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 instructional 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:
  - Fix typos, grammar of existing pages to improve quality, clarity, and readability.
  - Add new pages about ROSE-specific tutorials, how-tos, FAQ, workflow
  - start discussions on the Discussion Tab of an existing page about new suggestions of how things can be done better than the current practice.
- 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
- Occasionally, you may want to insert figures into a wiki page. You can do this by uploading file first through Left menu -> Toolbox->upload file
    - The upload link will direct you to Media Commons, more at link
- 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!

Tracking Wiki Changes

Learn how to "Track Changes": http://en.wikibooks.org/wiki/Help:Tracking_changes

Enable Email Notifications for Changes to this book

If you want to be notified of changes to this book, WikiBooks provides email notifications for changes to Wiki pages that you explicitly choose to watch.

To use this feature:

1. **Create an account** with WikiBooks: http://en.wikibooks.org/w/index.php?title=Special:UserLogin&returnto=Main+Page&type=signup

2. **Login** to WikiBooks and set your preferences (top right corner of the web page) for both email notifications and your watch list:

- **Email notification settings**
    - Preferences-> User profile-> E-mail notifications -> E-mail me when a page on my watchlist is changed (check this on)
- **Define your watchlist**
    - Preferences->Watchlist -> Advanced options -> you can select the options you want, such as "Add pages I edit to my watchlist" and "Add pages I create to my watchlist"
    - you can also individually watch and unwatch any wiki page: by click on the star on the page's tab list (after View history)

Caveat: we don't know if wikibooks supports users to watch one entire book. So far, you have to do this one page after another by editing them at some points.

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 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/

Virtual machine image

It can take quite some time to install ROSE for the first time. We provide a virtual machine image with a Ubuntu 10.04 OS and an installed ROSE within it.

You can just download it and play it using VMware Player

Download the virtual machine image:

- http://www.rosecompiler.org/Ubuntu-ROSE-Demo.tar.gz

- Demonstration user account (sudo user in Ubuntu):
  - account: demo
  - password: password
- **Warning**: it is a huge file of 4.8 GB

More information is at ROSE virtual machine image

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 1.47. 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

```
LD_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
LD_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.

Virtual machine image

The goal of this page is to document

- How users can download the virtual machine image (or virtual appliance) and use ROSE out of box.
- how the virtual machine image for a fully installed ROSE is created.

How to use the virtual machine image

Obtain the Virtual Machine Image

Download the virtual machine image created by using VMware Player:

- http://www.rosecompiler.org/Ubuntu-ROSE-Demo.tar.gz
- **Warning**: it is a huge file of 4.8 GB.
- Demonstration user account (sudo user in Ubuntu):
    - **account**: demo
    - **password**: password

LLNL users may not be able to download it due to limitations to max downloaded file size within LLNL.

Content of the VM Image

Copy&paste from README within the virtual machine

This is a virtual machine image for the ROSE source-to-source compiler framework.

sourcetree, cloned from github.com/rose-compiler/rose on July 21, 2012

- /home/demo/rose

buildtree

- /home/demo/buildrose

installation tree (--prefix path)

- /home/demo/opt/rose-inst

A script to set environment variables to use the installed ROSE tools

- /home/demo/set.rose.env

A test translator

- /home/demo/myTranslator

Some dot graphs of a simplest function

- /home/demo/dotGraphs

Install VMware Player

You have to install VMware Player to your machine to use the virtual machine image.

Goto http://www.vmware.com/go/downloadplayer/

Select the right bundle for your platform. For example: VMware-Player-4.0.4-744019.i386.txt

After downloading (assuming you are using Ubuntu 10.04)

- chmod a+x VMware-Player-4.0.4-744019.i386.txt
- sudo ./VMware-Player-4.0.4-744019.i386.txt
- follow the GUI to finish the installation

To start VMPlayer, goto Menu->Applications-> System Tools -> VMware Player

Open/Play the virtual machine

After downloading and untar the tar.gz package to a directory, use VMware player to open the configuration file of the directory.

How was the virtual machine made

Host Machine

We used Ubuntu 10.04 LTS as a host machine to create the virtual machine image.

```
uname -a
Linux 8core-ubuntu 2.6.32-41-generic-pae #91-Ubuntu SMP Wed Jun 13
12:00:09 UTC 2012 i686 GNU/Linux

cat /etc/*release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=10.04
DISTRIB_CODENAME=lucid
DISTRIB_DESCRIPTION="Ubuntu 10.04.4 LTS"
```

Configurations

VMware player has been installed onto the host machine, as described above.

Basic configuration for the virtual machine

Hardware

- Memory : 2 GB
- Processors: 2
- Hard Disk size: 15 GB: We would like to keep it small while having enough space for users.
    - 5GB is used for Ubuntu system files and
    - 10GB for the demonstration user's home directory
- Network Adapter: NAT: share the host's IP address

OS

- OS: Ubuntu 10.04 LTS
- Demonstration user account (sudo user in Ubuntu):
    - **account**: demo
    - **password**: password
- screen size: 1280x960 (4:3)

Download Ubuntu 10.04 LTS http://releases.ubuntu.com/lucid/ We currently use the i386 desktop ISO as the start point

- http://releases.ubuntu.com/lucid/ubuntu-10.04.4-desktop-i386.iso

Within the virtual machine

We installed Software Prerequisites

- sudo apt-get install gcc g++ gfortran
- sudo apt-get install autoconf automake libtool
- sudo apt-get install git-core bison flex texlive-full graphviz python-all-dev

We then installed ROSE

- See ROSE installation for details about how this was done.

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

Source: http://www.rosecompiler.org/ROSE_Tutorial/ROSE-Tutorial.pdf (chapter 2)

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.

Uses

Typical use cases

- without any options, test if ROSE can compile your code: replace the compiler used by your Makefile with identityTranslator
- turn on some built-in analysis, translation or optimization phases, such as -rose:openmp:lowering to support OpenMP
    - type "identityTranslator --help" to see all options
- debug a ROSE-based translator: the first step is often to use identityTranslator to rule out if it is a compilation problem using ROSE
- use the source of the identityTranslator as a start point to add custom analysis and transformation. The code in the identityTranslator is indeed the minimum code required for almost all kinds of ROSE-based tools.

Source code

identityTranslator.c

```
#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 manipulation of the AST here...

        // Generate source code from AST and call the vendor's compiler
        return backend(project);
}
```

Limitations

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.

TODO

- refactor the tools into a dedicated rose/tools directory. So they will always be built and available by default, regardless which languages are turned on or off

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).

Sanity check

We provide a set of sanity check for AST. We use them to make sure the AST is consistent. It is also highly recommended that ROSE developers to add a sanity check after their AST transformation is done. This has a higher standard than just correctly

unparsed to compilable code. It is common for an AST to go through unparsing correctly but fail on the sanity check.

The recommend one is

- AstTests::runAllTests(project); from src/midend/astDiagnostics. Internally, it calls the following checks:
  - TestAstForProperlyMangledNames
  - TestAstCompilerGeneratedNodes
  - AstTextAttributesHandling
  - AstCycleTest
  - TestAstTemplateProperties
  - TestAstForProperlySetDefiningAndNondefiningDeclarations
  - TestAstSymbolTables
  - TestAstAccessToDeclarations
  - TestExpressionTypes
  - TestMangledNames::test()
  - TestParentPointersInMemoryPool::test()
  - TestChildPointersInMemoryPool::test()
  - TestMappingOfDeclarationsInMemoryPoolToSymbols::test()
  - TestLValueExpressions
  - TestMultiFileConsistancy::test() //2009
  - TestAstAccessToDeclarations::test(*i); // named type test

There are some other functions floating around. But they should be merged into AstTests::runAllTests(project)

- FixSgProject(*project); //in Qing's AST interface
- Utility::sanityCheck(SgProject*)
- Utility::consistencyCheck(SgProject*) // SgFile*

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
```

Source: http://www.rosecompiler.org/ROSE_Tutorial/ROSE-Tutorial.pdf (Chapter 29 - Handling Comments, Preprocessor Directives, And Adding Arbitrary Text to Generated Code)

AST construction

SageBuilder and SageInterface namespaces provide functions to create AST pieces and manipulate them.

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 SageInterface 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

see more at ROSE Compiler Framework/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.

Generic Dataflow Framework

Introduction

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

Function, nodeState and FunctionState

Function and nodeState are two required parameters to run data flow analysis:

They are stored together inside FunctionState //functionState.h

functionState.h

genericDataflow/cfgUtils/CallGraphTraverse.h

function

An abstraction of functions, internally connected to SgFunctionDeclaration *decl

declared in ./src/midend/programAnalysis/genericDataflow/cfgUtils/CallGraphTraverse.h

constructors:

- Function::Function(string name) based on function name
- Function::Function(SgFunctionDeclaration* sample) // core constructor
- Function::Function(SgFunctionDefinition* sample)

CGFunction* cgFunc; // call graph function

Function func(cgFunc);

NodeFact

any information related to a CFG node.

- It has no dataflow 's IN/OUT concept
- not meant to evolve during the dataflow analysis

```
class NodeFact: public printable
{
        public:

                // returns a copy of this node fact
        virtual NodeFact* copy() const=0;

};
```

NodeState

Store information about multiple analyses and their corresponding lattices, for a given node (CFG node ??)

./src/midend/programAnalysis/genericDataflow/state/nodeState.h

It also provide static functions to

- initialize NodeState for all DataflowNode
- to retrieve NodeState for a given DataflowNode

```
class NodeState
{
     // internal types: map between analysis and set of lattices

     typedef std::map<Analysis*, std::vector<Lattice*> > LatticeMap;
     typedef std::map<Analysis*, std::vector<NodeFact*> > NodeFactMap;
     typedef std::map<Analysis*, bool > BoolMap;

        // the dataflow information Above the node, for each analysis
that
        // may be interested in the current node
        LatticeMap dfInfoAbove;  // IN set in a dataflow

        // the Analysis information Below the node, for each analysis
that
        // may be interested in the current node
        LatticeMap dfInfoBelow;  // OUT set in a dataflow,

        // the facts that are true at this node, for each analysis that
        // may be interested in the current node
        NodeFactMap facts;

        // Contains all the Analyses that have initialized their state
at this node. It is a map because
```

```
        // TBB doesn't provide a concurrent set.
        BoolMap initializedAnalyses;

// static interfaces

        // returns the NodeState object associated with the given
dataflow node.
        // index is used when multiple NodeState objects are associated
with a given node
        // (ex: SgFunctionCallExp has 3 NodeStates: entry, function
body, exit)
        static NodeState* getNodeState(const DataflowNode& n, int
index=0);


// most useful interface: retrieve the lattices (could be only one)
associated with a given analysis

      // returns the map containing all the lattices from above the
node that are owned by the given analysis
        // (read-only access)
        const std::vector<Lattice*>& getLatticeAbove(const Analysis*
analysis) const;

        // returns the map containing all the lattices from below the
node that are owned by the given analysis
        // (read-only access)
        const std::vector<Lattice*>& getLatticeBelow(const Analysis*
analysis) const;

}
```

FunctionState

./src/midend/programAnalysis/genericDataflow/state/functionState.h

A pair of Function and NodeState.

- it provides static functions to initialize all FunctionState And retrieve FunctionState

```
class FunctionState
{
        friend class CollectFunctions;
        public:
        Function func;
        NodeState state;
        // The lattices that describe the value of the function's
return variables
        NodeState retState;

        private:
        static std::set<FunctionState*> allDefinedFuncs;
```

```
        static std::set<FunctionState*> allFuncs;
        static bool allFuncsComputed;

    public:
        FunctionState(Function &func):
                func(func),
                state(/*func.get_declaration()->cfgForBeginning()*/)
        {}
  // We should use this interface --------------

  // 1. returns a set of all the functions whose bodies are in the
project
        static std::set<FunctionState*>& getAllDefinedFuncs();

  // 2. returns the FunctionState associated with the given function
        // func may be any declared function
        static FunctionState* getFuncState(const Function& func);
 ...
}
```

FunctionState* fs = new FunctionState(func); // empty From FuntionState to NodeState

```
/************************************
 *** UnstructuredPassInterAnalysis ***
 ************************************/
void UnstructuredPassInterAnalysis::runAnalysis()
{
        set<FunctionState*> allFuncs =
FunctionState::getAllDefinedFuncs(); // call a static function to get
all function state s

        // Go through functions one by one, call an intra-procedural
analysis on each of them
        // iterate over all functions with bodies
        for(set<FunctionState*>::iterator it=allFuncs.begin();
it!=allFuncs.end(); it++)
        {
                FunctionState* fState = *it;
                intraAnalysis->runAnalysis(fState->func, &(fState-
>state));
        }
}

// runs the intra-procedural analysis on the given function, returns
true if
// the function's NodeState gets modified as a result and false
otherwise
// state - the function's NodeState
bool UnstructuredPassIntraAnalysis::runAnalysis(const Function& func,
NodeState* state)
{
        DataflowNode funcCFGStart =
cfgUtils::getFuncStartCFG(func.get_definition(),filter);
```

```
        DataflowNode funcCFGEnd =
cfgUtils::getFuncEndCFG(func.get_definition(), filter);

        if(analysisDebugLevel>=2)
                Dbg::dbg <<
"UnstructuredPassIntraAnalysis::runAnalysis() function
"<<func.get_name().getString()<<"()\n";

        // iterate over all the nodes in this function
        for(VirtualCFG::iterator it(funcCFGStart);
it!=VirtualCFG::dataflow::end(); it++)
        {
                DataflowNode n = *it;
                // The number of NodeStates associated with the given
dataflow node
                //int numStates=NodeState::numNodeStates(n);
                // The actual NodeStates associated with the given
dataflow node
                const vector<NodeState*> nodeStates =
NodeState::getNodeStates(n);

                // Visit each CFG node
                for(vector<NodeState*>::const_iterator itS =
nodeStates.begin(); itS!=nodeStates.end(); itS++)
                        visit(func, n, *(*itS));
        }
        return false;
}
```

example: retrieve the liveness analysis's IN lattice

void getAllLiveVarsAt(LiveDeadVarsAnalysis* ldva, const NodeState& state, set<varID>& vars, string indent)

- LiveVarsLattice* liveLAbove = dynamic_cast<LiveVarsLattice*>(*(state.getLatticeAbove(ldva).begin()));

Lattices

Caveat: lattice vs. lattice value

- A lattice by definition is a set of values. However, an instance of lattice type in Generic dataflow framework is used to represent an individual value within a lattice also. Sorry for this confusing. We welcome suggestions to fix this.

Basics

See more at ROSE Compiler Framework/Lattice

Store the data flow analysis information attached to CFG nodes.

Fundamental operations:

- what to store: lattice value set, bottom, up , and anything in between
- initialization: LiveDeadVarsAnalysis::genInitState()
- creation: transfer function
- meet operation: a member function of the lattice

Example

- liveness analysis: the live variable set at the entry point of a CFG node:
- constant propagation: lattice values from no information (bottom) -> unkown --> constant --> too much information (conflicting constant values, top),

```
// blindly add all of that_arg's values into current lattice's value
set
void LiveVarsLattice::incorporateVars(Lattice* that_arg)

// retrieve a subset lattice information for a given expr. This lattice
may contain more information than those about a given expr.
Lattice* LiveVarsLattice::project(SgExpression* expr)

// add lattice (exprState)information about expr into current lattice's
value set: default implementation just calls meetUpdate(exprState)
bool LiveVarsLattice::unProject(SgExpression* expr, Lattice* exprState)
```

below/above vs IN/OUT

The concept is based on the original CFG flow direction

- above: the incoming edge direction
- below: the outcoming edge direction

IN and OUT depends on the direction of the problem, forward vs. backward

- forward direction: IN == above lattice, OUT = below lattice
- backward direction: IN == below lattice, OUT = above lattice

common utility lattices

the framework provides some pre-defined lattices ready for use.

lattice.h/latticeFull.h

- BoolAndLattice

LiveVarsLattice

```
class LiveVarsLattice : public FiniteLattice
{
        public:
        std::set<varID> liveVars;  // bottom is all live variables,
top is the empty set, meet brings down the lattice -> union of
variables.
    ...
 };


// Meet operation: simplest set union of two lattices:

// computes the meet of this and that and saves the result in this
// returns true if this causes this to change and false otherwise
bool LiveVarsLattice::meetUpdate(Lattice* that_arg)
{
        bool modified = false;
        LiveVarsLattice* that =
dynamic_cast<LiveVarsLattice*>(that_arg);

        // Add all variables from that to this
        for(set<varID>::iterator var=that->liveVars.begin(); var!=that-
>liveVars.end(); var++) {
                // If this lattice doesn't yet record *var as being
live
                if(liveVars.find(*var) == liveVars.end()) { // this if
() statement gives a chance to set the modified flag.
                                                        //
otherwise, liveVars.insert() can be directly called.
                        modified = true;
                        liveVars.insert(*var);
                }
        }

        return modified;
}
```

Transfer function

basics: Data_flow_analysis#flow.2Ftransfer_function

- IN = sum of OUT (predecessors)
- OUT = GEN + (IN - KILL)

The impact of program constructs on the current lattices (how to change the current lattices).

