The Nix Build Farm: A Declarative Approach to Continuous Integration

Eelco Dolstra, Eelco Visser
Delft University of Technology

Abstract

There are many tools to support continuous integration (the process of automatically and continuously building a project from a version management repository). However, they do not have good support for variability in the build environment: dependencies such as compilers, libraries or testing tools must typically be installed manually on all machines on which automated builds are performed. The *Nix package manager* solves this problem: it has a purely functional language for describing package build actions and their dependencies, allowing the build environment for projects to be produced automatically and deterministically. We have used Nix to build a continuous integration tool, the *Nix build farm*, that is in use to continuously build and release a large set of projects.

1. Introduction

Continuous integration (Fowler and Foemmel 2006), also known as "daily builds", is a simple technique to improve the quality of the software development process. An automated system continuously or periodically checks out the source code of a project, builds it, runs tests, and produces reports for the developers. Thus, various errors that might accidentally be committed into the code base are automatically caught. Such a system allows more in-depth testing than what developers could feasibly do manually:

- *Portability testing*: The software may need to be built and tested on many different platforms. It is infeasible for each developer to do this before every commit.
- Likewise, many projects have very large test sets (e.g., regression tests in a compiler, or stress tests in a DBMS) that can take hours or days to run to completion.
- Many kinds of static and dynamic analyses can be performed as part of the tests, such as code coverage runs and static analyses.

Email addresses: `e.dolstra@tudelft.nl` (Eelco Dolstra), `visser@acm.org` (Eelco Visser).

– It may also be necessary to build many different *variants* of the software. For instance, it may be necessary to verify that the component builds with various versions of a compiler.
– Developers typically use incremental building to test their changes (since a full build may take too long), but this is unreliable with many build management tools (such as Make), i.e., the result of the incremental build might differ from a full build.
– It ensures that the software can be built from the sources under revision control. Users of version management systems such as CVS and Subversion often forget to place source files under revision control.
– The machines on which the continuous integration system runs ideally provides a clean, well-defined build environment. If this environment is administered through proper SCM techniques, then builds produced by the system can be reproduced. In contrast, developer work environments are typically not under any kind of SCM control.
– In large projects, developers often work on a particular component of the project, and do not build and test the composition of those components (again since this is likely to take too long). To prevent the phenomenon of "big bang integration", where components are only tested together near the end of the development process, it is important to test components together as soon as possible (hence *continuous integration*).

In its simplest form, a continuous integration tool sits in a loop building and releasing software components from a version management system. For each component, it performs the following tasks:

(i) It obtains the latest version of the component's source code from the version management system.
(ii) It runs the component's build process (which presumably includes the execution of the component's test set).
(iii) It presents the results of the build (such as error logs) to the developers, e.g., by producing a web page.

Examples of continuous integration tools include CruiseControl (ThoughtWorks 2005), Tinderbox (Mozilla Foundation 2005), Sisyphus (van der Storm 2005), Anthill (Urbancode 2005) and BuildBot. However, these tools have various limitations. First, they do not manage the *build environment*. The build environment consists of the dependencies necessary to perform a build action, e.g., compilers, libraries, etc. Setting up the environment is typically done manually, and without proper SCM control (so it may be hard to reproduce a build at a later time). Manual management of the environment scales poorly in the number of configurations that must be supported. For instance, suppose that we want to build a component that requires a certain compiler X. We then have to go to each machine and install X. If we later need a newer version of X, the process must be repeated all over again. An ever worse problem occurs if there are conflicting, mutually exclusive versions of the dependencies. Thus, simply installing the latest version is not an option. Of course, we can install these components in different directories and manually pass the appropriate paths to the build processes of the various components. But this is a rather tiresome and error-prone process.

The second problem with existing continuous integration tools is that they do not easily support variability in software systems. A system may have a great deal of build-time variability: optional functionality, whether to build a debug or production version, different versions of dependencies, and so on. (For instance, the Linux kernel now has

over 2,600 build-time configuration switches.) It is therefore important that a continuous integration tool can easily select and test different instances from the configuration space of the system to reveal problem, such as erroneous interactions between features. In a continuous integration setting, it is also useful to test different combinations of versions of subsystems, e.g., the head revision of a component against stable releases of its dependencies, and vice versa, as this can reveal various integration problems.

This paper briefly describes the *Nix build farm*, a continuous integration tool that solves these problems. ("Build farm" refers to the large set of different machines necessary for portability testing (Hemel 2003).) It is built on top of the *Nix package manager* (Dolstra et al. 2004b,a; Dolstra 2006; Nix project 2008), which has a purely functional language for describing package build actions and their dependencies. This allows the build environment for projects to be produced automatically and deterministically, and variability in components to be expressed naturally using functions.

