ROSE User Manual:
A Tool for Building
Source-to-Source Translators
Draft User Manual
(version 0.9.11.115)

**Daniel Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke,
Markus Schordan, Rich Vuduc, and Qing Yi**
Lawrence Livermore National Laboratory
Livermore, CA 94550
925-423-2668 (office) 925-422-6278 (fax)
dquinlan@llnl.gov, liao6@llnl.gov, panas2@llnl.gov, matzke1@llnl.gov,
markus@complang.tuwien.ac.at, richie@cc.gatech.edu,
qingyi@cs.utsa.edu
Project Web Page: www.rosecompiler.org
UCRL Number for ROSE User Manual: UCRL-SM-210137-DRAFT
UCRL Number for ROSE Tutorial: UCRL-SM-210032-DRAFT
UCRL Number for ROSE Source Code: UCRL-CODE-155962
ROSE User Manual (pdf)
ROSE Tutorial (pdf)
ROSE HTML Reference (html only)

This ROSE User Manual is a very unevenly edited manual and contains many passages which simply seemed to its editors like a good idea at the time (from the *Hitchhiker's Guide To The Galaxy*).

September 12, 2019

September 12, 2019

Preface

Welcome to the ROSE Compiler Framework Project. The purpose of this project is to provide a mechanism for construction of specialized source-to-source translators (sometime referred to less precisely as *preprocessors*). ROSE provides simple programmable mechanisms to read and rewrite the abstract syntax trees generated by separate compiler front-ends. ROSE includes the Edison Design Group (EDG) front-end (in binary form within public distributions), and is internally based upon SAGE III, thus ROSE is presently specific to the generation of C and C++ source-to-source based compilers (*translators*, more precisely). Other language front-ends may be appropriate to add to ROSE in the future (current work with Rice University is focused on the addition of Open64's front-end to ROSE as part of support for FORTRAN 90).

ROSE makes it easy to build complex source-to-source translator (preprocessor) tools, and thus supports research work in many areas:

- Performance Optimization
- General Program Transformations
- Instrumentation
- Program Analysis
- Interface Generation
- Automated Check-pointing
- Software Security Analysis
- Software Verification
- Automated Unit Test Generation
- ... and much more ...

Acknowledgments

The Intermediate Representation (IR) used in ROSE is called SAGE III. SAGE III is something that we have built based on SAGE II, which was never completed or widely distributed. SAGE II was based on SAGE++, the improvements over SAGE++ were significant. SAGE II was the first version of SAGE to use the Edison Design Group (EDG) frontend. We want to thank the original developers of SAGE++ and SAGE II for their work, from which we learned a lot through use of their object-oriented IR interface.

We chose the name SAGE III to give sufficient credit to the original developers of SAGE++ and SAGE II, who also suggested that we call what we are doing SAGE III. ROSE, of course, builds on SAGE III and adds numerous additional mechanisms, including:

- Loop Optimizations (called by ROSE users)
- Abstract Syntax Tree (AST) Attributes (tree decoration)
- A family of AST Traversals (as used with Attribute Grammars)
- AST Rewrite mechanisms
- AST Query Mechanisms
- C and C++ code generation from SAGE III
- AST File IO
- Significant robustness for C, C99, and C++ (handles large DOE applications)
- AST Visualization
- and more ...

SAGE III is an automatically generated body of software that uses ROSETTA, a tool we wrote and distribute with ROSE. ROSETTA is an IR generator that, as its largest and most sophisticated test, generates SAGE III. The connection code that was used to translate EDG's AST to SAGE II was derived loosely from the EDG C++ source generator and has formed the basis of the SAGE III translator from EDG to SAGE III's IR. Under this license we exclude the EDG source code and the translation from the EDG AST in distributions and make available only a binary of those parts with use EDG (front-end AST translation), and the source to all of ROSE (which does not depend on EDG). No part of the EDG work is visible to the user of ROSE. We can make the EDG source available only to those who have the free EDG research license. We want to thank the developers at Edison Design Group (EDG) for making their work so widely available under their research license program.

Markus Schordan was the first post-doctorate researcher on the ROSE project; he made significant contributions while employed at Lawrence Livermore National Laboratory (LLNL), including the AST traversal mechanism. We continue to work with Markus, who is now at Vienna University of Technology as an Associate Professor. We were also fortunate to leverage a significant portion of Qing Yi's thesis work (under Ken Kennedy) and we would like to thank her for that work and the work she did as a post-doc at Lawrence Livermore National Laboratory. We continue to work with her, although she is now at the University of Texas at San Antonio.

