Continuous Integration and Release Management
with Nix

Eelco Dolstra

Institute of Information & Computing Sciences
Utrecht University, The Netherlands

July 14, 2004
Continuous Integration

- When developing multiple components of a system in parallel, we must *integrate* them at some point.
- “Integrate often” is more effective than “Big-Bang Integration”.
- So compositions of components should be built as often as possible.
- ⇒ Requires a build farm.
Stratego/XT is a compiler/toolset for the Stratego program transformation language.

Tiger is a compiler for the Tiger language, written in Stratego.

The Stratego/XT developers want to know:
- Whether Stratego/XT builds on/in a variety of platforms / configurations.
  - E.g., GCC 3.3 vs. GCC 3.4, SDF 2.2 vs SDF 2.3
- Whether changes to the Stratego compiler breaks existing Stratego code.

The Tiger developers want to know:
- Whether Tiger builds on/in a variety of platforms / configurations.
- Whether Tiger is compatible with previous/current releases of Stratego/XT, and with the HEAD branch.
The advantages:

- Tiger acts as a real-world regression test for Stratego/XT.
- Stratego/XT developers get feedback about unintentional breakage in the HEAD.
- Tiger developers get early feedback about incompatible changes to Stratego/XT.
1. Continuous Integration
2. Release Management
3. Build Farms
4. The Solution: Nix
5. Implementing a Build Farm with Nix
6. Conclusion
We want to build releases of components...
... automatically, since many steps are involved, e.g.,
- Make sure that all tests succeed.
- Build a source distribution.
- Build binary distributions for a variety of platforms.
- Upload (publish) to a server.
- Announce the release on a mailing list.

⇒ Requires a build farm.
Outline

1. Continuous Integration
2. Release Management
3. Build Farms
4. The Solution: Nix
5. Implementing a Build Farm with Nix
6. Conclusion
Other Advantages of Build Farms

- Allows more tests than can feasibly be run by a developer prior to commit:
  - Build on multiple platforms.
  - Build multiple configurations / variants.
  - Run time-consuming test sets.

- Distribute pre-built components to other developers.
Current Build Farms

- Mozilla Tinderbox, Cruise Control, AutoBuild, ...
- Disadvantages:
  - Typically build single components, not compositions.
  - Do not manage dependencies:
    - E.g., “package X requires GCC 3.3, GTK 2.4, Bison 1.875c, ...
    - This is left to the sysadmin of the build farm machines.
    - But dependencies can conflict: “package Y does not build on GCC 3.3”.
    - And dependencies can evolve: “package X now requires GCC 3.4” \( \Rightarrow \) requires \( \Theta(n) \) sysadmin time.
Nix is a system for software deployment, i.e., for distributing software from the developer to the user.

Features:

- Correctness: complete deployment. No missing dependencies—Nix always deploys closures of components.
- Support for multiple versions/variants of components.
- Users can have different versions/variants installed at the same time.
- Automatic garbage collection of unused components.
- Atomic upgrades / rollbacks.
- Simple language for describing components and compositions.
- Transparent source/binary deployment.
  - Basic model is source deployment, with binary deployment as a transparent optimisation.
Nix Expressions

- Nix expressions describe how components and compositions can be built.
- Simple functional language.

Nix expression for Stratego/XT: `strategoxt.nix`

```nix
{stdenv, fetchurl, aterm, sdf}:
  stdenv.mkDerivation {
    name = "strategoxt-0.10";
    builder = ./builder.sh;
    src = fetchurl {
      url = ftp://.../strategoxt-0.10.tar.gz;
      md5 = "526a28e84248b649bb098b22d227cd26";
    };
    inherit aterm sdf;
  }
```
Nix Expressions

- Nix expressions describe how components and compositions can be built.
- Simple functional language.

Build script for Stratego/XT: builder.sh

tar xvfz $src
cd strategoxt-*
./configure --prefix=$out \
  --with-aterm=$aterm \
  --with-sdf=$sdf
make
make install
Nix Expressions

- Nix expressions describe how components and compositions can be built.
- Simple functional language.

Composition: composition.nix

```nix
let { 
    stdenv = import (...)/stdenv.nix; 
    aterm = (import (...)/aterm.nix) { inherit stdenv; }; 
    sdf = (import (...)/sdf.nix) { inherit stdenv; }; 
    strategoxt = (import ./strategoxt.nix) { 
        inherit stdenv aterm sdf; 
    }; 
    body = strategoxt; 
} 
```
To build and install Stratego/XT:

$ nix-env -if ./composition.nix strategoxt

When a new version comes along:

$ nix-env -uf ./composition.nix strategoxt

If it doesn’t work:

$ nix-env --rollback

Delete unused components:

$ nix-collect-garbage
User Operations (Channels)

Subscribe to a channel:

$ nix-channel --add http://.../nix-stable-pkgs

Update all packages to the latest versions:

$ nix-channel --upgrade
On the producer side:

$ nix-push ./composition.nix http://server/cache

On the client side:

$ nix-pull http://server/cache

Installation will now reuse pre-built components, *iff* they are exactly the same.
All packages are stored in isolation from each other in the file system (in subdirectories of the Nix store—typically /nix/store).

