpreter for the `checkoutScript`, `buildScript`, `packageScript` or `fingerprintScript` scripts.

  For each `*Script` entry there is a variant for the different supported languages. For example the `buildScript` has the `buildScriptBash` and `buildScriptPwsh` siblings. They hold the scripts for the respective languages. This schema extends to the other scripts too.

  The selection which language is used at build time is done during execution. By default bash scripts are used. The language may be configured globally in `config.yaml` by setting *scriptLanguage* or on in the recipe/class by the *scriptLanguage* key.

- Add support for expressions in `if` fields.

  Traditionally `if` fields (e.g. in *depends*) are strings that can use substitutions to produce a boolean result. These strings are then considered false if they were empty, "0" or "false". In any other case they are considered to be true. Example:

  ```
  if: "$(or,$(eq,${FOO},bar),${BAZ})"
  ```

  Instead you can now write:

  ```
  if: !expr |
      "${FOO}" == "bar" || "${BAZ}"
  ```

  which is much more readable and can be properly indented if the expression gets complex. The new syntax is allowed at all `if` and `fingerprintIf` keys.

- Bob gained a simple `import` SCM that allows importing files from the project directly in to a src workspace. It copies the directory specified in `url` to the workspace. See *checkoutSCM* for more details.

- Relax *whitelist* name schema.

  Real environment variables can almost have any character in them. Bob now allows to white list any variable name that is supported by the OS.

- Allow Windows paths for URL SCM.

  On Windows it is now allowed to use fully qualified paths in the `url` SCM, e.g. `C:\tmp.txt`, `file:///C:/tmp.txt` or `\\server\path`.

- The `BOB_HOST_PLATFORM` variable is automatically populated.

  When building on multiple platforms the recipes will have to make platform specific decisions. The standard `BOB_HOST_PLATFORM` variable provides a standard way to identify the host platform type. See *{checkout,build,package}Vars* for more details.

- The git SCM now supports shallow clones.

  By setting the `shallow` attribute on a git SCM the number of commits that are fetched from the tip of the remote branch(es) is limited. This can improve initial clone times considerably. Likewise a `singleBranch` attribute was added too which is implicitly enabled if `shallow` is used. Because it is a regular SCM property the user can override it as needed from `default.yaml` via *scmOverrides*:

```
scmOverrides:
    - match:
        scm: git
      set:
        shallow: 1
```

See *checkoutSCM* for more details.

Bob build / bob dev

- Gained an option to build without audit trail.

  The generation of the audit trail is usually barely noticeable. But if a large number of repositories is checked out it can add a significant overhead nonetheless. This release adds a `-A` / `--without-audit` option so that the user can skip the generation of an audit trail to save this time.

  Without an audit trail it is not possible anymore to upload an artifact because vital information is missing. Consequently the generation of an audit trail is skipped if the audit trail of a dependency is missing or if it cannot be read. Otherwise the information would be incomplete.

Bob query-path

- `query-path` will now show a message if the query matched no packages at all or if an expected directory is missing. This is printed on stderr so that it does not interfere with existing scripts.
- Added `-q` option.

  Adds the possibility to silence the error messages provided by query-path on missing packages and paths.
- Added `--fail` option.

  This option enables non-zero return codes in the case of missing packages and/or paths.

Visual Studio project generator

- Built Windows executables are recognized and can be directly executed from Visual Studio. This includes debugging them but requires that the executables are built with the Microsoft compiler, though. Visual Studio cannot debug executables built by MSYS2 gcc.

Plugins

- The bob path is now passed to generator plugins.

  Starting with `apiVersion` 0.17 the generator plugin will get the path of the Bob executable. The plugin may use it to generate project files that work even if Bob is not in `$PATH`. See *Generators* for more details.

Changed behaviour

Other behavioural changes

- The `bobMinimumVersion` comparison is now fully SemVer compatible. Pre-release versions are considered to precede the final release version. When using pre-release versions of Bob the inferred version is based on

the next anticipated version, e.g. when the last tag was 0.17.0 the calculated pre-release version will be 0.17.1-devXXX. This does not impact released versions.

Backwards incompatible changes

- Recipe or class YAML files that start with a dot (`.`) are ignored. Some editors generate such hidden temporary files while editing recipes. It's still allowed to use command line configuration files (`bob -c ...`) or include files (`default.yaml`) starting with a dot.
- Fixed two bugs related to fingerprinting of build steps. Under certain recipe conditions it could happen that fingerprinting was not correct and even lead to inconsistent behaviour between subsequent invocations. The fix might break the discovery of binary artifacts that were created with older versions of Bob if the project used fingerprints and had actually triggered those bugs.
- Fixed another bug related to fingerprints. Jenkins builds uploaded incorrect fingerprint prediction files for non-relocatable packages. This might lead to unneeded checkouts for a user but the correct binary artifacts will still be found.
- Binary artifacts of the major platforms are always separated: POSIX systems (e.g. Linux), native Windows and MSYS2 on Windows. Artifacts that are built on these platforms are not shared between each other, even when building the same recipes. The reason is that the file systems and how they are seen by Python differ too much. It is not possible to reliably share these artifacts without introducing occidental false sharing.

