Eli System Administration Guide

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

Eli embodies an understanding of how to generate text processors from specifications of their behavior. This understanding has been developed over many years by a host of people, and is expressed in a large number of tools and work flow patterns. The generation process creates many intermediate artifacts, which must be passed among the tools. A change in a specification may make it necessary to rebuild some of these artifacts by executing particular tools.

Odin is a program capable of managing a collection of artifacts and tools to provide a product that is up-to-date with respect to all of the raw material on which it is based. The tools and associated work flow patterns are described to Odin by a *derivation graph*. Modularity in the derivation graph specification is achieved by gathering related tools and the work flow patterns connecting them into *packages*.

In order to use Odin to derive specific products from specific raw material, a *cache* must be available. A cache is represented in the computer by a single directory. It contains a derivation graph built from some set of packages, plus all of the artifacts created when deriving products defined by that derivation graph. Each user may establish an arbitrary number of caches, and each cache may be shared by an arbitrary number of users.

1 Package Management

Eli is a collection of packages. Each package is represented in the computer by a directory. A package directory must contain a file whose name is the name of the directory with the suffix `dg`. That file specifies the derivation graph for the package. For example, the package defining the tools and work flow for constructing a scanner is represented by a directory named '`gla`', and contains a derivation graph specification file named `gla.dg`. Some packages contain no tools, and therefore their derivation graphs are specified by empty files.

1.1 Versioned packages

The packages in the Eli distribution directory are *versioned*. A file named '`version`' is included in each package directory. That file consists of a single line that is a three-part version number. It is updated whenever a change to the package affects the behavior of that package; repairing defects in a package's tools does not require a version change.

If the package's work flow is changed without altering its functionality, the third component of the version number is incremented. When the local functionality is changed, the second component of the version is incremented and the third zeroed; when the change is visible to other packages, the first component is incremented and the others zeroed.

Versioned packages allow each user to decide on a project-by-project basis when to upgrade (see Chapter 3 [Installation], page 7).

1.2 Package collections

When the understanding of a process is embodied in a *collection* of packages, as in the case of Eli, all of the directories representing those packages are stored as subdirectories of a single directory. In the Eli distribution there are two such directories, '`Eli/pkg`' and '`Odin/pkg`'. '`Eli/pkg`' embodies the understanding of text processor generation, whereas '`Odin/pkg`' embodies the understanding of more general processes such as linking object files, executing a command line, and formatting a document.

In addition to the package directories themselves, each collection's directory contains a file named '`PKGLST`'. '`PKGLST`' is a text file, each line of which is the name of one of the package directories. Thus '`Eli/pkg/PKGLST`' contains the line '`gla`' to indicate that the '`gla`' directory represents a package.

'`PKGLST`' allows a collection's directory to contain additional files and non-package directories. The names of these files and directories will not appear in '`PKGLST`', and therefore they will not be mistaken for packages in the collection. It is also possible to temporarily exclude packages from the collection simply by removing their names from '`PKGLST`'; the package directories themselves remain unchanged (see Section 4.2 [Maintaining distributed packages], page 9).

2 Cache Management

Each cache directory contains a derivation graph built from some set of packages, plus all of the artifacts created when providing products defined by that derivation graph. A user may want to have more than one cache for several reasons:

- A cache used on a completed project can be deleted without losing artifacts associated with ongoing projects.
- A user may collaborate with different people on different projects, each having its own cache.
- Different derivation graphs may be appropriate for different projects.

In order to work with more than one cache, the user must be able to refer to each by a unique name; in order to have caches with different derivation graphs, the user must be able to control the set of packages from which a cache is built.

2.1 Name of the cache

When the `eli` command is executed, it specifies the name of a cache directory as follows:

- If the `eli` command has the parameter `-c 'dir'`, then the name of the cache directory is '`dir`'`/$ODINVIEW`. '`dir`' must be an absolute path name.
- If the `eli` command does not have the `-c` parameter, then the name of the cache directory is `$ODIN/$ODINVIEW`. The value of the environment variable `$ODIN` must be an absolute path name.
- If the environment variable `ODIN` is not set, Then the name of the cache directory is `$HOME/.ODIN/$ODINVIEW`.
- If the environment variable `ODINVIEW` is not set, then the result of the command `uname -n` is used if it is not empty (`uname -n` normally returns the name of the machine executing the command).
- If the command `uname -n` returns an empty string, then the string `local` is used.

