Hydra: A Declarative Approach to Continuous Integration[1]

Eelco Dolstra, Eelco Visser

Department of Software Technology,
Faculty of Electrical Engineering, Mathematics and Computer Science (EWI),
Delft University of Technology, The Netherlands

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. In this paper we present *Hydra*, a continuous build tool based on *Nix*, a package manager that 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 so significantly reduces the effort to maintain a continuous integration environment.

1. Introduction

Hydra is a tool for continuous integration testing and software release that uses a purely functional language to describe build jobs and their dependencies. Continuous integration (Fowler and Foemmel 2006) 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.

[1] Note to reviewers: the software system and associated materials described in this paper are available at `http://nixos.org/hydra-scp`.

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 a 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. A continuous integration system, on the other hand, can afford 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 provide a clean, well-defined build environment. If this environment is administered through proper software configuration management (SCM) techniques, then builds produced by the system can be reproduced. By 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*).
- It allows software to be *released* by automatically creating packages that users can download and install. To do this manually represents an often prohibitive amount of work, as one may want to produce releases for many different platforms: e.g., installers for 32-bit and 64-bit Windows and Mac OS X, RPM or Debian packages for certain Linux distributions, and so on.

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:

1. It obtains the latest version of the component's source code from the version management system.
2. It runs the component's build process (which presumably includes the execution of the component's test set).
3. It presents the results of the build (such as error logs and releases) 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 (Warner 2008). 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 even 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 3,400 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 problems, 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.

In this article we describe *Hydra*, a continuous integration tool that solves these problems. 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; and as such is an ideal fit for a continuous build system.

We give an overview of Nix and its build language in Section 2. Then, in Section 3, we describe Hydra's architecture and show how the Nix expression language can be used to declare jobs for a continuous build system. Section 4 shows how software releases can be derived on the fly as dynamic views on the set of finished builds. We discuss our experience with Hydra and its predecessors in Section 5. Related work is discussed in Section 6, and directions for future research in Section 7.

2. The Nix Package Manager

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 furthermore 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/fhmqjrs5sj7b477f8wnlf2588s2jdccb-firefox-3.0.4/
```

where fhmqjrs5sj7b... 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 build actions have the same inputs, then only one build will be performed. 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 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 introduces the essential concepts of the Nix expression language. It shows an example of a specification that builds Bison, a parser generator, along with its dependencies. Bison has numerous dependencies, such as the C compiler, the C library, the shell, standard Unix tools such as sed and grep, the Make tool, and the M4 macro processor. All of these have to be present in order to build the Bison package. At top-level, the expression in Figure 1 is an attribute set, a set of name/value pairs that can recursively reference each other (rec{...} at point 1). For instance, the attribute bison has the value stdenv.mkDerivation {...} 2.

The most important value in the Nix expression language is the *derivation*, which is an (as far as Nix is concerned) atomic build action that produces a single path in the Nix store. Derivations can have dependencies on other derivations, which will be built prior to building the referring derivation. Thus, derivation values describe a dependency graph of build actions, similar to a Makefile. For instance, if we issue the command

```
$ nix-build example.nix -A bison
```

then Nix will evaluate the attribute bison in Figure 1, and then perform all derivations in the dependency graph of bison's top-level derivation by performing the associated build actions.

For instance, Bison has a dependency on GNU M4 (due to the line inherit gnum4 3, which causes the gnum4 value in the surrounding lexical scope to be passed as an argument to the derivation). Thus, Nix will first build the gnum4 value 4 (after building *its* dependencies, of course), which might produce a path such as /nix/store/q70vsdncz0mz...-gnum4-1.4.9 in the Nix store. It will then compute a target path for the Bison package by hashing all its inputs, yielding e.g. /nix/store/m1n9g183gyxc...-bison-2.4. Finally, it will perform the build by executing the derivation's associated build action, which can be an arbitrary shell script. For instance, the sequence in Bison's buildCommand argument performs a typical Unix package build: it unpacks the sources, runs Bison's configure script to generate some Makefiles, and then runs Make to build and install the package. Nix passes the intended output location in the store through the environment variable out. All arguments of the derivation are likewise passed through environment variables. For instance, the environment variables gnum4 and src will hold the Nix store paths containing the M4 package and the downloaded source code of the package, respectively. Figure 2 shows parts of the Nix store following the building of the bison attribute in Figure 1.

```
rec { 1

  bison = stdenv.mkDerivation { 2
    name = "bison-2.4";

    src = fetchurl {
      url = mirror://gnu/bison/bison-2.4.tar.bz2;
      sha256 = "0c9sv03wsqnqc7wfpa51yc9yy1i3kdgsrjg7qchx0sk8zr11cvqf";
    };

    inherit gnum4; 3

    buildCommand = ''
      tar xf $src
      cd bison-*
      export PATH=$gnum4/bin:$PATH
      ./configure --prefix=$out
      make
      make install
    '';
  };

  gnum4 = stdenv.mkDerivation { 4
    name = "gnum4-1.4.9";
    ...
  };

  stdenv = {
    mkDerivation = args: derivation { ... gcc ... };
  };

