ROSE Compiler Framework/Print version

Contents

About the Book

The goal of this book is to have a **community documentation** providing extensive and up-to-date instructional information about how to use the open-source ROSE compiler framework, developed at Lawrence Livermore National Laboratory .

While the ROSE project website (http://www.rosecompiler.org) already has a variety of official documentations, having a wikibook for ROSE **allows anybody to contribute** to gathering information about this software.

Again, please note that this wikibook is not the official documentation of ROSE. It is the community efforts contributed by anyone just like you.

How to contribute

If you want to contribute, please first tell if your contributions are relevant to this wikibook about ROSE

- Welcomed contributions: ROSE-specific tutorials, how-tos, FAQ, workflow
- What will be not be kept: Copy& paste of general guidelines of doing things: Please just summary them in the ROSE-relevant wikibook page and give reference, URL to it.

Once you are certain the relevance of your contributions. Please read how to do one example contribution.

- http://en.wikibooks.org/wiki/ROSE_Compiler_Framework/How_to_write_a_How-to
- You can just test water how to edit in wikibook using http://en.wikibooks.org/wiki/ROSE_Compiler_Framework/Sandbox
- Bottomline: make sure your contributions are visible in the print version of this book and are logically consistent with the rest of the content.
    - Link http://en.wikibooks.org/wiki/ROSE_Compiler_Framework/Print_version
- Thank you!

Mission and Milestones

Some brainstorming for milestones of ROSE. The order of the sections is used to reflect the priorities. The most important ones come first.

Mission Statement

The primary mission of the ROSE project is to optimize applications within the U.S. Department of Energy (DOE).

Goa: To develop the **BEST** open-source, [[w:Source-to-source_compiler|source-to-source compiler] infrastructure in the world.

The ROSE team achieves this goal through:

- **cutting-edge research** on source- and high-level compiler analysis and optimization algorithms
- **best-practice software engineering** to encapsulate existing and newly-developed compiler techniques into easy-to-use APIs
- **pre-built ROSE tools** to perform program transformation, analysis and optimization of your code
- **an easy-to-use API** to help you to build your own customized, domain-specific compiler tools

Focus: To make advanced compiler techniques accessible to non-expert compiler or tool writers.

Benefit to You: Improved programmer productivity, source code correctness, performance, and efficiency.

ROSE Is...

- A **research platform**
- A **library** (and set of associated tools) to quickly and easily apply compiler techniques to your code
- A **compiler infrastructure** for you to write your own custom source-to-source translators to perform source code transformations, analyses, and optimizations.

Use cases

ROSE is intended to be for the following purposes:

- We use ROSE in our own daily work. In other words, we eat our own dog food.
    - Build ROSE translators to improve the code quality of ROSE, e.g., refactor code

    - Run static analysis tools built on ROSE to improve the code quality of ROSE, e.g. enforce coding conventions and catch errors not found by compilers and other commercial tools
    - Run dynamic analysis tools built on ROSE to catch run-time errors in ROSE
- ROSE is a DOE-fund software project within LLNL. A priority of ROSE is to serve the mission of LLNL and build up the software capability of DOE.
    - ROSE is being used to analyze, translate, and optimize DOE applications to make them run faster and more efficiently on high performance computing platforms.
- For average programmers
    - ROSE is funded by the Office of Science of DOE and is released under a BSD-like license. So everybody can obtain it to use the pre-built tools shipped with ROSE and/or build them custom tools using ROSE.

Core Technlogies

The uniqueness of ROSE lies on its high level program representation and its associated compiler analysis, translation, and optimizations.

AST

The Abstract Syntax Tree (AST) is the intermediate representation (IR) of programs in ROSE. It is important to have an intuitive program representation.

TODO:

- clean up name conventions of AST nodes, member functions, member data
- Using Doxygen to document each node for its purpose: like which languages are using it, the corresponding code constructs, etc
- document important data member and member functions
- use major versions to separate out big changes to names

Doxygen supports using a separated file to document classes/functions. For example, for SgProject node, we can add documents for it in docs/testDoxygen/SgProject.docs

bin(tools) directory

A place to provide popular and hardened tools built using ROSE. Users can use them directly and immediately get real sense about how powerful/useful ROSE can be. We also use these tools in our daily work.

tool list

- dot graph generator
- identityTranslator

- call graph generator
- control flow graph generator
- outliner
- inliner
- loop transformation tools
- constant folding tool
- static analysis tool (compass)
- runtime error checking tool(RTED)
- automatic parallelization tool (autoPar)

So sample tools from tutorials and immature projects will not conflict with real hardened tools.

By hardening these tools, we can really improve

- intra- and inter-procedural analysis
- global analysis using AST merge

Previous efforts in this direction:

- rose/tools: to be removed or merged into the new bin directory

Compilation

This is really no point to talk about a compiler if it cannot compile the code you are interested in.

Enrich the supported benchmarks for C, C++, Fortran, etc.

- ROSE itself: so we can eat our own dog food
- DOE/LLNL applications
- spec cpu benchmark: many conferences in compiler need proof of impact using industrial benchmarks
- Linux kernel
- boost C++ library
- Plum Hall test suites and others

Fix blocking bugs

Translation

With the AST representing an input program, the immediately possible thing is to do program translation by restructuring the AST.

List

- refine SageBuilder so no low level AST constructors are needed to create AST pieces
- refine SageInterface so AST insertion, deletion, copying is simpler

Sample translators built using ROSE

- AST outliner
- OpenMP implementation
- Cross language translation:
    - Fortran 77 to C
    - C++ to C

Analysis

The core of compiler technology is the set of sophisticated compiler analyses. These analyses are the basis for advanced compiler optimizations.

Implement and harden common analysis

- program representation for analysis: often raw AST may not be sufficient or efficient
    - control flow graph: intra-procedural and inter-procedural
    - dominance analysis
    - call graph
    - SSA, including array SSA
- data flow framework: enabling writing a set of data flow analyses, including
    - def-use
    - liveness analysis
    - constant propagation
    - side-effect analysis
    - alias, points-to analysis
    - dependence analysis

Extend the analysis support for multiple languages

- C/C++
- Fortran
- OpenMP

Optimization

Implementing an optimization often involving calling some relevant analyses and then doing some translations.

List

- constant folding
- partial redundancy elimination
- common subexpression elimination
- loop optimizations
- parallelization
- vectorization

Usability

Installation

The first step every ROSE user must encounter is to install ROSE. We should try our best to simplify, speedup, and automate this step as much as possible.

List

- ultimately: one step installation like apt-get install rose
- more realistic: two step process
    - apt-get install rose-prerequisite
    - ./build, configure, make, make install
- milestones, support two-step process on
    - ubuntu
    - Centos
- speedup the installation: goal is 30 minutes or less
- release a VM version of ROSE (use VeeWee, Vagrant, etc.)

Documentation

Types of documentation

- Doxygen documents
    - we promote source comment style so they can be automatically processed to generate class/function reference web pages.
- Design and implementation docs
    - written in LaTeX, reusable for writing academic papers, grant proposals.
    - separate them out from the rose source base
- Developer/User guide
    - we are experimenting with Markdown using github
    - this wikibook is another experiment
    - How-tos: like how to document an AST node
    - coding standard/convention: directory layout
- Tutorials
    - with embedded source code examples which must be tested with the latest version of ROSE
    - choice 1: keep adding the existing tutorial written in LaTex
    - choice 2: experimenting .md

API

ROSE essentially is a library encapsulating compiler techniques. We try to design intuitive, organized, and well documented API so programmers can easily leverage sophisticated compiler technology.

Checklist

- define top level namespaces to organize functionalities
- encapsulate core technologies of ROSE into simple functions

Portability

The goal here is to make ROSE widely available on mainstream platforms.

Support

- popular Linux distributions and their major versions
    - RedHat Enterprise Workstation or its open-source version, Centos
    - Ubuntu LTS
    - Fedora
- Mac OS X
- Windows: native support, not through Cygwin

Often, the key is to support

- more recent versions of GCC
- Boost C++ library

Workflow

Quality comes from a good process.

See details at Workflow

Streamlined, simplified, and automated workflow involving both users and developers to improve the qualify of ROSE and simplify our daily work.

- review of workflow changes

Current components and their roles:

- wikibook:
    - big pictures and milestones
    - instructional tutorials, how-tos, FAQ etc.
    - coding standard/convention: file names, directory layout

- mailing list: interaction with users, feel users' need
- redmine: create projects based on milestones and user input, create and track tasks
- Jenkins: continuous integration of new features, bugfixes
- github:
    - internal:code review only,
    - external: hosting code, issue tracking
    - "rosebot" to automate Github workflow: preliminary testing, policies (git-hooks), automatically add reviewers, etc.
- website: content management system hooked up with all other components

ROSE's Documentations

ROSE uses a range of materials to document the project.

- ROSE manual: the design, algorithm, and implementation details. Written in LaTeX, the content of the manual can come from published papers. It may contain intense academic citations and math formula.
- ROSE tutorial: code examples for tools built on top of ROSE, step-by-step instructions for doing things
- Doxygen web reference: class/namespace references of source code
- this wikibook: non-official, community documentation. Editable by anyone, aimed to supplement official documents and to collect tutorials, FAQ and quick pointers to important topics.

Obtaining ROSE

ROSE's source files are managed by git, a distributed revision control and source code management system. There are several ways to download the source tree:

- Private Git repos within LLNL
    - Private Git repository hosted within Lawrence Livermore National Laboratory: the internal file path is /usr/casc/overture/ROSE/git/ROSE.git: central repo of ROSE, mostly automatically updated by Jenkins only after incoming commits pass all regression tests
    - Private Git repository hosted by github.llnl.gov: used for daily pushes and code review
- Public repositories
    - Public Git repository hosted at https://github.com/rose-compiler/rose: the content is identical to the private Git repository's master branch at LLNL, except that the proprietary EDG submodule is not released.
    - Downloadable packages and a subversion repository (synchronized with stable snapshots of ROSE's git repository): https://outreach.scidac.gov/projects/rose/

git 1.7.10 or later for github.com

github requires git 1.7.10 or later to avoid HTTPS cloning errors, as mentioned at https://help.github.com/articles/https-cloning-errors

Ubuntu 10.04's package repository has git 1.7.0.4. So building later version of git is needed. But you still need an older version of git to get the latest version of git.

```
apt-get install git-core
```

Now you can clone the latest git

```
git clone https://github.com/git/git.git
```

Install all prerequisite packages needed to build git from source files(assuming you already installed GNU tool chain with GCC compiler, make, etc.)

```
sudo apt-get install gettext zlib1g-dev asciidoc libcurl4-openssl-dev
$ cd git  # enter the cloned git directory
$ make configure ;# as yourself
$ ./configure --prefix=/usr ;# as yourself
$ make all doc ;# as yourself
# make install install-doc install-html;# as root
```

Installation

ROSE is released as an open source software package. Users are expected to compile and install the software.