If the specified cache directory does not exist, a new cache is built and represented by a new directory with the specified name.

The environment variable `ODINVIEW` is commonly used in a classroom or laboratory having a number of identical computers that share a common file system. Since the computers share a common file system, every execution of the `eli` command will have the same value of `ODINVIEW`. As noted above, if `ODINVIEW` is not set then the default cache name will incorporate the name of the particular computer on which it was created. Therefore a user will have a different cache for each machine on which they have run the command `eli` without a `-c` parameter. This can be avoided if the user's startup script defines `ODINVIEW` as a constant value.

A different situation arises with a number of clusters of machines of different architectures, all sharing a common file system. Each cache is tied to a specific architecture, so there must be a distinct default cache name for each cluster if the `eli` command is to be used without a `-c` parameter. The user's startup script should therefore define `ODINVIEW` as the result of a command such as `arch`, which returns a string unique to the architecture of the machine running the command, rather than a constant value.

2.2 Packages in the cache

The set of packages from which a cache is built is determined either at the time that an `eli` command with the `-R` parameter is executed on that cache, or at the time a non-existent cache is specified. In either case, the packages making up the set are drawn from four sources, in order:

1. Parameters of the `eli` command, each consisting of `-p` followed by the absolute path name of a directory. An `eli` command may have an arbitrary number of such parameters, and they are considered in order from left to right.
2. The colon-separated list of absolute path names in the `ODINPATH` environment variable.
3. The standard Eli packages.
4. The standard Odin packages.

Each of the directories may represent a single package or a collection of packages with a '`PKGLST`' file (see Chapter 1 [Package Management], page 3).

The order in which the directories are considered is important, because only the first package with a given name is entered into the set from which the cache is built. Thus it is possible to override standard Eli and Odin packages by providing replacements whose names are the same as the names of the packages to be replaced (see Section 4.2 [Maintaining distributed packages], page 9).

If an `eli` command specifying an existing cache is executed without the `-R` parameter, then any `-p` parameters are ignored. The set of packages available is the set specified when that cache was originally created, or when it was last specified by an `eli` command with the `-R` parameter.

2.3 The `odin` command

Odin can be invoked directly with the `odin` command, although there is no need to do so except to create a cache without using any of the standard Eli packages.

Unless the `-R` parameter is given, the behavior of the `odin` and `eli` commands are identical when they specify an existing cache. If the specified cache does not exist, or if the `-R` parameter is given, then the sets of packages created by the two commands are different. When an `odin` command is executed in either of those cases, the packages making up the set are drawn from two sources, in order:

1. The colon-separated list of absolute path names in the `ODINPATH` environment variable.
2. The standard Odin packages.

This differs from the behavior of the `eli` command (see Section 2.2 [Packages in the cache], page 6). The `odin` command does not accept `-p` parameters and does not consider the standard Eli packages. The effect of `-p` parameters is obtained by building an appropriate colon-separated list and making it the value of the environment variable `ODINPATH` before executing the `odin` command.

3 Installation

Eli is distributed as source text, which must be configured and compiled before the system is usable. An extra (optional) *installation* step creates a library that takes advantage of versioned packages (see Section 1.1 [Versioned packages], page 3). There are two reasons for installing Eli:

- Existing packages are not overwritten by new versions, leaving current caches unchanged until they are reset by a `-R` parameter on the `eli` command (see Section 2.2 [Packages in the cache], page 6). Thus the timing of an upgrade can be tailored to individual projects.
- The distribution directory contains a lot of "scaffolding" that is used to build tools, but is not required to execute Eli. By installing the system and then deleting the distribution directory, you can eliminate all of this extraneous material.

3.1 Installing Eli

As described in the Eli distribution's '`README`' file, you can build the system by running `configure` followed by `make`. A subsequent invocation of `make install` will install Eli in the directory that you specified by the `--prefix` parameter when you configured the distribution. Typical settings are `--prefix=/usr/local` and `--prefix=/opt/eli`. (Your home directory is used if no `--prefix` parameter is supplied to the `configure` command.)

