Automating the Nengo build process

Trevor Bekolay

Centre for Theoretical Neuroscience technical report. Sept 17, 2010

Abstract

Nengo is a piece of sophisticated neural modelling software used to simulate large-scale networks of biologically plausible neurons. Previously, releases of Nengo were being created manually whenever the currently released version lacked an important new feature. While unit testing existed, it was not being maintained or run regularly. Good development practices are important for Nengo because it is a complicated application with over 50,000 lines of code, and depends on dozens of third-party libraries. In addition, being an open source project, having good development practices can attract new contributors. This technical report discusses the creation and automation of a back-end for Nengo, and how it integrates with the existing mechanisms used in Nengo development. Since the primary goal of the system was to avoid disturbing developers' workflow, the typical development cycle is made explicit and it is shown that the cycle is not affected by the new automated system.

1 Introduction

Nengo is the neural modelling software built by the Centre for Theoretical Neuroscience (CTN). It is based on the neural engineering framework (NEF)[1], which describes a method of building spiking neural network models that represent information, perform computation, and behave dynamically.

Nengo is currently used by researchers at the University of Waterloo, McGill, Yale, and Stanford. In order for it to be more widely used, Nengo is released as free software, under the Mozilla Public License Version 1.1.[2]

As more researchers begin to use and contribute to Nengo, it becomes more important to enforce good development practices. However, as a piece of software used for research, it is unlikely that many of those who wish to contribute to Nengo will have the time to read through extensive documentation detailing proper practices; the onus falls on the CTN, as the group most involved with Nengo, to closely monitor and maintain the quality of our codebase.

For these reasons, a number of tools were recently created and integrated into the development process of Nengo. Most of the tools are designed to run automatically, though some are simply available for those who find them useful.

This report begins with a discussion of the development environment that a typical researcher would use prior to the work described in this paper. Section 3 introduces Apache Ant, the build system that Nengo now uses. This section goes over the several options now available through the build scripts, such as running unit tests, creating a distribution folder, and running static analysis tools. Section 4 details the set-up and configuration of Hudson, a continuous integration server that was deployed on a CTN server. Section 5 concludes by discussing how the development environment has changed for a typical researcher.

[1]Eliasmith, Chris, and Charles H. Anderson. Neural engineering: computation, representation, and dynamics in neurobiological systems. MIT Press, 2003. `http://compneuro.uwaterloo.ca/`.

[2]A full copy of the Mozilla Public License can be viewed at `http://www.mozilla.org/MPL/`

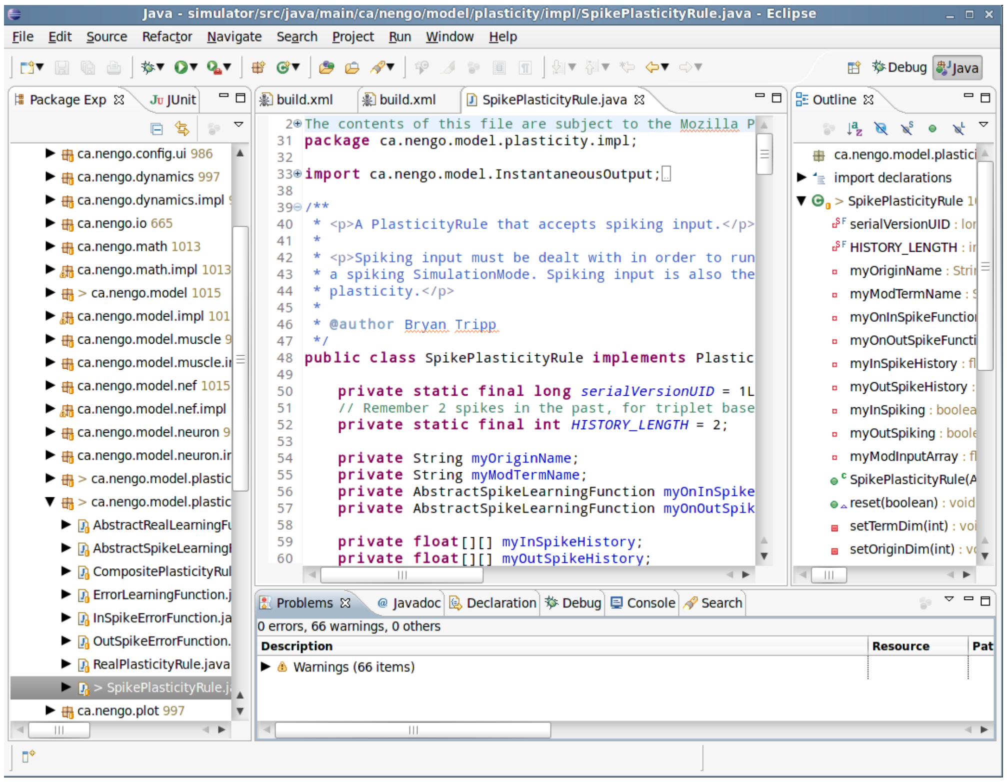
2 The development environment

Nengo is a Java application (with Python bindings through the Jython library). Its code is stored in a Subversion (SVN) server. A minimal environment for developing Nengo would consist of the following three elements.

1. Text editor (e.g. vi, emacs)
2. Java development kit, version 1.5 or later
3. Subversion (SVN) client

While this means that one could use only command-line utilities if preferred, it should be noted that some of Nengo's features – the graphical interface and plotting tools – require a graphical environment. However, simulations can be performed and data can be collected solely through the command line.