Platform Requirement

ROSE is portable to Linux and Mac OS X on IA-32 and x86-64 platforms. In particular, ROSE developers often use the following development environments:

- Red Hat Enterprise Linux 5.6 or its open source equivalent Centos 5.6
- Ubuntu 10.04.4 LTS. Higher versions of Ubuntu are NOT supported due to the GCC versions supported by ROSE.
- Mac OS X 10.5 and 10.6

Software Requirement

Here is a list for prerequisite software packages for installing ROSE

- GCC 4.0.x to 4.4.x , the range of supported GCC versions is checked by support-rose.m4 during configuration

    - gcc
    - g++
    - gfortran (optional for Fortran support)
- GNU autoconf >=2.6 and automake >= 1.9.5, GNU m4 >=1.4.5
- libtool
- bison (byacc),
- flex
- glibc-devel
- Sun Java JDK
- git
- boost library: version 1.36 to 1.48. Again the range of supported Boost versions is checked by support-rose.m4 during configuration
- ZGRViewer, a GraphViz/DOT Viewer: essential to view dot graphs of ROSE AST
    - install Graphviz first - Graph Visualization Software

Optional packages for additional features or advanced users

- libxml2-devel
- sqlite
- texlive-full, need for building LaTeX docs

Installing boost

The installation of Boost may need some special attention.

Download a supported boost version from
http://sourceforge.net/projects/boost/files/boost/

For version 1.36 to 1.38

```
./configure --prefix=/home/usera/opt/boost-1.35.0
 make
 make install
```

Ignore the warning like : Unicode/ICU support for Boost.Regex?... not found.

For version 1.39 and 1.48: create the boost installation directory first

In boost source tree

- ./bootstrap.sh --prefix=your_boost_install_path
- ./bjam install --prefix=your_boost_install_path --libdir=your_boost_install_path/lib

Remember to export LD_LIBRARY_PATH for the installed boost library, for example

```
D_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/leo/opt/boost_1.45.0_inst/lib
export PATH LD_LIBRARY_PATH
```

Installing Java JDK

Download Java SE JDK from
http://www.oracle.com/technetwork/java/javase/downloads/index.html

For example, you can download jdk-7u5-linux-i586.tar.gz for your Linux 32-bit system.

After untar it to your installation path, remember to set environment variables for Java JDK

```
# jdk path should be search first before other paths
PATH=/home/leo/opt/jdk1.7.0_05/bin:$PATH

# lib path for libjvm.so
D_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/leo/opt/jdk1.7.0_05/jre/lib/i386/
server

# Don't forget to export both variables!!
export PATH LD_LIBRARY_PATH
```

./build

In general, it is better to rebuild the configure file in the top level source directory of ROSE. Just type:

```
rose_sourcetree>./build
```

configure

The next step is to run configure in a separated build tree. ROSE will complain if you try to build it within its source directory.

There are many configuration options. You can see the full list of options by typing ../sourcetree/configure --help . But only --prefix and --with-boost are required as the minimum options.

```
mkdir buildrose
cd buildrose
../rose_sourcetree/configure --prefix=/home/user/opt/rose_tux284 --
with-boost=/home/user/opt/boost-1.36.0/
```

ROSE's configure turns on debugging option by default. The generated object files should already have debugging information.

Additional useful configure options

- Specify where a gcc's OpenMP runtime library libgomp.a is located. Only GCC 4.4's gomp lib should be used to have OpenMP 3.0 support
    - --with-gomp_omp_runtime_library=/usr/apps/gcc/4.4.1/lib/

make

In ROSE's build tree, type

```
cd buildrose
make -j4
```

will build the entire ROSE, including librose.so, tutorials, projects, tests, and so on. -j4 means to use four processes to perform the build. You can have bigger numbers if your machine supports more concurrent processes. Still, the entire process will take hours to finish.

For most users, building librose.so should be enough for most of their work. In this case, just type

```
make -C src/ -j4
```

make check

Optionally, you can type make check to make sure the compiled rose pass all its shipped tests. This takes hours again to go through all make check rules within projects, tutorial, and tests directories.

To save time, you can just run partial tests under a selected directory, like the buildrose/tests

```
make -C tests/ check -j4
```

make install

After "make", it is recommended to run "make install" so rose's library (librose.so), headers (rose.h) and some prebuilt rose-based tools can be installed under the specified installation path using --prefix.

set environment variables

After the installation, you should set up some standard environment variables so you can use rose. For bash, the following is an example:

```
ROSE_INS=/home/userx/opt/rose_installation_tree
PATH=$PATH:$ROSE_INS/bin
```

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROSE_INS/lib
# Don't forget to export variables !!!
export PATH LD_LIBRARY_PATH
```

try out a rose translator

There are quite some pre-built rose translators installed under $ROSE_INS/bin.

You can try identityTranslator, which just parses input code, generates AST, and unparses it back to original code:

```
identityTranslator -c helloWorld.c
```

It should generate an output file named rose_helloWorld.c, which should just look like your input code.

ROSE tools

ROSE is a compiler framework to build customized compiler-based tools. A set of example tools are provided as part of the ROSE release to demonstrate the use of ROSE. Some of them are also useful for daily work of ROSE developers.

We list and briefly explain some tools built using ROSE. They are installed under ROSE_INSTALLATION_TREE/bin .

identityTranslator

This is the simplest tool built using ROSE. It takes input source files , builds AST, and then unparses the AST back to compilable source code. It tries its best to preserve everything from the input file. But due to limitations of the frontends and the internal processing, it cannot generate 100% identical output compared to the input file.

Some notable changes it may introduce include:

- "int a, b, c;" are transformed to three SgVariableDeclaration statements,
- macros are expanded.
- extra brackets are added around constants of typedef types (e.g. c=Typedef_Example(12); is translated in the output to c = Typedef_Example((12));)
- Converting NULL to 0.

Supported Programming Languages

ROSE supports a wide range of main stream programming languages, with different degrees of maturity. The list of supported languages includes:

- C and C++: based on the EDG C++ frontend version 3.3.
  - An ongoing effort is to upgrade the EDG frontend to its recent 4.4 version.
  - Another ongoing effort is to use clang as an alternative, open-source C/C++ frontend
- Fortran 77/95/2003: based on the Open Fortran Parser
- OpenMP 3.0: based on ROSE's own parsing and translation support for both C/C++ and Fortran OpenMP programs.
- UPC 1.1: this is also based on the EDG 3.3 frontend

OpenMP support

ROSE supports OpenMP 3.0 for C/C++ (and limited Fortran support).

- The ROSE manual has a chapter (Chapter 12 OpenMP Support) explaining the details. pdf
- A paper was published for the uniqueness of the ROSE OpenMP Implementation pdf
- Frontend parsing source files (ompparser.yy and ompFortranParser.C) are located under https://github.com/rose-compiler/rose/tree/master/src/frontend/SageIII
- The transformation of OpenMP into threaded code is located in omp_lowering.cpp, under https://github.com/rose-compiler/rose/blob/master/src/midend/programTransformation/ompLowering
- The OpenMP runtime interface is defined in libxomp.h and xomp.c under the same ompLowering directory mentioned above

CUDA support

ROSE has an experimental connection to EDG 4.0, which helps us support CUDA.

To enable parsing CUDA codes, please use the following configuration options:

```
--enable-edg-version=4.0 --enable-cuda --enable-edg-cuda
```

Chapter 16 of ROSE User Manual has more details about this.

Abstract Syntax Tree (Intermediate Representation)

The main intermediate representation of ROSE is its abstract syntax tree (AST).

visualization of AST

We provide ROSE_INSTALLATION_TREE/bin/dotGeneratorWholeASTGraph to generate a dot graph of the detailed AST of input code.

To visualize the generated dot graph, you have to install

- ZGRViewer here: http://zvtm.sourceforge.net/zgrviewer.html#download.
- Graphviz: http://www.graphviz.org/Download.php.

A complete example

```
# make sure the environment variables(PATH, LD_LIBRARY_PATH) for the
installed rose are correctly set
which dotGeneratorWholeASTGraph
~/workspace/masterClean/build64/install/bin/dotGeneratorWholeASTGraph

# run the dot graph generator
dotGeneratorWholeASTGraph -c ttt.c

#see it
which run.sh
~/64home/opt/zgrviewer-0.8.2/run.sh

run.sh ttt.c_WholeAST.dot
```

text output of AST

just call: SgNode::unparseToString(). You can call it from any SgLocatedNode within the AST to dump partial AST's text format.

preprocessing info.

In addition to nodes and edges, ROSE AST may have some extra attributes attached for preprocessing information like #include, #if .. #else. They are attached before, after, or within a nearby lAST node (only the one with source location information.)

An example translator will traverse the input code's AST and dump information about the found preprocessing information,

```
exampleTranslators/defaultTranslator/preprocessingInfoDumper -c
main.cxx
-----------------------------------------------
Found an IR node with preprocessing Info attached:
(memory address: 0x2b7e1852c7d0 Sage type: SgFunctionDeclaration) in
file
/export/tmp.liao6/workspace/userSupport/main.cxx (line 3 column 1)
-------------PreprocessingInfo #0 ----------- :
classification = CpreprocessorIncludeDeclaration:
```