Installation will add the following files and directories to the specified directory (the '`man`', '`bin`', and '`lib`' sub-directories will be created by the installation process if they don't already exist):

'`man/man1/odin.1`'
: Man page for Odin.

'`bin/odin`'
: Command script

'`bin/eli`'
: Command script

'`lib/Eli`'
: The library of Eli packages.

'`lib/Odin`'
: The library of Odin packages.

Each package directory in these libraries contains a subdirectory for each distinct version of that package, plus meta-information defining the latest version. When a cache is built from a set of packages, the meta-information is used to establish links to the latest version in each set. The packages themselves are not copied to the cache.

When a new version of Eli is installed, the following changes are made to the libraries:

1. If the '`version`' file of a package has not changed, but some of the other files in that package have changed, the new files replace the old in the appropriate subdirectory.
2. Otherwise, a new subdirectory of the package directory in the library is created. It's name is the content of the '`version`' file. The necessary files are copied into this new directory, and the meta-information is updated to indicate that the new directory is the latest.

Since each cache links to a specific version subdirectory of the package, changes will only affect it if the package version has not changed. If the package version has changed, the previous subdirectory remains undisturbed. Thus an existing cache will be affected only by defect repairs that do not alter any visible behavior (see Section 1.1 [Versioned packages], page 3).

3.2 Installing additional packages

Locally-developed packages can be added to existing libraries, or new libraries can be created to hold them. A new library is typically created as a new subdirectory of the directory specified by the `--prefix` parameter when Eli was configured.

For example, you might create the directory '`/opt/eli/Packages`' to hold additional packages that should be a part of the set from which every cache was built. If you then set the environment variable `ODINPATH` to `/opt/eli/Packages`, the command `eli -R` would create a cache from those packages, the standard Eli packages, and the standard Odin packages (see Section 2.2 [Packages in the cache], page 6).

You can add a package directory or package collection directory '`dir`' to a library directory '`lib`' by the following Eli request:

```
-> dir +d_dest=(lib) :installpkg
```

Directory '`dir`' must be built (if necessary) and all irrelevant files (e.g. '`Makefile`', '`README`', '`CVS`', source files) removed before this request is made. If '`dir`' is a package directory, it must contain '`*.dg`' and '`version`' files; if it is a package collection directory, it must contain a '`PKGLST`' file (see Section 1.1 [Versioned packages], page 3).

Packages installed via `:installpkg` behave exactly like the standard versioned packages: repairs will be effective in existing caches but changes in behavior will not (see Section 3.1 [Installing Eli], page 7).

4 Examples of Package Use

Each example uses a cache directory with an explicit name given by a `-c` parameter. If you don't generally work on more than one project at a time, then you may wish to omit the `-c` parameter and use the same cache for every project (see Section 2.1 [Name of the cache], page 5).

All of these examples assume that you have installed eli in directory '`/opt/eli`' (see Section 3.1 [Installing Eli], page 7).

4.1 Using a single additional package

The Eli packages understand LALR(1) grammars and how to use them to generate deterministic parsers. Unfortunately, some languages are very difficult to describe using such grammars.

A package named `bisonglr` is available from the Eli web site. It uses the Bison parser generator's "GLR" facility to create parsers for grammars that cannot easily be rewritten to make them LALR(1). This package assumes that you have the Bison parser generator available in your path, under the name `bison`. No change in specifications is required; the package handles all of the necessary translations and integration.

Suppose that you have downloaded the `bisonglr` package and unpacked it to your home directory. Then the following command builds a cache called `GLRCACHE` in your home directory. Assuming that `$ODINPATH` is empty, `GLRCACHE` is built from the `bisonglr` package plus the standard Eli and Odin packages (see Section 2.2 [Packages in the cache], page 6).

```
eli -c $HOME/GLRCACHE -R -p $HOME/bisonglr
```

Note that absolute path names are used for both the cache directory and the package directory.