  gcc = derivation { ... };
}
```

Figure 1: example.nix: Nix expression for building Bison and some of its dependencies

The Nix expression language provides many features, such as functional abstraction, to allow packages to be described conveniently. For instance, the expression for Bison makes no explicit reference to its dependencies on the C compiler, Make, and so on. These are all common dependencies shared by almost all Unix packages, and it would be tedious to be explicit about them. Rather, there is a function stdenv.mkDerivation that wraps the built-in primitive function derivation to set up an environment that already contains those common packages. (Nix functions have the syntax *args*: *body*.)

As a package manager, Nix is used to deploy software to end-user machines. For instance, a user would typically issue a command like nix-env -i bison, which would make the path /nix/store/m1n9g183gyxc...-bison-2.4/bin appear automatically in the user's search path. Nix has many capabilities that exceed conventional package managers such as RPM (Foster-Johnson 2003): in addition to the ability to have multiple versions of a package installed at the same

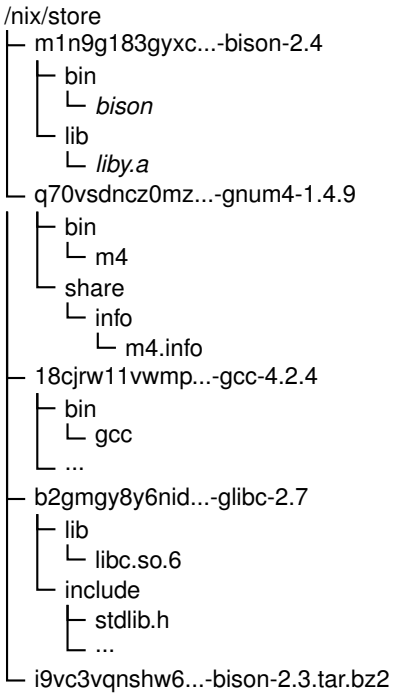


Figure 2: The Nix store after building Bison

time, the immutable nature of objects in the Nix store allows packages to be upgraded or rolled back atomically; they can be garbage-collected automatically when they are not used; Nix allows different users to have different views on the set of installed packages; and so on.

However, for the purpose of this paper, we are interested in Nix not so much as an end-user deployment tool, but as a high-level Make-like build tool for describing the build actions we want to perform in a continuous build system. For example, the Nix expression in Figure 1 could be submitted to a Nix-based continuous build tool, which would call Nix to build the package and all its dependencies, and make the results available to developers and end users. That's precisely what Hydra accomplishes.

3. Hydra

Hydra is a continuous build system that checks out the sources of software projects from version management repositories, uses Nix to build those projects, and makes the results available through a web interface.

Each project must provide a Nix expression that describes precisely how to build the various artifacts that one wants to build and release. As an example, we show how a simple Unix package, *PatchELF*, can be tested and released using Hydra. (PatchELF is a utility for performing certain modifications on ELF executables[2].) Testing and releasing this package involves a number of

[2] `http://nixos.org/patchelf.html`

actions:

- The sources from the version management repository (in this case a Subversion repository) must be turned into a proper *source distribution* (or "tarball" in Unix parlance) 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. Installable packages, such as RPMs, for these platforms must also be generated.
- A code coverage analysis is also performed by building an instrumented binary, running the test suite and generating a coverage report.

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 Hydra 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.

Figure 3 shows part of the Nix expression for building PatchELF in Hydra[3]. Hydra requires that the expression for a project evaluates to an attribute set, where each attribute denotes a *job* within the project. Thus, the expression in Figure 3 defines two jobs: tarball (for building the source distribution from the Subversion sources), and build (for performing a build, check and installation of a source distribution, i.e., essentially doing ./configure && make && make check && make install).

The value of each attribute is a *function* from certain inputs that will be supplied by Hydra, to a derivation that Hydra will then build. For instance, the attribute tarball [5] is a function that takes three arguments[6]: patchelfSrc (the pristine sources retrieved from the Subversion repository), nixpkgs (a copy of the Nix Packages collection (Nixpkgs), a set of Nix expressions for almost 1800 packages, containing the dependencies that we need for building and testing PatchELF) and officialRelease (a Boolean argument that specifies whether this is an "official" release or just a test release, discussed further in Section 4). The language construct with e_1; e_2 evaluates the expression e_1, which should yield an attribute set, and adds the attributes to the lexical scope of the expression e_2. For instance, the with at [7] makes the packages and support functions defined in Nixpkgs available in the body of the tarball function. Among these is releaseTools.makeSourceTarball, a useful function that runs Autoconf and Automake on the sources in the src argument, and then runs make dist to produce a source distribution which is placed in the Nix store. Likewise, releaseTools.nixBuild builds a package; the main difference with stdenv.mkDerivation is that it also runs make check to perform any tests provided by the package.

After writing the Nix expression, we make Hydra build it by adding the project through Hydra's web interface. This entails creating a project and defining within the project a so-called *jobset*: a specification of the location of a Nix expression, along with possible values for the function arguments of the jobs defined by the Nix expression. Figure 4 shows Hydra's overview

[3]The full Nix expression for PatchELF can be found at `https://svn.nixos.org/repos/nix/patchelf/trunk/release.nix`.