```
  String format = #include "all_headers.h"

relative position is = before
```

Please read more about this topic from ROSE tutorial: "Chapter 29 Handling Comments, Preprocessor Directives, And Adding Arbitrary Text to Generated Code" . You can download it from http://www.rosecompiler.org/ROSE_Tutorial/ROSE-Tutorial.pdf

Program Translation

With its high level intermediate representation, ROSE is suitable for building source-to-source translators. This is achieved by re-structuring the AST of the input source code, then unparsing the transformed AST to the output source code.

Expected behavior of a ROSE Translator

A translator built using ROSE is designed to act like a compiler (gcc, g++,gfortran ,etc depending on the input file types).

So users of the translator only need to change the build system for the input files to use the translator instead of the original compiler.

SageBuilder and SageInterface

The official guide for restructuring/constructing AST **highly recommends** using helper functions from SageBuilder and SageInterfaces namespaces to create AST pieces and moving them around. These helper functions try to be stable across low-level changes and be smart enough to transparently set many edges and maintain symbol tables.

Users who want to have lower level control may want to directly invoke the member functions of AST nodes and symbol tables to explicitly manipulate edges and symbols in the AST. But this process is very tedious and error-prone.

Steps for writing translators

Generic steps:

- prepare a simplest source file (a.c) as an example input of your translator
  - avoid including any system headers so you can visualize the whole AST
  - use ROSE_INSTALLATION_TREE/bin/dotGeneratorWholeASTGraph to generate a whole AST for a.c
- prepare another simplest source file (b.c) as an example output of your translator
  - again, avoid including any system headers

    - use ROSE_INSTALLATION_TREE/bin/dotGeneratorWholeASTGraph to generate a whole AST for b.c
- compare the two dot graphs side by side
- use SageInterface or SageBuilder functions to restruct the source AST graph to be the AST graph you want to generate

Order to traverse AST

Naive pre-order traversal is not suitable for building a translator since the translator may change the nodes the traversal is expected to visit later on. Conceptually, this is essentially the same problem with C++ iterator invalidation.

To safely transform AST, It is recommended to use a reverse iterator of the statement list generated by a preorder traversal. This is different from a list generated from a post order traversal.

For example, assuming we have a subtree of : parent <child 1, child 2>,

- Pre order traversal will generate a list: parent, child 1, child2
- Post order traversal will generate a list: child 1, child2, parent.
- Reverse iterator of the pre order will give you : child2, child 1, and parent. Transforming using this order is the safest based on our experiences.

example translators

split one complex statement into multiple simpler statements

- ROSE/projects/backstroke/ExtractFunctionArguments.C

Program Analysis

ROSE have implemented the following compiler analysis

- call graph analysis
- control flow graph
- data flow analysis: including liveness analysis, def-use analysis, etc.
- dependence analysis
- side effect analysis

control flow graph

ROSE provides several variants of control flow graphs

virtual control flow graph

The virtual control flow graph (vcfg) is dynamically generated on the fly when needed. So there is no mismatch between the ROSE AST and its corresponding control flow graph. The downside is that the same vcfg will be re-generated each time it is needed. This can be a potentially a performance bottleneck.

Facts

- documentation: virtual CFG is documented in **Chapter 19 Virtual CFG** of ROSE tutorial pdf
- source files:
  - src/frontend/SageIII/virtualCFG/virtualCFG.h
  - src/ROSETTA/Grammar/Statement.code // prototypes of member functions for located nodes, etc.
  - src/frontend/SageIII/virtualCFG/memberFunctions.C // implementation of virtual CFG related member functions for each AST node
    - this file will help the generation of buildTree/src/frontend/SageIII/Cxx_Grammar.h
- test directory: tests/CompileTests/virtualCFG_tests
- a dot graph generator: generator a dot graph for either the raw or interesting virtual CFG.
  - source: tests/CompileTests/virtualCFG_tests/generateVirtualCFG.C
  - Installed under rose_ins/bin

static control flow graph

Due to the performance concern of virtual control flow graph, we developed another static version which persistently exists in memory like a regular graph.

Facts:

- documentation: **19.7 Static CFG** of ROSE tutorial pdf
- test directory: rose/tests/CompileTests/staticCFG_tests

static and interprocedural CFGs

Facts:

- documentation: **19.8 Static, Interprocedural CFGs** of ROSE tutorial pdf
- test directory: rose/tests/CompileTests/staticCFG_tests

Virtual function analysis

Facts

- Original contributor: Faizur from UTSA, done in Summer 2011
- Code: at src/midend/programAnalysis/VirtualFunctionAnalysis.

- implemented with the techniques used in the following paper: "Interprocedural Pointer Alias Analysis - http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.2382". The paper boils down the virtual function resolution to pointer aliasing problem. The paper employs flow sensitive inter procedural data flow analysis to solve aliasing problem, using compact representation graphs to represent the alias relations.
- some test files in the roseTests folder of the ROSE repository and he told me that the implementation supports function pointers as well as code which is written across different files (header files etc).
- documentation: Chapter 24 Dataflow Analysis based Virtual Function Analysis, of ROSE tutorial pdf

def-use analysis

If you want a def-use analysis, try this
http://www.rosecompiler.org/ROSE_HTML_Reference/classVariableRenaming.html

```
VariableRenaming v(project);
v.run();
v.getReachingDefsAtNode(...);
```

pointer analysis

https://mailman.nersc.gov/pipermail/rose-public/2010-September/000390.html

On 9/1/10 11:49 AM, Fredrik Kjolstad wrote: > Hi all, > > I am trying to use Rose as the analysis backend for a refactoring > engine and for one of the refactorings I am implementing I need > whole-program pointer analysis. Rose has an implementation of > steensgard's algorithm and I have some questions regarding how to use > this. > > I looked at the file steensgaardTest2.C to figure out how to invoke > this analysis and I am a bit perplexed: > > 1. The file SteensgaardPtrAnal.h that is included by the test is not > present in the include directory of my installed version of Rose. > Does this mean that the Steensgaard implementation is not a part of > the shipped compiler, or does it mean that I have to retrieve an > instance of it through some factory method whose static return type is > PtrAnal? I believe it is in the shipped compiler. And you're using the correct file to figure out how to use it. It should be in the installed include directory --- if it is not, it's probably something that needs to be fixed. But you can copy the include file from ROSE/src/midend/programAnalysis/pointerAnal/ as a temporary fix

> > 2. How do I initialize the alias analysis for a given SgProject? Is > this done through the overloaded ()?

The steensgaardTest2.C file shows how to set up everything to invoke the analysis. Right now you need to go over each function definition and invoke the analysis explicitly, as illustrated by the main function in the file. > > 3. Say I want to query whether two pointer

variables alias and I have > SGNodes to their declarations. How do I get the AstNodePtr needed to > invoke the may_alias(AstInterface&, const AstNodePtr&, const > AstNodePtr&) function? Or maybe I should rather invoke the version of > may_alias that takes two strings (varnames)? > To convert a SgNode* x to AstNodePtr, wrap it inside an AstNodePtrImpl object, i.e., do AstNodePtrImpl(x), as illustrated inside the () operator of TestPtrAnal in steensgaardTest2.C.

> 4. How do I query whether two parameters alias? > The PtrAnal class has the following interface method

```
    may_alias(AstInterface& fa, const AstNodePtr& r1, const AstNodePtr&
```

r2); It is implemented in SteensgaardPtrAnal class, which inherit PtrAnal class. To build AstInterface and AstNodePtr, you simply need to wrap SgNode* with some wrapper classes, illustrated by steensgaardTest2.C

-Qing Yi

```
void func(void) {
int* pointer;
int* aliasPointer;

pointer = malloc(sizeof(int));
aliasPointer = pointer;
*aliasPointer = 42;