There are many additional people to thank for our current status in the ROSE project:

- Contributing Collaborators:
  Markus Schordan (Vienna University of Technology), Rich Vuduc (Georgia Tech), and Qing Yi (University of Texas at San Antonio)

- Post-docs (including former post-docs):
  Chunhua Liao (from University of Houston), Thomas Panas (from Vaxjo University, Sweden), Markus Schordan (from University of Klagenfurt, Austria), Rich Vuduc (from University of California at Berkeley), and Jeremiah Willcock (from Indiana University), Qing Yi (from Rice University)

- Students:
  Gergo Barany (Technical University of Vienna), Michael Byrd (University of California at Davis), Gabriel Coutinho (Imperial College London), Peter Collingbourne (Imperial College London), Valentin David (University of Bergen, Norway), Jochen Haerdtlein (University of Erlanger, Germany), Vera Hauge (University of Oslo, Norway), Christian Iwainsky (University of Erlanger, Germany), Lingxiao Jiang (University of California at Davis), Alin Jula (Texas A&M), Han Kim (University of California at San Diego), Milind Kulkarni (Cornell University), Markus Kowarschik (University of Erlanger, Germany), Gary Lee (University of California at Berkeley and Purdue University), Chunhua Liao (University of Houston), Ghassan Misherghi. (University of California at Davis), Peter Pirkelbauer (Texas A&M), Bobby Philip (University of Colorado), Radu Popovici (Cornell University), Robert Preissl (xxx Austria), Andreas Saebjornsen (University of Oslo, Norway), Sunjeev Sikand (University of California at San Diego), Andy Stone (Colorado State University at Fort Collins), Ryan Stutsman (Stanford University), Danny Thorne (University of Kentucky), Nils Thuerey (University of Erlanger, Germany), Ramakrishna Upadrasta (Colorado State University at Fort Collins), Christian Wiess(Munich University of Technology, Germany), Jeremiah Willcock (Indiana University), Brian White (Cornell University), Gary Yuan (University of California at Davis), and Yuan Zhao (Rice University).

- Friendly Users:
  Paul Hovland (Argonne National Laboratory), Brian McCandless (Lawrence Livermore National Laboratory), Brian Miller (Lawrence Livermore National Laboratory), Boyana Norris (Argonne National Laboratory), Jacob Sorensen (University of California at San Diego), Michelle Strout (Colorado State University), Bronis de Supinski (Lawrence Livermore National Laboratory), Chadd Williams (University of Maryland), Beata Winnicka (Argonne National Laboratory), Ramakrisna xxx (Colorado State University at Fort Collins), and Andy Yoo (Lawrence Livermore National Laboratory)

- Support:
  Steve Ashby, David Brown, Bill Henshaw, Bronis de Supinski, and CASC management

- Funding:
  Fred Johnson (Department of Energy, DOE) and Mary Zosel (Lawrence Livermore National Laboratory)

...elling of ... names. To be clear, nobody is to blame for the poor state of the current version of the ROSE documentation (but myself).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 What is ROSE

ROSE is an open source compiler infrastructure for building tools that can read and write source code in multiple languages (C/C++/Fortran) and/or analyze binary executables (using the x86, Power-PC, and ARM instruction sets). The target audience for ROSE is people building tools for the analysis and transformation of software generally as well as code generation tools. ROSE provides a library (librose) that can be used to support the universal requirements of tools that do custom analysis and/or transformations on source code and custom analysis of binary forms of software. ROSE is portable across and expanding range of operating systems and work with an growing number of compilers.

ROSE provides a common level of infrastructure support to user-defined tools, so that they need not implement the complex support required for software analysis and transformation operations. For source code based tools these include parsing, common forms of compiler analysis, common transformations, and code generation. For binary analysis based tools these include disassembly, function boundary detection, and common forms of analysis. User defined tools may also mix source code and binary analysis to form more interesting tools for specialized purposes. ROSE is part of research work to unify the analysis of both source code and binaries within general compiler research and define mixed forms of static and dynamic analysis.

ROSE works by reading the source code and/or binary and generating an Abstract Syntax Tree (AST). The AST forms a graph representing the structure of the source code and/or binary executable and is held in memory to provide the fastest possible means of operating on the graph. The nodes used to define the AST graph are an *intermediate representation* (IR); common within compiler research as a way of representing the structure of software absent syntax details (commas, semi-colons, white-space, etc.). ROSE provides mechanisms to traverse and manipulate the AST. Finally, in the case of source code, ROSE provides mechanisms to regenerate source code from the AST.

As a trivial example, if the input source code program contains a variable declaration for an integer, all of this information will be available in the AST generated from the input code passed on the command line to any tool built using ROSE. Similarly, an automated transformation of the variable declaration held in the AST would be expressed using a traversal over the AST and code *semantic actions* to mutate the AST. Then the transformed source code would be generated (*unparsed*) from the AST. In the case of binaries (including executables, object files, and libraries), the AST will represent the structure of the binary. The AST for a binary also includes the binary file format (symbol table, debug format, import tables, etc.), disassembled instructions, all instruction operands, etc.