```
{
  tarball = 5
    { patchelfSrc, nixpkgs, officialRelease }: 6

    with import nixpkgs.path {}; 7

    releaseTools.makeSourceTarball {
      name = "patchelf-tarball";
      src = patchelfSrc;
      inherit officialRelease;
    };

  build =
    { tarball, nixpkgs, system }: 8

    with import nixpkgs.path {inherit system;};

    releaseTools.nixBuild {
      name = "patchelf-build";
      src = tarball;
      postInstall = ''
        echo "doc readme $out/share/doc/patchelf/README" \
          >> $out/nix-support/hydra-build-products
      '';
    };
}
```

Figure 3: release.nix: Nix expression for the PatchELF project

page for the PatchELF project, which displays some statistics about the project, as well as its jobsets. In this case there is only one jobset: one that builds jobs from the "trunk" of PatchELF's Subversion repository. The inputs are shown at the bottom. For instance, the patchelfSrc argument required by the tarball function is defined as a Subversion checkout of the repository at https://svn.nixos.org/repos/nix/patchelf/trunk. Likewise, officialRelease is defined as a Boolean with value false. Also, the jobset specifies that the Nix expression containing the job can be found in the file release.nix in the input patchelfSrc (i.e., in the PatchELF Subversion repository).

Hydra sits in a loop continuously fetching the Nix expression for each project's jobsets, and evaluating the jobs defined therein by calling each job function with the required arguments. For instance, the function tarball in Figure 3 might be called with arguments such as officialRelease = false, patchelfSrc = /nix/store/hfk7bx01c20z...-svn-export (the latter being the location in the Nix store to which Hydra checked out the patchelfSrc Subversion input), and nixpkgs = /nix/store/gvsxbi5vk9s6...-svn-export (likewise, the checkout of the nixpkgs input). With these arguments, the tarball function returns a derivation of a Nix store path such as /nix/store/pxyxajam2wq7...-patchelf-tarball. Hydra then consults its database to see if it has built this path before. If not, the derivation is added to Hydra's queue of pending builds. Eventually, the derivation is retrieved from the queue and executed by calling Nix.

When the build finishes, a registration of the success or failure is made in Hydra's database, which can be viewed through its web interface. Users can also be notified through email of build

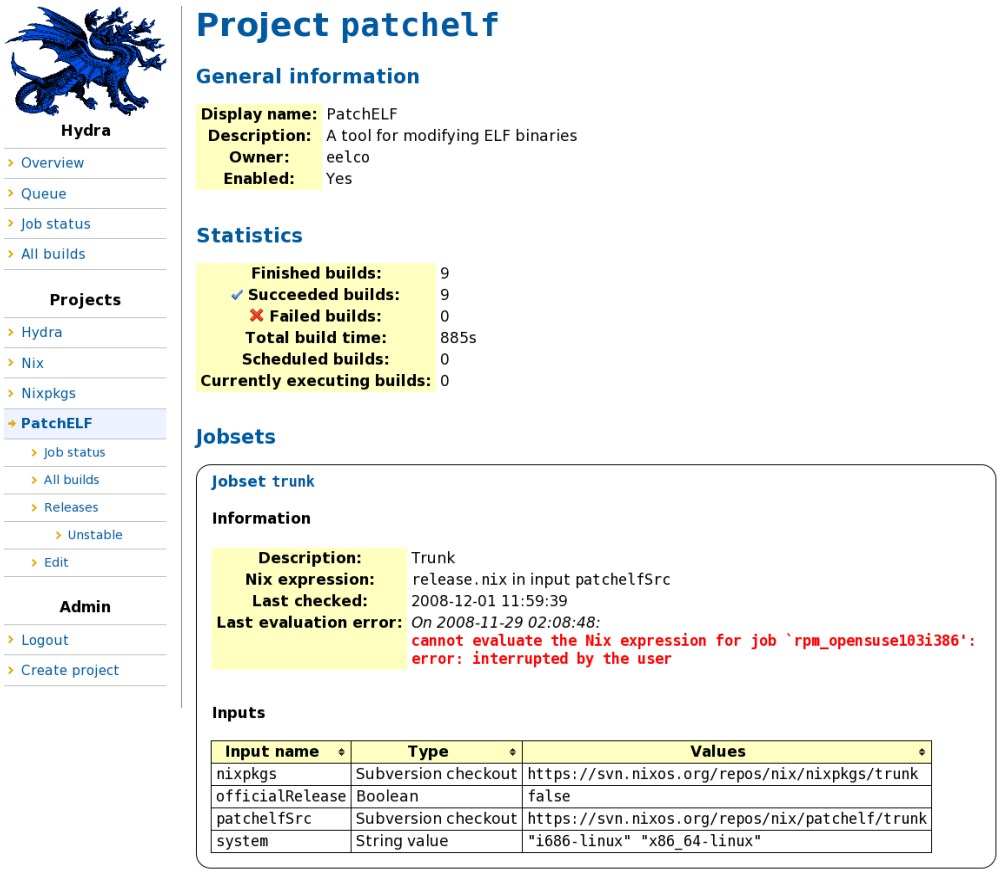
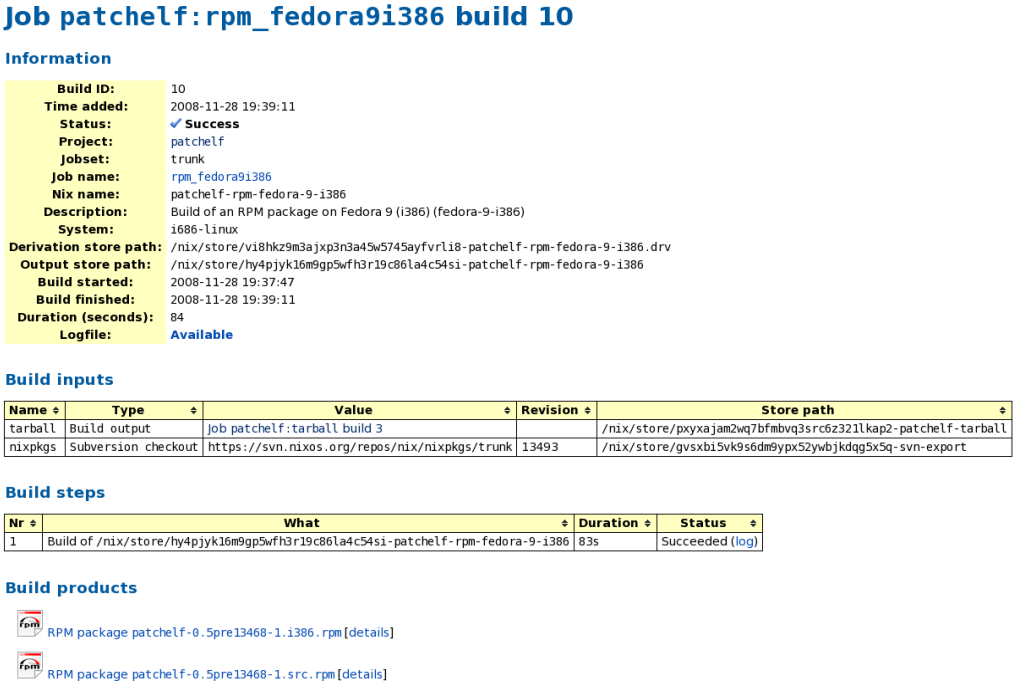


Figure 4: Hydra overview of the PatchELF project, with the definition of its inputs

failures. Figure 5 shows the result page of a build. It lists various statistics about the build, its inputs, and the "products" of the build, such as downloadable packages. It also provides access to the build logs. For instance, Figure 6 shows the build log for a failed build of the build job.

Jobs can have dependencies on the output of other jobs. For instance, the build job in Figure 3 depends on the output of the tarball job (because it compiles, tests and installs the source distribution created by the tarball job). Such a dependency is specified by giving the function for a job a formal parameter with a name equal to that of another job. Thus, the build function (at point 8) has a function argument named tarball. When Hydra sees such a function argument, it will look up the latest successful build of the named job in its database and pass it to the function. For example, build might be called with the argument tarball = /nix/store/pxyxajam2wq7...-patchelf-tarball. This enables very flexible integration testing: Hydra can easily test many different combinations of versions of packages against each other.