printf("%d\n", *pointer);
}

The SteensgaardPtrAnal::output function returns:
c:(sizeof(int )) LOC1=>LOC2
c:42 LOC3=>LOC4
v:func LOC5=>LOC6 (inparams: ) ->(outparams:  LOC7)
v:func-0 LOC8=>LOC7
v:func-2-1 LOC9=>LOC10
v:func-2-3 LOC11=>LOC12 (pending  LOC10 LOC13=>LOC14 =>LOC4 )
v:func-2-4 LOC15=>LOC16 =>LOC17
v:func-2-5 LOC18=>LOC14 =>LOC4
v:func-2-aliasPointer LOC19=>LOC14 =>LOC4
v:func-2-pointer LOC20=>LOC13 =>LOC14 =>LOC4
v:malloc LOC21=>LOC22 (inparams:  LOC2) ->(outparams:  LOC12)
v:printf LOC23=>LOC24 (inparams:  LOC16=>LOC17  LOC14=>LOC4 ) -
>(outparams:
 LOC25)
```

SSA

ROSE has implemented an SSA form. Some discussions on the mailing list: link.

Rice branch has an implementation of array SSA. We are waiting for their commits to be pushed into Jenkins. --Liao (discuss • contribs) 18:17, 19 June 2012 (UTC)

Generic dataflow framework

As the ROSE project goes on, we have collected quite some versions of dataflow analysis. It is painful to maintain and use them as they

- duplicate the iterative fixed-point algorithm
- scatter in different directories and
- use different representations for results.

An ongoing effort is to consolidate all dataflow analysis work within a single framework.

Quick facts

- original author: Greg Bronevetsky
- code reviewer: Chunhua Liao
- Documentation:
- source codes: files under ./src/midend/programAnalysis/genericDataflow
- tests: tests/roseTests/programAnalysisTests/generalDataFlowAnalysisTests
- currently implemented analysis
    - dominator analysis: dominatorAnalysis.h dominatorAnalysis.C
    - livedead variable analysis, or liveness analysis: liveDeadVarAnalysis.h liveDeadVarAnalysis.C
    - constant propagation: constantPropagation.h constantPropagation.C: TODO need to move the files into src/ from /tests

Dependence analysis

The interface for dependence graph could be found in DependencyGraph.h. The underlying representation is n DepGraph.h. BGL is required to access the graph.

Here are 6 examples attached with this email. In deptest.C, there are also some macros to enable more accurate analysis.

If USE_IVS is defined, the induction variable substitution will be performed. if USE_FUNCTION is defined, the dependency could take a user-specified function side-effect interface. Otherwise, if non of them are defined, it will perform a normal dependence analysis and build the graph.

Program Optimizations

ROSE provides the following program optimizations and tranformations:

- loop transformation, including loop fusion, fisson, unrolling, blocking, loop interchange, etc.

- inlining
- outlining
- constant folding
- partial redundancy elimination

Developer's Guide

We briefly describe the workflow of ROSE developers.

Basic skills for ROSE developers

These are some basic skills that you, as a ROSE developer, should have or acquire as you work on your project(s):

- Linux shell: bash is the default shell for ROSE. know common commands (grep, find, ...) and basic scripting (bash, ...)
- C++ programming - be conscious of applying consistent coding-style conventions and writing code that will be maintainable when you leave -
- Debugging: GDB will be indispensable to make sure your code works as expected
- Git - source code management. Get familiar with the basics of Git: http://git-scm.com/
- Build systems - GNU Autotools (autoconf, automake), GNU Make, GNU libtool, and CMake (primarily so you won't break our existing Windows port)
- LaTex - Document your work in ROSE/docs
- Be familiar with ROSE documents (tutorials, installation, and developer guides): http://rosecompiler.org/documents.html. This also includes the project's Doxygen documentation.
- Compiler - ROSE is a compiler project after all. Take some compiler courses
    - Read online free course materials related to compilers
    - Keep learning topics related to your projects

Expected Deliverables for Interns

Projects should result in some concrete deliverables, including but not limited to

- source codes: must past code review and Jenkins so the codes can be merged into rose's master branch. Unfortunately, we have no resources to fix/merge dangling branches after students leave.
- bug fixes: fix issues/bugs assigned to you. The goal is to close these issues after your work.
- documentation: write new chapters, improve existing ones, mostly in LaTeX format.

- presentation: give a talk to share what you have learn and/or what you have done. It is a group-learning process. We will also use some of your slides to report to our sponsors.
- publication: try to publish a paper during your stay with us.
- weekly status reports: please update your redmine project's tasks to reflect what you have done each week

Of course, not every project should produce all types of the deliverables.

code review

see Code Review for details

Workflow

Motivation and Goals

Quality comes from a good process.

The goal is to have a streamlined, simplified, and automated workflow involving both users and developers to

- improve the qualify of ROSE: source codes and documentations
- improve our productivity: optimize and simplify our daily work process so we can do more quality work using less time and other resources

Current workflow

Requirement Analysis

- external (https://github.com/rose-compiler/rose): start an issue to be discussed
- wikibook:
  - draft big pictures and milestones
- mailing list: interaction with users, feel users' need

Design

- wikibook: quick draft design documents and provoke discussion
- powerpoint slides: more formal communication about what is the design

Implementation

- redmine (http://hudson-rose-30:3000/): create projects based on milestones and user input, create and track tasks

- github:
    - internal (http://github.llnl.gov/):code review only,
    - external (https://github.com/rose-compiler/rose): public hosting code, issue tracking
    - "rosebot" to automate Github workflow: preliminary testing, policies (git-hooks), automatically add reviewers, etc.

Testing

- Jenkins ((http://hudson-rose-30:8080/)): continuous integration of new features, bugfixes

Documentation

- See more at ROSE Compiler Framework/Documentation

Publicity

- website (http://www.rosecompiler.org): content management system hooked up with all other components

Issue tracking

We actual have more issue trackers than we would like to have right now. This is caused by the different purposes of issue tracking and the balancing between transparency and privacy/security. Maintainability is also a big factor since internal issue trackers may have quite some down-time compared to commercial websites.

- SciDAC outreach center:
    - external issue tracker(in use): used by end-users only to collect user-submitted issues (ROSE bugs and feature requests). A nice thing about this tracker is that it allows submitting issues anonymously, very handy for busy and/or shy users. We should try to keep this quick channel open.
    - private issue tracker(phasing out): internal-submitted issues, which are not suitable for public view for any reasons. All issues should be moved to the internal issue tracker (redmine) we decide to use.
- Internal redmine (in use): the nice thing here is that it is a dedicated and intuitive project management web application. Tasks (like papers, presentations) can be not related to source code at all. Also important is that it is internal. We can have very secure internal project information. The downside is that it is hard to involve external collaborators.
    - Project specific issue tracking: each research project (and summer student project) has its own issue trackers
    - Internal issue tracking: any issues which are not suitable for public discussion for any reason.

- github.com: potentially handy to connect with issue trackers of github. But not full-featured as a project management software. Everything has to be tied to a git repo. It is not a general assumption. Another downside is that it is public, not suitable for our internal projects.
    - rosecompiler/rose 's issue tracking(in use). A very good way to involving collaborators without remote access to LLNL. Used to track anything which is general to ROSE and not specific to a ROSE-based project.
    - rosecompiler/rose-docs 's issue trackingn(phasing out)
- github enterprise within LLNL: having benefits of being internal. But still not a general project management software.
    - We only use it for code review. Issue tracking there is turned off.

https://github.com/blog/831-issues-2-0-the-next-generation

http://stackoverflow.com/questions/8888675/in-github-issue-tracker-can-non-admin-users-assign-users-and-labels

http://programmers.stackexchange.com/questions/129714/how-to-manage-github-issues-for-priority-etc

Proposal of workflow changes

Major workflow improvements and changes should be thoroughly tested and reviewed by staff members before deployment since they may have profound impact on the project

How to propose a workflow change

- submit a ticket on github.com's rose-public/rose issue tracker. in the ticket,
- (what is it) explain what change is proposed
- (benefits) why the changes: the long-term benefits for our productivity and quality of work
- (costs) the cost of the changes: learning curve, maintainability, purchase cost

Review of workflow change proposal

Review criteria:

- improve our productivity: optimize our workflow to allow us to do more quality and use less time and other resources
    - address what is slowing us down or distracting us
    - Simplify daily life (less hoops, the better). Side-by-side comparison of how many hoops we eliminated/automated by the workflow improvements
        - It is counterproductive to improve workflow by adding more hoops/steps/clicks into daily work.
- improve quality of work incrementally:

    - accepting incremental improvements is more realistic than asking for perfection for the first attempt.
    - workflow should allow quick new contributions and fast revision of existing contributions
- automation: new steps of workffow should be automated as much as possible.
- compatibility:
    - preserve existing work, not create anything from scratch
    - interact well with existing workflow
    - Or having a way to converting existing code/documents into the new form
- easy to use and maintain:
    - the more software tools we depend on, the harder to use and maintain our workflow. Similarly, the more formats/standards we enforce, the harder for developers to do their daily work
    - So adopting new required software components and new required technical formats/standards in our workflow should be very carefully reviewed for the associated **long-term benefits** and costs. Long-term means the range of 5 to 10 years and is not tied to a temporary thing we use now.
- preference of major contributors: whoever contributes the most should has a little bit more weight to say
- documentation: we require major changes to be documented and reviewed before deployment. Writing down things can help us clarify details and solicit wider comments (instead of limited to face-to-face meeting)
- conflict resolution: it will be common to have disagreements or even strong opinions. So we should try our best to resolve differences or we raise the difference to Dan to decide.

Coding Standard

Current Practice vs. Improvement Suggestions

This page documents the current practice of how we write code within the ROSE project. It also serves as a guideline for our code review process.

This is not a place to write down the new ideas/concepts/suggestions to be used in the future.

We do welcome suggestions for improvements and changes so we can do things faster and better.

- For suggestions, please follow the procedure defined in Proposal_of_workflow_changes
- The suggestions will be reviewed by the criteria defined in Review_of_workflow_change_proposal

Allowed Programming Languages

Essentially, only C++ is allowed. Any other programming language is an exception on a case-by-case consideration

Question: But Programming language XYZ is much better than C++ and I am really good at XYZ!!!

Answer: we can allow XYZ only if

- you can teach at least one of old dogs (staff members) of our team the new tricks to efficiently use XYZ
- you will be around in our team in the next 5 to 10 years to **maintain** all the code written in XYZ if none of the old dogs have time/interest to switch to XYZ
- you can prove that XYZ can interact well with the existing C++ codes in ROSE

Allowed Scripting Languages

Only two scripting languages are allowed

- bash shell scripting
- perl

Again, this is just a preference of the staff members and what we have now. Allowing uncontrolled number of scripting languages in a single project will make the project impossible to maintain and hard to learn.

Source comments

Please use styles recognizable by Doxygen. So headers and source files can be parsed and references can be automatically generated.

Language allowed

Only English is allowed. Period.

brief source comments

For single line source comments

```
//! brief description

/// brief description
```

detailed comments

For multiple-line comments

```
/**

   ... text..

 */

/**
 *
 *  ... text..
 *
 */

/*!
 *  ...text....
 */

/*!
   ...text....
 */


///
///  ...text
///

//!
//!  ...text
//!

/*******************************//**
 *         text
********************************/


////////////////////////////////////
///  ... text
////////////////////////////////////
```

combined

Often we need both brief and detailed comments at the same time:

```
/*! \brief Brief description.
 *         Brief description continued.
 *
 *  Detailed description starts here.
 */