2. The Nix build farm

The Nix package manager (`http://nixos.org/`) has precisely the properties needed to address the problems of managing the build environment and supporting variability. As a source-based deployment system, Nix has a *lazy purely functional language* (the *Nix expression language*) to describe how to build packages and how to compose them. This allows the build environment to be expressed in a self-contained and reproducible way, and it enables variability to be expressed by turning packages into *functions* of the desired variabilities. Laziness is important because it prevents function arguments, typically representing large package build actions, from being evaluated when they are not needed. The functional language also abstracts over multi-platform builds — Nix automatically dispatches the building of subexpressions to machines of the appropriate type.

Nix also stores packages in such a way that variants of packages do not interfere with each other (e.g., overwrite each other) and that prevents undeclared dependencies. The build result of each package instance is stored in the file system under a cryptographic hash of all inputs involved in building the package, such as its sources, build scripts, and dependencies such as compilers. For instance, a build of a particular package instance (e.g. Firefox) might be stored under

`/nix/store/1wxsnm40dvlfcb4cqff0kjb1dqllrr3v-firefox-2.0.0.14/`

where 1wxsnm40dvlf... is a 160-bit cryptographic hash. The directory /nix/store is called the *Nix store*. If any input differs between two package build actions, then the resulting packages will be stored in different locations in the file system and will not overwrite each other. Thus, conflicting dependencies such as different versions of a compiler no longer cause a problem; they are stored in isolation from each other. At the same time, if any two packages between different build farm jobs have the same inputs, they will be built only once. This prevents unnecessary rebuilds.

An added advantage of the hash approach is that it prevents undeclared build-time dependencies in build actions. If a build action does not explicitly declare an input, then it won't be able to find the input (e.g., the package won't appear in the C compiler's header search path). This is in contrast to build tools such as Make (Feldman 1979) or Ant (Apache Software Foundation 2005), which have no way to verify the completeness of

```
{nixpkgs, patchelfCheckout}:

...

rec {

  patchelfTarball = makeSourceTarball {
    src = patchelfCheckout;
    inherit stdenv;
    buildInputs = [autoconf automake];
  };

  patchelfNixBuild = doCoverageAnalysis: pkgs: nixBuild {
    src = patchelfTarball;
    inherit (pkgs) stdenv;
    lcovFilter = ["/nix/store/*" "tests/*"];
    inherit doCoverageAnalysis;
  };

  patchelfRPMBuild = diskImage: rpmBuild {
    inherit diskImage;
    src = patchelfTarball;
  };

  patchelfDebianBuild = diskImage: debianBuild { ... };

  patchelfRelease = makeReleasePage {
    fullName = "PatchELF";
    contactEmail = "e.dolstra@tudelft.nl";
    sourceTarball = patchelfTarball;
    nixBuilds = [
      (patchelfNixBuild false pkgsi686Linux)
      (patchelfNixBuild false pkgsx86_64Linux)
    ];
    coverageAnalysis = patchelfNixBuild true pkgsi686Linux;
    rpmBuilds = [
      (patchelfRPMBuild vmTools.diskImages.redhat9i386)
      (patchelfRPMBuild vmTools.diskImages.fedora2i386)
      (patchelfRPMBuild vmTools.diskImages.fedora3i386)
      (patchelfRPMBuild vmTools.diskImages.fedora5i386)
      (patchelfRPMBuild vmTools.diskImages.fedora7i386)
      (patchelfRPMBuild vmTools.diskImages.fedora8i386)
      (patchelfRPMBuild vmTools.diskImages.suse90i386)
      (patchelfRPMBuild vmTools.diskImages.opensuse103i386)
    ];
    nodistBuilds = [
      (patchelfDebianBuild vmTools.diskImages.ubuntu710i386)
      (patchelfDebianBuild vmTools.diskImages.debian40r3i386)
    ];
  };

}
```

Fig. 1. patchelf.nix: Nix expression for the PatchELF job

dependency specifications. (This is why Make users often have to do a make clean before a build to ensure that everything that should be rebuilt really is.) The hash approach also allows *runtime* dependencies to be found generically by scanning for hashes in binaries, a technique reminiscent of how conservative garbage collectors find pointers (Dolstra et al. 2004b).