ROSE provides a rich set of tools to support the analysis of software including the support for users to build their own forms of analysis and specialized transformations. As an example, ROSE includes a full OpenMP compiler built using the internal ROSE infrastructure for analysis and transformation. A wide assortment of AST traversals are provided to express both analysis and transformations of the AST. A set of common forms of analysis are provided (call graph, control flow, etc.) most work uniformly on both source code and binary executables. Visualization support in included to help users understand and debug their tools. GUI support is available to support building professional level tools using ROSE. ROSE is actively supported by a small group at LLNL and is used as a basis for compiler research work within DOE at LLNL.

Technically, ROSE is designed to build what are called *translators*, ROSE uses a source-to-source approach to define such translators. Note that translators are significantly more sophisticated than *preprocessors* but the terms are frequently confused. A translator must understand the source code at a fundamentally deeper level using a grammar for the whole language and on the whole source code, where as a preprocessor only understands the source code using a simpler grammar and on a subset of the source code. It is *loosely* the difference between any language compiler and the C preprocessor (cpp).

1.2 Why you should be interested in ROSE

ROSE is a tool for building source-to-source translators. You should be interested in ROSE if you want to understand or improve any aspect of your software. ROSE makes it easy to build tools that read and operate on source code from large scale applications (millions of lines). Whole projects may be analyzed and even optimized using tools built using ROSE. For example, ROSE is itself analyzed nightly using ROSE.

To get started immediately consult the ROSE User Manual, chapter *Getting Started* for details).

1.3 Problems that ROSE can address

ROSE is a mechanism to build source-to-source analysis or optimization tools that operate directly on the source code of large scale applications. Example tools that *have* been built include:

- OpenMP translator,
- Array class abstraction optimizer,
- Source-to-source instrumenter,
- Loop analyzer,
- Symbolic complexity analyzer,
- Inliner and outliner,
- Code coverage tools,
- and many more...

Example tools that *can* be built include:

- Custom optimization tools,

- Custom documentation generators,
- Custom analysis tools,
- Code pattern recognition tools,
- Security analysis tools,
- and many more...

1.4 Research Goals for ROSE

ROSE is a project that aims to define a new type of compiler technology that allows compilation techniques to address the optimization of user-defined abstractions. Due to the nature of the solution we provide, it is also an open compiler infrastructure that can be used for a wide number of other purposes.

User-defined abstractions are built from within an existing base language and carry specific semantic information that can't be communicated to the base language's compiler. In many situations, the semantic information could be useful within program optimization, but the base-language compiler is forced to ignore this semantic information because there is no way for applications to pass such additional information to the base-language compiler. Note that `#pragmas` only permit information that the base-language compiler might anticipate (expect) to be passed; it is not a meaningful mechanism to communicate arbitrary information about user-defined abstractions to a compiler. ROSE is a part of general research on *telescoping languages* (a term coined by Ken Kennedy at Rice University) and CELL languages (a term coined by Bjarne Stroustrup). It is part of general work to define domain-specific languages economically from general purpose languages.

FIXME: *Check CELL in recent work*

1.5 ROSE: A Tool for Building Source-to-Source Translators

ROSE represents a tool for building source-to-source translators. Such translators can be useful for many purposes:

- automated analysis and/or modification of source code
- instrumentation
- data extraction
- building domain-specific tools

An optimizing translator can be expected to both analyze the input source code and automatically generate transformations of the source code; the result being a new source code. If successful, the automatically generated source code will demonstrate better performance. ROSE is the tool that helps users write such source-to-source translators. Expected users would be library writers and tool developers, not necessarily the application developers. As a result, we expect the ROSE user to be more knowledgeable about programming languages issues than the average application developer.

ROSE translators are particularly useful as a way to bridge the gap between what we want compilers to do and what they actually do. This *semantic gap* is significant when optimizing user-defined abstractions (functions and/or data structures), because the base-language compiler has no knowledge of their semantics. The optimization is particularly important within scientific applications. Such applications are often expensive to build because they are exceedingly complex and must too often be written at low levels of abstraction to maintain significant performance on modern computer architectures. The modern computer architectures themselves also vary widely and make the optimization of software difficult.

1.6 Motivation for ROSE

The original motivation for the development of ROSE comes from work within the Overture Project to develop abstractions for numerical computation that are efficient and easy to use. Basically, C++ language mechanisms made the abstractions easy to use (if not tedious to build), but efficiency was more problematic since the optimization of low-level abstractions can be (and frequently is) not handled well by the compiler. Specifically the rich semantic information the library writer embeds into his abstractions can't be communicated to the compiler, so many optimizations are missed. ROSE has addressed this fundamental problem by simplifying how an optimized translator could be built and tailored to a library's abstractions to introduce optimizations that use the high-level semantics of user-defined abstractions.

