

ROSE Installation Guide:

January 31, 2012

January 31, 2012

0.1 ROSE Installation

0.1.1 Software/Hardware Requirements and Options

ROSE is general software and we ultimately hope to remove any specific software and hardware requirements. However, our goal is to be specific about where and how ROSE is developed and where it is regularly tested.

Required Hardware/Operating System

ROSE has been mainly developed on Linux/Intel platforms. The best supported platforms (including both 32-bit and 64-bit versions) include:

- Red Hat Enterprise Linux Client release 5.5 (Tikanga) or the open source equivalent, CentOS 5.5;
- Mac OS X 10.6.

Other Linux distributions should also work with ROSE. ROSE is being tested on about 20 different versions of Linux and Mac OS X. See section `refNML_testing` for current status. A port to Visual Studio is ongoing work. We will in time expand the portability of ROSE to other platforms as required.

If you have a specific requirement for ROSE to be ported to a target platform please let us know.

Software Requirements

- **ROSE:**

There are three distributions of ROSE: the *Distribution Version* for users (typical), the *External Development Version* for advanced users and collaborators, and the *Internal Development Version*

- **Distribution Version**

Provided as a tared and compressed file named `ROSE-0.9.5a.tar.gz`. It can be obtained from <https://outreach.scidac.gov/projects/rose>. This is the most typical way that users will see and work with ROSE. But it is less up-to-date compared to development versions.

- **External Development Version:git**

You can obtain it by typing `git clone http://www.rosecompiler.org/rose.git`

It enables people to have the quickest access to the most recent new features in ROSE. The external git repository is synchronized with the internal repository once its master branch is updated. It is highly recommended to be used by collaborators who regularly contribute back to us.

- **External Development Version: subversion**

It is available from the SciDAC Outreach Center's subversion repository: <https://outreach.scidac.gov/projects/rose>. It contains only a subset of ROSE (excluding the EDG part essentially) of the internal developer version. We are trying to gradually phase out the subversion repository to facilitate merging external contributions.

- **Internal Development Version: git repositories**

Internally, we maintain two git repositories internally, one for ROSE and the other for EDG source code. The EDG repository is linked to the ROSE repository as a git submodule. The details of getting

and building this version are located in the Developer Guide. This distribution is intended only for ROSE development team and external developers with access to our internal network file system.

- **git**: Required if you work with development versions of ROSE using git repositories.
- **subversion**: Required if you work with development versions of ROSE using subversion repositories.
- **wget**: Required for all external distribution/development versions.
Only the internal git development version has access to EDG source code. For all other cases, wget is used to automatically download prebuilt EDG binaries matching your platform, if the configure script detects that your platform is supported.
- **g++** : version 4.0.x to 4.4.x (Support for GCC 4.5 and higher versions is still ongoing work). For Mac OSX, please install XCode to have g++. .
- **gfortran**: version 4.2.x to 4.4.x. Required if you want to handle Fortran input. For Mac OS X 10.6, gfortran is not included in XCode. We use Apple Xcode gcc-42 (gfortran) add-on provided at <http://r.research.att.com/tools/> to get gfortran installed. Be sure to use a package matching your gcc build number there.
- **BOOST** : version 1.36.0 to 1.45.0
Visit www.boost.org for more details about BOOST and www.boost.org/users/download for download and installation instructions. Higher versions of boost may or may not work with ROSE. We only test rose using boost 1.36.0 to 1.45.0 (10 versions) right now due to resource limits. *Installation of Boost is such a common issue that we include simple directions for how to install Boost in section 0.1.2*
- **JAVA** : version >1.5.0_11
A SUN Java virtual machine (JVM) is needed. A Java compiler (JDK) is also required for development versions.
- **Autoconf** : version >= 2.59. Needed *ONLY* for development versions.
Autoconf is an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages.
- **Automake** : version >= 1.96. Needed *ONLY* for development versions.
Automake is a tool for automatically generating 'Makefile.in' files compliant with the GNU Coding Standards.
- **GNU Libtool**: version >=1.5.6. Needed *ONLY* for development versions. For MAC OS X, the default libtool used is not a GNU libtool. Please install GNU libtool and use it instead.
- **GNU Flex**: Needed *ONLY* for development versions.
Flex is used to generate annotationOpenMP directive scanners in ROSE.
- **GNU Bison**: Needed *ONLY* for development versions.
Yacc is used to generate some annotationOpenMP directive parsers in ROSE.
- **Doxygen** : Needed *ONLY* for development versions.
Most ROSE documentation is generated using LaTeX and Doxygen, thus Doxygen is required for ROSE developers that want to regenerate the ROSE documentation. This is not required for ROSE users, since

all documentation is included in the ROSE distribution. Visit www.doxygen.org for details and to download software. There are no ROSE-specific configure options to use Doxygen; it must only be available within your path.

- **ghostscript:** Needed *ONLY* for development versions. ps2pdf is need to generate ROSE documentations.
- **DOT (GraphViz) :** Required since many ROSE tests generate AST/CFG graph in dot format. ROSE uses DOT for generating graphs of ASTs, Control Flow, etc. DOT is also used internally by Doxygen. Visit www.graphviz.org for details and to download software. An example showing the use of the DOT to build graphs is in the ROSE Tutorial. There are no ROSE-specific configure options to use dot; it must only be available within your path.
- **LaTeX: texlive:** Needed *ONLY* for development versions. LaTeX is used for a significant portion of the ROSE documentation. LaTeX is included on most Unix systems. There are no ROSE specific configure options to use LaTeX; it must only be available within your path.
- **zgrviewer:** This is needed for ROSE Demos. We use it to visualize the AST tree saved in dot format. Zgrviewer is a dot file viewer that allows simple zooming, panning, etc. It is available on SourceForge. Zgrviewer is a java program and installs easily. however the detail memory setting for java are insufficient for large dot files generated within ROSE (and by the ROSE demos). We suggest a number of options to execute zgrviewer with more memory using java. A zgrviewer script (called zgrviewerExampleScript) is available in the ROSE/scripts directory (see the script for details).