Figure 1 shows the Nix expression for the build farm job for a simple Unix package, *PatchELF*, a tool to modify ELF executables. Testing and releasing this package involves a number of actions:

– The sources from the version management repository must be turned into a proper *source distribution* by generating a variety of files, such as a configure script. This is

done using tools such as Autoconf and Automake.
- This source distribution must then be compiled on a variety of platforms, in this case Linux distributions such as various versions of Fedora, openSUSE, Ubuntu and Debian.
- A code coverage analysis is also performed by building an instrumented binary, running the test suite and generating a coverage report.
- Finally, a web page is generated that contains build logs and the coverage report as well as end-user downloadable packages.

All these actions have specific dependencies that must be present, in the right versions, on the machines on which they are performed. For instance, building the source distribution requires specific versions of Autoconf and Automake, while the coverage analysis requires a tool called lcov. Different build farm jobs can have conflicting version dependencies, but this is no problem due to Nix's hashing scheme; different versions are installed in different locations of the file system automatically.

The expression in Figure 1 is a *function* that takes two arguments — nixpkgs and patchelfCheckout — and yields an *attribute set* (a set of name/value pairs) containing the build actions of the job. Functions are defined by the syntax args: body. Attribute sets are defined as $\{\mathsf{name}_1 = \mathsf{value}_1;\ \ldots\ \mathsf{name}_n = \mathsf{value}_n;\}$. The keyword rec before an attribute set makes the set recursive, i.e., values can refer to other attributes.

The build farm calls the job with a value for the function argument patchelfCheckout containing a copy in the Nix store of the sources of the package obtained from its version management repository. The argument nixpkgs contains the source of the Nix Packages collection, a set of Nix expressions for hundreds of common Unix packages (such as autoconf or gcc).

The attribute patchelfTarball is defined as the result of a call to the function makeSourceTarball, which applies Autoconf and Automake to build a source distribution (in .tar.bz2 format) in the Nix store. The attribute patchelfNixBuild is a function that builds the source distribution patchelfTarball on a platform defined implicitly by the function argument pkgs, which contains the dependencies of the build, such as the C compiler to be used. For instance, if called with the argument pkgsx86_64Linux, the build is performed on a 64-bit x86 Linux machine. The argument doCoverageAnalysis determines whether a normal build is done, or a build that uses lcov to generate a code coverage report.

The function nixBuild performs a "typical" Nix build, with the dependencies stored in the Nix store. For instance, the C compiler gcc would reside in a path such as /nix/store/13b5y3n2jfzx...-gcc-4.2.3/bin/gcc rather than /usr/bin/gcc. For portability testing, this is a problem: while it tests compatibility with platform characteristics such as endianness, it doesn't build in the "native" way for that platform, i.e., using tools in the conventional locations for that platform. Thus, there are also functions such as rpmBuild and debianBuild that perform a build action in a virtual machine instantiated and started on the fly (for Linux distributions based on the Red Hat Package Manager (Foster-Johnson 2003) and on the Debian Packager, respectively). In the virtual machine, the package is compiled and installed in the "native" way, e.g. in /usr/bin/patchelf, and a binary package for that particular platform is generated.

Finally, the attribute patchelfRelease uses the function makeReleasePage to generate a web page containing the results of the build jobs. A release page produced by patchelfRelease is shown in Figure 2.

The importance of a functional language for describing build jobs is that it makes it easy to build a package in various variants. For instance, we may want to test the