- lattices: stores IN and OUT information
- additional data members are necessary to store GEN and KILL set inside the transfer function.

class hierarchy:

```
class IntraDFTransferVisitor : public ROSE_VisitorPatternDefaultBase
{
protected:
  // Common arguments to the underlying transfer function
  const Function &func;  // which function are we talking about
  const DataflowNode &dfNode;  // wrapper of CFGNode
  NodeState &nodeState;   // lattice element state, context
information?
  const std::vector<Lattice*> &dfInfo;  // data flow information

public:

  IntraDFTransferVisitor(const Function &f, const DataflowNode &n,
NodeState &s, const std::vector<Lattice*> &d)
    : func(f), dfNode(n), nodeState(s), dfInfo(d)
  { }

  virtual bool finish() = 0;
  virtual ~IntraDFTransferVisitor() { }

 };



class LiveDeadVarsTransfer : public IntraDFTransferVisitor
{

};


class ConstantPropagationAnalysisTransfer : public
VariableStateTransfer<ConstantPropagationLattice>
{}
```

constant propagation

```
template <class LatticeType>
class VariableStateTransfer : public IntraDFTransferVisitor
{
  ...
};

class ConstantPropagationAnalysisTransfer : public
VariableStateTransfer<ConstantPropagationLattice> {};

void
ConstantPropagationAnalysisTransfer::visit(SgIntVal *sgn)
   {
     ROSE_ASSERT(sgn != NULL);
     ConstantPropagationLattice* resLat = getLattice(sgn);
     ROSE_ASSERT(resLat != NULL);
```

```
      resLat->setValue(sgn->get_value());
      resLat->setLevel(ConstantPropagationLattice::constantValue);
    }
```

live dead variable

Functions to convert program point to Generator and KILL set. For liveness analysis

- Kill (s) = {variables being defined in s}: //
- Gen (s) = {variables being used in s}

OUT = IN -KILL + GEN

- OUT is initialized to be IN set,
- transfer function will apply -KILL + GEN

```
class LiveDeadVarsTransfer : public IntraDFTransferVisitor

{
  LiveVarsLattice* liveLat;  // the result of this analysis

  bool modified;
  // Expressions that are assigned by the current operation
  std::set<SgExpression*> assignedExprs;  // KILL () set
  // Variables that are assigned by the current operation
  std::set<varID> assignedVars;
  // Variables that are used/read by the current operation
  std::set<varID> usedVars;   // GEN () set

public:
  LiveDeadVarsTransfer(const Function &f, const DataflowNode &n,
NodeState &s, const std::vector<Lattice*> &d, funcSideEffectUses
*fseu_)
    : IntraDFTransferVisitor(f, n, s, d), indent("    "),
liveLat(dynamic_cast<LiveVarsLattice*>(*(dfInfo.begin()))),
modified(false), fseu(fseu_)
  {
        if(liveDeadAnalysisDebugLevel>=1) Dbg::dbg << indent <<
"liveLat="<<liveLat->str(indent + "    ")<<std::endl;
        // Make sure that all the lattice is initialized
        liveLat->initialize();
  }

  bool finish();
 //  operationg on different AST nodes
  void visit(SgExpression *);
  void visit(SgInitializedName *);
  void visit(SgReturnStmt *);
  void visit(SgExprStatement *);
  void visit(SgCaseOptionStmt *);
  void visit(SgIfStmt *);
  void visit(SgForStatement *);
  void visit(SgWhileStmt *);
```

```
  void visit(SgDoWhileStmt *);
}


// Helper transfer function, focusing on handling expressions.
// live dead variable analysis: LDVA,
//  expression transfer: transfer functions for expressions
/// Visits live expressions - helper to LiveDeadVarsTransfer
class LDVAExpressionTransfer : public ROSE_VisitorPatternDefaultBase
{
  LiveDeadVarsTransfer &ldva;

public:

  // Plain assignment: lhs = rhs,  set GEN (read/used) and KILL
(written/assigned) sets
  void visit(SgAssignOp *sgn) {
    ldva.assignedExprs.insert(sgn->get_lhs_operand());

    // If the lhs of the assignment is a complex expression (i.e. it
refers to a variable that may be live) OR
    // if is a known expression that is known to may-be-live
    // THIS CODE ONLY APPLIES TO RHSs THAT ARE SIDE-EFFECT-FREE AND WE
DON'T HAVE AN ANALYSIS FOR THAT YET
    /*if(!isVarExpr(sgn->get_lhs_operand()) ||
      (isVarExpr(sgn->get_lhs_operand()) &&
      liveLat->isLiveVar(SgExpr2Var(sgn->get_lhs_operand()))))
      { */
    ldva.used(sgn->get_rhs_operand());
  }
...
}
```

call stack

```
(gdb) bt
#0  LDVAExpressionTransfer::visit (this=0x7fffffffcea0, sgn=0xa20320)

at ../../../../sourcetree/src/midend/programAnalysis/genericDataflow/si
mpleAnalyses/liveDeadVarAnalysis.C:228
#1  0x00002aaaac3d9968 in SgAssignOp::accept (this=0xa20320,
visitor=...) at Cxx_Grammar.C:143069
#2  0x00002aaaadc61c04 in LiveDeadVarsTransfer::visit (this=0xaf9e00,
sgn=0xa20320)

at ../../../../sourcetree/src/midend/programAnalysis/genericDataflow/si
mpleAnalyses/liveDeadVarAnalysis.C:384
#3  0x00002aaaadbbaef0 in ROSE_VisitorPatternDefaultBase::visit
(this=0xaf9e00, variable_SgBinaryOp=0xa20320)
at ../../../src/frontend/SageIII/Cxx_Grammar.h:316006
#4  0x00002aaaadbba04a in ROSE_VisitorPatternDefaultBase::visit
(this=0xaf9e00, variable_SgAssignOp=0xa20320)
at ../../../src/frontend/SageIII/Cxx_Grammar.h:315931
#5  0x00002aaaac3d9968 in SgAssignOp::accept (this=0xa20320,
visitor=...) at Cxx_Grammar.C:143069
```

```
#6  0x00002aaaadbcca0a in IntraUniDirectionalDataflow::runAnalysis
(this=0x7fffffffd9f0, func=..., fState=0xafbd18,
analyzeDueToCallers=true, calleesUpdated=...)

at ../../../../sourcetree/src/midend/programAnalysis/genericDataflow/an
alysis/dataflow.C:282
#7  0x00002aaaadbbf444 in IntraProceduralDataflow::runAnalysis
(this=0x7fffffffda00, func=..., state=0xafbd18)

at ../../../../sourcetree/src/midend/programAnalysis/genericDataflow/an
alysis/dataflow.h:74
#8  0x00002aaaadbb0966 in UnstructuredPassInterDataflow::runAnalysis
(this=0x7fffffffda50)

at ../../../../sourcetree/src/midend/programAnalysis/genericDataflow/an
alysis/analysis.C:467
#9  0x000000000040381a in main (argc=2, argv=0x7fffffffdba8)

at ../../../../../sourcetree/tests/roseTests/programAnalysisTests/gener
alDataFlowAnalysisTests/liveDeadVarAnalysisTest.C:101
```

Control flow graph and call graph

The generic dataflow framework works on virtual control flow graph in ROSE

Filtered virtual CFG

The raw virtual CFG may not be desirable for all kinds of analyses since it can have too many administrative nodes which are not relevant to a problem.

So the framework provides a filter parameter to the Analysis class. A default filter will be used unless you specify your own filter.

```
// Example filter funtion deciding if a CFGnNode should show up or not
bool gfilter (CFGNode cfgn)
{
  SgNode *node = cfgn.getNode();

  switch (node->variantT())
  {
    //Keep the last index for initialized names. This way the def of
the variable doesn't propagate to its assign initializer.
    case V_SgInitializedName:
      return (cfgn == node->cfgForEnd());

    // For function calls, we only keep the last node. The function is
actually called after all its parameters  are evaluated.
    case V_SgFunctionCallExp:
      return (cfgn == node->cfgForEnd());

   //For basic blocks and other "container" nodes, keep the node that
appears before the contents are executed
    case V_SgBasicBlock:
```

```
    case V_SgExprStatement:
    case V_SgCommaOpExp:
      return (cfgn == node->cfgForBeginning());

   // Must have a default case: return interesting CFGNode by default
in this example
    default:
      return cfgn.isInteresting();
  }
}

// Code using the filter function
int
main( int argc, char * argv[] )
{
  SgProject* project = frontend(argc,argv);
  initAnalysis(project);
  LiveDeadVarsAnalysis ldva(project);
  ldva.filter = gfilter; // set the filter to be your own one

  UnstructuredPassInterDataflow ciipd_ldva(&ldva);
  ciipd_ldva.runAnalysis();
  ....
}
```

Analysis driver

Key function:

bool IntraUniDirectionalDataflow::runAnalysis(const Function& func, NodeState* fState, bool analyzeDueToCallers, set<Function> calleesUpdated) // analysis/dataflow.C

Basic tasks: run the analysis by

- initialize data flow state: lattices and other information
- walk the CFG : find descendants from a current node
- call transfer function

Class hierarchy

- Analysis -> IntraProceduralAnalysis -> IntraProceduralDataflow -> IntraUnitDataflow --> IntraUniDirectionalDataflow (INTERESTING level)-> IntraBWDataflow -> LiveDeadVarsAnalysis

```
class Analysis {}; // an empty abstract class for any analysis

class IntraProceduralAnalysis : virtual public Analysis
//analysis/analysis.h ,  any intra procedural analysis, data flow or
not
{
  protected:
```

```
   InterProceduralAnalysis* interAnalysis;
  public:
    void setInterAnalysis(InterProceduralAnalysis* interAnalysis) //
connection to inter procedural analysis
    virtual bool runAnalysis(const Function& func, NodeState* state)=0;
// run this per function, NodeState stores lattices for each CFG node,
etc.
    virtual ~IntraProceduralAnalysis();
}


//No re-entry. analysis will be executed  once??,   data flow , intra-
procedural analysis
// now lattices are interested
class IntraProceduralDataflow : virtual public IntraProceduralAnalysis
//analysis/dataflow.h
{
// initialize lattice etc for a given dataflow node within a function
  virtual void genInitState (const Function& func, const DataflowNode&
n, const NodeState& state,
       std::vector<Lattice*>& initLattices, std::vector<NodeFact*>&
initFacts);

  virtual bool runAnalysis(const Function& func, NodeState* state, bool
analyzeDueToCallers, std::set<Function> calleesUpdated)=0;  // the
analysis on a function could be triggered by the state changes of
function's callers, or callees.

 std::set<Function> visited; // make sure a function is initialized
once when visited multiple times

}


class IntraUnitDataflow : virtual public IntraProceduralDataflow
{
  // transfer function: operate on lattices associated with a dataflow
node, considering its current state
 virtual bool transfer(const Function& func, const DataflowNode& n,
NodeState& state, const std::vector<Lattice*>& dfInfo)=0;

};


// Uni directional dataflow: either forward or backward, but not both
directions!
class IntraUniDirectionalDataflow : public IntraUnitDataflow {
public:
 bool runAnalysis(const Function& func, NodeState* state, bool
analyzeDueToCallers, std::set<Function> calleesUpdated);

protected:
  bool propagateStateToNextNode (
             const std::vector<Lattice*>& curNodeState, DataflowNode
curDFNode, int nodeIndex,
             const std::vector<Lattice*>& nextNodeState, DataflowNode
nextDFNode);
```

```
  std::vector<DataflowNode> gatherDescendants(std::vector<DataflowEdge>
edges,
                                                DataflowNode
(DataflowEdge::*edgeFn)() const);

        virtual NodeState*initializeFunctionNodeState(const Function
&func, NodeState *fState) = 0;
        virtual VirtualCFG::dataflow*
          getInitialWorklist(const Function &func, bool firstVisit,
bool analyzeDueToCallers, const set<Function> &calleesUpdated,
NodeState *fState) = 0;

        virtual vector<Lattice*> getLatticeAnte(NodeState *state) = 0;
        virtual vector<Lattice*> getLatticePost(NodeState *state) = 0;

        // If we're currently at a function call, use the associated
inter-procedural
        // analysis to determine the effect of this function call on
the dataflow state.
        virtual void transferFunctionCall(const Function &func, const
DataflowNode &n, NodeState *state) = 0;

        virtual vector<DataflowNode> getDescendants(const DataflowNode
&n) = 0;
        virtual DataflowNode getUltimate(const Function &func) = 0; //
ultimate what?   final CFG node?
};

class IntraBWDataflow  : public IntraUniDirectionalDataflow {//BW:
Backward
   public:

        IntraBWDataflow()
        {}

        NodeState* initializeFunctionNodeState(const Function &func,
NodeState *fState);

        VirtualCFG::dataflow*
          getInitialWorklist(const Function &func, bool firstVisit,
bool analyzeDueToCallers, const set<Function> &calleesUpdated,
NodeState *fState);

        virtual vector<Lattice*> getLatticeAnte(NodeState *state);
        virtual vector<Lattice*> getLatticePost(NodeState *state);

        void transferFunctionCall(const Function &func, const
DataflowNode &n, NodeState *state);

        vector<DataflowNode> getDescendants(const DataflowNode &n); //
next CFG nodes, depending on the direction
        { return gatherDescendants(n.inEdges(),
&DataflowEdge::source);  }

        DataflowNode getUltimate(const Function &func); // the last CFG
should be the start CFG of the function for a backward dataflow problem
```

```
        {   return cfgUtils::getFuncStartCFG(func.get_definition());   }

};
```

foward intra-procedural data flow analysis: e.g. reaching definition ()