1.7 ROSE as a Compiler Framework

ROSE contains compiler infrastructure. This is because a translator that reads source code in any language is essentially a compiler (or *translator*). The most precise understanding of a source code in any language is the process of compiling it. Source-to-source compilation can, however, skip the common back-end code generation (since source code is generated instead of object code in the form of an executable). ROSE translators pay particular attention to reconstruct the generated source code (including comments and CPP translator control directives [#include, #if, #else, #endif, etc.], and the original application's indentation and variable names, etc.).

ROSE is unique because it makes traditional compiler infrastructure accessible to library and tool developers who are not likely to have a significant compiler background. Still, some basic knowledge of an Abstract Syntax Tree (AST) is assumed (and, unfortunately, currently required).

Figure 1.1 shows the different phases of processing within ROSE.

1.8 ROSE Web Site

We have a ROSE Project Web page that can be accessed at the *ROSE* Web pages at **http://www.rosecompiler.org**.

This site is updated regularly with the latest documentation and software, as it is developed. [1].

1.9 ROSE Software/Documentation

ROSE is not yet released publicly on the Web, but is available within the SciDAC Performance Evaluation Research Center (PERC) project and through limited collaborations with the developers at universities and other laboratories. Since the spring of 2006, we have made ROSE available via a password protected web page to all who have ask for access. More information is available on the ROSE Web pages, located at:

[1]All ROSE documentation is still in development

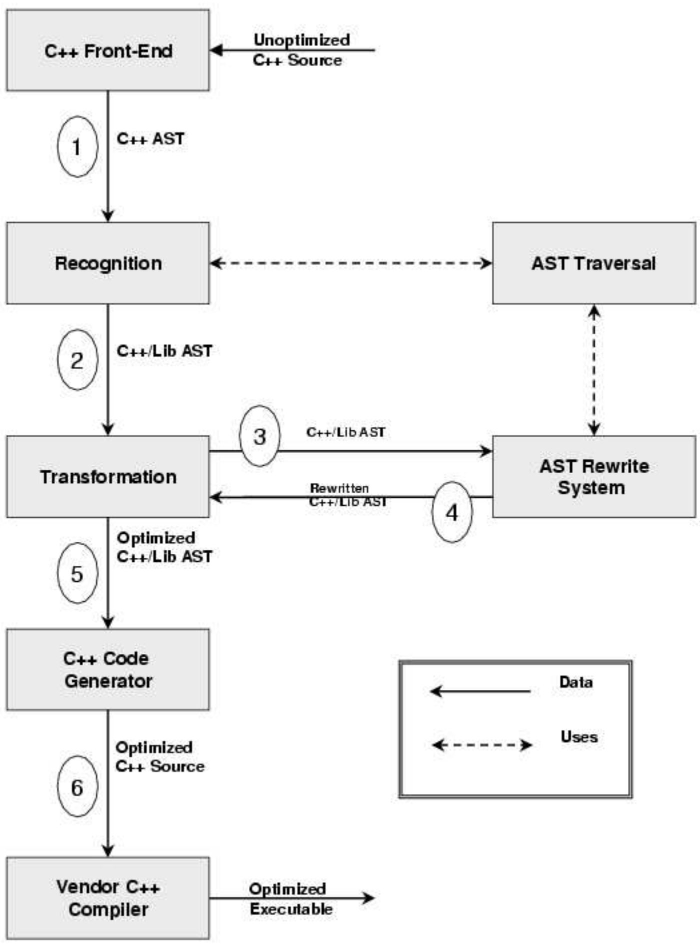

Figure 1.1: Different phases of internal processing within translators built using ROSE infrastructure

http://www.rosecompiler.org. Web pages are updated regularly (postscript versions of documentation are available as well).

1.10 About This Manual

This section includes a description of what this manual provides, how to use the manual, and the terminology related to the examples. An overview of the ROSE project is included. Error messages are contained in the Appendix (there are few at the moment). Further information is provided about the ROSE Web site, where more information is available and where the latest copy of the documentation is located. This Web site will also be the distribution site for ROSE, once it is made public; until then we welcome researchers to contact us directly to obtain pre-release versions of ROSE.