If you frequently worked on languages whose grammars required the `bisonglr` package, you could add it to the Eli package library by making the request:

```
-> $HOME/bisonglr +d_dest=(/opt/eli/lib/Eli) :installpkg
```

After making this request, the next time you build a cache the standard Eli packages will include `bisonglr`.

4.2 Maintaining distributed packages

The text for the Eli system is kept in a CVS repository. For the purposes of this example, assume that you have checked out a copy of the distribution directory '`Elidistrib`'. The name of your working directory will therefore be '`Elidistrib`'. The directory containing the standard Eli package collection is then '`Elidistrib/Eli/pkg`', and the directory containing the standard Odin package collection is '`Elidistrib/Odin/pkg`'.

You will use your working directory to develop and test changes to the standard packages, ultimately checking in the final version. During the development process, you want to run Eli with certain package directories from your working directory substituted for the corresponding directories in '`/opt/eli`'.

For example, suppose that your working directory is a subdirectory of your home directory and you need to debug the `gla` package. The following command will build the cache

'DBGCACHE' in which the gla package from your working directory takes the place of the standard gla package:

```
eli -c $HOME/DBGCACHE -R -p $HOME/Elidistrib/Eli/pkg/gla
```

If you need to work on several packages at once, a good strategy is to create a new package collection directory and populate it with symbolic links to package directories in your working directory:

```
mkdir $HOME/pkg
ln -s $HOME/Elidistrib/Eli/pkg/gla    $HOME/pkg
ln -s $HOME/Elidistrib/Eli/pkg/parser $HOME/pkg
ln -s $HOME/Elidistrib/Eli/pkg/pgs    $HOME/pkg
...
```

You then put the names of the packages you want to work on into '$HOME/pkg/PKGLST' and build the debug cache with the following command:

```
eli -c $HOME/DBGCACHE -R -p $HOME/pkg
```

Once you have finished debugging, you can leave the 'pkg' directory with its symbolic links in place. When you need to debug other packages, you add any new links and edit 'PKGLST' to name the packages on which you want to work. Links to currently-uninteresting packages remain for future debugging sessions, but because their names are not listed in 'PKGLST' they are ignored.

4.3 Developing additional packages

A package being developed requires a package directory. It is often useful to make that directory a subdirectory of the '$HOME/pkg' directory (see Section 4.2 [Maintaining distributed packages], page 9). It can then be used either singly or in combination with other packages being debugged.

All of the additional packages developed to date are kept in the Packages directory of the CVS repository. If you are developing an additional package for Eli, you might decide to check out the Packages directory and develop your package in a subdirectory of the resulting working directory. This is generally *not* a good idea. The reason is that you will ultimately want to use a CVS import command to add your package to the repository. That command expects a directory distinct from your working directory. A subsequent checkout with the -d parameter will add the imported directory to your working directory. If the package directory that you developed is a subdirectory of your working directory, it will have to be moved before the checkout takes place. Thus it is better develop the package in a directory (such as '$HOME/pkg') that is disjoint from your working directory.

5 Odin Execution

Odin is designed according to a client/server architecture. The server is responsible for maintaining the cache, and the client is responsible for interacting with the user and executing those tools not built into Odin. Normally the client and the server run as separate processes, possibly on different machines, communicating via network sockets.

5.1 Client execution

When you give an `eli` command, it starts an Odin client. The client determines a specific cache from the command (see Section 2.1 [Name of the cache], page 5). It then checks that cache to see whether a socket is associated with it. If not, the client establishes a socket for the cache and forks (see Section 5.3 [Interprocess communication], page 11). The forked process becomes the server for that cache, and listens for clients on the socket.

The client connects to the server via the socket. It continues to interact with you, accepting input and delivering reports. Whenever you make a request, the client packages that request and forwards it to the server. If the server needs to invoke a tool that is not built into Odin, then it sends an appropriate command line to the client to be executed.

Each client may have a colon-separated list of machines as the value of its `BuildHosts` variable (see Section "Buildhosts Maxbuilds" in *Eli User Interface Reference Manual*). The name `LOCAL` refers to the machine running the client; it is assumed if no list is given. The client executes the command line given it by the server on the first machine in the list that does not have a currently executing command.