- class IntraFWDataflow : public IntraUniDirectionalDataflow

Initialization: InitDataflowState

Used to initialized the lattices/facts for CFG nodes. It is an analysis by itself. unstructured pass

```
// super class: provides the driver of initialization: visit each CFG
node

class UnstructuredPassIntraAnalysis : virtual public
IntraProceduralAnalysis
{
public:
          // call the initialization function on each CFG node
          bool runAnalysis(const Function& func, NodeState* state);
         // to be implemented by InitDataflowState
        virtual void visit(const Function& func, const DataflowNode& n,
NodeState& state)=0;

}

bool UnstructuredPassIntraAnalysis::runAnalysis(const Function& func,
NodeState* state)
{
        DataflowNode funcCFGStart =
cfgUtils::getFuncStartCFG(func.get_definition());
        DataflowNode funcCFGEnd =
cfgUtils::getFuncEndCFG(func.get_definition());

        if(analysisDebugLevel>=2)
                Dbg::dbg <<
"UnstructuredPassIntraAnalysis::runAnalysis() function
"<<func.get_name().getString()<<"()\n";

        // iterate over all the nodes in this function
        for(VirtualCFG::iterator it(funcCFGStart);
it!=VirtualCFG::dataflow::end(); it++)
        {
                DataflowNode n = *it;
                // The number of NodeStates associated with the given
dataflow node
                //int numStates=NodeState::numNodeStates(n);
                // The actual NodeStates associated with the given
dataflow node
```

```
                const vector<NodeState*> nodeStates =
NodeState::getNodeStates(n);

                // Visit each CFG node
                for(vector<NodeState*>::const_iterator itS =
nodeStates.begin(); itS!=nodeStates.end(); itS++)
                        visit(func, n, *(*itS));
        }
        return false;
}
//-------------------- derived class provide link to a concrete
analysis, and visit() implementation
class InitDataflowState : public UnstructuredPassIntraAnalysis
{
        IntraProceduralDataflow* dfAnalysis;  // link to the dataflow
analysis to be initialized

        public:
        InitDataflowState(IntraProceduralDataflow* dfAnalysis/*,
std::vector<Lattice*> &initState*/)
        {
                this->dfAnalysis = dfAnalysis;
        }

        void visit(const Function& func, const DataflowNode& n,
NodeState& state);
};


void InitDataflowState::visit (const Function& func, const
DataflowNode& n, NodeState& state)
{
   ...
   dfAnalysis->genInitState(func, n, state, initLats, initFacts);
   state.setLattices((Analysis*)dfAnalysis, initLats);
   state.setFacts((Analysis*)dfAnalysis, initFacts);
   ....
}
```

worklist

list of CFG nodes, accessed through an iterator interface

auto_ptr<VirtualCFG::dataflow> workList(getInitialWorklist(func, firstVisit, analyzeDueToCallers, calleesUpdated, fState));

```
class iterator //Declared in cfgUtils/VirtualCFGIterator.h
{
public:
    std::list<DataflowNode> remainingNodes;
    std::set<DataflowNode> visited;
    bool initialized;
       protected:
```

```
        // returns true if the given DataflowNode is in the
remainingNodes list and false otherwise
        bool isRemaining(DataflowNode n);

        // advances this iterator in the given direction. Forwards if
fwDir=true and backwards if fwDir=false.
        // if pushAllChildren=true, all of the current node's unvisited
children (predecessors or successors,
        //    depending on fwDir) are pushed onto remainingNodes
        void advance(bool fwDir, bool pushAllChildren);

        public:
        virtual void operator ++ (int);

        bool eq(const iterator& other_it) const;

        bool operator==(const iterator& other_it) const;

        bool operator!=(const iterator& it) const;
...

};


void iterator::advance(bool fwDir, bool pushAllChildren)
{
        ROSE_ASSERT(initialized);
        /*printf("   iterator::advance(%d) remainingNodes.size()=%d\n",
fwDir, remainingNodes.size());
        cout<<"        visited=\n";
        for(set<DataflowNode>::iterator it=visited.begin();
it!=visited.end(); it++)
                cout << "            <"<<it->getNode()->class_name()<<"
| "<<it->getNode()<<" | "<<it->getNode()->unparseToString()<<">\n";*/
        if(remainingNodes.size()>0)
        {
                // pop the next CFG node from the front of the list
                DataflowNode cur = remainingNodes.front();
                remainingNodes.pop_front();

                if(pushAllChildren)
                {
                        // find its followers (either successors or
predecessors, depending on value of fwDir), push back
                        // those that have not yet been visited
                        vector<DataflowEdge> nextE;
                        if(fwDir)
                                nextE = cur.outEdges();
                        else
                                nextE = cur.inEdges();
                        for(vector<DataflowEdge>::iterator
it=nextE.begin(); it!=nextE.end(); it++)
                        {
                                DataflowNode nextN((*it).target()/*
need to put something here because DataflowNodes don't have a default
constructor*/);
                                if(fwDir) nextN = (*it).target();
```

```
                                else nextN = (*it).source();

                                /*cout << "      iterator::advance
"<<(fwDir?"descendant":"predecessor")<<": "<<

"<"<<nextN.getNode()->class_name()<<" | "<<nextN.getNode()<<" |
"<<nextN.getNode()->unparseToString()<<">, "<<

"visited="<<(visited.find(nextN) != visited.end())<<
                                                        "
remaining="<<isRemaining(nextN)<<"\n";*/

                                // if we haven't yet visited this node
and don't yet have it on the remainingNodes list
                                if(visited.find(nextN) == visited.end()
&&
                                        !isRemaining(nextN))
                                {
                                        //printf("   pushing back node
<%s: 0x%x: %s> visited=%d\n", nextN.getNode()->class_name().c_str(),
nextN.getNode(), nextN.getNode()->unparseToString().c_str(),
visited.find(nextN)!=visited.end());

remainingNodes.push_back(nextN);
                                }
                        }
                }

                // if we still have any nodes left remaining
                if(remainingNodes.size()>0)
                {
                        // take the next node from the front of the
list and mark it as visited
                        //visited[remainingNodes.front()] = true;
                        visited.insert(remainingNodes.front());
                }
        }
}



class dataflow :  public virtual iterator {};

class back_dataflow:  public virtual dataflow {};

void back_dataflow::operator ++ (int)
{
        advance(false, true);  // backward,  add all children
}


class IntraUniDirectionalDataflow : public IntraUnitDataflow
{  ...
   virtual VirtualCFG::dataflow*
          getInitialWorklist(const Function &func, bool firstVisit,
bool analyzeDueToCallers, const set<Function> &calleesUpdated,
NodeState *fState) = 0;
```

```
}
```

Implemented in derived classes:

- VirtualCFG::dataflow* IntraFWDataflow::getInitialWorklist ()
- VirtualCFG::dataflow* IntraBWDataflow::getInitialWorklist()

apply transfer function

b is a basic block in CFG

- $\mathrm{IN}[b] = \bigcup_{p \in pred[b]} \mathrm{OUT}[p]$ // information goes into b is the union/join of information comes out of all predecessor nodes of b
- $OUT[b] = \mathrm{GEN}[b] \cup (\mathrm{IN}[b] - \mathrm{KILL}[b])$// information goes out out S is the information generated by b minus information killed by b. This is the transfer function operating on b!!

```
bool IntraUniDirectionalDataflow::runAnalysis(const Function& func,
NodeState* fState, bool analyzeDueToCallers, set<Function>
calleesUpdated)
{

      // Iterate over the nodes in this function that are downstream
from the nodes added above
        for(; it != itEnd; it++)
        {
                DataflowNode n = *it;
                SgNode* sgn = n.getNode();

  ...
               for(vector<NodeState*>::const_iterator itS =
nodeStates.begin(); itS!=nodeStates.end(); )
                {
                        state = *itS;

                        const vector<Lattice*> dfInfoAnte =
getLatticeAnte(state);  // IN set
                        const vector<Lattice*> dfInfoPost =
getLatticePost(state);   // OUT set

                            // OUT = IN first     // transfer within
the node: from IN to OUT,
                        // Overwrite the Lattices below this node with
the lattices above this node.
                        // The transfer function will then operate on
these Lattices to produce the
                        // correct state below this node.

                        vector<Lattice*>::const_iterator itA, itP;
                        int j=0;
```

```
                        for(itA  = dfInfoAnte.begin(), itP  =
dfInfoPost.begin();
                            itA != dfInfoAnte.end() && itP !=
dfInfoPost.end();
                            itA++, itP++, j++)
                        {
                                if(analysisDebugLevel>=1){  //
                                        Dbg::dbg << "    Meet Before:
Lattice "<<j<<": \n        "<<(*itA)->str("            ")<<endl;
                                        Dbg::dbg << "    Meet After:
Lattice "<<j<<": \n        "<<(*itP)->str("            ")<<endl;
                                }
                                (*itP)->copy(*itA);
                                /*if(analysisDebugLevel>=1){
                                        Dbg::dbg << "    Copied Meet
Below: Lattice "<<j<<": \n        "<<(*itB)->str("            ")<<endl;
                                }*/
                        }

                        // ================== TRANSFER FUNCTION
==================
                        //  (IN - KILL ) + GEN
                        if (isSgFunctionCallExp(sgn))
                          transferFunctionCall(func, n, state);

                        boost::shared_ptr<IntraDFTransferVisitor>
transferVisitor = getTransferVisitor(func, n, *state, dfInfoPost);
                        sgn->accept(*transferVisitor);
                        modified = transferVisitor->finish() ||
modified;

                        // ================== TRANSFER FUNCTION
==================
        ...//
    }

}
```

propagate state to next (meetUpdate)

This is prove to be essential to propagate information along the path. Cannot commenting it out!!

??? not sure about the difference between this step and the step before (Meet Before () / Meet After)

meetUpdate() is called here also

```
// Propagates the dataflow info from the current node's NodeState
(curNodeState) to the next node's
//     NodeState (nextNodeState).
// Returns true if the next node's meet state is modified and false
otherwise.
bool IntraUniDirectionalDataflow::propagateStateToNextNode(
```

```
                      const vector<Lattice*>& curNodeState,
DataflowNode curNode, int curNodeIndex,
                      const vector<Lattice*>& nextNodeState,
DataflowNode nextNode)
{
        bool modified = false;
        vector<Lattice*>::const_iterator itC, itN;
        if(analysisDebugLevel>=1){
                Dbg::dbg << "\n        Propagating to Next Node:
"<<nextNode.getNode()<<"["<<nextNode.getNode()->class_name()<<" |
"<<Dbg::escape(nextNode.getNode()->unparseToString())<<"]"<<endl;
                int j;
                for(j=0, itC = curNodeState.begin(); itC !=
curNodeState.end(); itC++, j++)
                        Dbg::dbg << "        Cur node: Lattice "<<j<<":
\n            "<<(*itC)->str("            ")<<endl;
                for(j=0, itN = nextNodeState.begin(); itN !=
nextNodeState.end(); itN++, j++)
                        Dbg::dbg << "        Next node: Lattice
"<<j<<": \n            "<<(*itN)->str("            ")<<endl;
        }

        // Update forward info above nextNode from the forward info
below curNode.

        // Compute the meet of the dataflow information along the
curNode->nextNode edge with the
        // next node's current state one Lattice at a time and save the
result above the next node.
        for(itC = curNodeState.begin(), itN = nextNodeState.begin();
            itC != curNodeState.end() && itN != nextNodeState.end();
            itC++, itN++)
        {
                // Finite Lattices can use the regular meet operator,
while infinite Lattices
                // must also perform widening to ensure convergence.
                if((*itN)->finiteLattice())
                        modified = (*itN)->meetUpdate(*itC) ||
modified;
                else
                {
                        //InfiniteLattice* meetResult =
(InfiniteLattice*)itN->second->meet(itC->second);
                        InfiniteLattice* meetResult =
dynamic_cast<InfiniteLattice*>((*itN)->copy());
                        Dbg::dbg << "        *itN: " <<
dynamic_cast<InfiniteLattice*>(*itN)->str("            ") << endl;
                        Dbg::dbg << "        *itC: " <<
dynamic_cast<InfiniteLattice*>(*itC)->str("            ") << endl;
                        meetResult->meetUpdate(*itC);
                        Dbg::dbg << "        meetResult: " <<
meetResult->str("            ") << endl;

                        // Widen the resulting meet
                        modified =
dynamic_cast<InfiniteLattice*>(*itN)->widenUpdate(meetResult);
                        delete meetResult;
```

```
                }
        }

        if(analysisDebugLevel>=1) {
                if(modified)
                {
                        Dbg::dbg << "        Next node's in-data
modified. Adding..."<<endl;
                        int j=0;
                        for(itN = nextNodeState.begin(); itN !=
nextNodeState.end(); itN++, j++)
                        {
                                Dbg::dbg << "        Propagated:
Lattice "<<j<<": \n            "<<(*itN)->str("            ")<<endl;
                        }
                }
                else
                        Dbg::dbg << "        No modification on this
node"<<endl;
        }

        return modified;
}
```

stop condition

```
class IntraUniDirectionalDataflow : public IntraUnitDataflow
{
public:
         protected:
        // propagates the dataflow info from the current node's
NodeState (curNodeState) to the next node's  NodeState (nextNodeState)
           // return true if any state is modified.
        bool propagateStateToNextNode(
             const std::vector<Lattice*>& curNodeState, DataflowNode
curDFNode, int nodeIndex,
             const std::vector<Lattice*>& nextNodeState, DataflowNode
nextDFNode);

}
```

live dead variable

Backward Intra-Procedural Dataflow Analysis: e.g. liveness analysis (use --> backward --> defined)

- class IntraBWDataflow : public IntraUniDirectionalDataflow

```
class LiveDeadVarsAnalysis : public IntraBWDataflow {

  protected:
        funcSideEffectUses* fseu;
```

```
  public:
        LiveDeadVarsAnalysis(SgProject *project, funcSideEffectUses*
fseu=NULL);

 // Generates the initial lattice state for the given dataflow node, in
the given function, with the given NodeState
 void genInitState(const Function& func, const DataflowNode& n, const
NodeState& state,
                          std::vector<Lattice*>& initLattices,
std::vector<NodeFact*>& initFacts);


  boost::shared_ptr<IntraDFTransferVisitor> getTransferVisitor(const
Function& func, const DataflowNode& n,

NodeState& state, const std::vector<Lattice*>& dfInfo)
  { return boost::shared_ptr<IntraDFTransferVisitor>(new
LiveDeadVarsTransfer(func, n, state, dfInfo, fseu)); }

  bool transfer(const Function& func, const DataflowNode& n, NodeState&
state, const std::vector<Lattice*>& dfInfo) { assert(0); return
false; }


};
```

Inter-procedural analysis

Key: transfer function that is applied to call sites to perform the appropriate state transfers across function boundaries.

transfer function

```
void IntraFWDataflow::transferFunctionCall(const Function &func, const
DataflowNode &n, NodeState *state)
{
  vector<Lattice*> dfInfoBelow = state->getLatticeBelow(this);

  vector<Lattice*>* retState = NULL;
  dynamic_cast<InterProceduralDataflow*>(interAnalysis)->
    transfer(func, n, *state, dfInfoBelow, &retState, true);

  if(retState && !(retState->size()==0 || (retState->size() ==
dfInfoBelow.size()))) {
    Dbg::dbg << "#retState="<<retState->size()<<endl;
    for(vector<Lattice*>::iterator ml=retState->begin(); ml!=retState-
>end(); ml++)
      Dbg::dbg << "        "<<(*ml)->str("            ")<<endl;
    Dbg::dbg << "#dfInfoBelow="<<dfInfoBelow.size()<<endl;
    for(vector<Lattice*>::const_iterator l=dfInfoBelow.begin();
l!=dfInfoBelow.end(); l++)
      Dbg::dbg << "        "<<(*l)->str("           ")<<endl;
  }
```

```
  // Incorporate information about the function's return value into the
caller's dataflow state
  // as the information of the SgFunctionCallExp
  ROSE_ASSERT(retState==NULL || retState->size()==0 || (retState-
>size() == dfInfoBelow.size()));
  if(retState) {
    vector<Lattice*>::iterator lRet;
    vector<Lattice*>::const_iterator lDF;
    for(lRet=retState->begin(), lDF=dfInfoBelow.begin();
        lRet!=retState->end(); lRet++, lDF++) {
      Dbg::dbg << "    lDF Before="<<(*lDF)->str("        ")<<endl;
      Dbg::dbg << "    lRet Before="<<(*lRet)->str("        ")<<endl;
      (*lDF)->unProject(isSgFunctionCallExp(n.getNode()), *lRet);
      Dbg::dbg << "    lDF After="<<(*lDF)->str("        ")<<endl;
    }
  }
}
```