A more typical choice for a developer is to use the Eclipse IDE,[3] which is particularly well suited to developing Java applications. With a normal installation of Eclipse for Java developers, along with an SVN plugin such as Subversive,[4] Eclipse can replace all three of the elements above – it includes a sophisticated text editor that eases development through syntax highlighting and auto-completion, it compiles code continuously as it is changed, and it can do all common SVN tasks through a convenient graphical interface. For that reason, any developer capable of running Eclipse is recommended to use it, and most researchers currently contributing to Nengo use it as their development environment.


Java - simulator/src/java/main/ca/nengo/model/plasticity/impl/SpikePlasticityRule.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug Java
Package Exp JUnit
build.xml build.xml SpikePlasticityRule.java
Outline
Problems Javadoc Declaration Debug Console Search

```
The contents of this file are subject to the Mozilla P
package ca.nengo.model.plasticity.impl;

import ca.nengo.model.InstantaneousOutput;

/**
 * <p>A PlasticityRule that accepts spiking input.</p>
 *
 * <p>Spiking input must be dealt with in order to run
 * a spiking SimulationMode. Spiking input is also the
 * plasticity.</p>
 *
 * @author Bryan Tripp
 */
public class SpikePlasticityRule implements Plastic

    private static final long serialVersionUID = 1L
    // Remember 2 spikes in the past, for triplet base
    private static final int HISTORY_LENGTH = 2;

    private String myOriginName;
    private String myModTermName;
    private AbstractSpikeLearningFunction myOnInSpike
    private AbstractSpikeLearningFunction myOnOutSpik

    private float[][] myInSpikeHistory;
    private float[][] myOutSpikeHistory;
```

```
ca.nengo.config.ui 986
ca.nengo.dynamics 997
ca.nengo.dynamics.impl
ca.nengo.io 665
ca.nengo.math 1013
ca.nengo.math.impl 1013
> ca.nengo.model 1015
ca.nengo.model.impl 101
ca.nengo.model.muscle 9
ca.nengo.model.muscle.i
ca.nengo.model.nef 1015
ca.nengo.model.nef.impl
ca.nengo.model.neuron 9
ca.nengo.model.neuron.i
> ca.nengo.model.plastic
> ca.nengo.model.plastic
    AbstractRealLearningFu
    AbstractSpikeLearningI
    CompositePlasticityRul
    ErrorLearningFunction.j
    InSpikeErrorFunction.ja
    OutSpikeErrorFunction.
    RealPlasticityRule.java
    > SpikePlasticityRule.j
ca.nengo.plot 997
```

```
ca.nengo.model.plastici
import declarations
> SpikePlasticityRule 1
    serialVersionUID : lo
    HISTORY_LENGTH : i
    myOriginName : Stri
    myModTermName : S
    myOnInSpikeFunctio
    myOnOutSpikeFuncti
    myInSpikeHistory : f
    myOutSpikeHistory :
    myInSpiking : boolea
    myOutSpiking : boole
    myModInputArray : f
    SpikePlasticityRule(A
    reset(boolean) : void
    setTermDim(int) : voi
    setOriginDim(int) : v
```

0 errors, 66 warnings, 0 others

Description	Resource	Pat
Warnings (66 items)		

Figure 1: A screenshot of the Eclipse IDE, with Subversive installed.

[3]Eclipse is open source software available at http://www.eclipse.org/
[4]Subversive is available at http://www.eclipse.org/subversive/

2.1 Typical workflow

When a researcher wishes to make a change to the Nengo codebase, the following steps are typically followed. Note that there are no controls or procedural guidelines that dictate that these steps be followed in this order.

1. Check out a new copy of the Nengo codebase from the SVN repository, *or*
   SVN update the current local working copy of Nengo.
2. Implement desired change or new feature.
3. Test the change locally by running a few simulations to ensure that Nengo works as expected.
4. Commit the changes to the SVN repository, dealing with merge conflicts if necessary.

2.2 Ideal workflow

Nengo uses the JUnit unit testing framework to ensure that new code does not adversely affect existing functionality. While it does not adhere to the test-driven development workflow espoused by "extreme programming," unit testing is still an important method of ensuring code correctness.

Ideally, Nengo developers would follow the below steps to ensure that unit testing stays up to date. Tasks that differ from the typical workflow are bolded.

1. Check out a new copy of the Nengo codebase from the SVN repository.
2. Implement desired change or new feature.
3. **Write one or more unit tests to make explicit what the new change should do in the majority of situations, including edge cases and failure cases.**
4. **Run all unit tests to ensure old functionality is not affected.**
5. **Have code reviewed by a researcher familiar with that portion of the Nengo code.**
6. Commit the changes to the SVN repository, dealing with merge conflicts if necessary.
7. **Check out a fresh copy of the Nengo codebase from the SVN repository.**
8. **Run all unit tests again to ensure that Nengo builds and runs properly from a fresh checkout.**

The goal of the work described in the rest of this report is to automate as many of these tasks as possible. For those tasks that cannot be automated, tools are integrated to make the tasks as easy to accomplish as possible.