Optional Software:

More functionality within ROSE is available if one has additional (freely available) software installed:

- **libxml2-devel :** Optional for some features. Several optional features of ROSE need to handle XML files, such as roseHPCT and BinaryContextLookup.
- **SQLite :** ROSE users can store persistent data across separate compilation of files by storing information in an SQLite database. This is used by several features in ROSE (call-graph generation, for example) and may be used directly by the user for storage of user-defined analysis data. Such database support is one way to handle global analysis (the other way is to build the whole application AST). Visit www.sqlite.org for details and to download software. An example showing the use of the ROSE database mechanism is in the ROSE Tutorial. Use of SQLite requires special ROSE configuration options (so that the SQLite library can be added to the link line at compile time). See ROSE configuration options for more details (`configure --help`).
- **mpicc :** mpicc is a compiler for MPI development. If ROSE is configures with MPI enabled, one can utilize features in ROSE that allow for distributed parallel AST traversals.
- **Haskell (ghc) Compiler:** ROSE supports Haskell bindings so that ROSE tools can be written in Haskell. ROSE does not parse or permit source-to-source analysis and transformation of Haskell language. So we support Haskell but only in this specific way. See directions below for how to install the Haskell (ghc) compiler.

0.1.2 Building BOOST

The following is a quick guide on how to install BOOST. For more details please refer to www.boost.org. *Note that the install process is different for boost versions starting with 1.39.*

For Boost versions 1.36 through 1.38:

1. Download BOOST.
Download BOOST at www.boost.org/users/download.
2. Untar BOOST.
Type `tar -zxf BOOST-[VersionNumber].tar.gz` to untar the BOOST distribution.
3. Create a separate install tree.
Type `mkdir installTree` to create a location for the install files to reside (e.g. BOOST_INSTALL).
4. Run the `configure` script.
Type `./configure --prefix=[installTree]` to run the BOOST `configure` script. The path to the `configure` script may be either relative or absolute. The `prefix` option specifies the installation directory (e.g. BOOST_INSTALL).
5. Run `make`.
Type `make` to build all the source files.
6. Run `make install`.
Type `make install` to copy all build files into the install directory. BOOST is now available in your `installTree` (e.g. BOOST_INSTALL) to be used by ROSE.

Starting with Boost 1.39, Boost has a slightly different built process which will locted directories used by ROSE in to different locations (not copy them to the install tree). For Boost versions 1.39 or greater:

1. Download BOOST.
Download BOOST at www.boost.org/users/download.
2. Untar BOOST.
Type `tar -zxf BOOST-[VersionNumber].tar.gz` to untar the BOOST distribution.
3. Create a separate install tree.
Type `mkdir installTree` to create a location for the install files to reside (e.g. BOOST_INSTALL).
4. Run the `bootstrap.sh` script.
Type `./bootstrap.sh --prefix=[installTree]` to run the BOOST `bootstrap.sh` script. The path to the `bootstrap.sh` script may be either relative or absolute. The `prefix` option specifies the installation directory (e.g. BOOST_INSTALL). Note that `./bootstrap.sh --help` can be used to provide more information about how to install Boost.
5. Run `bjam`.
Type `./bjam install --prefix=[installTree]` to compile boost and copy all build files into the install directory. If you're compiling ROSE with the flag `_GLIBCXX_DEBUG`, you also need to use the flag when compiling boost; in that case the build command would be `./bjam debug define=_GLIBCXX_DEBUG install --prefix=[installTree]`. The BOOST library is now available in your `installTree` (e.g. BOOST_INSTALL) to be used by ROSE.

6. **Important:** In running `configure` to build ROSE it is sometimes also required to use the `--with-boost-libdir=BOOST_INSTALL/lib` in addition to the `--with-boost=BOOST_INSTALL` mentioned immediately below.

Use the `BOOST_INSTALL` directory on the `configure` line for ROSE using: `--with-boost=BOOST_INSTALL`.

Important: Don't forget to set your `LD_LIBRARY_PATH` environment variable to include the Boost lib (and sometimes `lib64`) directories.

Additional notes:

1. Note that the installation of Boost will frequently output warnings (e.g. `*(Unicode/ICU support for boost.regex?...not found).*`, these can be ignored.
2. Later versions of GNU will not work with older versions of Boost; as a data point the GNU g++ version 4.4.1 does not appear to ply well with Boost 1.37 (but works with Boost 1.42 just fine).

0.1.3 Using Insure++

It is possible to configure ROSE to use insure++ for static and dynamic analysis of the ROSE source code as part of development of ROSE or ROSE based tools. This is a feature that is experimental, and currently not working well (because our older version of insure++ (version 7.1.6) is failing to compile the Linux header files used in ROSE). The `configure` option to turn on the use of insure++ is: `-enable-insure`. This is all that is required if insure is in your path (at LLNL run: `source /usr/apps/insure++/default/setup.csh` for csh and `source /usr/apps/insure++/default/setup.sh` for bash). *Don't use Insure++ until you have some experience with ROSE development, it makes everything more complex..*

0.1.4 Building ROSE From a Distribution (ROSE-0.9.5a.tar.gz)

To simplify the descriptions of the build process, we define:

- **Source Tree**
Location of source code directory (there is only one source tree).
- **Compile Tree**
Location of compiled object code and executables. There can be many compile trees representing either different: `configure` options, compilers used to build ROSE and ROSE translators, compilers specified as backends for ROSE (to compile ROSE generated code), or architectures.