Names of component directories contain a cryptographic hash of all inputs involved in building the component:

- Input components (compilers, libraries, other tools).
- Sources.
- Build scripts.
- Variability parameters.
- System type on which the component is to be built.
Implementation

```
/nix/store
  eeeaf42e56b-subversion-0.32.1
    bin
    svn
    lib
      libsvn_wc.so
      libsvn_ra_dav.so
    a17fb5a6c48f-openssl-0.9.7c
      lib
        libssl.so.0.9.7
    8d013ea878d0-glibc-2.3.2
      lib
        libc.so.6
```
Outline

1. Continuous Integration
2. Release Management
3. Build Farms
4. The Solution: Nix
5. Implementing a Build Farm with Nix
6. Conclusion
So why is this useful for a build farm?

- The Nix expression language is ideal for describing the build tasks to be performed.
- The Nix expression language makes it easy to describe variant compositions.
- Nix manages the dependencies (and thus the build environment).
- Nix supports distributed builds in a semi-transparent way.
- The hashing scheme + complete dependencies allow builds to be reproduced reliably.
- Efficiency: due to the hashing scheme, we only rebuild things that have actually changed.
What each release should contain:

- A source distribution.
- Binary distributions for a number of platforms. (Test sets should also be run on each platform).
- Build logs.
Building a source distribution

# Bring in some standard packages (compilers, etc.)
pkgs = (import .../all-packages);
pkgsLinux = pkgs {system = "i686-linux"};

strategoxtTarball = revision: svnToSourceTarball revision {
    stdenv = pkgsLinux.stdenv;
    buildInputs = [pkgsLinux.autoconf pkgsLinux.automake ...];
};

svnToSourceTarball is a function that checks out sources from a specific revision from a Subversion repository (as specified by version).
Building a binary distribution for Linux

strategoxtBinary = revision: buildBinary
  (strategoxtTarball revision)
{
  stdenv = pkgsLinux.stdenv;
  withATerm = pkgsLinux.aterm;
  withSDF = pkgsLinux.sdf;
};

buildBinary performs a build of a source distribution.
Building a release page

```nix
strategoxtRelease = revision: makeReleasePage {
  stdenv = pkgsLinux.stdenv;
  sourceTarball = strategoxtTarball;
  binaries = [strategoxtBinary];
};
```

`makeReleasePage` creates a bunch of HTML and other files that should be uploaded to a server.
Instantiating

strategoxtHeadRelease = strategoxtRelease {
    url = https://svn.cs.uu.nl/repos/StrategoXT/trunk;
    rev = (HEAD revision);
};

strategoxt010Release = strategoxtRelease {
    url = https://svn.cs.uu.nl/repos/StrategoXT/tags/0.10;
    rev = 6812;
};
Nix expressions specify on what system a package is to be built.

derivation {
    name = "strategoxt-0.10";
    builder = ./builder.sh;
    system = "powerpc-darwin";
}

Normally, if we build this on (say) a i686-linux:

$ nix-env -i foo.nix ...
error: I am a ‘i686-linux’, but a ‘powerpc-darwin’ is required to build this.
Solution: we can configure Nix with a mapping from system types to machines (e.g., `powerpc-darwin \rightarrow bigmac.cs.uu.nl`).

If we then try to perform the build, Nix will:
- Send all build inputs to `bigmac.cs.uu.nl`.
- Run Nix on that machine to perform the build.
- Copy back the result.

Different subexpressions can require different system types (useful for build farms and cross-compilation).

Builds are performed in parallel.
Building for Multiple Platforms

pkgs = (import .../all-packages);
pkgsLinux = pkgs {system = "i686-linux"};
pkgsDarwin = pkgs {system = "powerpc-darwin"};

strategoxtBinary = pkgs: revision: buildBinary
    (strategoxtTarball revision)
{
    stdenv = pkgs.stdenv;
    withATerm = pkgs.aterm;
    withSDF = pkgs.sdf;
};

strategoxtBinaries = revision: [
    (strategoxtBinary pkgsLinux revision)
    (strategoxtBinary pkgsDarwin revision)
];
Making Tiger Releases

Building Tiger Binaries

tigerTarball = ...;

tigerBinary = revision: strategyxt: buildBinary
(tigerTarball revision)
{
    stdenv = pkgsLinux.stdenv;
    withStrategoXT = strategyxt;
};
Building Tiger Binaries

tigerHeadRelease = strategoxtRelease {
  url = https://svn.cs.uu.nl/repos/tiger/trunk;
  rev = (HEAD revision);
};

tiger12Release = strategoxtRelease {
  url = https://svn.cs.uu.nl/repos/tiger/tags/1.2;
  rev = ...;
};
Building Tiger Binaries

tigerBinaries = [
    (tigerBinary (tigerHeadRelease)
       (strategoxtBinary (strategoxtHeadRelease)))
    (tigerBinary (tigerHeadRelease)
       (strategoxtBinary (strategoxt010Release)))
    (tigerBinary (tiger12Release)
       (strategoxtBinary (strategoxtHeadRelease)))
    (tigerBinary (tiger12Release)
       (strategoxtBinary (strategoxt010Release)))
]
Building Tiger Binaries

tigerBinaries =
  [ tigerBinary t s
  | t <- [tigerHeadRelease tiger12Release]
  , s <- [strategoxtHeadRelease strategoxt010Release]
  ];
1. Continuous Integration
2. Release Management
3. Build Farms
4. The Solution: Nix
5. Implementing a Build Farm with Nix
6. Conclusion
The Nix build farm:

- Allows safe and efficient management of dependencies.
- Ensures reproducibility.
- Supports multi-platform builds.
- Is efficient: only changed things are rebuilt.
- Produces actual releases.

More information:
http://www.cs.uu.nl/groups/ST/Trace/Nix.