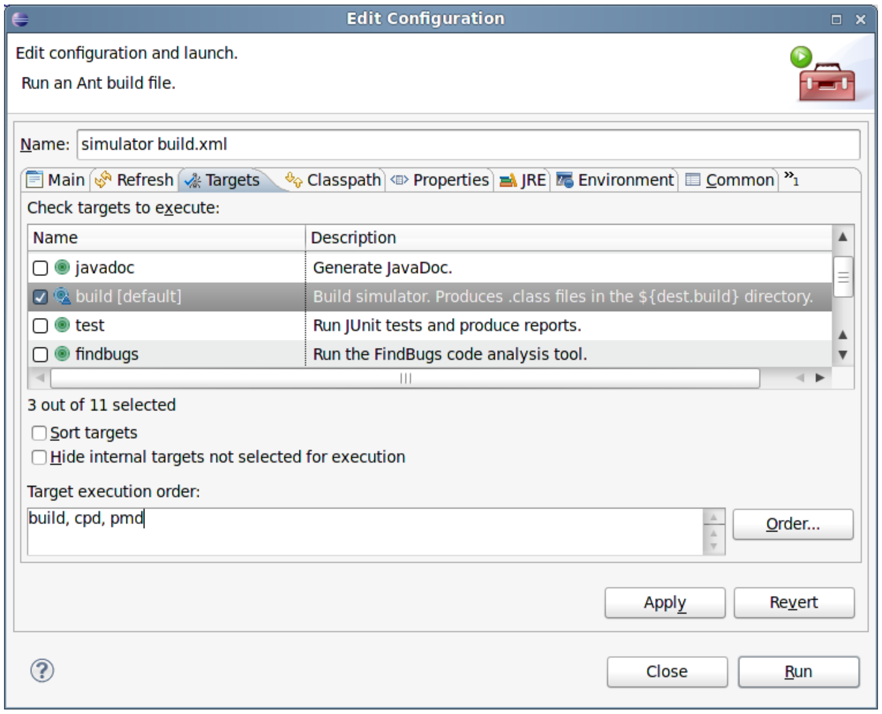
3 Apache Ant build scripts

Compiling a complicated large-scale application like Nengo is non-trivial to do from the command line because of the size of the codebase and a number of dependencies on third-party libraries. Eclipse makes building Nengo easier for developers working on their own machines, but Eclipse is not an appropriate environment for creating releases of Nengo. Extending Eclipse to do other tasks as part of the build process is also difficult because it may require that all developers change their Eclipse installation.

The traditional solution to this problem is a `makefile`. However, `makefiles` are platform dependent, meaning a `makefile` would have to be made for each operating system we would expect Nengo to be developed on.

The alternative system, which is used in many open source Java projects, is Apache Ant.[5] Ant is a platform independent build system like `make`. Ant build scripts are XML documents that define a number of **targets** – actions that a developer might want to perform – and how to perform those targets. The conventional name for an Ant build script is `build.xml`, placed in the root of a project folder; Nengo follows this convention by including a `build.xml` file in the root directories of both the `simulator` and `simulator-ui` projects.

Using Ant is also convenient for developers using Eclipse, as it has tools for running Ant build scripts installed by default. After checking out Nengo, a developer can right-click on `build.xml` in either the `simulator` or `simulator-ui` projects and choose any Ant target to run.



Edit configuration and launch.
Run an Ant build file.

Name: simulator build.xml

Check targets to execute:

Name	Description
☐ javadoc	Generate JavaDoc.
☑ build [default]	Build simulator. Produces .class files in the ${dest.build} directory.
☐ test	Run JUnit tests and produce reports.
☐ findbugs	Run the FindBugs code analysis tool.

3 out of 11 selected

☐ Sort targets
☐ Hide internal targets not selected for execution

Target execution order:
build, cpd, pmd

Figure 2: A screenshot showing Ant build script integration in Eclipse.

Table 1 gives a full listing of all the targets available to developers working on the `simulator` project. Table 2 gives a full listing of all the targets available to developers working on the `simulator-ui` project. These tables are current as of the writing of this report.

Targets marked as “internal” are targets that are not meant to be directly run by developers; they are automatically run at appropriate times through dependencies. For example, the `build` target requires that the `init` target be run first. The `build` target depends on the `init` target, so Ant will automatically run `init` before running `build`. In Ant, a target is marked as internal simply by not providing a description.

[5]Apache Ant is open source software available at `http://ant.apache.org/`

Target	**Description**
`init`	*Internal target*
`clean`	*Internal target*
`javadoc`	Generate JavaDoc
`build`	Build simulator. Produces .class files in the ${dest.build} directory.
`test`	Run JUnit tests and produce reports.
`findbugs`	Run the FindBugs code analysis tool.
`cpd`	Run the Copy-Paste Detector code analysis tool.
`pmd`	Run the PMD code analysis tool.
`checkstyle`	Run the CheckStyle code analysis tool.
`cobertura`	Do unit testing and generate Cobertura code coverage reports.
`all`	Generate all artifacts: javadoc, junit tests, FindBugs, CPD, PMD, CheckStyle, and Cobertura.

Table 1: Targets provided by `simulator/build.xml`.

Target	**Description**
`init`	*Internal target*
`clean`	*Internal target*
`javadoc`	Generate JavaDoc
`build-simulator`	*Internal target*
`build`	Build simulator-ui. Produces .class files in the ${dest.build} directory.
`test`	Run JUnit tests and produce reports.
`findbugs`	Run the FindBugs code analysis tool.
`cpd`	Run the Copy-Paste Detector code analysis tool.
`pmd`	Run the PMD code analysis tool.
`checkstyle`	Run the CheckStyle code analysis tool.
`cobertura`	Do unit testing and generate Cobertura code coverage reports.
`all`	Generate all artifacts: javadoc, junit tests, FindBugs, CPD, PMD, CheckStyle, and Cobertura.
`dist`	Create a distribution folder.

Table 2: Targets provided by `simulator-ui/build.xml`.