InterProceduralDataflow

```
InterProceduralDataflow::InterProceduralDataflow(IntraProceduralDataflo
w* intraDataflowAnalysis) :

InterProceduralAnalysis((IntraProceduralAnalysis*)intraDataflowAnalysis
)


 // !!! NOTE: cfgForEnd() AND cfgForBeginning() PRODUCE THE SAME
SgFunctionDefinition SgNode BUT THE DIFFERENT INDEXES
                        // !!!       (0 FOR BEGINNING AND 3 FOR END).
AS SUCH, IT DOESN'T MATTER WHICH ONE WE CHOOSE. HOWEVER, IT DOES MATTER
                        // !!!       WHETHER WE CALL genInitState TO
GENERATE THE STATE BELOW THE NODE (START OF THE FUNCTION) OR ABOVE IT
                        // !!!       (END OF THE FUNCTION). THE
CAPABILITY TO DIFFERENTIATE THE TWO CASES NEEDS TO BE ADDED TO
genInitState
                        // !!!       AND WHEN IT IS, WE'LL NEED TO CALL
IT INDEPENDENTLY FOR cfgForEnd() AND cfgForBeginning() AND ALSO TO MAKE
                        // !!!       TO SET THE LATTICES ABOVE THE
ANALYSIS
```

TODO: begin and end func definition issue is mentioned inside of this

simplest form:unstructured

Simplest form: No transfer action at call sites at all

```
class UnstructuredPassInterDataflow : virtual public
InterProceduralDataflow
{
```

```
        public:

        UnstructuredPassInterDataflow(IntraProceduralDataflow*
intraDataflowAnalysis)
                                :
InterProceduralAnalysis((IntraProceduralAnalysis*)intraDataflowAnalysis
), InterProceduralDataflow(intraDataflowAnalysis)
        {}

        // the transfer function that is applied to SgFunctionCallExp
nodes to perform the appropriate state transfers
        // fw - =true if this is a forward analysis and =false if this
is a backward analysis
        // n - the dataflow node that is being processed
        // state - the NodeState object that describes the dataflow
state immediately before (if fw=true) or immediately after
        //         (if fw=false) the SgFunctionCallExp node, as
established by earlier analysis passes
        // dfInfo - the Lattices that this transfer function operates
on. The function propagates them
        //          to the calling function and overwrites them with
the dataflow result of calling this function.
        // retState - Pointer reference to a Lattice* vector that will
be assigned to point to the lattices of
        //          the function call's return value. The callee may
not modify these lattices.
        // Returns true if any of the input lattices changed as a
result of the transfer function and
        //    false otherwise.
        bool transfer(const Function& func, const DataflowNode& n,
NodeState& state,
                      const std::vector<Lattice*>& dfInfo,
std::vector<Lattice*>** retState, bool fw)
        {
                return false;
        }

        void runAnalysis();
};

// simply call intra-procedural analysis on each function one by one.
void UnstructuredPassInterDataflow::runAnalysis()
{
        set<FunctionState*> allFuncs =
FunctionState::getAllDefinedFuncs();

        // iterate over all functions with bodies
        for(set<FunctionState*>::iterator it=allFuncs.begin();
it!=allFuncs.end(); it++)
        {
                const Function& func = (*it)->func;
                FunctionState* fState =
FunctionState::getDefinedFuncState(func);

                // Call the current intra-procedural dataflow as if it
were a generic analysi
                intraAnalysis->runAnalysis(func, &(fState->state));
```

```
        }
}
```

ContextInsensitiveInterProceduralDataflow

TODO

How to use one analysis

Call directly

Direct call: Runs the intra-procedural analysis on the given function and returns true if the function's NodeState gets modified as a result and false otherwise state - the function's NodeState

- bool IntraUniDirectionalDataflow::runAnalysis(const Function& func, NodeState* state, bool analyzeDueToCallers, std::set<Function> calleesUpdated);
- direct call with a simpler parameter list : not feasible, all intra procedural analysis has to have an inter procedural analysis set interally!

```
bool IntraProceduralDataflow::runAnalysis(const Function& func,
NodeState* state)
{
   // Each function is analyzed as if it were called directly by the
language's runtime, ignoring
  // the application's actual call graph
    bool analyzeDueToCallers = true;

    // We ignore the application's call graph, so it doesn't matter
whether this function calls other functions
   std::set<Function> calleesUpdated;

      return runAnalysis(func, state, analyzeDueToCallers,
calleesUpdated);
}
```

Through inter-procedural analysis

Invoke a simple intra-procedural analysis through the unstructured pass inter-procedural data flow class

```
int main()
{
  SgProject* project = frontend(argc,argv);
  initAnalysis(project);

  // prepare debugging support
  Dbg::init("Live dead variable analysis Test", ".", "index.html");
  liveDeadAnalysisDebugLevel = 1;
```

```
  analysisDebugLevel = 1;

  // basis analysis
  LiveDeadVarsAnalysis ldva(project);
     // wrap it inside the unstructured inter-procedural data flow
   UnstructuredPassInterDataflow ciipd_ldva(&ldva);
   ciipd_ldva.runAnalysis();

   .....

}
```

Retrieve lattices

Sample code:

```
// Initialize vars to hold all the variables and expressions that are
live at DataflowNode n
//void getAllLiveVarsAt(LiveDeadVarsAnalysis* ldva, const DataflowNode&
n, const NodeState& state, set<varID>& vars, string indent)
void getAllLiveVarsAt(LiveDeadVarsAnalysis* ldva, const NodeState&
state, set<varID>& vars, string indent)
{
        LiveVarsLattice* liveLAbove =
dynamic_cast<LiveVarsLattice*>(*(state.getLatticeAbove(ldva).begin()));
        LiveVarsLattice* liveLBelow =
dynamic_cast<LiveVarsLattice*>(*(state.getLatticeBelow(ldva).begin()));

        // The set of live vars AT this node is the union of vars that
are live above it and below it
        for(set<varID>::iterator var=liveLAbove->liveVars.begin();
var!=liveLAbove->liveVars.end(); var++)
                vars.insert(*var);
        for(set<varID>::iterator var=liveLBelow->liveVars.begin();
var!=liveLBelow->liveVars.end(); var++)
                vars.insert(*var);
}
```

How to debug

Trace the analysis

Turn it on

```
     liveDeadAnalysisDebugLevel = 1;
     analysisDebugLevel = 1;


// find code with
 if(analysisDebugLevel>=1) ...
```

check the web page dump using a browser

```
 firefox index.html
```

How to read the trace file: start from the beginning: information is ordered based on the CFG nodes visited. The order could be forward or backward order. Check if the order is correct first, then for each node visited

```
 ================================
  Copying incoming Lattice 0:
        [LiveVarsLattice: liveVars=[b]]
  To outgoing Lattice 0:
        [LiveVarsLattice: liveVars=[]]
 ================================
  Transferring the outgoing  Lattice ...
    liveLat=[LiveVarsLattice: liveVars=[b]]
    Dead Expression
        usedVars=<>
        assignedVars=<>
        assignedExprs=<>
        #usedVars=0 #assignedExprs=0
    Transferred: outgoing Lattice 0:
        [LiveVarsLattice: liveVars=[b]]
    transferred, modified=0
 ================================
 Propagating/Merging the outgoing  Lattice to all descendant nodes ...
    Descendants (1):
    ~~~~~~~~~~~~
    Descendant: 0x2b9e8c47f010[SgIfStmt | if(flag == 0) c = a;else c =
b;]

        Propagating to Next Node: 0x2b9e8c47f010[SgIfStmt | if(flag ==
0) c = a;else c = b;]
        Cur node: Lattice 0:
            [LiveVarsLattice: liveVars=[b]]
        Next node: Lattice 0:
            [LiveVarsLattice: liveVars=[a]]
        Next node's in-data modified. Adding...
        Propagated: Lattice 0:
            [LiveVarsLattice: liveVars=[a, b]]
    propagated/merged, modified=1
    ^^^^^^^^^^^^^^^^^

A real example: if (flag)  c = a; else c = b;  // liveness analysis, a,
b are live in two branches, they are propagated backward to if-stmt

   ------------------
    Descendants (1):  // from c =a back to if-stmt (next node)
    ~~~~~~~~~~~~
    Descendant: 0x2ac8bb95c010[SgIfStmt | if(flag == 0) c = a;else c =
b;]

        Propagating to Next Node: 0x2ac8bb95c010[SgIfStmt | if(flag ==
0) c = a;else c = b;]
        Cur node: Lattice 0:
            [LiveVarsLattice: liveVars=[a]]   // current node's lattice
        Next node: Lattice 0:
```

```
            [LiveVarsLattice: liveVars=[]]   // next node's lattice
before propagation
        Next node's in-data modified. Adding...
        Propagated: Lattice 0:
            [LiveVarsLattice: liveVars=[a]]  // propagate a into if-
stmt's lattice
    propagated, modified=1
    ^^^^^^^^^^^^^^^^^^

    ------------------
    Descendants (1):  // from c = b --> if-stmt
    ~~~~~~~~~~~~
    Descendant: 0x2ac8bb95c010[SgIfStmt | if(flag == 0) c = a;else c =
b;]

        Propagating to Next Node: 0x2ac8bb95c010[SgIfStmt | if(flag ==
0) c = a;else c = b;]
        Cur node: Lattice 0:
            [LiveVarsLattice: liveVars=[b]]
        Next node: Lattice 0:
            [LiveVarsLattice: liveVars=[a]]
        Next node's in-data modified. Adding...
        Propagated: Lattice 0:
            [LiveVarsLattice: liveVars=[a, b]]  // now both a and b are
propagated/ merged
    propagated, modified=1
    ^^^^^^^^^^^^^^^^^^
```

Dump cfg dot graph with lattices

A class analysisStatesToDot is provided generate a CFG dot graph with lattices information.

```
//AnalysisDebuggingUtils.C

  class analysisStatesToDOT : public UnstructuredPassIntraAnalysis
  {
    private:
      //    LiveDeadVarsAnalysis* lda; // reference to the source
analysis
      Analysis* lda; // reference to the source analysis
      void printEdge(const DataflowEdge& e); // print data flow edge
      void printNode(const DataflowNode& n, std::string state_string);
// print date flow node
      void visit(const Function& func, const DataflowNode& n,
NodeState& state); // visitor function
    public:
      std::ostream* ostr;
      analysisStatesToDOT (Analysis* l):  lda(l){ };
  };

namespace Dbg
{
//....
 void dotGraphGenerator (::Analysis *a)
```

```
  {
    ::analysisStatesToDOT eas(a);
    IntraAnalysisResultsToDotFiles upia_eas(eas);
    upia_eas.runAnalysis();
  }

} // namespace Dbg
```

Example use

```
// Liao, 12/6/2011
#include "rose.h"

#include <list>
#include <sstream>
#include <iostream>
#include <fstream>
#include <string>
#include <map>

using namespace std;

// TODO group them into one header
#include "genericDataflowCommon.h"
#include "VirtualCFGIterator.h"
#include "cfgUtils.h"
#include "CallGraphTraverse.h"
#include "analysisCommon.h"
#include "analysis.h"
#include "dataflow.h"
#include "latticeFull.h"
#include "printAnalysisStates.h"
#include "liveDeadVarAnalysis.h"

int numFails = 0, numPass = 0;

//-------------------------------------------------------------
int
main( int argc, char * argv[] )
   {

     SgProject* project = frontend(argc,argv);

     initAnalysis(project);

   // generating  index.html for tracing the analysis
     Dbg::init("Live dead variable analysis Test", ".", "index.html");
     liveDeadAnalysisDebugLevel = 1;
     analysisDebugLevel = 1;

     LiveDeadVarsAnalysis ldva(project);
     UnstructuredPassInterDataflow ciipd_ldva(&ldva);
     ciipd_ldva.runAnalysis();
   // Output the dot graph  ********************
    Dbg::dotGraphGenerator (&ldva);
```

```
    return 0;
}
```

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 ROSE developers should have, or acquire:

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

Milestones for a ROSE developers

Having been working with some interns with us, we roughly identify the following milestones for a ROSE developer

- Development environment: pick a platform of your choice (Linux or Mac OS), and get familiar with that specific platform (shell, editors, environment variable setting, etc.)
- Installing ROSE: being able to smoothly configure, compile, and install ROSE
- Build system: being able to add a project (first skeleton) into ROSE by modifying `Makefile.am`, etc.
- Contribution following ROSE Coding Standard and passing code review
    - **Documentation**: sufficient documentation about what you work is about
    - **Software Engineering**:
        - **Style guidelines**: Doxygen comments, naming conventions, where to put things, etc.
        - **Algorithm design**: documented by source comments how things are expected to work
        - **Coding implementation**: correctly implement the designed algorithm
    - **Tests**: Each contribution must have the accompanying tests to make sure it works as expected
- Continuous integration: pass Jenkins tests
    - Add a new test job if none of the existing ones tests your project

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

Development Guide

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.

Each ROSE developer is expected to push his/her work **at least once every three weeks**.

Major benefits of doing things incrementally

- You can have intermediate results along the path. So your sponsors will sleep better.
- You will get feedback early and frequently about if you are heading to the right direction.
- Your work will be tested and merged often into the master branch, avoiding the risks of merge conflicts.

See more tips about How to incrementally work on a project

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.

High Level Workflow

Requirement Analysis

- External (https://github.com/rose-compiler/rose): start an issue to be discussed

- Wikibook:
  - collect community input
- mailing list: interaction with users, feel users' need

Design

- Wikibook: community-based 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
  - Project-Specific Tasks
  - Private Issue Tracking
  - Private Documentation
    - Using redmine's wiki
- Github:
  - Internal (http://github.llnl.gov/): for code review only,
  - External (https://github.com/rose-compiler/rose): public hosting code, pubic issue tracking for general ROSE bugs and features.
  - "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

Proposing 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, provide the following information:
    - What is it: Explain what change is proposed
    - Why the changes: the long-term benefits for our productivity and quality of work
    - The cost of the changes: learning curve, maintainability, purchase cost

Reviewing Workflow Change Proposals

Review criteria

- Optimize
    - 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. Compare how we can eliminate or automate using the proposed workflow improvements.
        - It is counterproductive to improve workflow by adding more hoops/steps/clicks into daily work.
- Improve:
    - Allows the improvement of the quality of work incrementally:
    - Accepting incremental improvements is more realistic than asking for perfection in the first try.
    - Workflow should allow quick new contributions and fast revision of existing contributions
- Automate:
    - Additions to the workflow should be automated as much as possible.
- Preserve:
    - It must preserve existing work:
        - No creation of anything from scratch
    - Does it interact well with existing workflow
    - Is there a way to convert existing code/documents into the new form
- Simplicity:
    - 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
    - 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)

Coding Standard

What to Expect and What to Avoid

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

New code should follow the conventions described in this document from the very beginning.

Updates to **existing code** that follows a different coding style should only be performed if you are the maintainer of the code.

The order of sections in coding standard follows a top-down approach: big things first, then drill down to fine-grain details.

Five Principles

We use coding standard to reflect the principal things we value for all contributions to ROSE

- **Documentation**: What are the commits about? Is this reflected in README, source comments, or LaTex files within the same commits?
- **Style**: Is the coding style consistent with the required and recommended formats? Is the code clean and pleasant and easy to read?
- **Algorithm**: Does the code has sufficient comments about what algorithm is used? Is the algorithm correct and efficient(space and time complexity)?
- **Implementation**: Does the implementation correctly implement the documented algorithms?
- **Testing**: Does the code has the accompanying test translator and input to ensure the contributions do what they are supposed to do?
    - Is Jenkins being configured to trigger these tests? Local tests on developer's workstation do not count.

Avoid Coding Standard War

We directly quote text from http://www.parashift.com/c++-faq/coding-std-wars.html, as follows:

"Nearly every software engineer has, at some point, been exploited by someone who used coding standards as a **power play**. Dogmatism over minutia is the purvue of the intellectually weak. Don't be like them. These are those who can't contribute in any meaningful way, who can't actually improve the value of the software product, so instead of exposing their incompetence through silence, they blather with zeal about nits. **They**

can't add value in the substance of the software, so they argue over form. Just because "they" do that doesn't mean coding standards are bad, however.

Another emotional reaction against coding standards is caused by coding standards **set by individuals with obsolete skills**. For example, someone might set today's standards based on what programming was like N decades ago when the standards setter was writing code. Such impositions generate an attitude of mistrust for coding standards. As above, if you have been forced to endure an unfortunate experience like this, don't let it sour you to the whole point and value of coding standards. It doesn't take a very large organization to find there is value in having consistency, since different programmers can edit the same code without constantly reorganizing each others' code in a tug-of-war over the "best" coding standard."