When the client terminates, either because you have responded to a prompt with `C-d` or because the end of the input file has been reached in a batch run, it informs the server of this termination.

5.2 Server execution

A server is associated with a specific cache. The first client started on that cache creates the server by forking (see Section 5.1 [Client execution], page 11). Subsequent clients started on that cache do not fork new servers, but connect to the existing server via the socket already established for the cache. When all clients connected to the server have terminated, the server itself terminates and closes the socket.

The server for a cache receives all requests involving that cache, and is responsible for ensuring that requests involving specific files are sequenced so that all requests are effectively atomic.

Tools built into Odin are also executed by the server. If a client request requires execution of an external tool, the server builds a command line invoking the tool and passes it to the first client that is not currently executing such a command line. Thus the server may involve a client in running tools necessary to satisfy requests from other clients.

5.3 Interprocess communication

When a client is started on a cache (see Section 5.1 [Client execution], page 11), it checks whether the cache contains a file named '`SOCKET`'. '`SOCKET`' is normally a text file containing a single line giving the name of the machine on which the server is running and the port

number to which it is listening (but see Section 5.5 [Socket implementation], page 13). If '`SOCKET`' is not present, then the client creates it and forks a new server for the cache.

The client creates '`SOCKET`' by opening it with the `O_CREAT` and `O_EXCL` flags. If two or more clients are started simultaneously on a cache with no servers, and all attempt to create '`SOCKET`', this combination of flags ensures that only one of the create operations can succeed. Thus one of the clients will create the socket file and fork the server, while the file creation will fail on the others. After the client has determined that a server is present, or after its file creation fails, or after it has successfully forked a server, it connects to the socket defined by the cache's '`SOCKET`' file.

5.4 Execution scenarios

The simplest execution scenario involves a single person working on a single computer. There may be a number of caches, representing different projects, but normally the user will work with only one at a time. One client process and one server process will be running whenever the user is actually interacting with Odin.

A slightly more complex situation involves a team, all of whom run on the same machine. Again, the team may be working on several different projects so there may be several caches. More than one member of the team may, however, be working with a particular cache at the same time. In that case, there will be one server process associated with each active cache, and one client process associated with each active user. The server process will mediate all of the client requests on a particular cache to guarantee atomicity. If two clients' requests have products in common, each of those products will only be built once because the server knows that once the product is up-to-date it need not be rebuilt.

Students working on individual projects, using a collection of networked computers with a shared file system, can treat the situation as though they were running on a single machine. A small problem is that if they do not specify a cache explicitly with the `-c` parameter, they will have a distinct cache on each of the machines they use. In order to avoid this duplication, we advocate use of the `ODINVIEW` environment variable to define a single default cache name (see Section 2.1 [Name of the cache], page 5).

The most complex scenario is exemplified by students working in teams in a lab with multiple machines and a shared file system. Presumably each team member is sitting at a different machine. One student will start a client, resulting in a server running on that student's machine. Now a second student starts a client on a different machine, but referring to the project's common cache. At that point, one server and two clients are running. The server and one of the clients are both running on the first student's machine, and the other client is running on the second student's machine.

Having clients run on different machines improves performance. Remember that the client is responsible for executing tools that are not built into Odin, so running clients on different machines allows such tools to operate in parallel. Since much of the cost of a given derivation is the cost of running tools, this is an important benefit. Another way of executing tools on different machines is to use a `BuildHosts` list (see Section 5.1 [Client execution], page 11).

An Odin server makes very heavy use of cache files, many of which are quite small. Experience has shown that there is a serious performance degradation when the server must access the cache over the network. Thus it is useful to run an Odin server on the

computer where the files are actually stored. The educational computer laboratory again provides a good example: the shared file system is often stored on a central computer that serves the information to the workstations in the laboratory. One could start a session for the desired cache on the central computer, and simply leave that session open. Subsequent sessions for that cache, started on workstations, would use the server running on the central computer.

5.5 Socket implementation

Odin supports two socket implementations: unix internet (`AF_INET`) sockets and unix domain (`AF_UNIX`) sockets. The default is `AF_INET`; `AF_UNIX` is used only if the environment variable `ODIN_LOCALIPC` is set to `1`.