We *strongly* recommend that the **Source Tree** and the **Compile Tree** be different. This avoids many potential problems with the `make clean` rules. Note that the **Compile Tree** will be the same as the **Source Tree** if the user has *not* explicitly generated a separate directory in which to run `configure` and compile ROSE. If the **Source Tree** and **Compile Tree** are the same, then there is only one combined **Source/Compile Tree**. Alternatively, numerous different **Compile Trees** can be used from a single **Source Tree**. More than one **Compile Tree** allows ROSE to be generated on different platforms from a single source (either a generated distribution or checked out from svn/git repos). ROSE is developed and tested internally using separate **Compile Trees**.

The process for building ROSE from a ROSE *Distribution Version* is the same as for most standard software distributions (e.g those using autoconf tools). Specific details for particular operating system distributions can be found in various README files in the top of the source tree. In general:

1. Untar ROSE.

Type `tar -zxf ROSE-0.9.5a.tar.gz` to untar the ROSE distribution.

2. Build a separate compile tree.

Type `mkdir compileTree` to build a location for the object files and documentation (use any name you like for this directory).

3. Change directory to the new compile tree directory.

Type `cd compileTree; .` This changes the current directory to the newly created directory.

4. Add JAVA environment variables if your platform is a Linux box.

For example:

```
export JAVA_HOME=/usr/apps/java/jdk1.5.0_11
export LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/i386/server:$LD_LIBRARY_PATH
```

5. Add the Boost library path into your LD_LIBRARY_PATH. We put the path in front of any other paths to avoid some old versions of boost installation being picked up.

For example: For Linux, `LD_LIBRARY_PATH=/yopath/boost_1_36_0/lib:$LD_LIBRARY_PATH`

For MAC OS X: `export DYLD_LIBRARY_PATH=/yopath/boost_1_36_0-inst/lib:$DYLD_LIBRARY_PATH`

6. Run the configure script.

Type `{AbsoluteOrRelativePath}/configure --prefix=/yourpath/rose-inst --with-boost=[BOOST_installTree]` to run the ROSE configure script. The path to the configure script may be either relative or absolute. The prefix option on the configure command line is only required if you run `make install` (suggested), because the default location for installation is `/usr/local` and most users don't have permission to write to that directory. This is common to all projects that use autoconf. ROSE follows the GNU Makefile Standards as a result of using autoconf and automake tools for its build system. As of ROSE-0.8.9a, the default setting for the install directory (prefix) is the build tree. For more on ROSE configure options, see section 0.1.7. *Note: If configure is not present, you may have a development version. In that case, see the next section (0.1.5) for installation instructions.*

7. Run make.

Type `make` to build all the source files. See details of running `make` in parallel in section 0.1.8.

8. To test ROSE (optional).

Type `make check` to test the ROSE library against a collection of test codes. See details of running `make` in parallel 0.1.8.

9. To install ROSE, type `make install`.

Installation is optional, but suggested. Users can simplify their use of ROSE by using it from an installed version of ROSE. This permits compilation using a single include directory and the specification of only two libraries. See details of installing ROSE in section 0.1.9.

10. Testing the installation of ROSE (optional).

To test the installation and the location where ROSE is installed, against a collection of test codes (the application examples in `ROSE/tutorial`), type `make installcheck`. A sample `makefile` is generated.

0.1.5 Building ROSE from a Development Version (from SVN or GIT)

Building ROSE from an internal or external development version is very similar to building it from a distribution. The major difference is that for development versions, there is no configure script available .

You have to type `./build` in the source tree to generate the configure script and Makefile.ins. Once this is done, the rest steps are the same as those of building a distribution version.

0.1.6 Troubleshooting the ROSE Installation

There are a number of famous ways to screw up your installation of ROSE.

1. Message: **configure: error: Could not link against boost_filesystem-gcc41-nt**
 This message from running the `configure` command in ROSE (an initial step in building ROSE) indicates that your `LD_LIBRARY_PATH` (environment variable) is not set to to the location of the boost install tree. The ROSE configure scripts (autoconf) will test the linking to specific boost libraries and this is the first dynamic link library that it tests and so it will fail when many other tests on boost succeed because your `LD_LIBRARY_PATH` is finally required and is not properly set.
2. Message: **Making all in libltdl**
`make[2]: *** No rule to make target 'all'. Stop.`
 Run `glibtoolize --force` to rebuild the libtool support in ROSE for your machine at the top level of the source tree. If that does not work then give up on the libtool that came with the apple dev tools and just build your own libtool in your home directory.
3. **Don't build ROSE in the source tree, it is not tested often, but it should work.**
 Save yourself some trouble and build a separate compile tree. This will also allow you to build a number of different versions of ROSE with different options.
4. Message: **configure: error: Unable to find path to JVM library**
 This message from running the `configure` command in ROSE (an initial step in building ROSE) indicates either that your `LD_LIBRARY_PATH` (environment variable) is not set to to the location of the `libjvm.so` or that your machines java is not one that we support (e.g. non-Sun Java). If you don't require Java (e.g. don't need Fortran support) then consider skipping the java support by using `-without-java` on the configure command line. Alternatively, your `LD_LIBRARY_PATH` should contain the path to the file `libjvm.so`. The likely path is specified in the lines just before the message. The full message will appear as:

```

checking for Java... /usr/lib/jvm/java-1.5.0-ibm.x86_64/bin/./bin/java
checking for Java JVM include and link options... JavaJREDir = /usr/lib/jvm/java-1.5.0-ibm-1.5
JavaHomeDir = /usr/lib/jvm/java-1.5.0-ibm-1.5.0.8.x86_64
JavaJVMDir = /usr/lib/jvm/java-1.5.0-ibm-1.5.0.8.x86_64/jre/bin/classic
configure: error: Unable to find path to JVM library