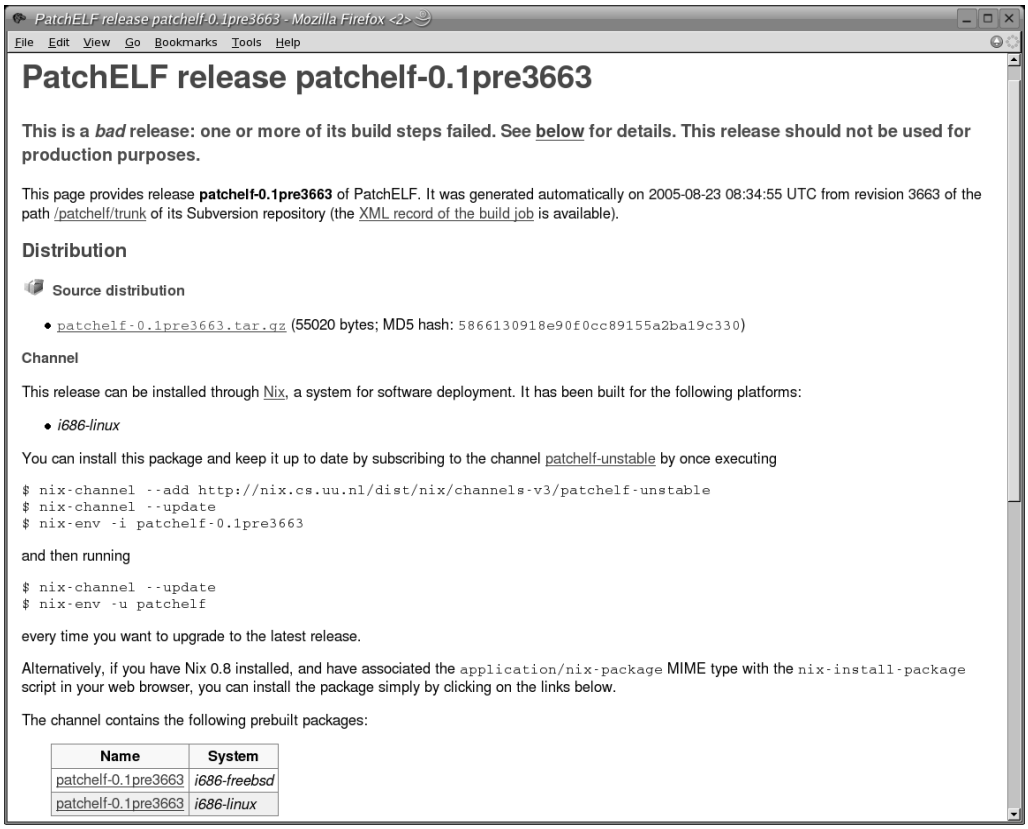

Fig. 2. Release page for PatchELF

package with several versions of the GNU C compiler. The package stdenv, which provides a standard Unix build environment and which patchelfNixBuild uses, contains a default version of GCC, but this can be overriden. For instance, we can add a function argument gcc to patchelfNixBuild, allowing specific versions of GCC to be passed in:

```
patchelfNixBuild = doCoverageAnalysis: pkgs: gcc: nixBuild {
  src = patchelfTarball;
  stdenv = overrideGCC pkgs.stdenv gcc;
  ...
};
```

We can then call this function with the desired versions of GCC:

```
nodistBuilds = [
  (patchelfNixBuild false pkgsi686Linux pkgsi686Linux.gcc295)
  (patchelfNixBuild false pkgsi686Linux pkgsi686Linux.gcc34)
  (patchelfNixBuild false pkgsi686Linux pkgsi686Linux.gcc42)
  (patchelfNixBuild false pkgsi686Linux pkgsi686Linux.gcc43)
];
```

Of course, if we also want to do these builds on x86_64, we can turn this expression into a function over pkgs, and call it once for each platform. Another example of the usefulness of the purely functional approach is the construction of virtual machine disk

images. For instance, the disk image in the value vmTools.diskImages.fedora8i386 is built by a function that creates a disk image from a subset of RPM packages that constitute an RPM-based Linux distribution. This image doesn't contain all RPMs; for instance, it doesn't contain the readline package. If we need this for our build, we can easily synthesise a new VM disk image that does contain it:

```
fedora8i386Extra = vm.diskImageFuns.fedora8i386 {
  packages = fedoraPackages ++ ["readline"];
};
...
rpmBuilds = [
  (patchelfRPMBuild fedora8i386Extra)
  ...
];
```

Thus, virtual machine images are not created manually, as is typical, but are instead automatically instantiated from a declarative specification.

The declarative specification of the PatchELF build farm job in Figure 1 abstracts over physical machines and platform types. That is, it doesn't mention on what particular machines specific build actions must be performed. Rather, Nix takes care of automatically distributing build actions to machines of the appropriate type (such as x86_64-linux), performing actions in parallel on multiple machines if possible. All function inputs (dependencies) are automatically copied to the machine on which the function is executed. For instance, the various calls to rpmBuild will be distributed over multiple machines, and the dependencies (such as the virtual machine software, the disk image, and the source distribution) will be copied automatically. No system administrator action is necessary to install dependencies on the appropriate machines.

3. Experience

The Nix build farm has been in use for a number of years by a variety of projects. It is used to build NixOS, a Linux distribution with a purely functional configuration model built on top of the Nix package manager, and the Nix Packages collection (Nixpkgs), which alone consists of over 600 package builds (see `http://nixos.org/releases`). Another major use is to build and release the Stratego/XT program transformation toolset (Visser 2004) and its ecosystem of related projects (see `http://strategoxt.org/releases`). Nixpkgs contains Nix expressions for many common dependencies such as compilers and runtime environments for many languages, making it easier to add jobs to the build farm.