`AF_INET` sockets can handle all of our execution scenarios (see Section 5.4 [Execution scenarios], page 12). When the environment involves a file system shared by several machines, however, `AF_UNIX` sockets can be used if and only if all interactions with a given cache at a given time involve a single machine. This restriction imposes an intolerable burden on a team, because each team member must find out whether a server is already running, and if so which machine it is running on, before they can start their own client. It also makes execution of a server on a central machine and clients on associated workstations impossible. Thus `ODIN_LOCALIPC` should be set to `1` only if Eli will not function otherwise.

The socket implementation is reflected in two cache files, '`SOCKET`' and '`ENV`'. If the socket implementation is `AF_UNIX`, then '`SOCKET`' is the actual socket on which the server is listening for client connections rather than a text file (see Section 5.3 [Interprocess communication], page 11).

'`ENV`' stores the values of important environment variables (including that of `ODIN_LOCALIPC`) that were in effect at the time that the cache was created or last reset by a `-R` parameter (see Section 2.2 [Packages in the cache], page 6). Thus a specific socket implementation technique is associated with each cache individually. That technique is used whenever a session begins on the cache; it can be changed only by establishing the appropriate value of `ODIN_LOCALIPC` and resetting the cache via the `-R` parameter.

Interprocess communication failures in Eli can occur in two contexts: communication between client and server, and communication between a client and a child running a command. A failure to communicate between client and server almost always results in the error report `Cannot connect to Odin server` when making the system originally. If there have been no other difficulties (e.g. stopping and restarting the make, re-configuring, etc.), then you should try the following:

```
make clean
ODIN_LOCALIPC=1 make
```

Please report all instances in which setting `ODIN_LOCALIPC`=1 is necessary to build the system (see Chapter 6 [Procedure for reporting a problem with Eli], page 15).

If you are able to build the system initially without setting `ODIN_LOCALIPC`=1, but the report `Cannot connect to Odin server` appears during normal operation, the problem is almost certainly that the server has been killed before it had the chance to remove file '`SOCKET`' from the cache. Resetting the cache with either the `-r` or `-R` parameter should fix this.

A failure to communicate between a client and a child running a command usually results in the system simply locking up. Lockups could also conceivably occur in the socket communication, so it is important to try to eliminate that possibility. If the parameter `-s` is given on the `eli` command line then only a single process, acting as both the client and the server, is used. The program still establishes the '`SOCKET`' file in the cache, but sockets will remain unused if you don't start any other clients on the same cache. Any lockups that occur in this mode of operation *cannot* be due to client/server interaction.

If the lockups are intermittent, be certain to make several test runs with the `-s` command line parameter before concluding that client/server interaction is the culprit.

6 Problem Reporting

Should you have a problem with Eli please e-mail a description to:

```
eli-project-users@lists.sourceforge.net
```

When reporting problems, please include the Eli version number (printed when Eli starts up, or available in the distribution in '`Eli/pkg/version/version.dg`') and the output from running the shell command `uname -a` on your system. Also, please give *complete* output illustrating the problem. If necessary, please include relevant specification files.

In some cases it will also be helpful to identify the version of the software in question. Most distributed files have RCS tags embedded in them. Most executables are generated from sources that have an initialized variable containing version information:

```
char rcsid[] = "$Id: misc.c,v 1.2 88/11/30 12:57:39 bob Exp $";
```

It is easy to identify the source versions from which binaries have been generated by running the RCS `ident` utility on them. For example, running `ident` on '`pkg/gla/gla_fe`' would yield something like the following:

```
gla_fe:
     $Id: driver.c,v 1.15 1996/03/22 19:28:48 kadhim Exp $
     $Id: dfltclp.c,v 1.8 1994/12/08 03:09:52 tony Exp $
     $Id: source.c,v 2.5 1995/01/16 02:19:21 waite Exp $
  ...
```

Sites that do not have `ident` can extract this information by one of the following commands:

Source `grep '\$Id:' 'file-name'`

Executable

`strings 'file-name' | grep '\$Id:'`

Index