//! A constructor.
/*!
    A more elaborate description of the constructor.
*/
```

README

For all major directory in ROSE, there should be a README explaining

- what this directory is about
- who added it and when

Each project directory must have a README to explain:

- What this project is about
  - Name of the project
  - Motivation: Why do we have this project
  - Goal: what do we want to achieve
- Design/Implementation: So next person can quickly catch up and contribute to this project
  - How do we design/implement it.
  - What is the major algorithm
- Brief instructions about how to use the project
  - installation
  - testing
  - or point out where to find the detailed documentation for this project
- Status
  - What is working now
  - Known limitations
- References and citations: for the underneath algorithms
- Authors and Dates

An example README can be found at

- https://github.com/rose-compiler/rose/blob/master/projects/OpenMP_Translator/README

Directory

Name Convention

Layout

TODO: big picture about where to put things within the ROSE git repository.

Files

Name Convention

Header files

Source files

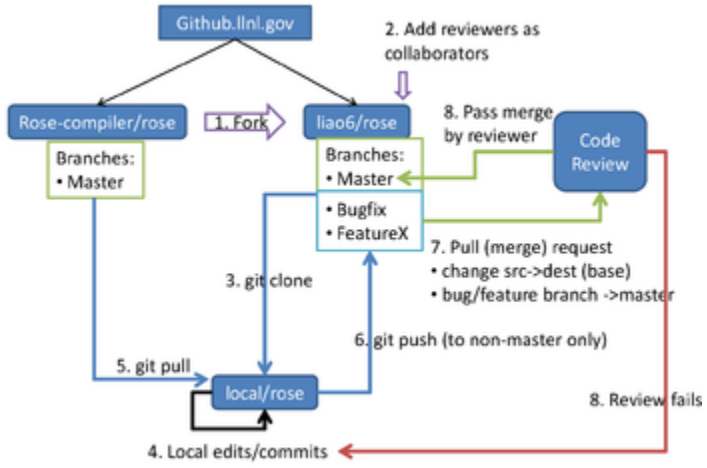
Classes and Variables

Try to use namespace when possible, avoid global variables or classes.

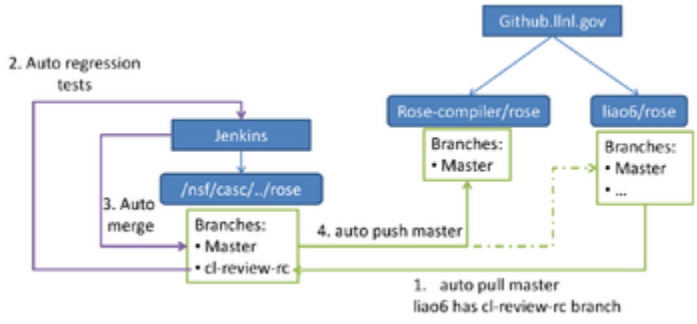
References

- http://www.possibility.com/Cpp/CppCodingStandard.html

Code Review Process



Code review using github.llnl.gov



Connection between github and Jenkins

Motivation

Without code review, developers have:

- added files into wrong directories, with improper names
- committed hundreds of reformatted files
- disabled tests to subvert our stringent Jenkins CI regression tests
- re-invented the wheel by implementing features that already exist

Goals

Our primary goals for code reviewing ROSE are to:

- share knowledge about the code: coder + reviewer will know the code, instead of just the coder
- group-study: learn through studying other peoples' code
- enforce policies for consistent usability and maintainability of ROSE code
- avoid reinventing the wheel and eliminating unnecessary redundancy
- safe-guarding the code: disallowing subversive attempts to disable or remove regression tests

Software

We are currently testing Github Enterprise and looking into the possibility of leveraging Redmine for internal code review.

In the past, we have looked at Google's Gerrit code review system.

Github

Releases: https://enterprise.github.com/releases

Support: https://support.enterprise.github.com

Developer Checklist

Read these tips and guidelines before sending a request for code review.

Coding Standards

Please go to Coding Standard for the complete guideline. Here we only summary some key points.

Your code should be written in a way that makes it easily maintainable and reviewable:

- write easy to understand code; avoid using exotic techniques which nobody can easily understand.
- add sufficient documentation (source-code comments, README, etc.) to aid the understandability of your code, your documentation should cover
    - why do you do this (motivation)
    - how do you do it (design and/or algorithm)
    - where are the associated tests (works as expected)
- before submission of your code for review, make sure
    - you have merged with the latest central repository's master branch without conflicts
    - your working copy can pass local tests via: make, make check, and make distcheck
    - you have fixed all compiler warnings of your code whenever possible
- submit a logical unit of work (one or more commits); something coherent like a bug fix, an improvement of documentation, an intermediate stage for reaching a big new feature.
- balance code submissions with a good ratio of [lines of code] and [complexity of code]. A good balance needs to be achieved to make the reviewer's life easier.
    - the time needed to review your code should not exceed 1 hour

Workflow

Steps for initializing code review:

One time setup

- log into http://github.llnl.gov using your OUN and PAC
- fork your own rose repo from rose-compiler/rose
    - Goto http://github.llnl.gov/rose-compiler/rose, click Fork on the right upper corner of the webpage
- add potential reviewers (liao6, too1, vanderbrugge1, aananthakris1, vanka1) into your forked repo as collaborators, so they can review and merge your branches into your master branch later on
- add hudson-rose as your collaborator also so it can automatically push latest commits to your master branch to keep it up-to-date
    - Goto youraccount/rose of github.llnl.gov, then click Admin -> Collaborators
- create your public-private key pair using ssh-keygen, and copy your public key into your profile with github.llnl.gov
    - this is necessary since only ssh is supported by github.llnl.gov for now. https is not yet supported.

Daily work process

- have a local git repo to do your work and submit local commits, you have two choices:

    - clone it from /nfs/casc/overture/rose/rose.git as we usually do before
    - clone your fork on github.llnl.gov to a local repo: use the ssh URL option for now since the https option won't work.
    - don't use branches, use separated git repositories for each of your tasks. So status/progress of one task won't interfere with other tasks.
- When ready to push your commits, synchronize with the latest rose-compiler/master to resolve merge conflicts, etc.
    - type: git pull origin master # this should always work since master branches on github.llnl.gov are automatically kept up-to-date
    - make sure your local changes can pass 1)make -j8, 2)make check -j8, and 3)make distcheck -j8
- push your commits to your fork's non-master branch, like bugfix-rc , featurex-rc. You have total freedom in creating any branches in your forked repo, with any names you like

```
  # If your local repository was cloned from
/nfs/casc/overture/ROSE/rose.git.
  # There is no need to discard it. You can just add the github.llnl's
repo as an additional remote repository and push things there:
  git remote add github-llnl-youraccount-rose
http://github.llnl.gov/youraccount/rose.git
  git push github-llnl-youraccount-rose HEAD:refs/heads/bugfix-rc
```

- add a pull(merge) request to merge bugfix-rc into your own fork's master,
    - please note that the default pull request will use rose-compiler/rose's master as the base branch (destination of the merge). Please change it to be your own fork's master branch instead.
    - Also make sure the source (head) branch of the pull (merge) request is the one your want (bugfix-rc in this example)
- notify a reviewer that you have a pull request (requesting to merge your bugfix-rc into your master branch)
    - You can assign the pull request to the reviewer so an email notification will be automatically sent to the reviewer
    - Or you can add discussion for the pull request using @revieweraccount
    - Or you can just email the reviewer
- waiting for reviewer's feedback:
    - if passes, reviewer should have merged your bugfix-rc into your master. Jenkins will automatically poll your master and do the testing/merging
    - if reviewer wants additional changes such as better naming, better places to put files, more source comments, accompanying regression tests, etc. Just repeat the process: do local edits, local commits, push to your remote branch, send merge request again
    - A third possible outcome is that reviewers may accept the commits. But some additional tasks are needed in the future to improve the code.

Reviewer Checklist

What to look as a code reviewers?

- Be familiar with the current Coding Standard as a general guideline to do the code review.
- allocate up to 1 hour each time to review 500 to 1000 lines of code: longer time may not pay off due to the attention span limit of human brains
- directory paths and file names: are files in the intuitive paths or conform to our convention? are the names readable?
    - source codes, test input, documentation files are added into the right directories
- clarify of the code: can somebody who did not write the code easily understand what the code does?
    - the reason/motivation for writing the code
    - name convention: variable, function, class names should be intuitive
    - source comments: sufficient explanations for what each function, class does, what is the algorithm used, what is the paper/book chapter the implementation is based on.
- no duplication or reinvent of the wheel: similar code already exists or can be extended
- refactored: can part of the code be refactored to be reusable by others?
- no big functions: a function with hundreds line of code
- make check rules are associated with each new feature to ensure the new feature will be tested and verified for expected behaviors
- No turning off/relaxing other make check rules to make developers' commits pass Jenkins
- make a decision for the code review
    - pass. The code does what it is supposed to do with clear documentation and test cases. Merge the pull request and close the review.
    - fail. Additional changes are needed, such as better naming, better places to put files, more source comments, accompanying regression tests, etc. Notify the developers the issues and ask for a new push with suggested improvements.
    - pass but with future tasks. The commits are accepted. But some additional tasks are needed in the future to improve the code. They can be put into a separate push later on.

what to avoid

- Judging code by whether it's what the reviewer would have written
    - Given a problem, there are usually a dozen different ways to solve it. And given a solution, there's a million ways to render it as code.
- degenerating into nitpicks:
    - perfectionism may hurt the progress. we should allow some non-critical improvements to be done in the next version/commits.
- feel obligated to say something critical: it is perfectly fine to say "looks good, pass"

- delay in review: we should not rush it but we should keep in mind that somebody is waiting for the review to be done to move forward

criticism

Code reviews often degenerate into nitpicks. Brainstorming and design reviews to be more productive.

- I think this makes sense, the early we catch the problems, the better. Design happens earlier. Design should be reviewed. The same idea applies to requirement analysis also. --Liao (discuss • contribs) 18:18, 22 June 2012 (UTC)

references

- http://www.possibility.com/wiki/index.php?title=CodeReviews
- http://scientopia.org/blogs/goodmath/2011/07/06/things-everyone-should-do-code-review/
- http://stackoverflow.com/questions/3730527/workflow-for-github-based-code-review
- http://stackoverflow.com/questions/4262693/what-to-look-for-in-a-code-review
- LLNL Internal URL: http://github.llnl.gov/

Frequently Asked Questions (FAQ)

We collect a list of frequently asked questions about ROSE, mostly from the rose-public mailing list link

How to search rose-public mailinglist for previously asked questions?

google.com supports search things within the scope of a URL. For example, if you have a problem with a keyword MY PROBLEM, you can try to search the mailing list by using the following keyword in google.com:

```
"MY PROBLEM site:https://mailman.nersc.gov/pipermail/rose-public/"
```

Compilation

How to speedup compiling ROSE?

Question It takes hours to compile ROSE, how can I speed up this process?

Answer:

- if you have multi-core processors, try to use make -j4 (make by using four processes).
- also try to only build librose.so under src/ by typing *make -C src/ -j4*
- Or only try to build the language support you are interested in during configure, such as
    - ../sourcetree/configure --enable-only-c # if you are only interested in C/C++ support
    - ../sourcetree/configure --enable-only-fortran # if you are only interested in Fortran support
    - ../sourcetree/configure --help # show all other options to enable only a few languages.

Can ROSE accept incomplete code?

https://mailman.nersc.gov/pipermail/rose-public/2011-July/001015.html

ROSE does not handle incomplete code. Though this might be possible in the future. It would be language dependent and likely depend heavily on some of the language specific tools that we use internally. This is however, not really a priority for our work. If you want to for example demonstrate how some of the internal tools we are using or alternative tools that we could use might handle incomplete code, this might be interesting and we could discuss it.

For example, we are not presently using Clang, but if it handled incomplete code that might be interesting for the future. I recall that some of the latest EDG work might handle some incomplete code, and if that is true then that might be interesting as well. I have not attempted to handle incomplete code with OFP, so I am not sure how well that could be expected to work. Similarly, I don't know what the incomplete code handling capabilities of ECJ Java support is either. If you know any of these questions we could discuss this further.

I have some doubts about how much meaningful information can come from incomplete code analysis and so that would worry me a bit. I expect it is very language dependent and there would be likely some constraints on the incomplete code. So understanding the subject better would be an additional requirement for me.

Can ROSE analyze Linux Kernel sources?

https://mailman.nersc.gov/pipermail/rose-public/2011-April/000856.html

Question: I'm trying to analyze the Linux kernel. I was not sure of the size of the code-base that can be handled by ROSE, and could not find references as to whether it has been tried on the Linux kernel source. As of now I'm trying to run the identity translator on the source, and would like to know if it can be done using ROSE, and if it has been successfully tested before.

Short answer: Not for now

Long answer: We are using EDG 3.3 internally by default and this version of EDG does not handle the GNU specific register modifiers used in the asm() statements of the Linux Kernel code. There might be other problems, but that was at least the one that we noticed in previous work on this some time ago. But we are working on upgrading the EDG frontend to be a more recent version 4.4.

Can ROSE compile C++ Boost library?

https://mailman.nersc.gov/pipermail/rose-public/2010-November/000544.html

not yet.

I know of a few cases where ROSE can't handle parts of Boost. In each case it is an EDG problem where we are using an older version of EDG. We are trying to upgrade to a newer version of EDG (4.x), but that version's use within ROSE does not include enough C++ support, so it is not ready. The C support is internally tested, but we need more time to work on this.

AST

How to find XYZ in AST?

The usually steps to retrieve information from AST are:

- prepare a simplest (preferrably 5-10 lines only), compilable sample code with the code feature you want to find (e.g array[i][j] if you are curious about how to find use of multi-dimensional arrays in AST), avoid including any headers (#include file.h) to keep the code small.
    - Please note: don't include any headers in the sample code. A header (#include <stdio.h> for example) can bring in thousands of nodes into AST.
- use dotGeneratorWholeASTGraph to generate a detailed AST dot graph of the input code
- use zgrviewer-0.8.2's run.sh to visulize the dot graph
- visually/manually locate the information you want in the dot graph, understand what to look and where to look
- use code (AST member functions, traversal, SageInteface functions, etc) to retrieve the information.

How does the AST merge work?

tests that demonstrate the AST Merge are in the directory:

```
tests/CompileTests/mergeAST_tests
```

(run "make check" to see hundreds of tests go by).

How to filter out header files from AST traversals?

https://mailman.nersc.gov/pipermail/rose-public/2010-April/000144.html Question: I want to exclude functions in #include files from my analysis/transformations during my processing.

By default, AST traversal may visit all AST nodes, including the ones come from headers.

So AST processing classes provide three functions :

- T traverse (SgNode * node, ..): traverse full AST , nodes which represent code from include files
- T traverseInputFiles(SgProject* projectNode,..) traverse the subtree of AST which represents the files specified on the command line
- T traverseWithinFile(SgNode* node,..): only the nodes which represent code of the same file as the start node

Should SgIfStmt::get_true_body() return SgBasicBlock?

https://mailman.nersc.gov/pipermail/rose-public/2011-April/000930.html

Both true/false bodies were SgBasicBlock before.

Later, we decided to have more faithful representation of both blocked (with {...}) and single-statement (without { ..}) bodies. So they are SgStatement (SgBasicBlock is a subclass of SgStatement) now.

But it seems like the document has not been updated to be consistent with the change.

You have to check if the body is a block or a single statement in your code. Or you can use the following function to ensure all bodies must be SgBasicBlock.

//A wrapper of all ensureBasicBlockAs*() above to ensure the parent of s is a scope statement with list of statements as children, otherwise generate a SgBasicBlock in between.

SgLocatedNode * SageInterface::ensureBasicBlockAsParent (SgStatement *s)

How to handle #include "header.h", #if, #define etc. ?

It is called preprocessing info. within ROSE's AST. They are attached before, after, or within a nearby AST node (only the one with source location information.)

An example translator is provided to traverse the input code's AST and dump information about the found preprocessing information,

```
exampleTranslators/defaultTranslator/preprocessingInfoDumper -c
main.cxx
---------------------------------------------
Found an IR node with preprocessing Info attached:
(memory address: 0x2b7e1852c7d0 Sage type: SgFunctionDeclaration) in
file
/export/tmp.liao6/workspace/userSupport/main.cxx (line 3 column 1)
-------------PreprocessingInfo #0 ----------- :
classification = CpreprocessorIncludeDeclaration:
  String format = #include "all_headers.h"

relative position is = before
```

SgClassDeclaration::get_definition() returns NULL?

If you look at the whole AST graph carefully, you can find defining and non-defining declarations for the same class.

A symbol is usually associated with a non-defining declaration. A class definition is associated with a defining declaration.

You may want to get the defining declaration from the non-defining declaration before you try to grab the definition.

Translation

Can ROSE identityTranslator generate 100% identical output file?

https://mailman.nersc.gov/pipermail/rose-public/2011-January/000604.html

Questions: Rose identityTranslator performs some modifications, "automatically".

These modifications are:

- Expanding the assert macro.
- Adding extra brackets around constants of typedef types (e.g. c=Typedef_Example(12); is translated in the output to c = Typedef_Example((12));)
- Converting NULL to 0.

How can I avoid these modifications?

Answer: No.

There is no easy way to avoid these changes currently. Some of them are introduced by the cpp preprocessor. Others are introduced by the EDG front end ROSE uses. 100% faithful source-to-source translation may require significant changes to preprocessing directive handling and the EDG internals.

We have had some internal discussion to save raw token strings into AST and use them to get faithful unparsed code. But this effort is still at its initial stage as far as I know.

How to build a tool inserting function calls?

https://mailman.nersc.gov/pipermail/rose-public/2010-July/000319.html

Question: I am trying to build a tool which insert one or more function calls whenever in the source code there is a function belonging to a certain group (e.g. all functions beginning with foo_*). During the ast traversal, how can I find the right place, i.e., there is a function in ROSE that searches for a string pattern or something similar?

Answers:

- In Chapter 28 AST Construction of the ROSE tutorial, there are examples to instrument function calls into the AST using traversals or a queryTree. I would approach this by checking the node for the specific SgFunctionDefinition (or whatever you need) and then check the name of the node to find its location.
- You can
    - use the AST query mechanism to find all functions and store them in a container. e.g Rose_STL_Container<SgNode*> nodeList = NodeQuery::querySubTree(root_node,V_Sg????);
    - Then iterate the container to check each function to see if the function name matches what you want.
    - use SageBuilder namespace's buildFunctionCallStmt() to create a function call statement.
    - use SageInterface namespace's insertStatement () to do the insertion.

How to copy/clone a function?

https://mailman.nersc.gov/pipermail/rose-public/2011-April/000919.html

We need to be more specific about the function you want to copy. Is it just a prototype function declaration (non-defining declaration in ROSE's term) or a function with a definition (defining declaration in ROSE's term)?

- Copying a non-defining function declaration can be achieved by using the following function instead:

```
// Build a prototype for an existing function declaration (defining or
nondefining is fine).
```

```
SgFunctionDeclaration* SageBuilder::buildNondefiningFunctionDeclaration
(const SgFunctionDeclaration *funcdecl, SgScopeStatement *scope=NULL)
```

Copying a defining function declaration is semantically a problem since it introduces redefinition of the same function. It is at least a hack to first introduce something wrong and later correct it. Here is an example translator to do the hack (copy a defining function, rename it, fix its symbol):

```
#include <rose.h>
#include <stdio.h>
using namespace SageInterface;

int main(int argc, char** argv)
{
  SgProject* project = frontend(argc, argv);
  AstTests::runAllTests(project);

// Find a defining function named "bar" under project

  SgFunctionDeclaration* func=
findDeclarationStatement<SgFunctionDeclaration> (project, "bar", NULL,
true);
  ROSE_ASSERT (func != NULL);

// Make a copy and set it to a new name
  SgFunctionDeclaration* func_copy =
isSgFunctionDeclaration(copyStatement (func));
  func_copy->set_name("bar_copy");

// Insert it to a scope
  SgGlobal * glb = getFirstGlobalScope(project);
  appendStatement (func_copy,glb);

#if 1  // fix up the missing symbol, this should be optional now since
SageInterface::appendStatement() should handle it transparently.
  SgFunctionSymbol *func_symbol =  glb->lookup_function_symbol
("bar_copy", func_copy->get_type());
  if (func_symbol == NULL);
  {
    func_symbol = new SgFunctionSymbol (func_copy);
    glb ->insert_symbol("bar_copy", func_symbol);
  }
#endif
  AstTests::runAllTests(project);
  backend(project);
  return 0;
}
```

Can I transform code within a header file?

https://mailman.nersc.gov/pipermail/rose-public/2011-May/000971.html

No. ROSE does not unparse AST from headers right now. A summer project tried to do this. But it did not finish.

https://mailman.nersc.gov/pipermail/rose-public/2010-August/000344.html

I guess ROSE does not support writing out changed headers for safety/practical reasons. A changed header has to be saved to another file since writing to the original header is very dangerous (imaging debugging a header translator which corrupts input headers). Then all other files/headers using the changed header have to be updated to use the new header file.

Also all files involved have to be writable by user's translators.

As a result, the current unparser skips subtrees of AST from headers by checking file flags (compiler_generated and/or output_in_code_generation etc.) stored in Sg_File_Info objects.

How to work with formal and actual arguments of functions?

https://mailman.nersc.gov/pipermail/rose-public/2011-June/001008.html

```
     //Get the actual arguments
     SgExprListExp* actualArguments = NULL;
     if (isSgFunctionCallExp(callSite))
         actualArguments = isSgFunctionCallExp(callSite)->get_args();
     else if (isSgConstructorInitializer(callSite))
         actualArguments = isSgConstructorInitializer(callSite)-
>get_args();
     ROSE_ASSERT(actualArguments != NULL);

     const SgExpressionPtrList& actualArgList =
actualArguments->get_expressions();

     //Get the formal arguments.
     SgInitializedNamePtrList formalArgList;
     if (calleeDef != NULL)
         formalArgList = calleeDef->get_declaration()->get_args();

     //The number of actual arguments can be less than the number of
formal arguments (with implicit arguments) or greater
     //than the number of formal arguments (with varargs)
```

Daily work

git clone returns error: SSL certificate problem?

Symptom:

```
git clone https://github.com/rose-compiler/rose.git
```

```
Cloning into rose...
error: SSL certificate problem, verify that the CA cert is OK. Details:
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate
verify failed while accessing https://github.com/rose-
compiler/rose.git/info/refs

fatal: HTTP request failed
```

The reason may be that you are behind a firewall which tweaks the original SSL certification.

Solutions: Tell cURL to not check for SSL certificates:

```
#Solution 1: Environment variable (temporary)
      $ env GIT_SSL_NO_VERIFY=true git pull