Architecture. Hydra consists of several parts. First, it uses the *Nix store* to store and cache builds, and Nix to perform build actions (derivations) in the store.



Job patchelf:rpm_fedora9i386 build 10

Information

Build ID:	10
Time added:	2008-11-28 19:39:11
Status:	✔ **Success**
Project:	patchelf
Jobset:	trunk
Job name:	rpm_fedora9i386
Nix name:	patchelf-rpm-fedora-9-i386
Description:	Build of an RPM package on Fedora 9 (i386) (fedora-9-i386)
System:	i686-linux
Derivation store path:	/nix/store/vi8hkz9m3ajxp3n3a45w5745ayfvrli8-patchelf-rpm-fedora-9-i386.drv
Output store path:	/nix/store/hy4pjyk16m9gp5wfh3r19c86la4c54si-patchelf-rpm-fedora-9-i386
Build started:	2008-11-28 19:37:47
Build finished:	2008-11-28 19:39:11
Duration (seconds):	84
Logfile:	**Available**

Build inputs

Name	Type	Value	Revision	Store path
tarball	Build output	Job patchelf:tarball build 3		/nix/store/pxyxajam2wq7bfmbvq3src6z321lkap2-patchelf-tarball
nixpkgs	Subversion checkout	https://svn.nixos.org/repos/nix/nixpkgs/trunk	13493	/nix/store/gvsxbi5vk9s6dm9ypx52ywbjkdqg5x5q-svn-export

Build steps

Nr	What	Duration	Status
1	Build of /nix/store/hy4pjyk16m9gp5wfh3r19c86la4c54si-patchelf-rpm-fedora-9-i386	83s	Succeeded (log)

Build products

RPM package patchelf-0.5pre13468-1.i386.rpm [details]

RPM package patchelf-0.5pre13468-1.src.rpm [details]

Figure 5: A build of one of the PatchELF jobs: a Fedora 9 RPM

Second, it has a *database* to store information about projects, jobsets and inputs; the queue of scheduled builds; and finished builds, including a record of each build's downloadable "products" and log files. (The database is currently an embedded SQLite database. This simplify Hydra's deployment; no database servers and accounts need to be configured.).

The third component is the *scheduler*, which continuously, for each jobset, checks out the inputs of the jobset, calls all functions in the Nix expression with the appropriate arguments, and adds the resulting derivations to the build queue in the database if they haven't been performed before.

The fourth component is the *queue runner*, which monitors the build queue filled by the scheduler. For each platform type (e.g. i686-linux), it selects the highest-priority pending builds from the queue up to a configurable per-platform maximum, and executes a separate process to build them. For instance, if we have four quad-core machines of type i686-linux available, we would set the limit for concurrent i686-linux builds to 16, allowing that many jobs to be executed in parallel. Each build's concurrently executing build process calls Nix to build the derivation, and then registers the success or failure of the build in the database.

The final component is the *web interface*, which allows users to browse builds, download build products, view the queue, define and manage projects, and so on. The web interface also produces "releases" of packages — sets of related builds for some revision of the project — dynamically as views on the database, as discussed in Section 4.