Must, Should and Can

The terms must, should and can have special meaning.

- A must requirement must be followed,
- A should is a strong recommendation,
- A can is a general guideline.

Got New Ideas, Suggestions

This is not a place to write down the new ideas/concepts/suggestions to be used in the future. If you have suggestions, put into the discussion tab link of this page.

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

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

Programming Languages

Core Languages

Only C++ is allowed. Any other programming language is an exception on a case-by-case basis.

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

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.

Naming Conventions

The order of sub-sections reflects a top-down approach for how things are added during the development cycle: from directory --> file --> namespace --> etc.

General

- Language: all names should be written in English since it is the preferred language for development, internationally
- fileName; // NOT: filNavn

Abbreviations and Acronyms

Avoid ambiguous abbreviations: obtain good balance between user-clarity and -productivity.

Abbreviations and acronyms should NOT be uppercase when used as name

- exportHtmlSource(); // NOT: exportHTMLSource();
- openDvdPlayer(); // NOT: openDVDPlayer();

File/Directory

Case:

- **`camelCase`** like `fileName.hpp`: This is consistent with existing names used in ROSE

File Extension:

- Header files: `.h` or `.hpp`
- Source files: `.cpp` or `.cxx`
    - `.C` should be avoided to work with file systems which do not distinguish between lower or upper case.

Namespaces

- A namespace should represent a **logical unit**, usually encapsulated in a single header file within a specific directory.
- **CamelCase** for namespaces, such as SageInterface, SageBuilder, etc.
    - avoid lower case names, bad names: sage_interface
- use singular for nouns within namespace names, avoid plural
- use full words, avoid abbreviations

Reason: the name convention of namespace is meant to be compatible with existing code and consistent with function names within namespaces.

- CamelCase namespace can nice be used with doSomething() like: NameSpace::doSomething()
- lower case namespace names may look inconsistent, such as name_space_1::doSomething()
- many existing namespaces in ROSE already follow CamelCase, as shown at link

[Note] Leo: I believe this should be more discussed with ROSE Compiler Framework/ROSE API.

Types

MUST be in mixed case starting with an uppercase letter, as in `SavingsAccount`

Variables

- **Length**: variables with a large scope should have long names, variables with a small scope can have short names
- ***Temporary* variables** used for temporary storage (e.g. loop indices) are best kept short. A programmer reading such variables should be able to assume that its value is not used outside of a few lines of code. Common scratch variables for integers are `i`, `j`, `k`, `m`, `n`. Optionally, you can use ii, jj, kk, mm, and nn, which are easier to highlight when looking for indexing bugs.

- **Case**: `camelCase`--mixed case starting with lowercase letter, as in `functionDecl`

- Variables are purposely to start with lowercase letter as compared to upper case letter for Types. So it is clear by looking at the first letter to know if a name is a variable or a type.

Booleans

Negated boolean variable names must be avoided. The problem arises when such a name is used in conjunction with the logical negation operator as this results in a double negative. It is not immediately apparent what !isNotFound means.

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

Collections

Plural form should be used on names representing a collection of objects. This enhances readability since the name gives the user an immediate clue as to the type of the variable and the operations that can be performed on its elements.

For example,

```
vector<Point> points;
int values[];
```

Constants

Named constants (including enumeration values): MUST be all uppercase using underscore to separate words.

For example:

```
int MAX_ITERATIONS, COLOR_RED;
double PI;
```

In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice:

```
int getMaxIterations() // NOT: MAX_ITERATIONS = 25
{
    return 25;
}
```

Generic

Generic variables should have the same name as their type. This reduces complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only. If for some reason this convention doesn't seem to fit it is a strong indication that the type name is badly chosen.

```
void setTopic(Topic* topic) // NOT: void setTopic(Topic* value)
                            // NOT: void setTopic(Topic* aTopic)
                            // NOT: void setTopic(Topic* t)

void connect(Database* database) // NOT: void connect(Database* db)
                                 // NOT: void connect (Database*
oracleDB)
```

Non-generic variables have a role. These variables can often be named by combining role and type:

```
Point  startingPoint, centerPoint;
Name   loginName;
```

Globals

Must always be fully qualified, using the scope-resolution operator `::`.

For example, `::mainWindow.open()` and `::applicationContext.getName()`

In general, the use of global variables should be avoided. Instead,

- Place variable into a namespace
- Use singleton objects

Private class variables

Private class variables should have underscore suffix. Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables.

For example,

```
class SomeClass {
  private:
    int length_;
}
```

An issue is whether the underscore should be added as a prefix or as a suffix. Both practices are commonly used, but the latter is recommended because it seem to best preserve the readability of the name. A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods and constructors:

```
  void setDepth (int depth)
  {
    depth_ = depth;
  }
```

Methods and Functions

Names representing methods or functions: MUST be **verbs** and written in **mixed case** starting with lower case to indicate what they return and procedures (void methods) after what they do.

- e.g. getName(), computeTotalWidth(), isEmpty()

A method name should **avoid duplicated object name**.

- e.g. line.getLength(); // NOT: line.getLineLength();

The latter seems natural in the class declaration, but proves superfluous in use, as shown in the example.

The terms **`get`** **and** **`set`** must be used where an attribute is accessed directly.

- e.g: employee.getName(); employee.setName(name); matrix.getElement(2, 4); matrix.setElement(2, 4, value);

The term **`compute`** can be used in methods where **something is computed**.

- e.g: valueSet->computeAverage(); matrix->computeInverse()

Give the reader the immediate clue that this is a potentially time-consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

The term **`find`** can be used in methods where **something is looked up**.

- e.g.: vertex.findNearestVertex(); matrix.findMinElement();

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

The term **`initialize`** can be used **where an object or a concept is established**.

- e.g: printer.initializeFontSet();

The american initialize should be preferred over the English initialise. Abbreviation init should be avoided.

The prefix **`is`** should be used for **boolean variables and methods**.

- e.g: isSet, isVisible, isFinished, isFound, isOpen

There are a few alternatives to the `is` prefix that fit better in some situations. These are the **`has`, `can` and `should`** prefixes:

- bool hasLicense();
- bool canEvaluate();
- bool shouldSort();

Directories

Naming Convention

List of common names

- `src`: to put source files, headers
- `include`: to put headers if you have many headers and don't want to put them all into ./src
- `tests`: put test inputs
- `docs`: detailed documentation not covered by README

Please use camelCase for your directory name.

- you should avoid leading Capitalization

Examples of preferred names

- roseExtensions
- roseSupport
- roseAPI

What to avoid

- rose_api
- rose_support

Layout

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

For each project directory under ./projects, it is our convention to have subdirectories for different files

- README: must have this
- ./src: for all your source files
- ./include: for all your headers if you don't want to put them all into ./src

- ./tests: for your test input files
- ./doc: for your more extensive documentation if README is not enough

Files

A single file should contain one *logical* unit, or feature. Keep it modular!

Naming Conventions

A file name should be specific and descriptive about what it contains.

You should use camelCase (lowercase character in the beginning)

- good example: fileName.h

What should be avoided

- start with capitalization,
- bad example using underscore: file_name.h

Bad file name

- functions.h
- file_name.h

References

- http://geosoft.no/development/cppstyle.html/cppstyle.html#Files
- A couple good points: http://www.records.ncdcr.gov/erecords/filenaming_20080508_final.pdf

Line length

- File content should be kept within 80 columns.

80 columns is a common dimension for editors, terminal emulators, printers and debuggers, and files that are shared between several people should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

Indentation

Avoid tabs for your code indentation, except in cases where tabs (`\t`) are required, e.g. `Makefile`s.

2 or 4 spaces is recommended for code indentation.

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Indentation of 1 is too small to emphasize the logical layout of the code. Indentation larger than 4 makes deeply nested code difficult to read and increases the chance that the lines must be split.

Characters

- Special characters like TAB and page break must be avoided.

These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

We already have a built-in perl script to enforce this policy.

Header files

File name:

- must be camelCase: such as fileName.h or fileName.hpp
- avoid file_name.h

Suffix

- For C header files: Use `.h`
- For C++ header files: Use `.h` or `.hpp`

Must have

- protected preprocesssing directives to prevent the header from being included more than once, example

```
#ifndef _HEADER_FILE_X_H_
#define _HEADER_FILE_X_H_

#endif //_HEADER_FILE_X_H_
```

- try to put your variables, functions, classes within a descriptive namespace.

- Include statements must be located at the top of a file only.
    - Avoid unwanted compilation side effects by "hidden" include statements deep into a source file.

What to avoid

- global variables, functions, or classes ; // they will pollute the global scope
- using namespace std;
    - this will pollute the global scope for each .cpp file which includes this header. using namespace should only be used by .cpp files. More explanations are at link and link2
- function definitions

References:

- http://www.parashift.com/c++-faq/hdr-file-ext.html

Source files

Again, file names should follow the name convention

- camelCase file name: e.g. sageInterface.cpp
- Avoid capitalization, spaces, special characters

Preferred suffix

- Use `.c` for C source files
- Use `.cpp` or `.cxx` for C++ source files

What to avoid

- capitalized `.C` for source files. This will cause some issue when porting ROSE to case-insensitive file systems.

References

- http://www.parashift.com/c++-faq/src-file-ext.html

README

File name should be README

what to avoid

- README.txt
- readme

Required Content

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

- What is in this directory
- What does this directory accomplish
- 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 complete documentation
- Status
  - What works
  - What doesn't work
- Known limitations
- References and citations: for the underlying algorithms
- Authors and Dates

Format

Format of README

- text format with clear sections and bullets
- optionally, you can use styles defined by w:Markdown

Examples

An example README can be found at

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

Source Code Documentation

The source code of ROSE is documented using the Doxygen documentation system.

General Guidelines

- English only
- Use valid Doxygen syntax (see "Examples" below)
- Make the code readable for a person who reads your code for the first time:
    - Document key concept, algorithm, functionalities
    - Cover your project, file, class/namespace, functions, and variables.
    - State your input and output clearly, specifically the meaning of the input or output
        - Users are more likely to use your code if they don't have to think about what the output means or what the input should be

TODO, not ready yet

- **Test** your documentation by generating it on your machine and then manually inspecting it to confirm its correctness

TODO: Generating Local Documentation

This does not work sometimes since we have a configuration file to indicate which directories to be scanned to generate the web reference html files

```
$ make doxygen_docs -C ${ROSE_BUILD}/docs/Rose/
```

Use //TODO

This is a recommended way to improve your code's comments.

While doing incremental development, it is often to have something you decide to do in the next iterations or you know your current implementation/functions have some limitations to be fixed in the future.

A good way is to immediately put a TODO source comments (// TODO blar blar ..) into the relevant code when you make such kind of decisions so you won't forget here is something you want to do next time.

The TODOs also serve as some handy flags within the code for other people if they want to improve your work after you are gone.

Examples

Single Line

Often a brief single line comment is enough

```
//! Brief description.
```

Multiple lines

Doxygen supports comments with more than one lines.

```
/**

   ... text..

 */

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


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

////////////////////////////////////
///  ... text <= 80 columns in length
////////////////////////////////////
```

Combined single line and multiple lines

Doxygen can generate a brief comment for a function and optionally show detailed comments if users click on the function.

Here are the options to support combined single-line and multiple-line source comments.

Option 1:

```
/**
 * \brief Brief description.
 *        Brief description continued.
 *
 * [Optional detailed description starts here.]
 */
```

Option 2:

```
/**
 \brief Brief description.
        Brief description continued.

 [Optional detailed description starts here.]
 */
```

Single line comment followed by multiple line comments':

You may extend an existing single line comment with a multiple line comments (Option 1 or 2). For example:

```
//! Brief description.
/**
 * Detailed description starts here.
 */
```

TODO: provide a full, combined example.

Functions

Rules

- Except for simple functions like getXX() and setXX(), all other functions should have at least one line comment to explain what it does
- Avoid global functions and global variables. Try to put them into a namespace.
- A function should not have more than 100 lines of code. Please refactor a big function into smaller, separated functions.
- Limit the unconditional printf() so your translator will print to screen hundreds lines of text output when processing multiple input files
    - use if condition to control printf() for debugging purposes such as if (SgProject::get_verbose() > 0)

Comments

Rules

- Please follow Doxygen style comments
- Please explain in sufficient details about how your function works: the algorithm steps.
    - Reviewers will check your algorithms in comments first then read your code to see if the code implements the algorithm correctly and efficiently.

Coding

Correctly implement the designed/documented algorithms

Code should be efficient in terms of both time and space (memory) complexity.

Please be aware that your translator may handle thousands of statements with even more AST nodes.

Classes

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

Name after what it is

Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.

- a class name should be a noun.

Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.

Explicit access

All sections (public, protected, private) should be identified explicitly. Not applicable sections should be left out.

Public members first

The parts of a class should be sorted public, protected and private.

The ordering is "most public first" so people who only wish to use the class can stop reading when they reach the protected/private sections.

Class variables

Class variables should NOT be declared public.

The concept of C++ information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C struct). In this case it is appropriate to make the class' instance variables public.

Avoid structs

Structs are kept in C++ for compatibility with C only, and avoiding them increases the readability of the code by reducing the number of constructs used. Use a class instead.

Statements

Loops

Only loop control statements must be included in the for() construction.

```
// Recommended way
sum = 0;
for (i = 0; i < 100; i++)
  sum += value[i]; sum += value[i];

// NOT allowed
 for (i = 0, sum = 0; i < 100; i++)
```

Increase maintainability and readability. Make a clear distinction of what controls and what is contained in the loop.

Loop variables should be initialized immediately before the loop.

Type conversions

Type conversions must always be done explicitly. Never rely on implicit type conversion.

```
  // recommended way
  floatValue = static_cast<float>(intValue);
  // NOT allowed
  floatValue = intValue;
```

By this, the programmer indicates that he is aware of the different types involved and that the mix is intentional.

Conditionals

The conditional should be put on a separate line.

```
 if (isDone)
 // NOT: if (isDone) doCleanup(); doCleanup();
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test is really true or not.

Complex conditional expressions must be avoided. Introduce temporary boolean variables instead

```
//recommended way
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry) { : }

// NOT: if ((elementNo < 0) || (elementNo > maxElement)|| elementNo ==
lastElement) { : }
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read, debug and maintain.

Statements to be avoided

The following statements should usually be avoided

- goto should not be used. Goto statements violate the idea of structured code. Only in some very few cases (for instance breaking out of deeply nested structures) should goto be considered, and only if the alternative structured counterpart is proven to be less readable.

- Executable statements in conditionals should be avoided. Conditionals with executable statements are just very difficult to read.

```
File* fileHandle = open(fileName, "w");
if (!fileHandle) { : }
// NOT: if (!(fileHandle = open(fileName, "w"))) { : }
```

AST translators

All ROSE-based translators should call AstTests::runAllTests(project) after all the transformation is done to make sure the translated AST is correct.

This has a higher standard than just correctly unparsed to compilable code. It is common for an AST to go through unparsing correctly but fail on the sanity check.

More information is at Sanity_check

Test cases

All contributions MUST have the accompanying test translator and input files to demonstrate the contributions work as expected.

- All tests MUST be triggered by the "make check" rule
- All test should have self-verification to make sure the correct results are generated
- All tests MUST be activated by at least one of the integration tests of Jenkins
    - This will ensure that no future commits can break your contributions.