This manual is divided into several principal chapters. Each chapter covers material that, in some cases, requires an understanding of previous chapters. These are intended to simplify your use of this manual. Each chapter is described briefly below:

- **Preface**
  This section briefly describes what this project is about.
- **Acknowledgments**
  This section acknowledges contribution by many people over several years to the development of the ROSE project.
- **Introduction**
  This chapter introduces why we have developed ROSE and some of its organization.
- **Getting Started**
  This chapter walks the user through the configuration, compilation, installation, and testing of ROSE. Installation requirements are also explained. A small set of tests are available which verify the installation.
- **Writing a Source-to-Source Translator**
  This chapter presents, by example, the details of writing a trivial translator using ROSE.
- **Overview of ROSE**
  This chapter presents details of specific features in ROSE.
- **AST Query Library**
  This chapter presents work that has been completed to support simple and complex queries on the AST.
- **AST Traversal**
  This chapter covers different ways to write AST traversals (operators on the AST). This chapter is required to understand the subsequent chapter on the AST Rewrite Mechanism.
- **AST Rewrite Mechanism**
  This chapter covers the details of how to use the mechanism within ROSE for modifying the AST. This chapter describes how to write general transformations on the Abstract Syntax Tree (AST). It builds on concepts from the previous chapter.
- **Program Analysis**
  This chapter explains what program analysis is available within ROSE.

- **Loop Transformations**
  This chapter explains the loop optimization work that has been done.
- **SAGE III Intermediate Representation**
  This chapter details issues specific to the IR used in ROSE.
- **Appendix**
  This contains information that has not yet made its way into the manual. Much of this information will later be integrated into the User Manual, but until then, it is provided for reference. This chapter will at some point contain a reference to error messages (there are few at present, most abort upon error, just like a compiler).
- **Developer's Appendix**
  This chapter contains information specific to development of ROSE, and thus mostly of use only for ROSE developers.
- **Frequently Ask Questions (FAQ)**
  This chapter contains a series of frequently ask questions (FAQ) about the ROSE project.
- **Glossary**
  Terms and definitions that simplify the documentation are included in this section. More will be added over time.

A later version of the manual will include performance data on different machines so that the use of different features in ROSE can be better understood. This work is incomplete at present (implemented, but not yet represented in the documentation).

Chapter 2

Getting Started

This chapter details how to build ROSE and how to begin to use ROSE to build a source-to-source translator. ROSE uses EDG and SAGE III internally. EDG is a commercial (and proprietary) C++ frontend that we are permitted to use to support our research work. SAGE III is loosely derived from SAGE II, which is derived from SAGE++. SAGE III is a rewrite of SAGE II and uses a similar object-oriented design and a similar interface (API). The developers of SAGE II suggested that we call our work on the C++ intermediate representation Sage III. We are thankful to the developers of SAGE II for their work.

2.1 ROSE Documentation and Where to Find It

To simplify user access to the ROSE documentation, the pre-built postscript files are included in the `ROSE/docs/Rose` directory of each ROSE distribution. These versions are always kept up-to-date by the automated build system that generates ROSE distributions:

- **ROSE Web Page** : The ROSE Web page is located at www.roseCompiler.org.
  The web page contains the ROSE manual, tutorial and developer API. The API provides details about IR nodes and their usage (interfaces). The documentation is generated by Doxygen.
- **ROSE offline Web content** : ROSE/docs/Rose/ROSE-0.9.11.115-HTML-Docs.ps.gz
  ROSE HTML documentation that is available without internet access.
- **MANUAL** : ROSE/docs/Rose/ROSE-0.9.11.115-UserManual.ps.gz
  This is the ROSE User Manual which explains basic concepts about and capabilities within ROSE.
- **TUTORIAL** : ROSE/docs/Rose/Tutorial/ROSE-0.9.11.115-Tutorial.tar.gz
  This is the ROSE Tutorial with numerous examples of how to use ROSE.
  The tutorial documentation is constructed using the following steps:
  1. Actual source code for each example translator in the ROSE/tutorial directory is included.
  2. Each example is compiled.
  3. Inputs to the examples are taken from the ROSE/tutorial directory.

4. Output generated from running each example is placed into the tutorial documentation.

Thus, the `ROSE/tutorial` contains exact examples, and each example may be manipulated (changing either the example translators or the inputs to the examples).

- **PAPERS** : ROSE/ROSE_RESEARCH_PAPERS.tar.gz
  These are the current ROSE related research papers.

The ROSE project maintains an external mailing list (see information at: www.roseCompiler.org and click on the **Mailing Lists** link for how to join).

2.2 ROSE Installation

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html.

2.2.1 Software/Hardware Requirements and Options

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html

Required Hardware/Operating System

See http://rosecompiler.org/ROSE_HTML_Reference/installation_hardware.html

Software Requirements

See http://rosecompiler.org/ROSE_HTML_Reference/installation_prerequisites.html

2.2.2 Building BOOST

See http://rosecompiler.org/ROSE_HTML_Reference/installing_boost.html

2.2.3 Using Insure++

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

2.2.4 Building ROSE From a Distribution (ROSE-0.9.11.115.tar.gz)

ROSE is no longer distributed as a tarball. Use the Git repositories instead.