3.1 Building Nengo

Compiling all of the Nengo code and handling all of the third-party dependencies is handled by the Ant target `build` in both `simulator` and `simulator-ui`.

simulator:init

The `init` target creates the directories that will receive output from other targets. Ant does not automatically create directories that do not yet exist – if you attempt to to use `simulator/build` as the destination for compiling the `simulator` code and it does not exist, Ant will fail. The directories set up by the `init` target are the following – these are defined in `simulator/build.properties`. *Note: variables in Ant are called "properties," and are used with the syntax ${var name}.*

- `${dest.artifacts} = simulator/artifacts`
- `${dest.build} = simulator/bin`
- `${dest.instrumented} = simulator/bin-instrumented`
- `${dest.test} = simulator/bin-test`

simulator:build – depends on simulator:init

The `build` target compiles all of the `.java` files in the `simulator/src/main` directory and places the generated `.class` files in `simulator/bin`. Of particular importance is that this target does not compile the source code in `simulator/src/test`.

simulator-ui:build – depends on simulator-ui:init and simulator-ui:build-simulator

The `build-simulator` target simply calls the `build` target in the `simulator` project. `simulator-ui` is designed to be an interface into `simulator`, so it requires it to be compiled before `simulator-ui` can be compiled. Other than this dependency, `init` and `build` do the same things as in the `simulator` project, only within the `simulator-ui` directory.

3.2 Unit testing

There are two Ant targets related to unit testing. These targets do identical work in `simulator` and `simulator-ui`.

test

The `test` target compiles the code found in `src/test`. It then runs all of test cases found in those compiled classes using `batchtest`, provided by the JUnit Ant task.[6] It produces reports in two formats: XML, which are output to `artifacts/xml/junit`, and HTML, which are output to `artifacts/html/junit`. HTML is generally more readable for developers, but the XML reports are easily parsed by other analysis tools.

cobertura

Cobertura is a tool used to determine code coverage – how much code the unit tests are actually testing. The report generated by Cobertura is extremely detailed, telling a developer whether each line of code is tested or not. The Cobertura target is more complicated than other code analysis targets, because it must "instrument" all of the source code – that is, it must recompile source files with additional information that is used by Cobertura to determine is a line of code is executed. Running the unit tests with these instrumented classes adds significantly to the time it takes for unit tests to run. The majority of time take on a full build (that is, the time taken to run the `all` target) is taken by the `cobertura` task. For full details on Cobertura, see its website at `http://cobertura.sourceforge.net/`.

[6]See `http://ant.apache.org/manual/Tasks/junit.html` for more details.

3.3 The dist target

Prior to the work described in this report, creating a "release" of Nengo was done manually whenever a useful new feature was added, and the code was determined to be stable. The `dist` target essentially codifies that process by performing all the tasks that were previously done manually. It introduces an additional level of control to the release process, as all of the tasks done for a release are now explicitly available to anyone with access to the SVN repository.

As part of this change, files that were previously included with releases of Nengo but were not checked into the SVN repository (e.g. example script files) are now tracked by the SVN repository, in the directory `simulator-ui/dist-files`. Not only does this mean that anyone with SVN access can create a release of Nengo if desired, it means that we can make a release of Nengo based on any version in the SVN repository after SVN revision 989. For more details on this change and other reorganizations done as a result, see `http://ctnsrv.uwaterloo.ca:8080/sventon/repos/Nengo/info?revision=989`.

The `dist` target only exists in `simulator-ui/build.xml`.

dist

The `dist` target does the following.

- Creates a new folder, `nengo-x`, where x is replaced with the current SVN revision number. For example, if one checks out SVN revision 995 and runs the `dist` target, it will create a `nengo-995` directory at the same level as the `simulator` and `simulator-ui` directories.
- Creates `nengo-x.jar`, a file that contains all of the non-test classes in `simulator` and `simulator-ui`, and places it in the `nengo-x` folder.
- Copies the source code for `simulator` and places it in `nengo-x/api/src`.
- Generates the Javadoc for `simulator` and places it in `nengo-x/api/docs`.
- Generates the files `nengo.bat` and `nengo`, used to launch Nengo on Windows and non-Windows systems, respectively. The files are based on templates that have the SVN revision number inserted into them so that the appropriate `nengo-x.jar` file is launched. Those files are copied into the `nengo-x` folder.
- Copies any files in `simulator-ui/dist-files` to the `nengo-x` folder.
- Uses `chmod` to set `nengo-x/nengo` and `nengo-x/external/pseudoInverse` to executables files (i.e. adds the 'x' flag to their permissions). *Note: this does not always seem to work properly.*

3.4 Code analysis

A common practice in development teams is to perform code reviews. The purpose of a code review is to ensure that at least two pairs of eyes have looked at all of the code that gets checked in to the source code repository. This practice does not guarantee high quality code, but it catches many mistakes early enough to fix easily.

Code reviews are difficult to organize for Nengo for several reasons. The developers of Nengo are not necessarily all in one physical location. In addition, the researchers that develop Nengo tend to be in specialized fields, and for that reason focus on small portions of the Nengo codebase – there are few people who have good knowledge of all of the code in Nengo.

Unit testing alleviates some of the need for code reviews, because they at the very least ensure that Nengo is functional and performing the operations that the unit tests exercise. In an effort to further scrutinize code automatically, static code analysis tools were applied to the Nengo codebase. These tools can be run using a number of Ant targets, all of which operate identically on `simulator` and `simulator-ui`.

Note that, unlike code reviews, the reports generated by these tools should not be taken as a list of problems to fix; they may indicate some areas of code that should be cleaned up or refactored, but attempting to fix all of the errors and warnings brought up by these tools is not a useful exercise. Reviewing the output from these tools when new code is checked in, however, can improve the coding style of a Nengo contributor.

findbugs

The `findbugs` target runs the FindBugs static analysis tool. FindBugs looks for three main types of issues.