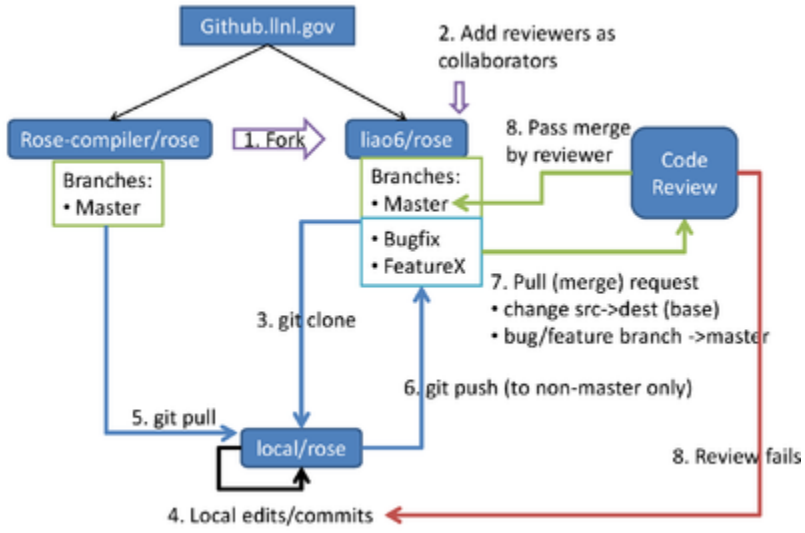
References

We list some external resources which are influential for us to define ROSE's coding standard

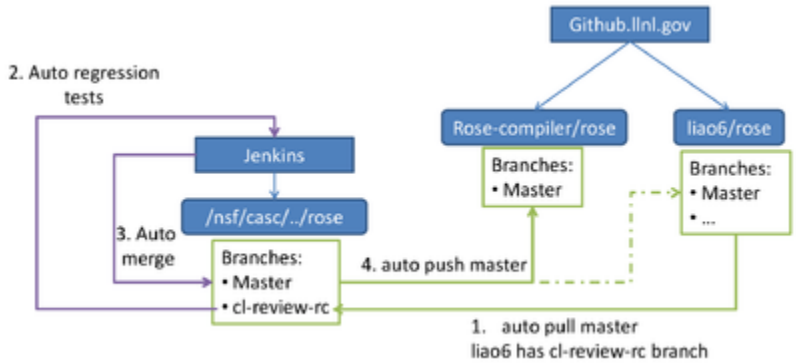
- http://www.possibility.com/Cpp/CppCodingStandard.html
- Sutter and Alexandrescu, C++ Coding Standards, 220 pgs, Addison-Wesley, 2005, ISBN 0-321-11358-6.
- http://www.parashift.com/c++-faq/coding-standards.html

- http://geosoft.no/development/cppstyle.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
- added 160MB MPI trace files into the git repository

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

rosebot

(*Under development*)

An automated pull request analyzer to perform various tasks:

- Automatically add reviewers to Pull Requests based on hierarchical configuration
- "Pre-receive hook" analyses: file sizes, quantity of files, proprietary source, etc.
- more...

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. Please avoid pushing thousands of lines at a time.
    - Please also avoid pushing any trivial (fixed a typo, commented out a single line etc.) to be reviewed.

One time setup

Steps for initializing code review: 1. **Login** to http://github.llnl.gov using your OUN and PAC.

2. **Fork** your own clone of the ROSE repository from http://github.llnl.gov/rose-compiler/rose.

- Go to http://github.llnl.gov/rose-compiler/rose
- Click the **Fork** button at the upper right corner of the webpage

3. **Add Collaborators**:

- Go to http://github.llnl.gov/<your_account>/rose
    - Click **Admin**
    - Click **Collaborators**
        - Add **candidate code reviewers**: liao6, too1. These developers will review and merge your work.
        - Add **admins**: hudson-rose. This user will automatically synchronize your master branch with /nfs/casc/overture/ROSE/git/ROSE.git:master.

4. **Create your public-private SSH key pair** using `ssh-keygen`, and add it to your github.llnl.gov account. (github.llnl.gov only supports the SSH protocol 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)
  - Double check the diff tab of your pull request **only shows the differences you made**, without other things brought in from the central repo. Or your own repo's master is out-of-sync with the central repo's master. Notify system admin (too1) for the problem or manually fix it using the troubleshooting section of this page.
- 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 within the pull request using @revieweraccount. **NOTE**: please only click "Comment on this issue"

once and manually refresh the web page. Github Enterprise has a bug so it cannot automatically shown the newly added comment. bug79
    - Or you can just email the reviewer
- waiting for reviewer's feedback:

Review results

- There might be three kinds of results
    - 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.
- What to do next
    - please look through the reviewer comments and try your best to address them
    - some of the comments should indicate some mandatory changes, please follow them
    - some of the comments may be just suggestions. Use your own judgement. The bottomline is the balance between quality and productivity.
    - Please **do not close** the pull request. You can push your new commits to the same branch again and comment on the pull request to indicate there are new updates. Please review them again. So the reviewer would not need to go to another pull request to see what were the previous comments before.

Reviewer Checklist

What to look out for as a code reviewer?

- Be familiar with the current Coding Standard as a general guideline to perform the code review.
- Allocate up to 1 hour at a time to review approximately 500-1000 lines of code: a longer time may not pay off due to the attention span limits of human brains

What to check

Five major things to check:

- **Documentation**: What are the commits about? Is this reflected in README, source comments, or LaTex files?
- **Style**: Does the coding style follow our standard? Is the code clean, robust, and maintainable?

- **Algorithm**: Does the code have sufficient comments about what algorithm is used? Is the algorithm correct and efficient (space and time complexity)?
- **Implementation**: Does the code correctly implement the documented algorithm(s)?
- **Testing**: Does the code have the accompanying test translator and input test codes to ensure the contributions do what they are supposed to do?
    - Is Jenkins being configured to trigger these tests (your work may require new pre-requisite software or configure options)? Local tests on developer's workstation do not count.

More details:

- **Naming conventions**: File and directory names follow our standards; clear and intuitive
    - **Directory structure**: source code, test code, and documentation files are added into the correct locations
- **Maintainability**: clarity of code; can somebody who did not write the code easily understand what the code does?
    - **No looong functions**: a function with hundreds of lines of code is a no-no
    - **Architecture/design**: the reasons and motivations for writing the code, and its design.
- **No duplication**: similar code may already exist or can be extended
- **Re-use**: can part of the code be refactored to be reusable by others?
- **Unit tests**: make check rules are associated with each new feature to ensure the new feature will be tested and verified for expected behaviors
- **Sanity**: no turning off, or relaxing, other tests to make the developer's commits pass Jenkins. In other words, **no cheating**.

Commenting

Reviewer comments should be clearly delimited into these three well-defined sections:

1. **Mandatory**: the details of the comment must be implemented in a new commit and added to the Pull Request before the code review can be completed.

2. **Recommended**: the details of the comment could represent a best-practice or, simply, it could be intended to provide some insight to the developer that they may have not thought about.

Both `Mandatory` and `Recommended` can be accompanied by the keyword `Nitpick`:

3. **Nitpick**: the details of the comment represent a fix that usually involves a spelling/grammatical or coding style correction. The main purpose of the **nitpick** indication is to let the developer know that you're not trying to be on their case and make their life difficult, but an error is an error, or there's a better way to do something.

Decisions

Make a clear and definitive decision for the code review:

- **Pass**: The code does what it is supposed to do with clear documentation and test cases. **Merge and close** the pull request.
- **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 set of commits and pushed later on.
- **Fail**. Additional work is needed, such as better naming, better places to put files, more source comments, add regression tests, etc. Notify the developers of the issues and ask for a new set of commits to be pushed addressing the corrections or improvements.

Who should review what

Ideally, every ROSE contributor should participate in code review as a reviewer at some point so the benefits of peer-review can fully be fulfilled.

However, due to the limited access to our internal github enterprise server, we currently have a centralized review process in which ROSE staff members (liao6, too1) serve as the default code reviewers. They are responsible for either reviewing the code themselves or delegate to other developers who either has better knowledge about the contributions or should be aware of the contributions.

We am actively looking at better options and will gradually expand the pool of reviewers so the reviewing step won't become a bottleneck.

TODO: use `rosebot` to automatically assign reviewers according to a hierarchical configuration of the source-tree.

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.

- 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.

Troubleshooting

master is out-of-sync

The master branch of each developer's git repository (http://github.llnl.gov) should be automatically synchronized with the central git repository's master branch (`/nfs/casc/overture/ROSE/git/ROSE.git`). In rare cases, it could be out-of-sync. Here is an example to perform a manual synchronization:

1. Clone your Github repository:

```
$ cd ~/Development/projects/rose
$ git clone git@github.com:<user_oun>/rose.git
Cloning into ROSE...
remote: Counting objects: 216579, done.
remote: Compressing objects: 100% (55675/55675), done.
remote: Total 216579 (delta 159850), reused 211131 (delta 155786)
Receiving objects: 100% (216579/216579), 296.41 MiB | 35.65 MiB/s,
done.
Resolving deltas: 100% (159850/159850), done.
```

2. Add the central repository as a remote repository:

```
$ git remote add central /nfs/casc/overture/ROSE/git/ROSE.git
$ git fetch central
From /nfs/casc/overture/ROSE/git/ROSE.git
 * [new branch]      master     -> central/master
 ...
```

3. Push the central master branch to your Github's master branch:

```
-bash-3.2$ git push central central/master:refs/heads/master
Total 0 (delta 0), reused 0 (delta 0)
To git@github.llnl.gov:<user_oun>/rose.git
   16101fd..563b510  central/master -> master
```

master cannot be synchronized

In rare cases, your repository's master branch cannot be automatically synchronized. This is most likely due to merge conflicts. You will receive an error message through an automated email, resembling the following (last updated on 7/24/2012):

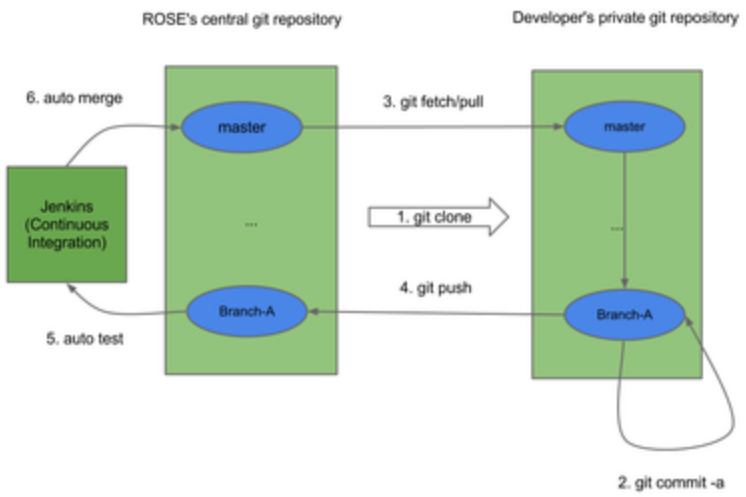
```
To git@github.llnl.gov:lin32/rose.git
! [rejected]        origin/master -> master (non-fast forward)
error: failed to push some refs to 'git@github.llnl.gov:lin32/rose.git'

---

Your master branch at [github.llnl.gov:lin32/rose.git] cannot be
automatically updated with
[/nfs/casc/overture/ROSE/git/ROSE.git:master]

Please manually force the update:

Add the central repository as a remote, call it "nfs":

  $ git remote add nfs /nfs/casc/overture/ROSE/git/ROSE.git

1. First, try to manually perform a merge in your local repository:

  # 1. Checkout and update your Github's master branch
  $ git checkout master
  $ git pull origin master

  # 2. Merge the central master into your local master
  $ git pull nfs master
  <no merge conflicts>

  # 3. Synchronize your local master to your Github's master
  $ git push origin HEAD:refs/head/master

2. Otherwise, try to resolve the conflict.

3. Finally, if all else fails, force the synchronization:

  $ git push --force origin nfs/master:refs/heads/master

  WARNING: your master branch on Github will be overriden so make sure
  you have sufficient backups, and take precaution.
```

Please simply follow the email's instructions to force the update of your Github's master branch.

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/

Continuous Integration



ROSE Continuous integration using Git and Jenkins (Code Review Omitted for simpler explanation)

Motivation

Without automated continuous integration, we had frequent incidents like:

- Developer A commits something to our central git repository's master branch. The commits contain some bugs which break our build and take a long time to have a fix. Then the central master branch is left to a corrupted state for weeks so nobody can check out/in anything.
- Developer A does a lot of wonderful work offline for months. But his work later is found to be incompatible with another developer's work. His work has unsolvable merge conflicts.

Overview

The ROSE project uses a workflow that automates the central principles of continuous integration in order to make integrating the work from different developers a non-event. Because the integration process only integrates with ROSE the changes that passes all tests we encourage all developers to stay in sync with the latest version.

A high level overview of the development model used by ROSE developers.

- Step 1: Taking advantage of the distributed source code repositories based on git, each developer should first clone his/her own repository from our central git repository (or its mirrors/clones/forks).

- Step 2: Then a feature or a bugfix can be developed in isolation within the private repository. He can create any number of private branches. Each branch should relate to a feature that this developer is working on and be relatively short-lived. The developer can commit changes to the private repository without maintaining an active connection to the shared repository.
- Step 3: When work is finished and locally tested, he can pull the latest commits from the central repo's master branch
- Step 4: He then can push all accumulated commits within the private repository to his branch within the shared repository. We create a dedicated branch within the central repository for each developer and establish access control of the branch so only an authorized developer can push commits to a particular branch of the shared repository.
- Step 5-6 (automated): Any commits from a developer's private repository will not be immediately merged to the master branch of the shared repository.

In fact, we have access control to prevent any developer from pushing commits to the master branch within the shared repository. A continuous integration server called Jenkins is actively monitoring each developer's branch within the central repository and will initiate comprehensive commit tests upon the branch once new commits are detected. Finally, Jenkins will merge the new commits to the master branch of the central repository if all tests pass. If a single test fails, Jenkins will report the error and the responsible developer should address the error in his private repository and push improved commits again.

As a result, the master branch of the central git repository is mostly stable and can be a good candidate for our external release. On top of the master branch of the central git repository, we further have more comprehensive release tests in Jenkins. If all the release tests pass, an external release based on the master branch will be made available outside.

Tests on Jenkins

We use Jenkins (http://hudson-rose-30:8080/) to test commits added to developer's release candidate branches at the central git repository.

The tests are organized into three categories

- **Integration**: tests used to check if the new commits can pass various "make check" rules, compatibility tests, portability tests, configuration tests, and so on. If all tests pass, the commits will be merged (or **integrated**) into the master branch of the central repository.
- **Release**: tests used to test the updated master branch of the central repository for additional set of tests using external benchmarks. If all tests pass, the head of the master will be **released** as a stable snapshot for public file package releases(generated by "make dist").
- Others: for informational purpose now, not being used in our production workflow.

So for each push (one or more commits to a -rc branch), it will go through two stages: Integration test and Release test stage.

It is each developer's responsibility to make sure their commits can **pass BOTH stage** by **fixing any bugs** discovered by the tests.

Check Testing Results

It is possible to manually tracking down how you commits are doing within the test pipeline within Jenkins (http://hudson-rose-30:8080/). But it can be tedious and overwhelming.

So we provide a dashboard (http://sealavender:4000/) to summarize the commits to your release candidate branch(-rc) and the pass/fail status for each integration tests

Frequently Failed Jobs

C6-ROSE-distcheck

http://hudson-rose-30:8080/job/C6-ROSE-distcheck/

Problem:

```
make[3]: *** No rule to make target `README', needed by `distdir'.
Stop.
...
make: *** [distdir] Error 1

*************************************************************************
*********
*** FAILED make distcheck step
*************************************************************************
*********
```

The problem is that some files are not automatically added into the software distribution during "`make dist`". For example, input test code files, `README` files, and configuration files.

Prognosis:

Overview of "`make distcheck`":

1. "`make dist`": ensure that you can successfully create a tarball distribution of the software package.

2. The tarball distribution is un-tarred (unpackaged) and the tests are then carried out on this distribution source tree.

3. "`$ROSE/configure`": configure the software package.

4. "`make all`": build the software package.

5. "`make check`": execute the software package's regression tests.

Files are not always automatically added into the distribution tarball (during "`make dist`"). However, most source and header files will be automatically added if they are used to build libraries and executables (specified within Automake makefiles, `Makefile.am`).