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html

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

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html

2.2.6 TroubleShooting the ROSE Installation

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

1. Message: `configure: error: Could not link against boost_filesystem-gcc41-mt`
   This message from running the `configure` command in ROSE (an initial step in building ROSE) indicates that your `LD_LIBRARY_PATH` (environment variable) is not set to to the location of the boost install tree. The ROSE configure scripts (autoconf) will test the linking to specific boost libraries and this is the first dynamic link library that it tests and so it will fail when many other tests on boost succeed because your `LD_LIBRARY_PATH` is finally required and is not properly set.

2. Message: `Making all in libltdl`
   `make[2]: *** No rule to make target 'all'. Stop.`
   Run `glibtoolize --force` to rebuild the libtool support in ROSE for your machine at the top level of the source tree. If that does not work then give up on the libtool that came with the apple dev tools and just build your own libtool in your home directory.

3. **Don't build ROSE in the source tree, it is not tested often, but it should work.**
   Save yourself some trouble and build a separate compile tree. This will also allow you to build a number of different versions of ROSE with different options.

4. Message: **configure: error: Unable to find path to JVM library**
   This message from running the `configure` command in ROSE (an initial step in building ROSE) indicates either that your `LD_LIBRARY_PATH` (environment variable) is not set to to the location of the *libjvm.so* or that your machines java is not one that we support (e.g. non-Sun Java). If you don't require Java (e.g. don't need Fortran support) then consider skipping the java support by using *–without-java* on the configure command line. Alternatively, your `LD_LIBRARY_PATH` should contain the path to the file *libjvm.so*. The likely path is specified in the lines just before the message. The full message will appear as:

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

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

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

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

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

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

   Report from use:

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

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

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

2.2.7 ROSE Configure Options

For GNU autotools configuration options and documentation, run "configure –help".

Output of `configure --help` is detailed in Figures 2.2.7 (Part 1) and 2.2.7 (Part 2):

2.2.8 Running *GNU Make* in Parallel

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html

2.2.9 Installing ROSE

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html

2.2.10 Getting Help

See http://rosecompiler.org/ROSE_HTML_Reference/index.html

2.2.11 ROSE and the NMI Compile Farm

The NSF Middleware Initiative (NMI) has provides us with time on their system to support the robustness of ROSE across multiple platforms. ROSE is not tested on a wide range of platforms (see table 2.2.11). The prerequisites used for each platform (machine and operating system) are generated in the table from the input test descriptions located in the directory `ROSE/scripts/nmiBuildAndTestFarm/build_configs`.

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

NMI OS and machine (platform) Prerequisites for ROSE:

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

2.2.12 Installation Details for Specific Platforms

See http://rosecompiler.org/ROSE_HTML_Reference/installation.html

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

Fedora 11

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

Intel C++ Compiler

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

2.2.13 Installing ROSE under Windows

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

Setup

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

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

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

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

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

Debugging in VS 8

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

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

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

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

- General and Language options as above
- Debugging: enter the following under arguments "–help"

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

Hudson specific

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

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

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

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

Extending ROSE with the Windows SDK

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

2.2.14 Options for Static vs. Dynamic Linking of Executables

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

- dynamic linking (default): **–enable-shared**

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

2.2.15 Options to Control the Size of ROSE Executables

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

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

The sizes of executables built using dynamic linking are always trivially small, the significant different matters when executables are built using static linking (see section 2.2.14). For a statically linked executable (ROSE-based tool) the size with symbol tables (debugging information) can be 150-200 Meg per executable; this value is for the g++ compiler, other compilers will vary in the sizes of the executables they generate. Turning off the debug information generated by g++ will reduce the size of the same executable to be about 40 Meg. Executables can be stripped of symbol table information further using the `strip` utility (Linux). Executables using ROSE that are stripped will be about 25% smaller (about 30 Meg).

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

2.3 Building Translators Using ROSE

At this point you should have installed ROSE. For examples of ROSE translators see the ROSE-0.9.11.115-Tutorial.tar.gz and the examples in the `ROSE/tutorial` directory.

2.4 Robustness of ROSE

A significant focus of the ROSE project is on the robustness of the software supporting our project. We have based the C and C++ support upon the use of the EDG frontend (the same commercial quality frontend used by most commercial C++ compilers). ROSE is a research project at a Department or Energy (DOE) national laboratory. As such, it must handle DOE laboratory applications that scale to a million lines of code or more. ROSE is not an academic research project, nor is it a commercial product. This section will layout what we do to test ROSE, what parts we consider to be *robust*, and exactly what we mean by *robust*.

2.4.1 How We Test ROSE

ROSE Regression Tests

Our regression test of collected bugs reported over several years helps prevent the reintroduction of old bugs during the development process. Additional test codes and applications codes help provide more complete testing of

ROSE.