# Solution 2: git-config (permanent)
      # set local configuration
      $ git config --local http.sslVerify false

# Solution 2:  set global configuration
      $ git config --global http.sslVerify false
```

What is the best IDE for ROSE developers?

https://mailman.nersc.gov/pipermail/rose-public/2010-April/000115.html

There may not be a widely recognized best integrated development environment. But developers have reported that they are using

- vim
- emacs
- KDevelop
- Source Navigator
- Eclipse
- Netbeans

The thing is that ROSE is huge and has some ridiculously large generated source file (CxxGrammar.h and CxxGrammar.C are generated in the build tree for example). So many code browsers may have trouble in handling ROSE.

Portability

What is the status for supporting Windows?

https://mailman.nersc.gov/pipermail/rose-public/2011-December/001349.html

We have not finished the Windows work yet. IT is on our list of things to do. It was started and ROSE internally compiles using MS Visual Studio (using project files generated from the Cmake build that we maintain and test within our release process for ROSE) but does not pass our tests. So it is not ready. The distribution of the EDG binaries for Windows is another step that would come after that. We don't know at present when this will be done, it is important, but not a high priority for our DOE specific work, but important for other work. The effort required is something that we could discuss. If you want to call me that would be the best way to proceed. Send me email off of the main list and we can set that up.

https://mailman.nersc.gov/pipermail/rose-public/2011-March/000798.html

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. {\em 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.}

How-tos

Quick, short, and focused tutorials about how to do common tasks as a ROSE developer.

Please create a new wikibook page for each how-to topic. Each how-to wiki page should NOT contain any level one (=) or level two(==) heading so it can be included at the correct levels in the print version of this wikibook.

How to write a How-to

Quick, short, and focused tutorials about how to do common tasks as a ROSE developer. Please create a new wikibook page for each how-to topic. Each how-to wiki page should NOT contain any level one (=) or level two(==) heading so it can be included at the correct levels in the print version of this wikibook.

Create a new page

- optional step: create an account and log in
- Goto: http://en.wikibooks.org/wiki/ROSE_Compiler_Framework/How-tos
- Click on **Edit** tab on the right top of the How-tos page
- Copy and paste one existing How-to to the end of the page, for example:

```
==[[ROSE Compiler Framework/How to write a How-to|How to write a How-
to]]==
{{:ROSE Compiler Framework/How to write a How-to}}
```

- rename three places of the pasted text with the desired page name, for example

```
==[[ROSE Compiler Framework/How to do XYZ|How to do XYZ]]==
{{:ROSE Compiler Framework/How to do XYZ}}
```

- click **save page**
- You will see red text trying to link to the not yet existing **How to do XYZ** page
- click any of the red text, it will bring you to an editing window to add content of your new how-to page
- you can now add new content and save it.

Rules of the content

- Only level three headings and higher are allowed in a how-to page. This is necessary for the how-to page to be correctly included into the final one-page print version of this wikibook. Sorry about this restriction.
  - Again, please don't use level one (=) or level two (==) headings in a how-to page!
- Keep each how-to short and focused. Readers are expected to only spend 30-minutes or much less to quickly learn how to do something using ROSE.
- After you created a new how-to page and saved your contributions. Please go to the print version to make sure it shows up correctly.
  - Here is the link: http://en.wikibooks.org/wiki/ROSE_Compiler_Framework/Print_version
  - Having new content show up in the print version will make sure it is really visible and consistent with the rest of the book.
- please specify the how-to topic is the current practice or the proposed new ways of doing things. So we can have clear guideline for code review for what is mandatory and what is optional.

How to incrementally work on a project

Developing a big, sophisticated project entails many challenges. To mitigate some of these challenges, we have adopted several best practices: incremental development, code review, and continuous integration.

Incremental Development

Developing new functionality in small steps, where the resulting code at each step is a useful improvement over the previous state. Contrast to developing an entire feature fully elaborated, with no points along the way at which it's externally usable.

Code Review

See Code Review in ROSE.

Continuous Integration

Incorporating changes from work in progress into a shared mainline as frequently as possible, in order to identify incompatible changes and introduced bugs as early as possible. The integrated changes need not be particular increments of functionality as far as the rest of the system is concerned.

In other words, incremental development is about making one's work valuable as early as possible, and potentially about getting a better sense of what direction it should take, while continuous integration is about reducing the risks that result from codebase divergence as multiple people do development in parallel.

The question of whether to conditionalize new code is an interesting one. By doing so, one narrows the scope of continuous integration to just checking for surface incompatibilities in merging the changed code. Without actually running the new code against the existing tests, the early detection of introduced bugs is lost. In exchange, multiple people working in the same part of the codebase become less likely to step on each other's toes, because the relevant code changes are distributed more rapidly.

Divide and Conquer

Here are some tips on how to divide up a big project into smaller, bite-sized pieces so each piece can be incrementally developed, code reviewed, and integrated.

- **Input**: define different sets of test inputs based on complexity and difficulty. Tackle simpler sets first.
- **Output**: define intermediate results leading to the final output. Often, results `A` and `B` are needed to generate `C`. So the project can have multiple stages, based on the intermediate results.
- **Algorithm**: complex compiler algorithms are often just enhanced versions of more fundamental algorithms. Implement the fundamental algorithms first to gain insight and experience. Then, afterward, you can implement the full-blown versions.
- **Language:** for projects dealing with multiple languages, focus on one language at a time.
- **Platform**: limit the scope of supported platforms: Linux, Ubuntu, OS X (**TODO**: add reference to ROSE supported platforms)
- **Performance**: Start with a basic, working implementation first. Then try to optimize its performance, efficiency.
- **Scope**: your translator could first focus on working at a function scope, then grow to handle an entire source file, or even multiple files, at the same time.
- **Skeleton then meat**: a project should be created with the major components defined first. Each component can be enriched separately later on.
- **Annotations** (manual vs. automated): Performing one compiler task often requires results from many other tasks being developed. Defining **source code annotations** as the interface between two tasks can decouple these dependencies

in a clean manner. The annotations can be first manually inserted. Later the annotations can be automatically generated by the finished analysis.
- **Optional vs. Default**: introducing a flag to turn on/off your feature. Make it as a default option when it matures.

How to set up the makefile for a translator

In this How-to, you will create a makefile to compile and test your own custom ROSE translator.

You may want to first look at "How-to install ROSE": ROSE Compiler Framework/Installation.

Environment variables

You must have the proper environment variable set so you translator can find the librose.so during execution.

```
export
LD_LIBRARY_PATH=${ROSE_INSTALL}/lib:${BOOST_INSTALL}/lib:$LD_LIBRARY_PA
TH
```

Translator Code

Here is a simplest ROSE translator.

```
// ROSE translator example: identity translator.
//
// No AST manipulations, just a simple translation:
//
//    input_code > ROSE AST > output_code

#include <rose.h>

int main (int argc, char** argv)
{
    // Build the AST used by ROSE
    SgProject* project = frontend(argc, argv);

    // Run internal consistency tests on AST
    AstTests::runAllTests(project);

    // Insert your own manipulations of the AST here...

    // Generate source code from AST and invoke your
    // desired backend compiler
    return backend(project);
}
```

Makefile

Here is a sample makefile. Please make sure replacing some leading spaces of make rules with leading Tabs if you copy & paste this sample.

```
## A sample Makefile to build a ROSE tool.
##
## Important: remember that Makefile recipes must contain tabs:
##
##     <target>: [ <dependency > ]*
##         [ <TAB> <command> <endl> ]+

## ROSE installation contains
##   * libraries, e.g. "librose.la"
##   * headers, e.g. "rose.h"
ROSE_INSTALL=/path/to/rose/installation

## ROSE uses the BOOST C++ libraries
BOOST_INSTALL=/path/to/boost/installation

## Your translator
TRANSLATOR=my_translator
TRANSLATOR_SOURCE=$(TRANSLATOR).cpp

## Input testcode for your translator
TESTCODE=input_code_ifs.cpp

#--------------------------------------------------------------
# Makefile Targets
#--------------------------------------------------------------

all: $(TRANSLATOR)

# compile the translator and generate an executable
# -g is recommended to be used by default to enable debugging your code
$(TRANSLATOR): $(TRANSLATOR_SOURCE)
        g++ -g $(TRANSLATOR_SOURCE) -I$(BOOST_INSTALL)/include/boost -
I$(ROSE_INSTALL)/include -L$(ROSE_INSTALL)/lib -lrose

# test the translator
check: $(TRANSLATOR)
        ./$(TRANSLATOR) -c -I. -I$(ROSE_INSTALL)/include $(TESTCODE)

clean:
        rm -rf $(TRANSLATOR) *.o rose_* *.dot
```

How to debug a translator

It is rare that your translator will just work after your finish up coding. Using gdb to debug your code is indispensable to make sure your code works as expected. This page shows examples of how to debug your translator.

A translator not built by ROSE's build system

If the translator is built using a makefile using libtool. The debugging steps of your translator are just classic steps to use gdb.

- make sure your translator is compiled with -g option so there is debugging information in your object codes

A typical debugging session:

- set a break point
- examine the execution path to make sure the program goes the path your expect
- examine the data to check the values are what you expect

```
# how to print out information about a AST node
#-------------------------------------
(gdb) print n
$1 = (SgNode *) 0xb7f12008
(gdb) print n->sage_class_name()
$2 = 0x578b3af "SgFile"
(gdb) print n->get_parent()
$7 = (SgNode *) 0x95e75b8

#-------------------------------------
# When displaying a pointer to an object, identify the actual (derived)
type of the object
# rather than the declared type, using the virtual function table.
#-------------------------------------
(gdb) set print object on
(gdb) print astNode
$6 = (SgPragmaDeclaration *) 0xb7c68008


# unparse the AST from a node
#-------------------------------------
(gdb) print n->unparseToString()

# print out Sg_File_Info
#-------------------------------------
(gdb) print n->get_file_info()->display()
```

A translator shipped with ROSE

ROSE turns on debugging support by default so the translators shipped with ROSE should already have debugging information available.

However, ROSE uses libtool so the executables in the build tree are not real. You have two choices:

- Find the real executable in the .lib directory then debug the real executables there
- Use libtool command line as follows:

```
libtool --mode=execute gdb --args ./built_in_translator file1.c
```

How to add a new project directory

Many work within ROSE start as a project. They will be moved/refactored into ROSE/src later on once they mature.

Here we should how to add a new project into directory.

A basic example

Many projects start as a translator, analyzer or optimizer, which takes into input code and generate output.

A **basic** sample commit which adds a new project directory into ROSE:
https://github.com/rose-compiler/rose/commit/edf68927596960d96bb773efa25af5e090168f4a

Please look through the diffs so you know what files to be added and changed for a new project.

Essentially, a basic project should contain

- a README file explaining what this project is about, algorithm, design, implementation, etc
- a translator acts as a driver of your project
- additional source files and headers as needed to contain the meat of your project
- test input files
- Makefile.am to
    - compile and generator your translator
    - contain **make check** rule so your translator will be invoked to process your input files and generate expected results

To connect your project into ROSE's build system, you also need to

- Add one more subdir entry into projects/Makefile.am for your project directory
- Add one line into config/support-rose.m4 for **EACH** new Makefile (generated from each Makefile.am) used by your projects.

How to fix a bug

If you are trying to fix a bug (your own or a bug assigned to you to fix). Here are high level steps to do the work

Reproduce the bug

You can only fix a bug when you can reproduce it. This step may be more difficult than it sounds. In order to reproduce a bug, you have to

- find proper the input file
- find a proper translator: a translator shipped with ROSE is easy to find. But be patient and sincere when you ask for a translator written by users.
- find a similar/identical software and hardware environment: a bug may only appear on a specific platform when a specific software configuration is used

Possible results for this step:

- You can reproduce the bug reliably. Bingo! Go to the next step.
- You cannot reproduce the bug. Either the bug report is invalid or you have to keep trying.
- You can reproduce the bug once a while. Oops. This is kind of difficult situation.

Find causes of the bug

Once you can reproduce the bug. You have to identify the root cause of the bug.

Common steps involved

- simplify the input code as much as possible: It can be very hard to debug a problem with a huge input. Always try to prepare the simplest possible code which can just trigger the bug.
    - Often, you have to use a binary search approach to narrow down the input code: only use half of the input at a time to try. Recursively cut the input file into two parts until no further cut is possible while you can still trigger the bug.
- forward tracking: for the translator, it usually takes input and generate intermediate results before the final output is generated. Using a debugger to set break points at each critical stages of the code to check if the intermediate results are what you expect.
- backwards tracking: similar to the previous techniques. But you just back tracking the problem.

Fix the bug

Any bug fix commit should contain

- a regression test: so make check rules can make sure the bug is actually fixed and no further code changes will make the bug relapse.

Lessons Learned

Here we try to collect things we usually try to avoid:

Formating/Indending other people's code

Lesson:

- A developer tried to understand a staff member's source code. But he found that the code's indentation was not right for him. So he re-formatted the source files and committed the changes. Later, the staff member found that his codes were changed too much and he could not read the codes anymore.

Solution:

- Please don't reformat codes you do not own or will maintain.

Using branches of a same repository for different tasks

Lesson:

- A developer used different branches of the same git repository to do different tasks: fixing bugs, adding a new feature, and documenting something. Later on he found that he could not commit and push the work for one task since the changes for other tasks are not ready.

Solution:

- using separated git repositories for different tasks. So the status of one task won't interfere with the progress of other tasks.

Testing

ROSE uses Jenkins to implement a contiguous integration software development process. It leverages a range of software packages to test its correctness, robustness, and performance. The software used by the ROSE's Jenkins include:

- SPEC CPU 2006 benchmark: a subset is supported for now
- SPEC OMP benchmark: a subset is supported for now
- NAS parallel benchmark: developed by NASA Ames Research Center. Both C (customize version) and OpenMP versions are used
- Plum Hall C and C++ Validation Test Suites: a subset is supported for now

Modena Test Suite

1. Clone the Modena test suite repository:

```
  $ git clone ssh://rose-dev@rose-git/modena
```

2. Autotools setup

```
  $ cd modena
  $ ./build.sh
  + libtoolize --force --copy --ltdl --automake
  + aclocal -I ./acmacros -I ./acmacros/ac-archive -I
/usr/share/aclocal
  + autoconf
  + automake -a -c
  configure.ac:4: installing `./install-sh'
  configure.ac:4: installing `./missing'
```

3. Environment bootstrap

```
  $ source /nfs/apps/python/latest/setup.sh
```

4. Build and test!

```
  $ mkdir buildTree
  $ cd buildTree
  $ ../configure \
          --with-
sqlalchemy=${HOME}/opt/python/sqlalchemy/0.7.5/lib64/python2.4/site-
packages \
          --with-target-java-interpreter=java \
          --with-target-java-compiler=testTranslator \
          --with-target-java-compiler-flags="-ecj:1.6" \
          --with-host-java-compiler-flags="-source 1.6"
```

Who is using ROSE

We are aware of the following ROSE users (people who write their own ROSE-based tools). They are the reason of the ROSE's existence. Feel free to add your name if you are using ROSE.

Universities

- University of California, San Diego, CUDA code generator link
- University of Utah, compiler-based parameterized code transformation for autotuning
- University of Oregon, performance tools
- University of Wyoming, OpenMP error checking

DOE national laboratories

- Argonne National Laboratory, performance modeling

TODO List

What is missing (so you can help if you want)

How to backup/mirror this wikibook?

Just in case this website is down, how to download a backup of this wiki book?

How to set up a mirror wiki website containing the wikibook of ROSE?

Maintain the print version

It is possible that new chapters are added but they are not reflected in the one-page print version. So periodical synchronization is needed by including more chapters or re-arranging their order in the one-page print version.

Observations:

- A print version is similar to a source file with included contents, each included chapter will have a first level of heading
- Because the first level heading (=) is used by the print version page to include all chapters, all included pages/chapters should NOT contain any first level heading.

With the basic understanding of how this work, you can now edit the print version's wiki page:

- **Print version**

More at: http://en.wikibooks.org/wiki/Help:Print_versions

Maintain the better pdf file

The pdf version automatically generated from the print version page is rudimentary. It has no table of content and pagination etc.

So we used a manual process to generate better pdf file. We need to occasionally repeat this process to have a up-to-date and better pdf file.

Here are the manual steps:

- Use your web browser to open and save the print version to your own computer as "web page complete"
- use the HTML-compatible word processor of your choice to open the html file, convert html to a format the word processor, and add paginate the book.

- In Microsoft Word, this can done by
  - opening the saved HTML file
  - saving it to a word file
  - adding table of content by selecting *Insert* > *Field* > *Index and Tables* > *TOC* or *Preferences*-> *Table of contents* for Word 2012 or later.
  - adding page numbers to the footer
  - save it to a pdf file with a name like ROSE_Compiler_Framework.pdf
  - upload to wikibooks

To add a link to your wikibook page, insert

```
{{PDF version|pdf file name without .pdf|size kb, number pages|file
description}}
```

For example

```
{{PDF version|ROSE_Compiler_Framework|840 kb, 48
pages|ROSE_Compiler_Framework}}
```

More background about pdf verions: at: http://en.wikibooks.org/wiki/Help:Print_versions

Sandbox

Some common tricks to write things on wikibooks/wikipedia (both are using the mediawiki software).

How to create a new page

Usually you have to start a new page from an existing wikipage.

Go to the wiki page you want to have a link to the new page you want to create

- click the edit tab the existing page
- at the place you want to have a link to the new page, use
- `[[ROSE_Compiler_Framework/name of the page]]`

  .

- If there is already a page with the desired name. It will become a link to the page.
- If not, the link is red so you can click the red link to enter editing model to add content to the page.

Please link the new page to the print version of this wikibook so it can be visible in the print out.

- To edit the print version, go to http://en.wikibooks.org/w/index.php?title=ROSE_Compiler_Framework/Print_version&action=edit

How to do XYZ in wiki?

The best way is to goto en.wikipedia.com and find a page with the output you want. Then pretend to edit the page (by clicking edit) to see the source used to generate the output.

For example, you want to know how C++ syntax highlighting is obtained in wikibook. Go to en.wikipedia.com and find the page for C++. There must be sample code snippet.

Then you pretend to edit it to see the source:
http://en.wikipedia.org/w/index.php?title=C%2B%2B&action=edit&section=6

You will see the source code generating the syntax highlighting:

```
<source lang="cpp">

# include <iostream>

int main()
{
   std::cout << "Hello, world!\n";
}
</source>
```

Syntax highlighting

Copied from
http://en.wikipedia.org/w/index.php?title=C%2B%2B&action=edit&section=6

```
<source lang="cpp">

# include <iostream>

int main()
{
   std::cout << "Hello, world!\n";
}
</source>
```

Can generate the following highlighted code:

```
# include <iostream>

int main()
{
   std::cout << "Hello, world!\n";
```

}

Math formula

You can pretend to edit this section to see how math formula are written.

More resources are at

- http://en.wikipedia.org/wiki/Help:Formula
- http://www.mediawiki.org/wiki/Manual:Math

$$\sum_{j=1}^{N}(Si,j) = 1$$

$$\begin{aligned}\log_2(n!) &= \log_2(n) + \log_2(n-1) + \log_2(n-2) + ... + \log_2(1) \\ &\quad \log_2(n) + \log_2(n) + \log_2(n) + ... + \log_2(n) \\ &\quad nlog_2(n)\end{aligned}$$

$$\begin{aligned}\log_2(n!) &= \log_2(n) + \log_2(n-1) + \log_2(n-2) + ... + \log_2(1) \\ &< \log_2(n) + \log_2(n) + \log_2(n) + ... + \log_2(n) \\ &= nlog_2(n)\end{aligned}$$

$$\begin{aligned}z &= a \\ f(x,y,z) &= x + y + z\end{aligned}$$

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\,dt = \frac{e^{-x^2}}{x\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{(2n)!}{n!(2x)^{2n}}$$

Retrieved from "http://en.wikibooks.org/w/index.php?title=ROSE_Compiler_Framework/Print_version&oldid=2373918"

Category:

- ROSE Compiler Framework

What do you think of this page? Reliability: (unsure) **Completeness:** (unsure) **Neutrality:** (unsure) **Presentation:** (unsure) Submit

Re-review this revision

Quality: poor/unrated minimal average good
Comment: Accept revision Unaccept revision

Personal tools

- Liao
- My discussion
- My preferences
- My watchlist
- My contributions
- Log out

Namespaces

- Book
- Discussion

Variants

Views

- Read
- Latest draft
- Edit
- View history
- Unwatch

Actions

- Move

Search

Navigation

- Main Page
- Help
- Browse
- Cookbook
- Wikijunior
- Featured books
- Recent changes
- Donations

- Random book

Community

- Reading room
- Community portal
- Bulletin Board
- Help out!
- Policies and guidelines
- Contact us

Toolbox

- What links here
- Related changes
- Upload file
- Special pages
- Permanent link
- Cite this page
- Page rating

Sister projects

- Wikipedia
- Wikiversity
- Wiktionary
- Wikiquote
- Wikisource
- Wikinews
- Commons

Print/export

- Create a collection
- Download as PDF
- Printable version

- 
a
WIKIMEDIA
project
- 
Powered By
MediaWiki