Nix itself is written in C++, using the ATerm term manipulation library (van den Brand et al. 2000) to concisely implement the language evaluation machinery using term rewriting (Dolstra 2008). One might ask whether the Nix expression language could not be implemented as an embedded DSL in a language such as Haskell. While this would have definite advantages, such as the availability of the libraries of such a host language, it has the significant downside of creating a dependency on a large piece of software such as GHC. A deployment tool should itself be easy to deploy, which is not the case if it has large external dependencies. Second, embedded DSLs make it harder to provide syntax convenient to the domain at hand.

4. Future work

The main focus of future research is automatic testing in large configuration spaces. When testing a component with a large amount of variabilities, the build farm should automatically select interesting configurations in order to maximize the amount of useful knowledge that it produces for the developers. For instance, if a certain configuration succeeds and another does not, the build farm should explore the configuration space, building different configurations, to discover which parameter causes the failure. Similarly, if a certain configuration fails while it did not previously, the build farm should try to isolate the specific commit that introduced the fault.

Another interesting direction is to discover possibly troublesome configurations using source code analysis. For instance, nested #ifdefs in C programs conditional on options in the configuration space may indicate a potential feature interaction that must be tested specifically.

5. Conclusion

A build farm is an indispensable tool in any software development project, but existing implementations have serious limitations: they are hard to maintain because the build environment is not managed, have poor reproducibility, and have no explicit support for building variants of systems. The Nix-based build farm, by virtue of its purely functional component description language, solves these issues.

With respect to tool development, the Nix build farm supports the development of other tools in several ways. First, continuous integration improves the quality of tools built in the build farm, just as with any other package. Second, the continuous release of source and binary packages by the build farm improves dissemination of the tool, and thus the underlying research results. This should help to prevent the common problem of interesting tools fading into obscurity because they are not available in an easily installable form. Finally, the build farm serves as a test bed for analysis tools (such as dynamic tools like coverage analysers or static tools like FindBugs (Hovemeyer and Pugh 2004)), which can be plugged into the build farm and applied to the large "corpus" of software built by it.

Acknowledgements. The authors would like to thank the anonymous reviewers for their comments. This research was supported by the NIRICT LaQuSo Build Farm project.

References

Apache Software Foundation, 2005. Apache Ant. `http://ant.apache.org/`, accessed 15 August 2005.

Dolstra, E., Jan. 2006. The purely functional software deployment model. Ph.D. thesis, Faculty of Science, Utrecht University, The Netherlands, `http://www.cs.uu.nl/~eelco/pubs/phd-thesis.pdf`.

Dolstra, E., Apr. 2008. Maximal laziness — an efficient interpretation technique for purely functional DSLs. In: Eighth Workshop on Language Descriptions, Tools and

Applications (LDTA 2008). Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, to appear.

Dolstra, E., de Jonge, M., Visser, E., Nov. 2004a. Nix: A safe and policy-free system for software deployment. In: Damon, L. (Ed.), 18th Large Installation System Administration Conference (LISA '04). USENIX, Atlanta, Georgia, USA, pp. 79–92.

Dolstra, E., Visser, E., de Jonge, M., May 2004b. Imposing a memory management discipline on software deployment. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004). IEEE Computer Society, pp. 583–592.

Feldman, S. I., 1979. Make—a program for maintaining computer programs. Software—Practice and Experience 9 (4), 255–65.

Foster-Johnson, E., 2003. Red Hat RPM Guide. John Wiley & Sons, also at `http://fedora.redhat.com/docs/drafts/rpm-guide-en/`.

Fowler, M., Foemmel, M., 2006. Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`.

Hemel, A., Aug. 2003. Using buildfarms to improve code. In: UKUUG Linux 2003 Conference.

Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. SIGPLAN Notices 39 (12), 92–106.

Mozilla Foundation, 2005. Tinderbox. `http://www.mozilla.org/tinderbox.html`.

Nix project, 2008. Nix homepage. `http://nixos.org/`.

ThoughtWorks, 2005. Cruise Control. `http://cruisecontrol.sourceforge.net/`.

Urbancode, 2005. Anthill. `http://www.urbancode.com/projects/anthill/default.jsp`, accessed 21 August 2005.

van den Brand, M. G. J., de Jong, H. A., Klint, P., Olivier, P. A., 2000. Efficient annotated terms. Software—Practice and Experience 30, 259–291.

van der Storm, T., Sep. 2005. Continuous release and upgrade of component-based software. In: 12th International Workshop on Software Configuration Management (SCM-12).

Visser, E., Jun. 2004. Program transformation with Stratego/XT: Strategies, tools, and systems in StrategoXT-0.9. In: Lengauer, C., et al. (Eds.), Domain-Specific Program Generation. Vol. 3016 of Lecture Notes in Computer Science. Spinger-Verlag, pp. 216–238.