```
5. Previously installed version of Boost library.
 Some machines have a default version of Boost already installed (for example in `/usr/include/boost`). This always the wrong version since the OS installation of Boost lags by several years. ROSE now attempts to detect this and use the

-isystem g++ option to have the explicitly specified version of boost from the configure command-line be search before the system include directories. This works well where a machine has a previously installed version of Boost, but it will fail when used with SWIG (so don't use `--with-javaport` where a previous system installation of Boost is detected). The ROSE configure scripts will detect the presence of a previously installed version of Boost and issue a warning message to not use `--with-javaport`. Also if no previously installed version of Boost is detected the configuration will report this as well and make clear that it will use the Boost include directory with a `-I` option.

6. libtoolize not available (or old version)

The problem is that ROSE is calling `libtoolize` or `glibtoolize` and it seems that you don't have it on your machine (called by the build script). You will need it, it is a requirement. The build script will run this to build you the required libtool support. Since this happens upstream of `configure` we don't have a test for it. The clue is the output:

```
ls: cannot access libltdl/*: No such file or directory
libtoolize: cannot list files in '/usr/share/libtool/libltdl'
```

If you build libtool on your machine and add the installed libtool `bin` directory to your path, then it should work. I often use `libtool-2.2.4.tar` when I have this problem on a new platform.

Report from user:

```
Reason for the problem: I am not building libtools from source, instead using
the packages from the Linux distribution repository. On my distribution
(Ubuntu 8.04, X86_64), the libltdl3 (and libltdl3-dev) does not come with
the libtool package. After installing the libtools, I still need to install
both the libltdl3 and libltdl3-dev package. That is the issues of unable to
find libltdl folders.
```

7. ROSE fails to compile after `svn update`:

We have seen this problem and had it reported and we don't understand it. It does however disappear after a fresh checkout from SVN into an empty directory. If you figure this out please let us know. Where this has happened to us, we were using `svn` version 1.4.6, where as our `svn` repository is more commonly (within development) had work checked in using `svn` version 1.5.1; since a lot changed from `svn` version 1.4 to version 1.5, this may be the issue.

```
make[2]: Entering directory '<Your ROSE compile tree path>/src/frontend/SageIII'
  COMPILING preproc.lo
/home/dquinlan/ROSE/svn-rose/src/frontend/SageIII/preproc.lex: In function 'ROSEAttributesList* getPreprocessorDirectives(std::string)':
/home/dquinlan/ROSE/svn-rose/src/frontend/SageIII/preproc.lex:961: error: conversion from 'std::_Rb_tree_iterator<std::pair<const std::string, ROSEAttributesList*>>' to 'std::string' is ambiguous
/home/dquinlan/ROSE/svn-rose/src/frontend/SageIII/preproc.lex:963: error: no match for 'operator!=' in 'iltr != (&mapFilenameToAttributes)->std::map<std::string, ROSEAttributesList*>'
/home/dquinlan/local/gcc/3.4.3/bin/../../../../lib/gcc/i686-pc-linux-gnu/3.4.3/../../../../include/c++/3.4.3/bits/stl_tree.h:213: note: candidates are: bool operator!=(const std::string&, const std::string&) const
/home/dquinlan/ROSE/svn-rose/src/ROSETTA/Grammar/Node.code:50: note:      bool operator!=(const rose_rva_t&, const rose_rva_t&) const
/home/dquinlan/ROSE/svn-rose/src/ROSETTA/Grammar/Support.code:3984: note:      bool operator!=(const Sg_File_Info&, const Sg_File_Info&) const
make[2]: *** [preproc.lo] Error 1
make[2]: Leaving directory '<Your ROSE compile tree path>/src/frontend/SageIII'
make[1]: *** [all-recursive] Error 1
make[1]: Leaving directory '<Your ROSE compile tree path>/src/frontend/SageIII'
make: *** [all] Error 2
```

0.1.7 ROSE Configure Options

A few example configure options are:

- Minimal configuration

```
../ROSE/configure --with-boost=[BOOST_installTree]
```

This will configure ROSE to be compiled in the current directory (separate from the **Source Tree**). The installation (from `make install`) will be placed in `/usr/local`. Most users don't have permission to write to this directory, so we suggest always including the *prefix option* (e.g. `--prefix='pwd'`).

- Minimal configuration (preferred)

```
../ROSE/configure --prefix='pwd' --with-boost=[BOOST_installTree]
```

Configure in the current directory so that installation will also happen in the current directory (a `install` subdirectory will be built).

- Debugging

Note that all possible debug modes are turned on by default. These include:

- `-g` (compiler debug flag), and
- `-Wall` (compiler warnings flag).

Additional modes are possible, one of which is now tested within ROSE internal release level testing:

- `-D_GLIBCXX_DEBUG` (stl and other stdlib debug flag (used)),
- `-D_GLIBCXX_DEBUG_PEDANTIC` (another stl debug flag we consider excessive (not used)).