Solution:

- Add the culprit files into the `EXTRA_DIST` Automake variable in the appropriate `Makefile.am` file.
- Test "`make distcheck`" on your local machine before you push next time!

Reference:

- Basics of distribution: http://www.delorie.com/gnu/docs/automake/automake_91.html

C2-ROSE-language-matrix-linux

ROSE has configuration options to turn on desired language support. So this job is used to test -enable-only-LANGUAGE option for fortran, c-and-cxx, java, php, and binary-analysis.

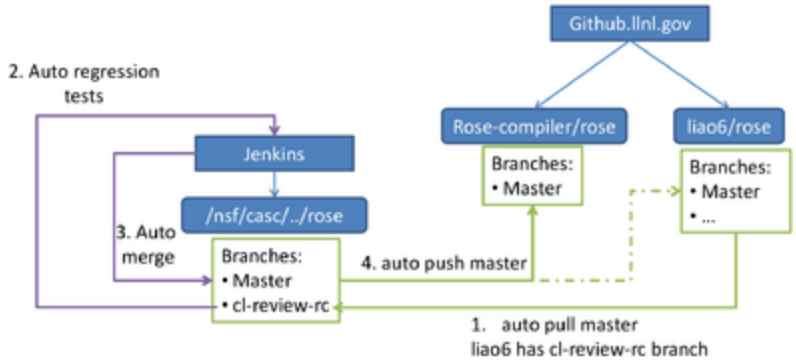
Many new developers are not aware of this so their pushes often fail on this test.

To pass this test, make sure you have conditionals in your Makefile.am to turn on language specific rules.

For example, for your Fortran-specific work, using the following conditional

```
if ROSE_BUILD_FORTRAN_LANGUAGE_SUPPORT
 Fortran_to_C:
  $(MAKE) -C ../src Fortran_to_C
else
  ..
endif
```

Connection to Code Review



Connection between Github Enterprise and Jenkins

In reality, most LLNL developers are now asked to push things to Github Enterprise for code review first instead of directly pushing to our central git repository.

- Auto pull: we have another Jenkins at (https://hudson-rose-30:8443/jenkins/) which serves as the bridge between Github Enterprise and Jenkins.
  - For each private repositories on Github Enterprise, we have a Jenkins job to monitor the master branch for approved pull (merge) request. If there is any new approved commits, the job will transfer the commits to the central repository's -reviewed-rc branch for that developer.
- Auto push: A Jenkins job is responsible for propagating latest central master contents to all private repositories on github.llnl.gov
  - http://hudson-rose-30:8080/job/Commit-sync-github

TODO

High priority

- enable email notification for the final results of each test:
- incrementally add more compilation tests using external benchmarks to be integration tests.

References

- Files used to generate the figure: feel free to add new versions as new slides: link

Frequently Asked Questions (FAQ)

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

General

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/"

How many lines of source code does ROSE have?

Excluding the EDG submodule and all source code comments, the core of ROSE (rose/src) has about 674,000 lines of C/C++ source code as of July 11, 2012.

Including tests, projects, and tutorial directories, ROSE has about 2 Million lines of code.

Some details are shown below:

```
[rose/src]./cloc-1.56.pl .
    3076 text files.
    2871 unique files.
     716 files ignored.

http://cloc.sourceforge.net v 1.56  T=26.0 s (91.7 files/s, 39573.3
lines/s)
-------------------------------------------------------------------------
--------
Language                     files          blank        comment
code
-------------------------------------------------------------------------
--------
C++                            908          75280          93960
354636
C                              123          12010           3717
199087
C/C++ Header                   915          28302          38412
121373
Bourne Shell                    17           3346           4347
25326
Perl                             4            743           1078
7888
Java                            18           1999           4517
7096
m4                               1            747             20
6489
Python                          34           1984           1174
5363
make                           148           1682           1071
3666
C#                              11            899            274
2546
```

```
SQL                              1              0              0
1817
Pascal                           5            650             31
1779
CMake                          168           1748           4880
1702
yacc                             3            352            186
1544
Visual Basic                     6            228            421
1180
Ruby                            11            281            181
809
Teamcenter def                   3              3              0
606
lex                              2            103             47
331
CSS                              1             95             32
314
Fortran 90                       1             34              6
244
Tcl/Tk                           2             29              6
212
HTML                             1              8              0
15
-------------------------------------------------------------------------
--------
SUM:                          2383         130523         154360
744023
-------------------------------------------------------------------------
--------
```

How large is ROSE?

To show top level information only (in MB): du -msl * | sort -nr

```
170     tests
109     projects
90      src
19      docs
16      winspecific
16      ROSE_ResearchPapers
15      binaries
7       scripts
5       LicenseInformation
4       tutorial
4       autom4te.cache
2       libltdl
2       exampleTranslators
2       configure
2       config
2       ChangeLog
```

Sort directories by their sizes in MegaBytes

```
 du -m | sort -nr >~/size.txt
709     .
250     ./.git
245     ./.git/objects
243     ./.git/objects/pack
170     ./tests
109     ./projects
90      ./src
76      ./tests/CompileTests
50      ./tests/RunTests
40      ./tests/RunTests/FortranTests
34      ./tests/RunTests/FortranTests/LANL_POP
29      ./tests/RunTests/FortranTests/LANL_POP/netcdf-4.1.1
27      ./src/3rdPartyLibraries
23      ./tests/roseTests
23      ./src/frontend
22      ./tests/CompileTests/Fortran_tests
21      ./tests/CompilerOptionsTests
19      ./docs
18      ./tests/CompileTests/RoseExample_tests
18      ./src/midend
18      ./docs/Rose
16      ./winspecific
16      ./ROSE_ResearchPapers
15      ./tests/CompileTests/Fortran_tests/gfortranTestSuite
15      ./binaries/samples
15      ./binaries
14      ./tests/CompileTests/Fortran_tests/gfortranTestSuite/gfortran.d
g
14      ./src/roseExtensions
11      ./projects/traceAnalysis
10      ./tests/CompileTests/A++Code
10      ./tests/CompilerOptionsTests/testCpreprocessorOption
10      ./tests/CompilerOptionsTests/A++Code
10      ./src/roseExtensions/qtWidgets
10      ./src/frontend/Disassemblers
10      ./projects/symbolicAnalysisFramework
10      ./projects/SATIrE
10      ./projects/compass
9       ./winspecific/MSVS_ROSE
9       ./tests/RunTests/A++Tests
9       ./tests/roseTests/binaryTests
9       ./src/frontend/SageIII
9       ./projects/symbolicAnalysisFramework/src
9       ./docs/Rose/powerpoints
8       ./winspecific/MSVS_project_ROSETTA_empty
8       ./projects/simulator
7       ./tests/RunTests/FortranTests/LANL_POP_OLD
7       ./tests/CompileTests/Cxx_tests
7       ./src/midend/programTransformation
7       ./src/midend/programAnalysis
7       ./src/3rdPartyLibraries/libharu-2.1.0
7       ./scripts
7       ./projects/symbolicAnalysisFramework/src/mpiAnal
7       ./projects/RTC
6       ./winspecific/MSVS_ROSE/Debug
6       ./tests/RunTests/FortranTests/LANL_POP/netcdf-4.1.1/ncdap_test
```

```
6       ./tests/roseTests/programAnalysisTests
6       ./src/3rdPartyLibraries/ckpt
6       ./src/3rdPartyLibraries/antlr-jars
6       ./projects/SATIrE/src
5       ./tests/RunTests/FortranTests/LANL_POP/pop-distro
5       ./tests/RunTests/FortranTests/LANL_POP/netcdf-4.1.1/libcf
5       ./tests/CompileTests/ElsaTestCases
5       ./src/ROSETTA
5       ./src/3rdPartyLibraries/qrose
5       ./projects/DatalogAnalysis
5       ./projects/backstroke
5       ./LicenseInformation
5       ./docs/Rose/AstProcessing
```

To list files based on size

```
 find . -type f -print0 | xargs -0 ls -s | sort -k1,1rn
241568 ./.git/objects/pack/pack-
f366503d291fc33cb201781e641d688390e7f309.pack
13484 ./tests/CompileTests/RoseExample_tests/Cxx_Grammar.h
10240 ./projects/traceAnalysis/vmp-hw-part.trace
6324 ./tests/RunTests/FortranTests/LANL_POP_OLD/poptest.tgz
5828 ./winspecific/MSVS_ROSE/Debug/MSVS_ROSETTA.pdb
4732 ./.git/objects/pack/pack-
f366503d291fc33cb201781e641d688390e7f309.idx
4488 ./binaries/samples/bgl-helloworld-mpicc
4488 ./binaries/samples/bgl-helloworld-mpixlc
4080 ./LicenseInformation/edison_group.pdf
3968 ./projects/RTC/tags
3952 ./src/frontend/Disassemblers/x86-InstructionSetReference-NZ.pdf
3908 ./tests/CompileTests/RoseExample_tests/trial_Cxx_Grammar.C
3572 ./winspecific/MSVS_project_ROSETTA_empty/MSVS_project_ROSETTA_empt
y.ncb
3424 ./src/frontend/Disassemblers/x86-InstructionSetReference-AM.pdf
2868 ./.git/index
2864 ./projects/compassDistribution/COMPASS_SUBMIT.tar.gz
2864 ./projects/COMPASS_SUBMIT.tar.gz
2740 ./ROSE_ResearchPapers/2007-
CommunicatingSoftwareArchitectureUsingAUnifiedSingle-ViewVisualization-
ICECC
S.pdf
2592 ./docs/Rose/powerpoints/rose_compiler_users.pptx
2428 ./src/3rdPartyLibraries/ckpt/wrapckpt.c
2408 ./projects/DatalogAnalysis/jars/weka.jar
2220 ./scripts/graph.tar
1900 ./src/3rdPartyLibraries/antlr-jars/antlr-3.3-complete.jar
1884 ./src/3rdPartyLibraries/antlr-jars/antlr-3.2.jar
1848 ./src/midend/programTransformation/ompLowering/run_me_defs.inc
1772 ./src/3rdPartyLibraries/qrose/docs/QROSE.pdf
1732 ./tests/CompileTests/Cxx_tests/longFile.C
1724 ./src/midend/programTransformation/ompLowering/run_me_task_defs.in
c
1656 ./ChangeLog
1548 ./tests/roseTests/binaryTests/yicesSemanticsExe.ans
1548 ./tests/roseTests/binaryTests/yicesSemanticsLib.ans
```

```
1480 ./ROSE_ResearchPapers/1997-ExpressionTemplatePerformanceIssues-
IPPS.pdf
1408 ./docs/Rose/powerpoints/ExaCT_AllHands_March2012_ROSE.pptx

...
```

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 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.

How to add new AST nodes?

There is a section named "1.7 Adding New SAGE III IR Nodes (Developers Only)" in ROSE Developer's Guide (http://www.rosecompiler.org/ROSE_DeveloperInstructions.pdf)

But before you decide adding new nodes, you may consider if AstAttribute (user defined objects attached to AST) would be sufficient for your problem.

For example, the 1st version of the OpenMP implementation in ROSE started by using AstAttribute to represent information parsed from pragmas. Only in the 2nd version we introduced dedicated AST nodes.

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).

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)
```

How to translate multiple files scattered in different directories of a project?

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.

```
On 07/25/2012 11:20 AM, Fernando Rannou wrote:
> > Hello
> >
> > We are trying to use ROSE to refactor  a big project consisting of
> > several  *.cc and *.hh files, located at various directories. Each
> > class is defined in a *.hh file and implemented in a *.cc file.
> > Classes include (#include) other class definitions. But we have
only
> > found single file examples.
> >
> > Is this possible? If so, how?
> >
> >
> > Thanks
```

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.
    - Again, 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.

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.

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.

Code Review

See Code Review in ROSE.

How to create a translator

Translator basically converts one AST to another version of AST. The translation process may add, delete, or modify the information stored in AST.

Overview

A ROSE-based translator usually has the following steps

1. Search for the AST nodes you want to translate.
2. Perform the translation action on the found AST nodes. This action can be one of two major variants

- Updating the existing AST nodes
- Creating new AST nodes to replace the original ones. This is usually cleaner approach than patching up existing AST and is better supported by SageBuilder and SageInterface functions.
- Optionally update other related information for the translation.

First Step

Get familiar with the ASTs before and after your translation. So you know for sure what your code will deal with and what AST you code will generate.

The best way is to prepare simplest sample codes and carefully examine the whole dot graphs of them.

Design considerations

It is usually a good idea to

- separate the searching step from the translation step so one search (traversal) can be reused by all sorts of translations.
- When design the order of searching and translation, be careful about if the translation will negatively impact on the searching
  - pre-order traversal may cause AST nodes to be visited are modified, similar to the effect of iterator invalidation.
  - please use post-order, or reverse order of pre-order for your traversal hooked up with translation

Searching for the AST node

There are multiple ways to find things you want to translate in AST.

- Via AST Query: Node query returns a list of AST nodes in the same type. This is often enough to simple translations

```
Rose_STL_Container<SgNode*> ProgramHeaderStatementList =
NodeQuery::querySubTree (project,V_SgProgramHeaderStatement);
for (Rose_STL_Container<SgNode*>::iterator i =
ProgramHeaderStatementList.begin(); i !=
ProgramHeaderStatementList.end(); i++)
{
    SgProgramHeaderStatement* ProgramHeaderStatement =
isSgProgramHeaderStatement(*i);
    ...
}
```

- Through AST traversal: walks through whole AST using different orders (pre-order or post order). More sophisticated translations may need this. Also post-

order traversal is recommended to avoid modifying things the traversal will hit later on (similar problem as iterator invalidation in C++)

- The AST traversal gives visit() functions to hook up your translation functions. A switch statement is can be used for handling different types of AST node.

```
class f2cTraversal : public AstSimpleProcessing
{
  public:
    virtual void visit(SgNode* n);
};

void f2cTraversal::visit(SgNode* n)
{
  switch(n->variantT())
  {
    case V_SgSourceFile:
      {
        SgFile* fileNode = isSgFile(n);
        translateFileName(fileNode);
      }
      break;
    case V_SgProgramHeaderStatement:
      {
        ...
      }
      break;
    default:
      break;
  }
}
```

Performing Translation

The translations you want to do often depend on the types of the AST nodes you visit. For example you can have a set of translation functions defined in your namespace

- void translateForLoop(SgForLoop* n)
- void translateFileName(SgFile* n)
- void translateReturnStatement(SgReturnStmt* n), and so on

Other tips

- Reference ROSE doxygen website for information of each AST node.
- Use SageBuilder if you want to create new AST node. Update SageBuilder you cannot find the one you need.
- Look up in SageInterface for the translation functions you need. If there is none, then write your own function.
- Update the information, or create the new AST node you need.
- Replace the existing AST node with your updated or new AST node.

Updating Tree

- You might need to handle some details, like removing symbol, updating parent, and symbol table.
- Be careful to use deepDelete() and deepCopy(). Some information might not be updated properly. For example, deepDelete might not update your symbol table.

Verify the correctness

You can use wholeAST graph to verify your translation.

All ROSE-based translators should call AstTests::runAllTests(project) after all the transformation is done to make sure the translated AST is correct.

This has a higher standard than just correctly unparsed to compilable code. It is common for an AST to go through unparsing correctly but fail on the sanity check.

More information is at Sanity_check

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, the --prefix path
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) -o $(TRANSLATOR) -
I$(BOOST_INSTALL)/include -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
```

A complete example

The sample Makefile prepared within ROSE virtual machine image.

```
demo@ubuntu:~/myTranslator$ cat makefile
## 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=/home/demo/opt/rose-inst

## ROSE uses the BOOST C++ libraries
BOOST_INSTALL=/home/demo/opt/boost-1.40.0

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

## Input testcode for your translator
TESTCODE=hello.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 -
I$(ROSE_INSTALL)/include -L$(ROSE_INSTALL)/lib -lrose -o $(TRANSLATOR)

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

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


demo@ubuntu:~/myTranslator$ make check
```

```
g++ -g myTranslator.cpp -I/home/demo/opt/boost-1.40.0/include -
I/home/demo/opt/rose-inst/include -L/home/demo/opt/rose-inst/lib -lrose
-o myTranslator
./myTranslator -c -I. -I/home/demo/opt/rose-inst/include hello.cpp
```

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 build 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 a proper 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 collect things we try to avoid:

Formatting/Indenting 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 not 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.