Distributed builds. An input of a jobset can have multiple possible values. For instance, Figure 4 shows that the system string argument (used by the build job) can take the values i686-linux and

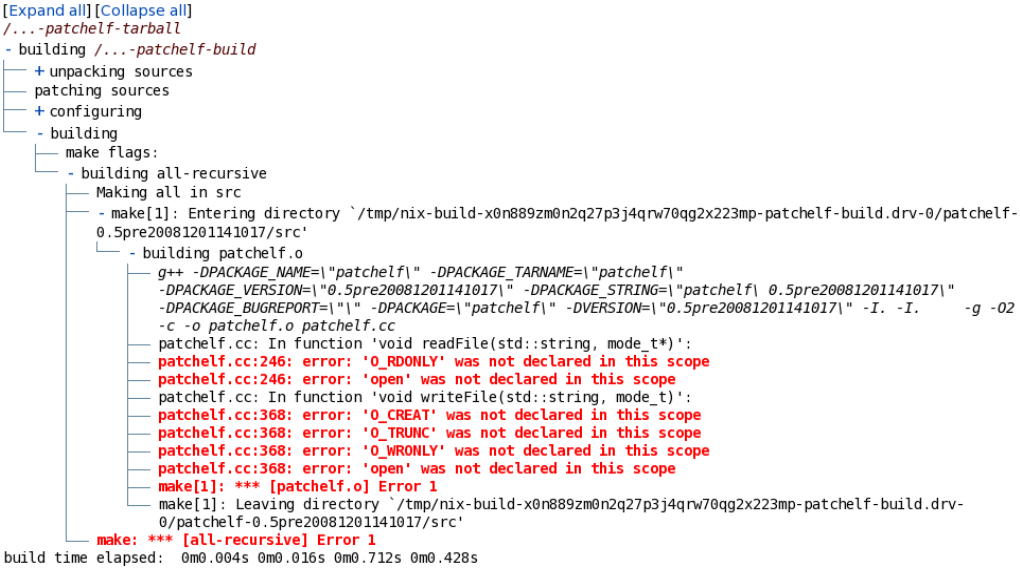


Figure 6: The build log for a failed build of PatchELF's build job.

x86_64-linux. The current default policy of the Hydra scheduler is to simply try all possible combinations of variant argument values. Thus, it will add two builds of the job build to the queue when a commit in the patchelf repository occurs: one with system = "i686-linux", and one with system = "x86_64-linux".

The system argument demonstrates Nix's support for distributed multi-platform builds, which Hydra inherits. Each derivation has an attribute called system, which defines the type of platform on which the build is to be performed, e.g. x86_64-linux (for a build on the Linux operating system on a 64-bit Intel/AMD CPU) or powerpc-darwin (for a Mac OS X build on a PowerPC CPU). If Nix is asked to build a derivation for a platform not equal to that of the platform it's running on, Nix can automatically forward the derivation to a remote machine of the appropriate type, e.g. by copying all the dependencies of the build to the remote machine via the ssh protocol, running Nix on the remote machine to perform the build, and finally copying the output path from the remote Nix store to the local store. This is also possible when the derivation's platform type does match the local platform type, allowing distributed, parallel builds. Thus, Nix expressions abstract over the physical machines necessary to perform a build: they just declare the required platform type, and Nix takes care of forwarding them to the appropriate machines.

Build products. In addition to the jobs shown in Figure 3, there are many other artifacts that we can build to test and release PatchELF. For instance, Nixpkgs provides a function releaseTools.coverageAnalysis, which, like releaseTools.nixBuild, compiles and runs the tests of a Unix package. However, unlike releaseTools.nixBuild, it does not install the package into the Nix store; rather, it compiles the package with coverage instrumentation, and uses the coverage data resulting from running make check to generate a coverage report in HTML format in $out/coverage/.

It is easy to define functions such as releaseTools.coverageAnalysis, since Nix doesn't care what output a derivation produces; the only requirement is that it creates the store path $out. Thus, any kind of build product — whether it's a "native" build such as performed by release-

Tools.nixBuild, an RPM package build, a user manual, or a static or dynamic analysis report — can be accommodated. However, a mechanism is required to "publish" such build results in Hydra, e.g., allow them to be viewed or downloaded through Hydra's web interface.

The interface that Hydra uses to accomplish this is fairly simple: derivations declare a build product by writing the type of the product and its full path to $out/nix-support/hydra-build-products. For instance, the build script for releaseTools.coverageAnalysis ends with the following lines, after generating the coverage report in $out/coverage/:

```
echo "report coverage $out/coverage" \
  >> $out/nix-support/hydra-build-products
```

which declares a product of type report and subtype coverage. After a successful build, Hydra reads the build's hydra-build-products file and registers the listed build products in the database. The web interface then provides appropriate access to the products. For instance, products of type file are offered for download, while products of type doc or report are browseable. (See e.g. Figure 8.) It's up to the web interface to present the product in a nice way, e.g. by showing a Debian icon for a Debian package. Thus, presentation aspects are kept separate from the build jobs.

Declarative virtual machines. An interesting aspect of Hydra is the construction of RPM and Deb packages. RPM packages are used by Linux distributions such as Red Hat Enterprise Linux, Fedora and openSUSE, while Deb packages are used by Debian and Ubuntu. To build packages for such distributions, we need to perform the build on a machine containing the appropriate Linux distribution, e.g. Fedora 10. It is, of course, inconvenient to have a large number of such machines. Instead, there is a function releaseTools.rpmBuild that performs the build inside a virtual machine, given a disk image:

```
rpm_fedora10i386 =
  { tarball, nixpkgs }:

  with import nixpkgs.path {};

  releaseTools.rpmBuild rec {
    name = "patchelf-rpm-fedora-10-i386";
    src = tarball;
    diskImage = vmTools.diskImages.fedora10i386;
  };
```

The rpmBuild function performs the build using KVM[4], a virtualisation system for Linux. That is, its build script dynamically starts a virtual machine with the given virtual disk, performs a build of an RPM package inside the VM, copies the resulting package to $out, and stops the VM. The interesting aspect is that vmTools.diskImages.fedora10i386 is a derivation that builds the disk image for the Linux distribution *from a declarative specification* of the packages that we want in the disk image. It is the result of a call to the function makeImageFromRPMDist, which, given a manifest of all the RPM packages provided by an RPM-based Linux distribution, as well as the names of a set of top-level packages, returns a disk image containing the closure of those

[4] http://kvm.qumranet.com/

top-level packages (i.e. the packages and their dependencies). Likewise, there are functions that accomplish the same for Deb-based Linux distributions.

This approach makes it very simple to specify different disk images in which the build should take place. For instance, the default image vmTools.diskImages.fedora10i386 contains only a small set of basic packages, similar to Nix's stdenv. We can, however, easily specify that we want additional packages from the distribution:

```
diskImage =
  vmTools.diskImageFuns.fedora10i386 {
    packages = fedoraPackages ++ ["readline" "firefox"];
  };
```

The evaluation of the diskImage attribute will produce a derivation that downloads all required RPMs from Fedora 10 (including Readline, Firefox and their dependencies) and generates a disk image containing those RPMs. Thus, virtual machine images are not created manually, as is typical, but are instead automatically instantiated from a declarative specification.

As a final example of the benefits of a functional build language, note that we can abstract the rpm_fedora10i386 job into a function that builds PatchELF RPMs for a whole set of RPM-based distributions:

```
let

  makeRPM =
    diskImageFun: 9
    { tarball, nixpkgs }: 10

    with import nixpkgs.path {};

    releaseTools.rpmBuild rec {
      name = "patchelf-rpm-${diskImage.name}";
      src = tarball;
      diskImage = diskImageFun vmTools.diskImages;
    };