But these last flags require a specially compiled version of Boost (because they change the size of iterators). The use of these tests are equivalent to the configure line:

```
../ROSE/configure --with-CXX_DEBUG="-g -D_GLIBCXX_DEBUG -D_GLIBCXX_DEBUG_PEDANTIC"
--with-CXX_WARNINGS=-Wall --prefix='pwd' --with-boost=[BOOST_installTree]
```

Configure as above, if you like, but with debugging and warnings turned on. Note that if you compile ROSE with `_GLIBCXX_DEBUG`, you should also compile boost with the same debug flag (Section 0.1.2). As mentioned the default is to turn on the use of `-g` and `-Wall` only.

- Adding Fortran support

```
../ROSE/configure --prefix='pwd' --with-boost=[BOOST_installTree] --with-java
```

The Open Fortran Parser will also be enabled, allowing ROSE to process Fortran code. The programs `java`, `javac`, and `jar` must be either in your `PATH` or in `$JAVA_HOME/bin`.

- Adding SQLite support

```
../ROSE/configure --with-sqlite3=/home/dquinlan/SQLite/sqliteCompileTree --prefix='pwd'
--with-boost=[BOOST_installTree]
```

Configure as above, but permit use of SQLite database for storage of analysis results between compilation of separate files (one type of support in ROSE for global analysis).

- Adding parallel distributed memory analysis support (using MPI)

```
../ROSE/configure --prefix='pwd' --with-mpi --with-gcc-omp --with-boost=[BOOST_installTree]
```

Configure as above, but with MPI and OpenMP support for ROSE to run AST traversals in parallel (distributed and shared memory).

- Adding IDA Pro support

```
../ROSE/configure --prefix='pwd' --with-binarysql --with-boost=[BOOST_installTree]
```

The `binarysql` flag allows ROSE to read a binary file previously stored as a sql file (e.g. fetched from IDA Pro).

- Adding support for SWIG (Python connection)

```
../ROSE/configure --prefix='pwd' --with-javaport=yes SWIG=swig --with-boost=[BOOST_installTree]
```

`--with-java`

This allows ROSE to be build with javaport, a support that connects ROSE to Java via SWIG. The Eclipse plug-in to ROSE is based on this work.

- Additional Examples

More detailed documentation on configure options can be found by typing `configure --help`, or see figure 0.1.7 for complete listing.

Output of `configure --help` is detailed in Figures 0.1.7 (Part 1) and 0.1.7 (Part 2):

0.1.8 Running *GNU Make* in Parallel

ROSE uses general Makefiles and is not dependent on *GNU Make*. However, *GNU Make* has an option to permit compilation in parallel and we support this. Thus you may use `make` with the `-j<n>` option if you want to run `make` in parallel (a good value for `n` is typically twice the number of processors in your computer). We have paid special attention to the design of the ROSE *makefiles* to permit parallel `make` to work; we also use it regularly within development work.

0.1.9 Installing ROSE

Installation (using `make install`) is optional, but suggested. Users can simplify their use of ROSE by using it from an installed version of ROSE. This permits compilation using a single include directory and the specification of only two libraries, as in:

```
g++ -I{\<install dir>/include} -o executable executable.C
    -L{\<install dir>/lib} -lrose -ledg -lm ${RT_LIBS}
```

See the example makefile in

`ROSE/exampleTranslators/documentedExamples/simpleTranslatorExamples/exampleMakefile`

in Section ?? for exact details of building a translator on your machine (setup by `configure` and tested by `make installcheck`). Note that the tutorial example codes are also tested by `make installcheck` and the `example_makefile` there can also serve as an example.

`autoconf` uses `/usr/local` as the default location for all installations. Only `root` has write privileges to that directory, so you will likely get an error if you have not overridden the default value with a new location. To change the location, you need to have used the `--prefix={install_dir}` to run the `configure` script. You can rerun the `configure` script without rebuilding ROSE.

0.1.10 Haskell Support

Haskell (ghc) is available at <http://www.haskell.org/ghc/>. To install ghc (from the binary distribution) run "configure" and "make install". Specifically (for example):

```
configure --prefix=/home/dquinlan/local/Haskell/ghc-6.10.4_install and make install. (This assumes that the path in your .bashrc includes ghc, e.g.:
```

```
/home/dquinlan/local/Haskell/ghc-6.10.4_install/bin
```

That should be all that is required. ROSE will detect the existence of "ghc" at "configure" time and setup what is required internally. To turn off the use of Haskell support, use the configure option `-with-haskell=no`.

Note that we enforce and require version *6.10.x* of ghc, this is because we assume some default modules that have been removed in more recently released versions of ghc.

0.1.11 Testing ROSE

A set of test programs is available. Type `make check` to run your build version of ROSE using these test codes. Several years of contributed bug reports and internal test codes have been accumulated in the `ROSE/tests` directory.

Extra tests are available for development versions of ROSE. ROSE developers are highly recommended to run `make dist` and `make distcheck` to make sure that the modified development versions can be used to create functioning distributions.

0.1.12 Getting Help

We have three mailing lists for core developers (those who have write access to the internal repository), all developers (anyone who has write access to the internal or external repository) and all users of ROSE. They are:

- `rose-core@nersc.gov`, web interface: <https://mailman.nersc.gov/mailman/listinfo/rose-core>.
- `rose-developer@nersc.gov`, web interface: <https://mailman.nersc.gov/mailman/listinfo/rose-developer>.
- `rose-public@nersc.gov`, web interface: <https://mailman.nersc.gov/mailman/listinfo/rose-public>.