1. **Correctness** An apparent coding mistake. For example, FindBugs identified a code path in `ca.nengo.config.ui.NewConfigurableDialog.showDialog` that will always result in a null-pointer exception.
2. **Bad practice** Violations of what is commonly accepted as good programming practice. For example, FindBugs identified that the `clone()` function in `ca.nengo.model.nef.impl.DecodedOrigin` does not call `super.clone()`, which can cause issues with creating clones of `DecodedOrigin`.
3. **Style** Confusing code that may cause programming errors in the future. For example, FindBugs identified that the static field `ca.nengo.util.impl.NodeThreadPool.myNumThreads` is written to in a certain method, which may cause issues with other instances of the class.

For more details on FindBugs, see its website at `http://findbugs.sourceforge.net/`.

cpd

The `cpd` target runs the Copy-Paste Detector static analysis tool. It searches through Nengo's codebase looking for chunks of code that are exact duplicates of other chunks of code. The motivation behind this is that duplicate code should never need to occur; where duplicate code seems necessary, it is almost always a better choice to extract that logic into a helper function. For more details, see the Copy-Paste Detector's website at `http://pmd.sourceforge.net/cpd.html`

pmd

The `pmd` target runs the PMD static analysis tool. PMD is similar in scope to FindBugs, searching code for issues such as possible bugs, suboptimal code, and overcomplicated expressions. The specific items that it looks for are different from FindBugs, however, so despite some overlap between the issues they find, having both is still useful.

checkstyle

The `checkstyle` target runs the CheckStyle static analysis tool. Unlike the other tools that look through code for possible errors, the CheckStyle tool simply looks at the structure of the text that makes up the code, ensuring that the programming style matches a set of rules. For example, CheckStyle can check to ensure that there are no "tab" characters in the code, and that all indentation is done with spaces. CheckStyle necessitates the creation of a set of rules that Nengo programmers should follow.[7]

4 The Hudson continuous integration server

Having the Ant build scripts available is helpful to developers who wish to do things like run unit tests, create release versions of Nengo, and analyze source code. However, the vast majority of developers are not concerned with these things – nor should they be bothered with them. Instead, we can automate the running of these tasks using a **continuous integration server**.

The idea of a continuous integration server is that every time code is committed to the SVN repository, a fresh checkout is done and the code is built and tested automatically. In this way, even if a developer does not follow best practices for development, other developers can see if the latest version of the code in the

[7]This set of rules has not yet been decided upon; CheckStyle is currently set up to use the conventions outlined by Oracle at `http://www.oracle.com/technetwork/java/codeconv-138413.html`.

repository will build successfully and pass all of the unit tests. The continuous integration server can, in fact, run any Ant target upon an SVN commit, so static code analysis reports can also be generated.

Continuous integration is one of the tools created as a result of the Agile Software Development methodology. Eclipse does this on a local scale; when a programmer makes a change to source code in Eclipse, it will rebuild the project in the background, giving nearly immediate feedback on how the source code changes affect the project. A continuous integration server does this for the whole project; when a programmer makes a change to source code in the the repository, the project is rebuilt and analyzed.

After evaluating the available options, it was determined that the **Hudson** continuous integration server would be the best option.[8] Hudson is a Java servlet, which means that the server that will be used for continuous integration needs to have a servlet container set up. **Tomcat 6** was chosen as the Java servlet container that is the most robust and easy to set up. Hudson also integrates with a number of SVN repository browsers to easily view the code present in an SVN repository and see how files have changed over time. **Sventon**[9] was chosen as the best SVN repository browser for our needs.

4.1 Server set-up

The server chosen to set Hudson up on is the main SVN server for the Centre for Theoretical Neuroscience. This server is publicly accessible at `http://ctnsrv.uwaterloo.ca/`, but many of the features are restricted to lab members through password protection.

The SVN server uses a version of Ubuntu Desktop as its operating system. To set it up for Hudson, it required the following general steps.

1. Install a Java development kit (JDK).

   While Java 1.6 has been available since December 2006, it was decided to put JDK 1.5 on the server. This is because Nengo does not explicitly use any features of Java 1.6, and some researchers are forced to use Java 1.5 in their computing environment. Using JDK 1.5 on this server means that we can catch errors that occur from developers with Java 1.6 on their local machines using features introduced in 1.6.

2. Install Tomcat 6.

3. Download the latest version of Hudson and place it in Tomcat's "webapps" directory.

A complete list of the commands performed on the server is included as Appendix A.