in {
  ... other jobs ...
  rpm_fedora5i386 = makeRPM (diskImages: diskImages.fedora5i386);
  rpm_fedora9i386 = makeRPM (diskImages: diskImages.fedora9i386);
  rpm_fedora10i386 = makeRPM (diskImages: diskImages.fedora10i386);
  rpm_opensuse103i386 = makeRPM (diskImages: diskImages.opensuse103i386);
  ...
}
```

That is, we turn the original rpm_fedora10i386 into a function that takes as its first argument 9 a function that selects the desired disk image from vmTools.diskImages, and returns a Hydra job function 10: i.e. a function from the Hydra inputs tarball and nixpkgs to a derivation. (The first argument has to be a function returning a disk image, rather than a disk image directly, because vmTools.diskImages is not in scope at top-level in the Nix expression; it comes from Nixpkgs and appears in the scope of the makeRPM function due to the with construct.)

Release Set patchelf:unstable

[Edit]

	#	Release	Date	Source distribution	Build on i686-linux	Build on x86_64-linux	Coverage analysis	Debian 4.0 (i386)	Ubuntu 8.04 (i386)	Fedora 10 (i386)	Fedora 9 (i386)
✗	320	patchelf-0.5pre20081129011355	2008-11-30 19:53:20	✓	✗	✗	✗	✗	✗	✗	✗
✗	276	patchelf-0.5pre20081128155734	2008-11-28 18:13:39	✓	✗	✗	✗	✗	✗	✗	✗
✗	265	patchelf-0.5pre20081128155734	2008-11-28 17:09:30	✓	✗	✗	✗	✗	✗	✗	✗
✗	256	patchelf-0.5pre20081128142327	2008-11-28 15:34:28	✓	✗	✗	✗	✗	✗	✗	✗
✓	247	patchelf-0.5pre20081128122223	2008-11-28 13:24:59	✓	✓	✓	✓	✓	✓	✓	✓
✗	231	*No name*	2008-11-27 20:09:49	✗							
✓	222	patchelf-0.5pre20081127190807	2008-11-27 20:10:08	✓	✓	✓	✓	✓	✓	✓	✓
✓	213	patchelf-0.5pre20081127190702	2008-11-27 20:08:49	✓	✓	✓	✓	✓	✓	✓	✓
✓	201	patchelf-0.5pre20081127185906	2008-11-27 20:05:41	✓	✓	✓	✓	✓	✓	✓	✓
✓	193	patchelf-0.5pre20081126172420	2008-11-27 19:29:26	✓	✓	✓	✓	✓	✓	✓	✓
✓	185	patchelf-0.5pre20081126171013	2008-11-26 18:12:55	✓	✓	✓	✓	✓	✓	✓	✓
✓	177	patchelf-0.5pre20081126170445	2008-11-26 18:06:30	✓	✓	✓	✓	✓	✓	✓	✓
✓	148	patchelf-0.5pre20081126143451	2008-11-26 15:42:53	✓	✓	✓	✓	✓	✓	✓	✓
✓	140	patchelf-0.5pre20081126132553	2008-11-26 15:19:47	✓	✓	✓	✓	✓	✓	✓	✓
✗	127	patchelf-0.5pre20081126122812	2008-11-26 13:30:58	✓	✓	✗	✓	✓	✓	✓	✓
✗	122	patchelf-0.5-pre20081126121343	2008-11-26 13:30:11	✓	✓	✓	✓		✓	✗	✗
✗	112	patchelf-0.5-pre20081126005947	2008-11-26 13:19:21	✓	✓	✓	✓	✗	✗		✗
?	105	patchelf-0.5-pre20081125181215	2008-11-25 19:32:55	✓							

Figure 7: Synthesized releases of PatchELF

4. Release Management with Hydra

For each commit into a project's version management repository, Hydra executes the jobs defined in the project's Nix expression, resulting in separate build records in Hydra's database (e.g., the build of a Fedora 9 RPM shown in Figure 5). This is not necessarily convenient for users, who may wish to see all builds associated with a revision of the project together. Such a set of related builds is a *release* of the project. For example, Figure 8 shows a web page for PatchELF release patchelf-0.5pre13468 (i.e. the pre-release of version 0.5 built from Subversion revision 13468).

A precursor of Hydra, also based on Nix, had a *release-centric* model (Dolstra and Visser 2008). This means that the Nix expression for a project evaluates to a single top-level derivation, which produces an HTML page as output. The HTML page and the files linked from it are then uploaded to a web server and served as static files. The dependencies of the top-level derivation are derivations that build source distributions, RPM packages, and so on. However, several years of experience revealed several flaws in this model. First, the failure of one subderivation causes the entire release to fail. Thus, even relatively unimportant artifacts (say, an RPM package for some old, little-used Linux distribution) could potentially hold up new releases for a long time, unless the Nix expression for the project was modified. Second, since the subderivations for each artifact are not first-class entities that the continuous build system knows about, the user cannot get or set information about these subtasks, such as change the scheduling priority of Fedora 10 RPM builds. Finally, by statically generating an HTML page, the presentation of the release is statically fixed.

Instead, Hydra supports releases as a *dynamic view* on the database of builds. It allows a user to declaratively define, through the web interface, what builds together constitute a release. Indeed, it is possible to define several such views, called *release sets*. For instance, for PatchELF, we would define a release in the "unstable" release set as a build of the tarball job with the officialRelease argument set to false; plus a set of other builds that have that tarball build as input, such as builds of the the build job and the fedora10i386 job. Likewise, the "stable" release set is based on builds of the tarball job with officialRelease set to true. Incidentally, the makeSourceTarball function uses the officialRelease argument to determine the name of the release: if it's false, then the Subversion revision number is appended to the project's version (e.g. 0.5pre13468).

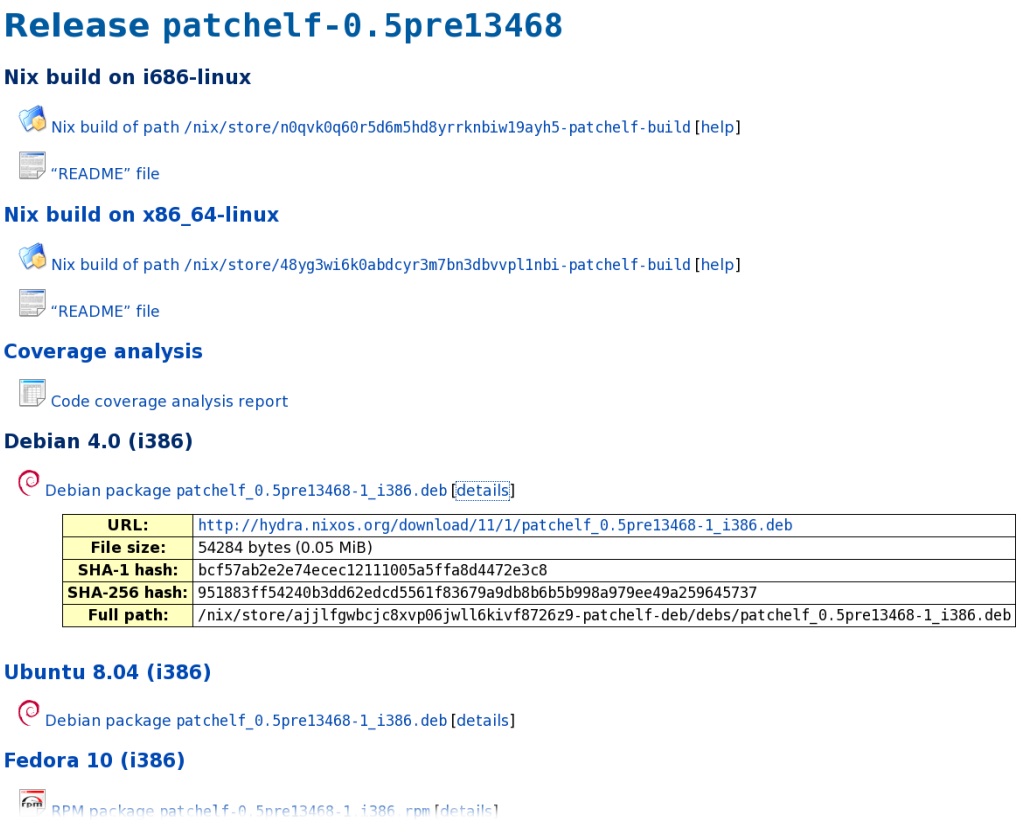


Figure 8: A synthesized release of PatchELF

Figure 7 shows the unstable release view on PatchELF. Releases are *successful* if there are successful builds for all constituent jobs; *incomplete* if there are no builds for some constituent jobs; and *failed* if there are only failed builds for some constituent jobs.

For end users, the dynamically generated page in Figure 8 is very convenient: it shows a specific release from the unstable release view, namely patchelf-0.5pre13468. All build products from the constituent builds are presented together, allowing users to easily download the approriate package for their platform.