0.1.13 ROSE and the NMI Compile Farm

The NSF Middleware Initiative (NMI) has provides us with time on their system to support the robustness of ROSE across multiple platforms. ROSE is not tested on a wide range of platforms (see table 0.1.13). The prerequisites used for each platform (machine and operating system) are generated in the table from the input test descriptions located in the directory `ROSE/scripts/nmiBuildAndTestFarm/build_configs`.

For More information about NMI, see <http://nmi.cs.wisc.edu/>. To see the details of the ROSE nightly tests click on the link: *Run Results* and select the project, *rose compiler*, from the pull down menu.

NMI OS and machine (platform) Prerequisites for ROSE:

NMI OS and machine (platform) Configure Options for ROSE:

0.1.14 Installation Details for Specific Platforms

These are comments collected from users about what platform specific details they have discovered and shared.

Mac OS X

Mac OS X does not appear to come with `wget` and it should be installed separately so that ROSE can find and download it's required EDG binaries from the web (ROSE web site).

Mac OS X v10.5, Leopard We only build EDG binaries for the default GNU 4.0 compiler.

Mac OS X v10.6, Snow Leopard The non-standard STL hashmap implementation in Mac OS X v10.6 is broken and so ROSE can fail when using this OS. We are working on a fix to use more standard features of STL.

Fedora 11

Some trivia with regard to libtool and building the SVN version of Rose: Fedora (11), at least, has a package, libtool-ltdl-devel that is needed (and separate from the libtool package) to make autoconfiguration work right. The symptom is that the post-configure build enters libltdl, and finds no Makefile and fails immediately. Before running Rose's `./build`, make sure the libtool-ltdl-devel RPM is installed.

Intel C++ Compiler

The Intel compiler can run out of space compiling some of the larger files in ROSE. Although not previously seen by anyone on the ROSE team, one user has reported that the Intel compiler option `-override-limits` was required. As used on the following configure line: `/nfs/casc/aleamr/yana-local/rose/build/rose-sourcetree/configure CXX=icc CC=icc CXXFLAGS=-override-limits -prefix='pwd' -with-boost=jpath to boost` More information is on this option and when to use it is at: <http://software.intel.com/en-us/articles/internal-threshold-was-exceeded>. The problem that this appears to fix is that on some machines the ROSE file `AST_FILE_IO.C` will fail with the error *memory limit error*, this flag to the Intel compiler will fix this (the `mcpcom` process goes above 2.5g memory for this file).

0.1.15 Installing ROSE under Windows

Under Windows ROSE uses CMake. This is a project that is currently under development. As of November 2010 we are able to compile and link the src directory. We are also able to run example programs that link against librose and execute the frontend and backend. *However, this is an internal capability and not available externally yet since we don't distribute the Windows generated EDG binaries that would be required. Also the current support for Windows is still incomplete, ROSE does not yet pass its internal tests under Windows.*

Setup

Under Windows we use the following tools for compilation and development:

- Microsoft Visual Studio (9.0) : We currently use the Debug mode
- CMake 2.8.0 : One only needs to specify the source and build directory. All configuration options should be chosen correctly during makefile generation.
- Bison 2.4.1 : Used by ROSE
- Flex 2.5.4 : Used by ROSE
- Git 1.6.5 : We utilize the git shell for debugging and command line operations
- Boost 1.37 : Boost is required by ROSE

In order to have Hudson test ROSE under Windows, we have additional tools set up:

- `pegeant.exe` : Allows to avoid entering ssh keys when connecting to git repository
- Hudson client : The client can be started through the web page. All that is required is Java to be installed.

Currently under `tux270-0` Flex, `cmake`, `boost` and `Bison` are installed under `c:/ROSE`. In addition, a ROSE test branch for `ROSE-tps` is installed in order to test ROSE under Windows locally without Hudson. The directory `c:/tools` contains `pegeant.exe` and other useful tools like `plink.exe` and `puttygen.exe`. See further notes below. Finally required ssh keys are located in `c:/putty_keys`.

Debugging in VS 8

ROSE compiles now in release mode and debug mode. Debug mode was a challenge before because adding type-information (RTTI) caused the ROSE dll to be too large.

To use VS with debugging information right click on the ROSE.dll project and chose PROPERTIES/C-C++ INFO:

- GENERAL: turn on Debugging Information Format (/Z7) - do not use DB
- Language: Enable Runtime type information
- Linker: Turn off incremental linking (slightly slower but takes too much memory)

To test whether ROSE (src) compiles, links and a test program can run, modify the qualifiedName project in VS:

- General and Language options as above
- Debugging: enter the following under arguments “-help”

This should allow you to run “qualifiedName -help” in debug mode and all options of rose are print out.

Hudson specific

Setting up SSH keys on Windows Server (logged in as hudson-rose):

- Generate the keys:
 - Generate the keys, run “plink -agent tux269” and enter your own login and then password. to login to tux269 and then exit.
- Copy the keys onto tux270-0:
 - Double Click on “My_Computer” icon
 - Goto C:, tools, pageant.exe
 - Double click on pageant.exe
 - Should display a computer with a hat on it in the bottom right corner of the screen.
 - Right click on icon (of computer with a hat on it)
 - click on “Add Key” (this will cause a window to put up called “Select Private Key File”)
 - Goto “My Computer TUX270-0”, “Local Disk(C:)”, “putty_keys”
 - Click on file “id_rsa.ppk” (this will load the private key for use by ssh).
 - Now try to run the git command: `git clone ssh://hudson-rose@tux269/usr/casc/overture/ROSE/git/ROSE.git c:/ROSE/hudson/workspace/test-windows/label/windows-server`

Now in order to run a job from Hudson under Windows we need to start the client on the Windows side. Go to the hudson webpage and chose “Manage Hudson”/”Manage Nodes”/”tux270” and the start the JNLP agent. On the Windows side a client job is active now and whenever the Windows job on Hudson runs the client is activated.