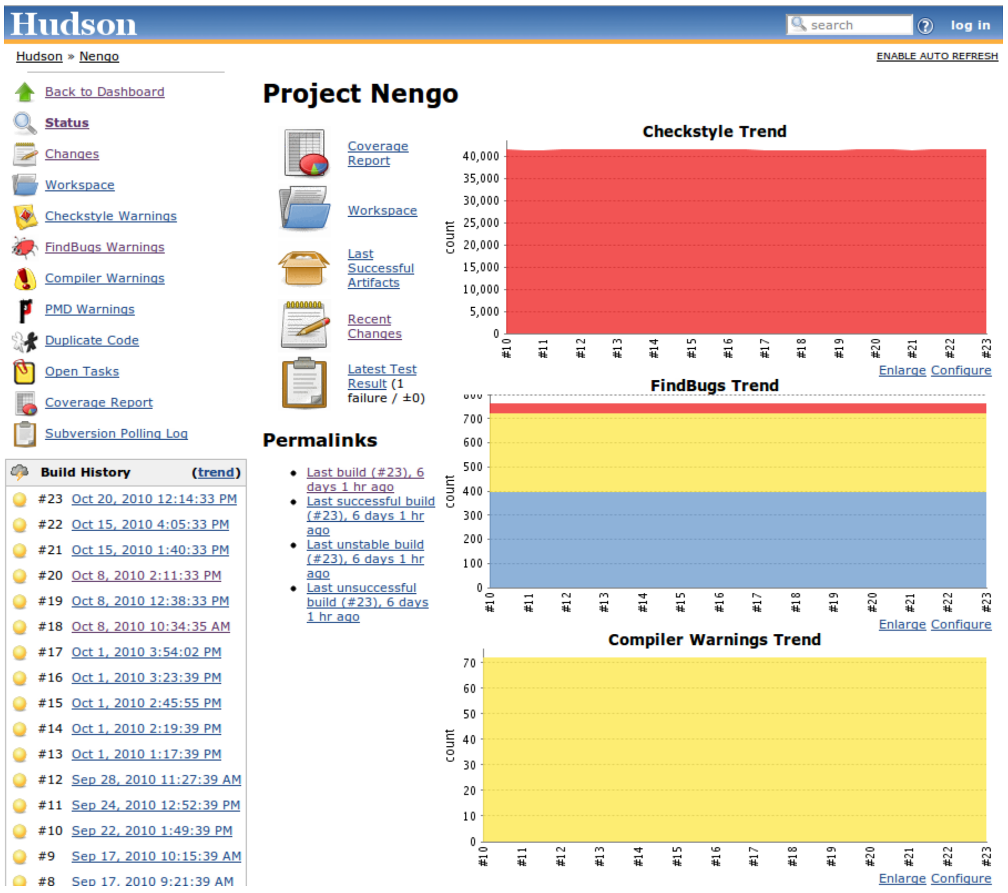
One important note is that the server already has Apache 2 installed on it, duplicating some of the functionality of Tomcat 6 (serving static websites, for example). It is possible to integrate these two closely, such that Apache forwards requests to certain URLs to Tomcat and handles the rest itself. However, since Tomcat by default only handles requests from port `8080`, the potential benefit of removing `:8080` from URLs that Tomcat handles was not seen as worth the extra set up time.

4.2 Hudson

Hudson is a piece of software that can monitor various sources, such a source code repositories, and build and test software projects when certain conditions are met (e.g. new code is committed, six hours have passed). While Hudson can be extended greatly through plugins to perform many complicated functions, its use in the Nengo project is relatively straightforward. Hudson monitors the SVN repository continually. When a

[8]Hudson is open source software available at `http://hudson-ci.org/`.
[9]Sventon is open source software available at `http://www.sventon.org/`.

new commit is made, Hudson runs the `all` targets for both `simulator` and `simulator-ui`, which runs JUnit tests, generates a coverage report for those tests, produces Javadoc, and runs all of the static analysis tools. It then calls the `dist` target in `simulator-ui`, producing a `nengo-x` directory tagged with the SVN revision number. This directory can then be used as a release version of Nengo.



- Last build (#23), 6 days 1 hr ago
- Last successful build (#23), 6 days 1 hr ago
- Last unstable build (#23), 6 days 1 hr ago
- Last unsuccessful build (#23), 6 days 1 hr ago

Build History	(trend)
#23	Oct 20, 2010 12:14:33 PM
#22	Oct 15, 2010 4:05:33 PM
#21	Oct 15, 2010 1:40:33 PM
#20	Oct 8, 2010 2:11:33 PM
#19	Oct 8, 2010 12:38:33 PM
#18	Oct 8, 2010 10:34:35 AM
#17	Oct 1, 2010 3:54:02 PM
#16	Oct 1, 2010 3:23:39 PM
#15	Oct 1, 2010 2:45:55 PM
#14	Oct 1, 2010 2:19:39 PM
#13	Oct 1, 2010 1:17:39 PM
#12	Sep 28, 2010 11:27:39 AM
#11	Sep 24, 2010 12:52:39 PM
#10	Sep 22, 2010 1:49:39 PM
#9	Sep 17, 2010 10:15:39 AM
#8	Sep 17, 2010 9:21:39 AM

Figure 3: The main Nengo project screen in Hudson.

Hudson provides a web interface through which options can be modified and builds monitored. The main Nengo project page, found at `http://ctnsrv.uwaterloo.ca:8080/hudson/job/Nengo/`, gives an overview of the project. Everything that a typical developer would want to do can be accessed through this screen.

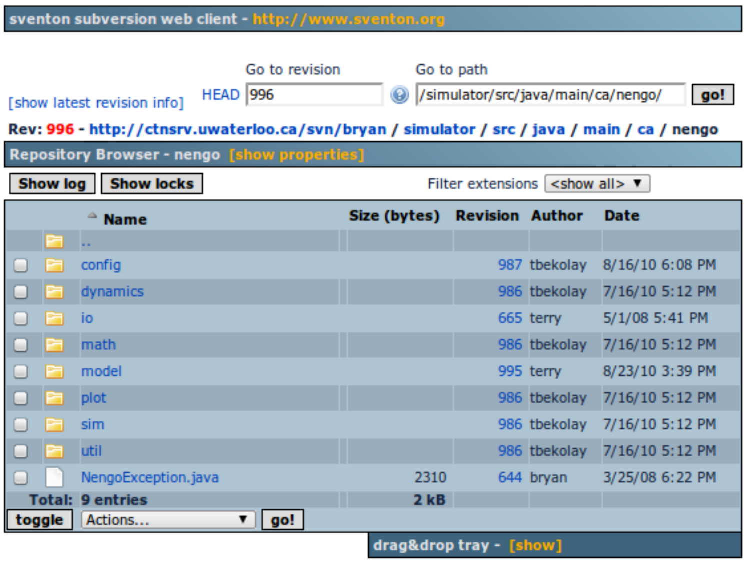
Whenever Hudson detects a change to the SVN repository, it does a fresh checkout of all of the code contained in it. It puts all of that in a "Workspace" folder. It then runs the Ant tasks described above. Any files created by Ant can also be seen in the workspace. Clicking on the *Workspace* link in the middle column shows the current workspace folder, which is usually in a post-build state, with `nengo-x`, `simulator`, and `simulator-ui` directories. When navigating through the workspace through the web interface, zip files can be created at any point, containing all of the files and folders contained in the current directory.