Hydra allows an unstable release to be turned into a stable release by enabling a user to select the patchelf-0.5pre12345 tarball build and *re-schedule* it with the same inputs, except that officialRelease is changed to true. After the successful completion of this build, Hydra will then proceed to build the other PatchELF jobs that depend on the tarball job. This will eventually lead to a release in the stable release view named patchelf-0.5.

5. Experience

Hydra and previous incarnations of continuous build systems based on the Nix package manager have been in use since 2004 by a variety of projects. It is used to build NixOS (Dolstra and Löh 2008), a Linux distribution with a purely functional configuration model built on top of

Nix, and the Nix Packages collection (Nixpkgs), which alone represents over 600 binary package builds (see `http://nixos.org/releases`). Another major use is to build and release the Stratego/XT program transformation toolset (Bravenboer et al. 2008) and its ecosystem of related projects (see `http://strategoxt.org/releases`).

Nix is written in about 20 thousand lines of 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 (see e.g. (Sloane 2002)). 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.

Hydra itself is written in Perl and uses the Catalyst web application framework (Catalyst community 2008). Hydra leans heavily on functionality provided in the Nix Packages collection to allow build jobs to be written concisely and easily. As shown in Section 3, Nixpkgs contains Nix expressions for many common dependencies such as compilers and runtime environments for many languages, as well as specific support functions for release-related build and test actions. The Nix Packages collection has been in development since 2003, has had numerous contributors, and contains around 54 thousand lines of Nix expression code for nearly 1800 packages.

6. Related Work

Hydra's primary advantage over other continuous build systems is the use of a declarative language to specify the complete dependency graph of a build job, including all associated build actions. This makes the management of the build environment (i.e., dependencies) quite easy and scalable. This aspect is largely ignored by tools such as CruiseControl (ThoughtWorks 2005), Tinderbox (Mozilla Foundation 2005) and BuildBot (Warner 2008), which expect the environment to be managed by the machine's administrator. Anthill (Urbancode 2005) has the notion of "dependency groups" that allows an ordering between build jobs.

Many continuous build tools are language-specific (or at least language-biased), such as CruiseControl or CruiseControl.NET. Nix is completely language-agnostic: any build action can be accommodated, and conveniently automated using the appropriate function abstractions.

Most continuous integration tools are targeted at testing, not producing releases. Sisyphus (van der Storm 2005) on the other hand is a continuous integration system that is explicitly intended to support deployment of upgrades to clients. It uses a destructive update model, which makes it easy to use with existing deployment tools, but bars side-by-side versioning and rollbacks.

7. 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, Hydra 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, Hydra

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, Hydra 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.

8. Conclusion

A continuous build system is an indispensable tool in any software development project, but existing implementations have serious limitations: they place a heavy maintenance burden on users because the build environment is not managed, have poor reproducibility, and have no explicit support for building variants of systems. Hydra, by virtue of the purely functional component description language provided by the underlying Nix package manager, solves these issues.

The software described in this paper is available at `http://nixos.org/hydra-scp`. This includes the source code of Hydra, Nix and Nixpkgs, various binary distributions of Nix, a binary distribution of Hydra installable through Nix that includes Hydra's 120 runtime dependencies, the extensive Nix and Nixpkgs user manuals, and links to numerous examples of Hydra build jobs.

Acknowledgements. The authors would like to thank the anonymous reviewers of the WASDeTT workshop for their comments. This research was supported by the NIRICT LaQuSo Build Farm project. Martin Bravenboer, Armijn Hemel, Merijn de Jonge and Roy van den Broek contributed to the various incarnations of the Nix build farm. We also want to thank the many contributors to the Nix project.

References

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

Bravenboer, M., Kalleberg, K. T., Vermaas, R., Visser, E., June 2008. Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming 72 (1-2), 52–70, special issue on experimental software and toolkits.

Catalyst community, 2008. Catalyst — the elegant MVC framework. `http://www.catalystframework.org/`, accessed 1 December 2008.

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., Löh, A., Sep. 2008. NixOS: A purely functional Linux distribution. In: ICFP 2008: 13th ACM SIGPLAN Intl. Conf. on Functional Programming. ACM Press.

Dolstra, E., Visser, E., Jul. 2008. The Nix build farm: A declarative approach to continuous integration. In: Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008).
URL `http://smallwiki.unibe.ch/wasdett2008/`

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`.

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

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

Sloane, A. M., 2002. Post-design domain-specific language embedding: A case study in the software engineering domain. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02). IEEE Computer Society, Washington, DC, USA.

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).

Warner, B., 2008. BuildBot. `http://buildbot.net/`, accessed 1 December 2008.