Currently Hudson is configured in a way that when a00-ROSE-from-scratch passes, a downstream is started and a01-ROSE-WINDOWS is started. The a01-ROSE-WINDOWS job will activate the client on tux270-0 and run the Windows test. In addition rose-hdsn-win-1 has been configured as a second Windows test node for Hudson.

Extending ROSE with the Windows SDK

Some functions used in Linux are not available under Windows - but they are available through the Windows SDK. For instance, `realpath()` in linux is available as `PathCanonicalize()` under Windows using the Windows SDK. You need to include : `#include "Shlwapi.h"` and the following library into the project : `shlwapi.lib` (part of SDK). More information can be found at: <http://msdn.microsoft.com/en-us/windows/bb980924.aspx> http://en.wikipedia.org/wiki/Microsoft_Windows_SDK

0.1.16 Options for Static vs. Dynamic Linking of Executables

ROSE will by default build dynamically linked executables. ROSE supports both static and dynamic linking. Some developers want the space savings of dynamic linking (also might link faster) and some want the simplicity of static linked executables (since they can be easily moved in a single step). The configure options to support linking options are:

- dynamic linking (default): `-enable-shared`
- dynamic and static linking (builds the static libraries, but executables are by default linked dynamically): `-enable-static`
- static linking only: `-enable-static -disable-shared`

0.1.17 Options to Control the Size of ROSE Executables

ROSE by default turns on the compiler's generation of debugging information (symbol tables and dwarf2 debug information). This is done to support development which will nearly always be easier with the symbol tables generated with the debugging options (typically "-g"). However, the volume of code in ROSE will cause a lot of debugging information to be generated (around 150Meg, just for the debugging information in a statically linked executable).

To turn off the generation of debug information in the executables use the configure options: `--with-CXX_DEBUG=no --with-C_DEBUG=no` Any other values will use those explicit value to turn on debugging information in the compilation of ROSE.

The sizes of executables built using dynamic linking are always trivially small, the significant different matters when executables are built using static linking (see section 0.1.16). For a statically linked executable (ROSE-based tool) the size with symbol tables (debugging information) can be 150-200 Meg per executable; this value is for the g++ compiler, other compilers will vary in the sizes of the executables they generate. Turning off the debug information generated by g++ will reduce the size of the same executable to be about 40 Meg. Executables can be stripped of symbol table information further using the `strip` utility (Linux). Executables using ROSE that are stripped will be about 25% smaller (about 30 Meg).

Further work in ROSE could likely reduce the size of ROSE based executables, but it is the judgment of the ROSE team that current work work is sufficient to generate small enough executables. If users have need for smaller executables for ROSE based tools, please contact the ROSE team directly.

```

configure --help Option Output (Part 1)

'configure' configures ROSE 0.9.5a to adapt to many kinds of systems.

Usage: ../../configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
-h, --help                display this help and exit
  --help=short            display options specific to this package
  --help=recursive       display the short help of all the included packages
-V, --version            display version information and exit
-q, --quiet, --silent    do not print 'checking...' messages
  --cache-file=FILE      cache test results in FILE [disabled]
-C, --config-cache       alias for '--cache-file=config.cache'
-n, --no-create          do not create output files
  --srcdir=DIR           find the sources in DIR [configure dir or '..']

Installation directories:
--prefix=PREFIX          install architecture-independent files in PREFIX
                        [/usr/local]
--exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                        [PREFIX]

By default, 'make install' will install all the files in
'/usr/local/bin', '/usr/local/lib' etc.  You can specify
an installation prefix other than '/usr/local' using '--prefix',
for instance '--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
--bindir=DIR             user executables [EPREFIX/bin]
--sbindir=DIR           system admin executables [EPREFIX/sbin]
--libexecdir=DIR        program executables [EPREFIX/libexec]
--datadir=DIR           read-only architecture-independent data [PREFIX/share]
--sysconfdir=DIR        read-only single-machine data [PREFIX/etc]
--sharedstatedir=DIR    modifiable architecture-independent data [PREFIX/com]
--localstatedir=DIR     modifiable single-machine data [PREFIX/var]
--libdir=DIR            object code libraries [EPREFIX/lib]
--includedir=DIR        C header files [PREFIX/include]
--oldincludedir=DIR     C header files for non-gcc [/usr/include]
--infodir=DIR           info documentation [PREFIX/info]
--mandir=DIR            man documentation [PREFIX/man]

Program names:
--program-prefix=PREFIX  prepend PREFIX to installed program names
--program-suffix=SUFFIX  append SUFFIX to installed program names
--program-transform-name=PROGRAM run sed PROGRAM on installed program names

X features:
--x-includes=DIR        X include files are in DIR
--x-libraries=DIR       X library files are in DIR

System types:
--build=BUILD           configure for building on BUILD [guessed]
--host=HOST             cross-compile to build programs to run on HOST [BUILD]

Optional Features:
--disable-FEATURE      do not include FEATURE (same as --enable-FEATURE=no)
--enable-FEATURE[=ARG] include FEATURE [ARG=yes]
--disable-dependency-tracking speeds up one-time build
--enable-dependency-tracking do not reject slow dependency extractors
--enable-ssl            Enable use of SSL library (MD5 checksums)
#####
--enable-only-binary-analysis(=yes)
                        Enable ONLY Java support in ROSE (Warning:
                        '--enable-only-binary-analysis=no' and
                        '--disable-only-binary-analysis' are no longer
                        supported)
#####
--enable-only-c(=yes)   Enable ONLY C support in ROSE (Warning:
                        '--enable-only-c=no' and '--disable-only-c' are no
                        longer supported)
#####
--enable-only-cxx(=yes) Enable ONLY C++ support in ROSE (Warning:
                        '--enable-only-cxx=no' and '--disable-only-cxx' are
                        no longer supported)