Keeping the whole workspace for each Nengo build would take a lot of hard drive space; even with zip compression, the workspace takes up over 100 MB of space. Further, since those files and all previous revisions are already in the SVN repository, it's redundant information. However, the artifacts produced by the build process (compiled classes, reports, etc.) are kept so that, if one wants to know about the state of Nengo at a certain point, or wants to get a release of a specific build of Nengo, it can just be downloaded from Hudson rather than regenerated from scratch. The artifacts for the current build can be accessed by clicking on the *Last Successful Artifacts* link in the middle column. With zip compression, the artifacts for a build take up approximately 20 MB of space, much less than keeping the whole workspace.

Unit testing results can be seen by clicking the *Latest Test Result* link in the middle column. The coverage report, produced by Cobertura, can be seen by clicking the *Coverage Report* link. Details on the results of running the static analysis tools can be seen by clicking on the appropriate link in the left column. Plugins for Hudson also produce graphs showing how those analysis reports have changed over time in the right column.

A brief list of recent changes to the SVN repository can be seen by click on the *Recent Changes* link in the middle column. More details on changes to the SVN repository can be seen by clicking on the *Sventon* link on any of the changes listed in Hudson.

4.3 Sventon



sventon subversion web client - http://www.sventon.org

[show latest revision info] Go to revision: HEAD 996 Go to path: /simulator/src/java/main/ca/nengo/ go!

Rev: 996 - http://ctnsrv.uwaterloo.ca/svn/bryan / simulator / src / java / main / ca / nengo

Repository Browser - nengo [show properties]

Show log Show locks Filter extensions <show all>

Name	Size (bytes)	Revision	Author	Date
..				
config		987	tbekolay	8/16/10 6:08 PM
dynamics		986	tbekolay	7/16/10 5:12 PM
io		665	terry	5/1/08 5:41 PM
math		986	tbekolay	7/16/10 5:12 PM
model		995	terry	8/23/10 3:39 PM
plot		986	tbekolay	7/16/10 5:12 PM
sim		986	tbekolay	7/16/10 5:12 PM
util		986	tbekolay	7/16/10 5:12 PM
NengoException.java	2310	644	bryan	3/25/08 6:22 PM
Total: 9 entries	2 kB			

toggle Actions... go!

drag&drop tray - [show]

Figure 4: Viewing a previous revision of a directory in Sventon.

Sventon is an SVN repository browser implemented as a Java servlet. Because it's a Java servlet, it can be run in a very similar manner as Hudson, making its inclusion as part of the server set up natural.

Sventon allows a developer to look through an SVN repository, from the first revision to the `HEAD`, and anywhere in between. Sventon provides an easy-to-use interface to many of SVN's useful functions, such as `blame`, which lists which developer is responsible for committing each line of code in a file.

Sventon also can show the difference between any two revisions of a file in a number of different presentation styles, including the side-by-side `diff`.

The Nengo repository can be viewed with Sventon at `http://ctnsrv.uwaterloo.ca:8080/sventon/`.

5 The development environment revisited

Despite the addition of Ant build scripts and the Hudson continuous integration server, the workflow for a Nengo contributor has not significantly changed. In fact, a developer does not have to change the way they contribute to Nengo at all. However, there are still some areas that require developer intervention, and it is those tasks that developers should be aware of and encouraged to perform.

Recall that the “ideal workflow” listed in section 2.2 added several new steps. Below, we examine how the changes described in this report have made these steps either automatic or easier to perform.

3. **Write one or more unit tests to make explicit what the new change should do in the majority of situations, including edge cases and failure cases.**

   This is the one task that cannot be reasonably automated. Writing test cases requires that a developer take the time to appropriately test their code. This is made somewhat easier with the Ant scripts, as the `test` target allows developers to run all unit tests at once and see the results. In addition, small changes were made to the Eclipse project file that allows developers to run individual unit tests from within Eclipse.

4. **Run all unit tests to ensure old functionality is not affected.**

   Again, a developer can now run the `test` target in the Ant build scripts. However, because Hudson will automatically run these tests when new code is committed, this step is not necessary. If the newly committed code breaks a unit test, developers will be notified, and the code either fixed, or rolled back to a previous version.

5. **Have code reviewed by a researcher familiar with that portion of the Nengo code.**

   As discussed in section 3.4, code reviews are not reasonable for most changes made to Nengo. However, unit testing and the integration of static analysis tools provide many of the same benefits as code reviews.

7. **Check out a fresh copy of the Nengo codebase from the SVN repository.**
8. **Run all unit tests again to ensure that Nengo builds and runs properly from a fresh checkout.**

   These two steps are performed by Hudson automatically once code is committed to the SVN repository.

With the integration of Ant build scripts and the Hudson continuous integration server, we have gotten very close to the ideal workflow without adding an additional burden on researchers contributing to Nengo.

Additional work needs to be done in encouraging developers to write unit tests for the code that they commit to the Nengo SVN repository, because this task cannot be automated. Possible ways to do this include writing an easy-to-follow guide on writing unit tests, posting it on the website, and sending a link to researchers that contribute to Nengo. Another resource to prepare would be some template test classes that can be used as a starting point for writing new unit tests.

More experienced Nengo developers should also consider writing unit tests for currently untested portions of code. The Cobertura coverage report helps these developers identify code that needs unit testing.