Elsa Regression Tests

Recent work has included the a separate regression test suit from the Elsa project (an open source C++ parser project). This is tested infrequently at this point, but will be folded into standard ROSE regression tests in the future. We wish to thanks Scott McPeak for the use of his rather large collection of tests that he uses within Elsa (about 1000 test codes that test many corners of the C, C99, and C++ language).

Plum Hall C and C++ Compiler Test Suite

This is a commercial C and C++ compiler test suit that was purchased for us by the DOE Advanced Simulation and Computing (ASC) program. We appreciate their substantial support of ROSE. They also fund part of the ROSE project, but these test codes are REALLY hard.

Nightly cron jobs

Nightly regression tests are run on ROSE, these are easy to setup using the command `crontab -e`, this will bring up an editor, then put in the following lines:

```
# Time Spec, 1st column: minute, 2nd column: hours, 3rd column: day, 4th column: month, 5th column: year?;
# then followed by the command to be run at the time specified by the time spec:
55 12 * * * cd /home/dquinlan/ROSE/svn-rose/scripts && ./roseFreshTest ./roseFreshTestStub-xyz.sh
```

Then build a special roseFreshTestStub-xyz.sh file (examples are in the *ROSE/scripts* directory); it holds the required paths for the environment to be setup.

2.4.2 What Parts of ROSE Are Robust

We consider the compiler construction issues – IR, code generation, AST traversal support, and low level AST transformation mechanisms – to be robust. These are the mechanisms that are dominantly tested by the regression suits and application codes. Specifically, a ROSE translator is built that does no transformation (e.g. *IdentityTransformation.C* in the ROSE Tutorial). Input files are processed with this translator, and the following steps are tested for each source file:

- EDG's AST is built internally.
- ROSE's AST (the SAGE III AST) is built from the EDG AST.
- EDG's AST is deleted.
- ROSE's AST traversals are tested.
- ROSE's AST Attribute Mechanism is tested in each IR node.
- ROSE's AST internal tests are done (all tests must pass).
- ROSE's Code Generator is used to regenerate the source code.
- Vendor compiler compiles the ROSE-generated source code.

Note that separate tests to run the executables generated form the vendor compiler's compilation of the ROSE generated sources are not automated. This is not yet a standard test in ROSE, just verified infrequently.

2.4.3 What Parts of ROSE Are *Not* Robust

Basically, the program analysis lags in robustness. The robustness of the program analysis and optimization in ROSE has only recently become a focus. This work is not yet as durable as the compiler construction aspects of ROSE. The development of the ROSE infrastructure requires that we can first compile and transform large scale applications before we address complex program analysis and its robustness.

2.5 Submitting a Bug Report

The rule is simple: the better quality the bug report, the higher priority it gets. All good bug reports include a very simple example that demonstrates the bug, and only that bug, so that it is clearly reproducible. We welcome your submission of good quality bug reports. You may also send email directly to *dquinlan *at* llnl *dot* gov*. Any bug report you submit will be added as a test code and used to test future versions of ROSE (please add **ROSE bug report** to the subject line). At a later point we will use a more formal bug tracking mechanism.

2.6 Getting a Free EDG License for Research Use

ROSE source code is released under BSD license to make it as easy as possible to use. ROSE uses the EDG (www.edg.com) C++ front-end to parse C++ code internally. No part of the EDG source code is visible to the user or ROSE. ROSE distributes a binary version of the EDG work for a limited but growing range of platforms (32-bit and 64-bit Linux, Mac OSX, etc.). Since ROSE does not yet routinely package a separate binary for more than this range of platforms, we can optionally provide the EDG source code as part of the distribution of ROSE. However, we only give out ROSE source code that includes the EDG source code to research groups that also get a free research license for the EDG source code (available from EDG).

We are particularly thankful to the EDG people for providing such a good quality C++ front-end and for allowing it to be used for research work in C++. They have permitted research work specific to the C++ language to address the complexity of real application written in C++, which would not otherwise be practical or within the scope of a research project.

To get a version of ROSE, we encourage you to contact EDG to obtain their research license. Instructions for getting an EDG license:

- Send email to these three fellows at EDG:
  - Steve Adamczyk jsa at edg.com
  - John Spicer jhs at edg.com
  - Daveed Vandevoorde daveed at edg.com

I suggest sending the email to all of them at the same time so that they can see that you have sent email to the other two, since I really don't know which one is the correct person to contact. At some point we might get more information about a better approach.

The content of the email can be something like:

- We would like to work with the ROSE project at Lawrence Livermore National Laboratory (LLNL) which is using the EDG front-end for research on C++ optimization. They have asked that we obtain a research license in order to use ROSE for our research work with them.

They will then contact you (by email) and give you the location of the license form to fill out and get signed. They will either let you know where to get the EDG software or suggest that you get our version of their code directly from us. We will then give you all of ROSE, which includes (at present) the source code to the EDG front-end. You will not need a version of EDG directly from them.