Create Exacting Tests Early and Often

Lesson:

- A developer created tests that were too broad, mostly because they were included late in development. This led to passes that should not have passed, that is passing all tests even though the code had been broken.

Solution:

- Make sure that tests check results carefully. This is made much easier by making sure your functions have precisely ONE intention. E.g. if you need to transform data and operate on the transformed data, split the transformation and the operation into two functions (at least).

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 (customized 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"
```

Jenkins

using external benchmarks

The way we set it up is to

- In the benchmark, we change the benchmark's build system to call the ROSE tool (identityTranslator or your RTED tool) installed.
- In the Jenkins test job,
  - Build and install the tested ROSE, prepare environment variables.
  - Go to the benchmark with modified build system. Build and run the benchmark.

Basically, the test job should simulate how a ROSE tool would be used by end-users, not by tweaking ROSE for each different benchmarks.

Lattices

Introduction

Lattices are mathematical structures. They can be used as a general way to express an order among objects. This data can be exploited in data flow analysis.

Lattices can describe transformations effected by basic blocks on data flow values also known as flow functions.

Lattices can describe data flow frameworks when instantiated as algebraic structures consisting of a set of data flow values, a set of flow functions, and a merge operator.

Poset

Partial ordering: $\leq$

A **partial ordering** is a binary relation $\leq$over a set P which is reflexive, antisymmetric and transitive, i.e.

- Reflexive x<=x
- Anti-Symmetric, if $x \leq y, y \leq x$then x=y
- Transitive: if $x \leq y, y \leq z$then $x \leq z$

Partial orders should not be confused with total orders. A total order is a partial order but not vice versa. In a total order any two elements in the set P can be compared. This is not required in a partial order. Two elements that can be compared are said to be **comparable**

A **partially ordered set**, also known as a **poset**, is a set with a partial order.

Given a poset there may exist an infimum or a supremum. However, not all posets contain these.

Given a poset P with set X and order $\leq$:

An **infimum** of a subset S of X is an element a of X such that

- $a \leq x$for all x in S and
- for all y in X, if for all x in S, $y \leq x$then $y \leq a$

The dual of this notion is the **supremum** which has the definition of infimum if you switch $\leq$with $\geq$

If we simply pick an element of X that satisfies the first condition we have a **lower bound**. The second condition ensures that we have (if it exists) the **unique** greatest lower bound. Similarly for suprema.

A lattice is a particular kind of poset. In particular, a lattice L is a poset P(X, $\leq$where For any two elements of the lattice a and b, the set {a, b} has a **join** and a **meet**

The join and meet operations MUST satisfy the following conditions

- 1) The join and meet must commute
- 2) The join and meet are associative
- 3) The join and meet are idempotent, that is, x join itself or x meet itself are both x.

If the lattice contains a meet it is a meet-semilattice, if a lattice contains a join it is a join-semilattice, similarly there exists a meet-semilattice

(Definitions obtained from wikipedia with minimal modification)

Lattice Definition

Definition of a Lattice (L, $\wedge$, $\vee$)

- L is a poset under $\leq$ such that
  - Every pair of elements has a unique greatest lower bound (meet) and least upper bound (join)
  - Not every poset is a lattice: greatest lower bounds and least upper bounds need not exist in a poset.

Infinite vs. Finite lattices

- Infinite: An infinite lattice does not contain an 0 (bottom) or 1 (top) element, even though every pair of elements contains a greatest lower bound and a least upper bound on the entire underlying set. By the definition of unbounded or infinite sets we know that given X an unbounded set given any x in X we can find an x' that is greater than x (under some ordering, in this case the lattice). Similarly for greatest lower bounds.
- a finite/bounded lattice: the underlying set itself has a greatest lower bound and a least upper bound, For now we will call the greatest lower bound 0 and the least upper bound 1.
  - if a $\leq$ x, for all x in L, then a is the 0 element of L, $\bot$, recall that this is a unique element
  - if a $\geq$ x for all x from L, then a is the 1 element of L, $\top$

Meet $\wedge$ is a binary operation such that a $\wedge$ b take the greatest lower bound of the set (this is guaranteed by the definition lattice.

Similarly Join $\vee$ returns the least upper bound of the set, guaranteed to exist by the definition of a lattice.

To recap, a lattice L is a triple {X, $\wedge$, $\vee$} composed of a set, a Meet function, and a Join function

Properties of Meet and $\wedge$.

- We refer to the $\vee$ as $\vee$ and the $\wedge$ as J
- Closure: If x and y belong to L, then there exists a unique z and a unique w from L such that x $\vee$ y = z, and x $\wedge$ y = w
- Commutativity: for all x, y in L, x $\vee$ y = y meet x, x $\wedge$ y = y $\wedge$ x:
- Associativity: (x $\vee$ y) $\vee$ z = x $\vee$ (y $\vee$ z), similarly in the $\wedge$ operation
- There are two unique elements of L called bottom (_|_), and top (T) , such that for all x, x $\vee$ _|_ = _|_ and x $\wedge$ T = T

- Many lattices, with some exceptions, notably the lattice corresponding to constant propagatioin, are also distributive: x ⋁y ⋀z = (x ⋀z) ⋁(y ⋀z)

Lattices and partial order:

$x \sqsubseteq y$ if and only if $x \sqcap y = x$

A **strictly ascending chain** is a sequence of elements of a set X such that, for x_i in X, $x_1, x_2, ..., x_n$ has the property $\bot = x_1 \sqsubset x_2 \sqsubset ... \sqsubset x_n = \top$. The greatest is the chain with final index n such that n is the greatest such final index among all strictly ascending chains.

The **height** of a lattice is defined as the length of the longest strictly ascending chain it contains.

If a data-flow analysis lattice has a finite height and a monotonic flow function then we know that the associated data flow analysis algorithm will terminate.

- Example: If the greatest strictly ascending chain of a lattice L is finite and it takes finitely many steps to reach the top, we can infer that the associated data flow algorithm terminates.

(wikipedia used for definitions)

Example: Bit vector Lattices

- The elements of the set are bit vectors
- The bottom is the 0 vector
- The top is a 1 vector
- Meet is a bitwise And
- Join is a bitwise Or

BV^n denotes the lattice of bit vectors of length n.

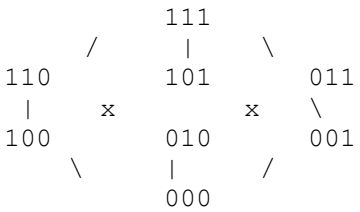
Constructing complex lattices from multiple less complex lattices

- Example: The product operation which combines (concatenates) lattices elementwise
    - The product of two lattices L1 and L2 with meet operators M1, M2, respectively: L1 x L2
    - The elements in the lattice: {<x1, x2> | x1 from L1, x2 from L2}
- The meet operator: <x1, x2> M <y1, y2> = <x1 M y1, x2 M y2>
- The join operation: <x1, x2> J <y1, y2> = <x1 J y1, x2 J y2>

- Example:

- BV^n is the product of n copies of the trivial bit vector attice BV^1 with bottom 0 and top 1

Graphical Representation BV^3



Here meet and join operators induce a partial order on the lattice elements

x is less than or equal to (<=) y if an only if x M y = x

For the BV^3: 000<= 010 <= 101<=111

The partial order on the lattice is:

- Transitive x <= y and y <= z, then x <=z
- Antisymmetric: if x<=y and y<=x, then x = y
- Reflexive: for all x: x<=x:

The height of the lattice is the length of its longest strictly ascending chain:

- The maximal n such that there exists a strictly ascending chain x1, x2, ..., xn such that
- Bottom = x1 < x2 < xn = Top

For BV^3 lattice, height = 4

monotone function

In mathematics, a monotonic function (or monotone function) is a function that preserves the given order.

a function mapping a lattice to itself: f: L -> L, is monotone if for all x, y from L

- x<=y ==> f(x)<=f(y)

monotone -> order preserving

example monotone function: f: BV^3 -> BV^3

- f (<x1 x2 x3>) = <x1 1 x3>

tuples of lattices

simple analyses may require very complex lattices

e.g. reaching constants: V 2^(v*c) where v is number of variables and c is the constants

solution: use a tuple of lattices, one per variable.

V = coonstant U { Top, Bottom}

integer value: ICP

e.g. used for constant propagation

elements: Top, Bottom, all the integers, the Booleans

- n M Bottom = Bottom
- n J Top = Top
- n J n = n M n = n
- integers and Booleans m,n, if m != n, then m M n = Bottom, m J n = Top
    - as a result the lattice has three levels: top element, all other elements, bottom element
    - from higher level to lower level: join operation
    - from lower level to higher level: meet operation

Relevance to data flow analysis

A lattice provides a set of flow values to a particular data flow analysis.

Lattices are used to argue the existence of a solution obtainable through fixed-point iteration

- At each program point a lattice represents an IN[p] or OUT[p] set (flow value)
- meet: merge flow values, e.g. set union, deal with control flow branches merge
- Top usually represents the best information (initial flow value). Note people can use top to represent worst-base information also!!
- The bottom value represents the worst-base information
- if BOTTOM <= x <= y <= TOP , then x is a conservative approximation of y. e.g. x is a superset

e.g. liveness analysis

bitvector for all variables x_1, x_2, ..., x_n

First step: design the lattice values

- top value: empty set {}, initial value, knowing nothing
- bottom value: all set {x_1, x_2, ..., x_n}: max possible value, knowing every variable is live

n = 3, 3 variable case: a flow value==> a set of live variable at a point

S = {v1, v2, v3}

value set: 2^3 = { empty, {v1},{v2}, {v3}, {v1, v2}, {v1,v3}, {v2, v3}, {v1, v2, ve} }

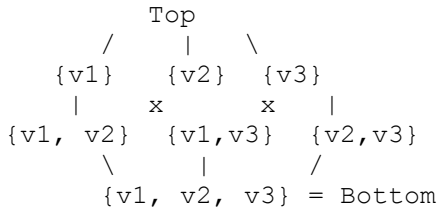
Design lattice

- top value, best case: none live { T } // top
- bottom value, worst ase: all live {v1, v2, v3}

Design meet: set Union (Or operation): bring the value down to the bottom, context insensitive

- design partial order <= --> $\supseteq$

In between, a partial order: inferior/conservative solutions are lower on the lattice



Flow function F: $f_n(X) = Gen_n \cup (X - Kill_n), \forall_n\}$

reaching definition

Value: 2^n n = number of all definitions

top: empty set: knowing nothing, initial value

bottom: all set: all definitions are reaching definition

Meet operation: set union: bring down the levels of values, from unknowing to knowing

C++ Programming

ROSE is written in C++. Some users have suggested to mention the major C++ programming techniques used in ROSE so they can have more focused learning experiences as C++ beginners.

Design Patterns: ROSE uses some common design patterns

- visitor pattern: used to create the AST traversal.

Good API Design

Google: "How to Design a Good API and Why it Matters" by Joshua Bloch

TODO: convert from Markdown

Characteristics of a Good API

- Easy to learn
- Easy to use, even without documentation
- Hard to misuse
- Easy to read and maintain code that uses it
- Sufficiently powerful to satisfy requirements
- Easy to extend
- Appropriate to audience

The Process of API Design

- Gather true requirements in the form of **use-cases**
- Start with a short 1-page specification
  - Agility trumps completeness
  - Collect a lot of feedback
- Use your API early and often
  - [Test-Driven Development (TDD)](http://en.wikipedia.org/wiki/Test-driven_development)

    [T]he repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.

    -

    - Doubles as examples/tutorials and unit tests
- Maintain realistic expectations
    - You won't be able to please everyone... aim to displease everyone equally
    - Expect to evolve API; mistakes happen; real-world usage is necessary

General Principles

- **When in doubt, leave it out**. You can always add, but you can never remove.
- **Just because you can doesn't mean you should**
- [Power-to-weight ratio](http://en.wikipedia.org/wiki/Power-to-weight_ratio)

```
> [A] measurement of actual performance [power / weight]
```

- **Don't give users a gun to shoot themselves with**
    - **Information hiding**: minimize the accessibility of *everything*

Documentation Matters

- Class: what an instance represents

```
* Method: contract between method and calling client (preconditions,
postconditions, and side-effects)
* Parameter: indicate units, form, ownership
```

Pre- and Post- Conditions

- The **precondition** statement indicates what must be true before the function is called.
- The **postcondition** statement indicates what will be true when the function finishes its work.

```
/// \post <return_value>.empty() == false
```

API vs. Implementation

Implementation details should not impact the API. Don't let implementation details "leak" into the API.

Performance

- Design for usability, refactor for performance
- Do not warp the API to gain performance
- Effects of API design decisions on performance are real and permanent:
    - `Component.getSize()` returns `Dimension`
    - `Dimension` is mutable

    - Each `getSize` call must allocate `Dimension`
    - Causes *millions* of needless object allocations

"Harmonize"

- API must coexist peacefully with platform
    - Do what is customary (standard)
    - Avoid obsolete parameter and return types
    - Mimic patterns in core APIs and language
    - Take advantage of API-friendly features: generics, varargs, enums, default arguments
- Don't make the client do anything the module could do
    - Reduce need for **boilterplate code**
- Don't violate the [Principle of Least Astonishment](http://en.wikipedia.org/wiki/Principle_of_least_astonishment)

```
  > The design should match the user's experience, expectations, and
mental models...aims to exploit users' pre-existing knowledge as a way
to minimize the learning curve
```

- Provide programmatic access to all data available in string form => no client string parsing necessary
- Overload with care: ambiguous overloadings

Names Matter

- Largely self-explanatory (avoid cryptic abbreviations)
- Be consistent (e.g. same word means same thing)
- Strive for symmetry
- Should read like [prose](http://en.wikipedia.org/wiki/Prose)

```
 > [T]he most typical form of language, applying ordinary grammatical
structure and natural flow of speech rather than rhythmic structure (as
in traditional poetry).
      if (car.speed() > 2 * SPEED_LIMIT)
        generateAlert("Watch out for cops!");
```

Input Parameters

- **interface types over classes**: flexibility, performance
- **most specific possible type**: moves error from runtime to compile time
- use `double` (64 bits) rather than `float` (32 bits): precision loss is real, performance loss negligible
- **consistent ordering**:

```
#include <string.h>
char *strcpy (char *dest, char *src);
```

```
void bcopy   (void *src,  void *dst, int n); // bad!
```

- **short parameter lists**: 3 or fewer; more and users will have to refer to docs; identically typed params harmful
    - Two techniques for shortening: 1) break up method, 2) create helper class to hold parameters

Return Values

- Avoid values that demand *exceptional* processing

```
 > For example, return a `zero-length array` or `empty collection`, not `null`
```

Exceptions

- don't force client to use exceptions for control flow
- don't fail silently
- favor unchecked exceptions
- include failure-capture diagnostic information
- **Fail fast**: report errors ASAP
    - **Compile time**: static typing, generics
    - **Run time**: error on first bad method invocation (should be **failure-atomic**)

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>
```

How to add comments which are only visible to editor, not readers of a page?

Use the HTML comments: for example, the following comment will not show up in the paper rendered. But it is visible to editor to reminder why things are done in certain way.

```
<!-- Please keep the pixel size to 400 so they are clean in the pdf
version, Thanks!  -->
[[File:Rose-compiler-code-review-1.png|thumb|400px|Code review using
github.llnl.gov]]
```

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

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

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

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

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

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

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

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

- This page was last modified on July 26, 2012, at 22:23.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of Use for details.

- Privacy policy
- About Wikibooks
- Disclaimers
- Mobile view

- 

-