A Setting up ctnsrv.uwaterloo.ca

The machine that serves the `ctnsrv.uwaterloo.ca` domain runs Ubuntu Desktop with Apache 2 and Subversion server software installed. Hudson was determined as a good choice for a continuous integration server for Nengo, so the server needed to have Java 1.5, Tomcat 6, Hudson and Sventon installed in it. The commands that accomplished this are given below. If these tools need to be set up on another similar server, these commands should be used as a guideline.

A.1 Setting up Tomcat

The first step is to install Java 1.5.

Open up `/etc/apt/sources.list` with a command such as

```
sudo pico /etc/apt/sources.list
```

To the bottom of this file, append

```
deb http://us.archive.ubuntu.com/ubuntu/ jaunty multiverse
deb http://us.archive.ubuntu.com/ubuntu/ jaunty-updates multiverse
```

In a terminal window, run the following commands

```
sudo apt-get update
sudo apt-get install sun-java5-jdk
```

Accept if prompted.

The next step is to download and install Tomcat 6. The location of Tomcat's distribution tar file may have changed since the writing of this report.

```
cd ~
wget http://apache.parentinginformed.com/tomcat/tomcat-6/ \
     v6.0.29/bin/apache-tomcat-6.0.29.tar.gz
tar xvzf apache-tomcat-6.0.29.tar.gz
sudo mv apache-tomcat-6.0.29 /usr/share/tomcat6
```

Next, we will create a file in `/etc/init.d` so that Tomcat 6 will start automatically when the computer boots up.

Create a new text file, `/etc/init.d/tomcat6`, with a command such as

```
sudo pico /etc/init.d/tomcat6
```

Paste the following text into that file.

```
# Tomcat auto-start
#
# description: Auto-starts tomcat
# processname: tomcat
# pidfile: /var/run/tomcat.pid

export JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun
export JAVA_OPTS="-Xmx1024M -Xms256M -server -Djava.awt.headless=true"
export TOMCAT_HOME=/usr/share/tomcat6
export CATALINA_OPTS="-DHUDSON_HOME=/srv/hudson"

case $1 in
start)
        /bin/su tomcat6 $TOMCAT_HOME/bin/startup.sh
        ;;
stop)
        /bin/su tomcat6 $TOMCAT_HOME/bin/shutdown.sh
        ;;
restart)
        /bin/su tomcat6 $TOMCAT_HOME/bin/shutdown.sh
        /bin/su tomcat6 $TOMCAT_HOME/bin/startup.sh
        ;;
esac
exit 0
```

Next, we will set the appropriate permissions on this file and then create necessary symbolic links to it.

```
sudo chmod 755 /etc/init.d/tomcat6
sudo ln -s /etc/init.d/tomcat6 /etc/rc1.d/K99tomcat
sudo ln -s /etc/init.d/tomcat6 /etc/rc2.d/S99tomcat
```

Finally, we must create a user, `tomcat6`, that Tomcat will be run as, so that it only has access to the files that we say it should have access to. In this process, we will also create a folder for Hudson and give the `tomcat6` user access to it.

```
sudo useradd tomcat6 -g nogroup -d /usr/share/tomcat6/ -p'*' -r
cd /srv
sudo mkdir hudson
sudo chown -R tomcat6:nogroup hudson
cd /usr/share/tomcat6
sudo chown -R tomcat6:nogroup webapps temp logs work conf
sudo /etc/init.d/tomcat6 start
```

At this point, Tomcat should be running. To test this, open a browser window and navigate to either `http://localhost:8080` or `http://ctnsrv.uwaterloo.ca:8080`.

If you see a tomcat and some text, then Tomcat is running properly.

One area of Tomcat that may be useful for future administration is the Tomcat Manager. To access it, we must first edit Tomcat's list of users.

```
sudo pico /usr/share/tomcat6/conf/tomcat-users.xml
```

Paste the following into this file, changing the asterisks to the appropriate password.

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="manager"/>
  <user username="ctnuser" password="*****" roles="manager"/>
</tomcat-users>
```

Use that username and password to log into the manager at `http://localhost:8080/manager/html` or `http://ctnsrv.uwaterloo.ca:8080/manager/html`.

A.2 Integrating Tomcat with Apache

It is possible to integrate Tomcat with Apache, so that rather than using port 8080, and URLs that begin with `http://ctnsrv.uwaterloo.ca:8080`, Apache can forward any URLs that match a certain pattern to Tomcat.

This is done with the `mod_jk` module. Details on `mod_jk` can be found at `http://tomcat.apache.org/connectors-doc/`.

This process was not seen as essential for the goal of using Hudson internally.

A.3 Setting up Hudson and Sventon

Hudson and Sventon are distributed as `WAR` files. If Tomcat 6 is set up properly, and you are able to access the manager at `http://ctnsrv.uwaterloo.ca:8080/manager/html`, then you can deploy those `WAR` files simply by downloading them to your local machine, and uploading them to the server through the manager interface (under the heading "WAR file to deploy").

The latest Hudson `WAR` file can be found at `http://hudson-ci.org/latest/hudson.war`. The latest Sventon `WAR` file can be found at `http://www.sventon.org/`.

You can also do this manually through the command line.

```
# Assuming that hudson.war and sventon.war are in your home directory
cd ~
sudo mv hudson.war /usr/share/tomcat6/webapps
sudo mv sventon.war /usr/share/tomcat6/webapps
cd /usr/share/tomcat6/webapps
sudo chown tomcat6:nogroup hudson.war sventon.war
```

After deploying these `WAR` files, Hudson should be available at `http://ctnsrv.uwaterloo.ca:8080/hudson` and Sventon should be available at `http://ctnsrv.uwaterloo.ca:8080/sventon`.