```

Figure 1: Example output from `configure --help` in ROSE directory (Part 1).

```

configure --help Option Output (Part 2)
#####
--enable-only-fortran(=yes)
    Enable ONLY Fortran support in ROSE (Warning:
    '--enable-only-fortran=no' and
    '--disable-only-fortran' are no longer supported)
#####
--enable-only-java(=yes)
    Enable ONLY Java support in ROSE (Warning:
    '--enable-only-java=no' and '--disable-only-java'
    are no longer supported)
#####
--enable-only-php(=yes) Enable ONLY PHP support in ROSE (Warning:
    '--enable-only-php=no' and '--disable-only-php' are
    no longer supported)
#####
--enable-only-python(=yes)
    Enable ONLY Python support in ROSE (Warning:
    '--enable-only-python=no' and
    '--disable-only-python' are no longer supported)
#####
--enable-only-cuda(=yes)
    Enable ONLY Cuda support in ROSE (Warning:
    '--enable-only-cuda=no' and '--disable-only-cuda'
    are no longer supported)
#####
--enable-only-openssl(=yes)
    Enable ONLY OpenSSL support in ROSE (Warning:
    '--enable-only-openssl=no' and
    '--disable-only-openssl' are no longer supported)
#####
--enable-languages=LIST Build specific languages:
    all , none , binaries , c , c++ , cuda , fortran , java , openssl , php , python
    (default=all)
#####
--enable-binary-analysis
    Enable binary analysis support in ROSE (default=yes)
#####
--enable-c
    Enable C language support in ROSE (default=yes).
    Note: C++ support must currently be simultaneously
    enabled/disabled
#####
--enable-cxx
    Enable C++ language support in ROSE (default=yes).
    Note: C support must currently be simultaneously
    enabled/disabled
#####
--enable-cuda
    Enable Cuda language support in ROSE (default=yes)
#####
--enable-fortran
    Enable Fortran language support in ROSE
    (default=yes)
#####
--enable-java
    Enable Java language support in ROSE (default=yes).
    Note: --without-java turns off support for ALL
    components in ROSE that depend on Java, including
    Java language support
#####
--enable-php
    Enable PHP language support in ROSE (default=yes)
#####
--enable-python
    Enable Python language support in ROSE (default=no)
#####
--enable-openssl
    Enable OpenSSL language support in ROSE (default=yes)
#####
--disable-gcc-version-check
    Disable GCC version 4.0.x - 4.4.x verification check
--enable-rtedupc
    Enable UPC support in ROSE (default=no)
--enable-compass2
    build the Compass2 static analysis tool under
    projects/
--disable-projects-directory
    Disable compilation and testing of the ROSE/projects
    directory
--disable-tests-directory
    Disable compilation and testing of the ROSE/tests
    directory
--disable-tutorial-directory
    Disable compilation and testing of the ROSE/tutorial
    directory
--enable-smaller-generated-files
    ROSETTA generates smaller files (but more of them so
    it takes longer to compile)
--enable-internalFrontendDevelopment
    Enable development mode to reduce files required to
    support work on language frontends

```

Figure 2: Example output from configure --help in ROSE directory (Part 2).

| NMI Platform (OS and Machine) Prerequisites | |
|---|--|
| x86_64_deb_5.0 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3" |
| x86_64_fedora_12-updated | : "boost-1.36.0, libtool-2.2.6b" |
| x86_64_macos_10.5-updated | : "boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3, wget" |
| x86_64_rhap_5 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b" |
| x86_64_rhap_5.2 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b" |
| x86_64_rhap_5.3 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b" |
| x86_64_rhas_4 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63" |
| x86_deb_5.0 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, libxml2-2.7.3" |
| x86_macos_10.4 | : "boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3" |
| x86_rhap_5 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b" |
| x86_rhas_3 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, autoconf-2.59, automake-1.10" |
| x86_rhas_4 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63" |
| x86_sles_9 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3, tar-1.14" |
| x86_suse_10.0 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3" |
| x86_suse_10.2 | : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, libxml2-2.7.3" |

Figure 3: Example NMI machine preques used for nightly tests.

| NMI Platform (OS and Machine) Configure Options | |
|---|--|
| x86_64_deb_5.0 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_64_fedora_12-updated | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_64_macos_10.5-updated | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_64_rhap_5 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_64_rhap_5.2 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_64_rhap_5.3 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_64_rhas_4 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java --disable-rosehd" |
| x86_deb_5.0 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_macos_10.4 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_rhap_5 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_rhas_3 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_rhas_4 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_sles_9 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_suse_10.0 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |
| x86_suse_10.2 | : "--with-boost=/prereq/boost-1.36.0 --with-CXX_WARNINGS=Wall --without-java" |

Figure 4: Example NMI machine configure options used for nightly tests.