  A similar distinction is done on Windows regarding the capability to create symlinks. Some scripts and archive utilities will behave differently if it is possible to create symlinks on Windows or not. This capability is not granted by default but only available to administrative shells or user accounts that were given the `SeCreateSymbolicLinkPrivilege` privilege. Builds with and without the symlink capability are now treated differently because there there would be false sharing if symlinks are actually used.
- Environment variable names in `default.yaml` are now correctly validated.

  Environment variables that are defined in the recipes must not begin with `BOB_` because this namespace is reserved for future usage by Bob. This was not enforced yet for variables in `default.yaml`.

Performance improvements

- Audit trail generation is not blocking execution of other parallel jobs anymore. Previously Bob was not scheduling new jobs while an audit trail was generated.
- The audit trail of a checkout step is now only updated if something has changed in the workspace. If no updates were found and the checkout workspace did not change the old audit trail record is still considered valid. This saves significant time for large checkouts because no SCM status need to be scanned.

CHAPTER 2

Copyright

CHAPTER 3

Indices and tables

- genindex
- search

Index

C

copy() (*bob.input.PluginState method*), 56

D

doesProvideTools() (*bob.input.Step method*), 58

G

getAllDepSteps() (*bob.input.Package method*), 57
getAllDepSteps() (*bob.input.Step method*), 58
getArguments() (*bob.input.Step method*), 58
getBuildStep() (*bob.input.Package method*), 58
getCheckoutStep() (*bob.input.Package method*), 58
getDigestScript() (*bob.input.Step method*), 58
getDirectDepSteps() (*bob.input.Package method*), 58
getEnv() (*bob.input.Step method*), 59
getEnvironment() (*bob.input.Sandbox method*), 60
getEnvironment() (*bob.input.Tool method*), 60
getExecPath() (*bob.input.Step method*), 59
getIndirectDepSteps() (*bob.input.Package method*), 58
getJenkinsScript() (*bob.input.Step method*), 59
getLabel() (*bob.input.Step method*), 59
getLibraryPaths() (*bob.input.Step method*), 59
getLibs() (*bob.input.Tool method*), 61
getMetaEnv() (*bob.input.Package method*), 58
getMounts() (*bob.input.Sandbox method*), 60
getName() (*bob.input.Package method*), 58
getName() (*bob.input.Recipe method*), 57
getNetAccess() (*bob.input.Tool method*), 61
getPackage() (*bob.input.Step method*), 59
getPackageName() (*bob.input.Recipe method*), 57
getPackageStep() (*bob.input.Package method*), 58
getPath() (*bob.input.Tool method*), 61
getPaths() (*bob.input.Sandbox method*), 60
getPaths() (*bob.input.Step method*), 59
getRecipe() (*bob.input.Package method*), 58
getSandbox() (*bob.input.Step method*), 59
getScript() (*bob.input.Step method*), 59
getSettings() (*bob.input.PluginSetting method*), 56
getStack() (*bob.input.Package method*), 58
getStep() (*bob.input.Sandbox method*), 60
getStep() (*bob.input.Tool method*), 61
getTools() (*bob.input.Step method*), 59
getValue() (*bob.input.PluginProperty method*), 55
getVariantId() (*bob.input.Step method*), 59
getWorkspacePath() (*bob.input.Step method*), 59

I

inherit() (*bob.input.PluginProperty method*), 55
isBuildStep() (*bob.input.Step method*), 60
isCheckoutStep() (*bob.input.Step method*), 60
isDeterministic() (*bob.input.Step method*), 60
isEnabled() (*bob.input.Sandbox method*), 60
isPackageStep() (*bob.input.Step method*), 60
isPresent() (*bob.input.PluginProperty method*), 55
isRelocatable() (*bob.input.Package method*), 58
isRelocatable() (*bob.input.Step method*), 60
isRoot() (*bob.input.Recipe method*), 57
isShared() (*bob.input.Step method*), 60
isValid() (*bob.input.Step method*), 60

M

merge() (*bob.input.PluginSetting method*), 56

O

onEnter() (*bob.input.PluginState method*), 56
onFinish() (*bob.input.PluginState method*), 57
onUse() (*bob.input.PluginState method*), 57

P

Package (*class in bob.input*), 57
PluginProperty (*class in bob.input*), 55
PluginSetting (*class in bob.input*), 55
PluginState (*class in bob.input*), 56

R

Recipe (*class in bob.input*), 57

S

Sandbox (*class in bob.input*), 60
Step (*class in bob.input*), 58

T

Tool (*class in bob.input*), 60

V

validate() (*bob.input.PluginProperty static method*), 55
validate() (*bob.input.PluginSetting static method*), 56