`configure --help` Option Output (Part 1)

```
`configure' configures ROSE 0.9.11.115 to adapt to many kinds of systems.

Usage: /export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/configure [OPTION]... [VA

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

Defaults for the options are specified in brackets.

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

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

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

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR            user executables [EPREFIX/bin]
  --sbindir=DIR           system admin executables [EPREFIX/sbin]
  --libexecdir=DIR        program executables [EPREFIX/libexec]
  --sysconfdir=DIR        read-only single-machine data [PREFIX/etc]
  --sharedstatedir=DIR    modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR     modifiable single-machine data [PREFIX/var]
  --libdir=DIR            object code libraries [EPREFIX/lib]
  --includedir=DIR        C header files [PREFIX/include]
  --oldincludedir=DIR     C header files for non-gcc [/usr/include]
  --datarootdir=DIR       read-only arch.-independent data root [PREFIX/share]
  --datadir=DIR           read-only architecture-independent data [DATAROOTDIR]
  --infodir=DIR           info documentation [DATAROOTDIR/info]
  --localedir=DIR         locale-dependent data [DATAROOTDIR/locale]
  --mandir=DIR            man documentation [DATAROOTDIR/man]
  --docdir=DIR            documentation root [DATAROOTDIR/doc/rose]
  --htmldir=DIR           html documentation [DOCDIR]
  --dvidir=DIR            dvi documentation [DOCDIR]
  --pdfdir=DIR            pdf documentation [DOCDIR]
  --psdir=DIR             ps documentation [DOCDIR]

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

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

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

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
```

`configure --help` Option Output (Part 2)

```
                          '--enable-only-binary-analysis=no' and
                          '--disable-only-binary-analysis' are no longer
                          supported)
  --enable-only-c(=yes)   Enable ONLY C support in ROSE (Warning:
                          '--enable-only-c=no' and '--disable-only-c' are no
                          longer supported)
  --enable-only-cxx(=yes) Enable ONLY C++ support in ROSE (Warning:
                          '--enable-only-cxx=no' and '--disable-only-cxx' are
                          no longer supported)
  --enable-only-fortran(=yes)
                          Enable ONLY Fortran support in ROSE (Warning:
                          '--enable-only-fortran=no' and
                          '--disable-only-fortran' are no longer supported)
  --enable-only-java(=yes)
                          Enable ONLY Java support in ROSE (Warning:
                          '--enable-only-java=no' and '--disable-only-java'
                          are no longer supported)
  --enable-only-php(=yes) Enable ONLY PHP support in ROSE (Warning:
                          '--enable-only-php=no' and '--disable-only-php' are
                          no longer supported)
  --enable-only-python(=yes)
                          Enable ONLY Python support in ROSE (Warning:
                          '--enable-only-python=no' and
                          '--disable-only-python' are no longer supported)
  --enable-only-cuda(=yes)
                          Enable ONLY Cuda support in ROSE (Warning:
                          '--enable-only-cuda=no' and '--disable-only-cuda'
                          are no longer supported)
  --enable-only-opencl(=yes)
                          Enable ONLY OpenCL support in ROSE (Warning:
                          '--enable-only-opencl=no' and
                          '--disable-only-opencl' are no longer supported)
  --enable-languages=LIST Build specific languages:
                          all,none,binaries,c,c++,cuda,fortran,java,opencl,php,python
                          (default=all)
  --enable-binary-analysis
                          Enable binary analysis support in ROSE (default=yes)
  --enable-c              Enable C language support in ROSE (default=yes).
                          Note: C++ support must currently be simultaneously
                          enabled/disabled
  --enable-cxx            Enable C++ language support in ROSE (default=yes).
                          Note: C support must currently be simultaneously
                          enabled/disabled
  --enable-cuda           Enable Cuda language support in ROSE (default=yes)
  --enable-fortran        Enable Fortran language support in ROSE
                          (default=yes)
  --enable-java           Enable Java language support in ROSE (default=yes).
                          Note: --without-java turns off support for ALL
                          components in ROSE that depend on Java, including
                          Java language support
  --enable-php            Enable PHP language support in ROSE (default=yes)
  --enable-python         Enable Python language support in ROSE (default=no)
  --enable-opencl         Enable OpenCL language support in ROSE (default=yes)
  --enable-rtedupc        Enable UPC support in ROSE (default=no)
  --enable-compass2       build the Compass2 static analysis tool under
                          projects/
  --enable-projects-directory
                          Toggle compilation and testing of the the
                          ROSE/projects directory (disabled by default)
  --disable-tests-directory
                          Disable compilation and testing of the ROSE/tests
                          directory
  --disable-tutorial-directory
                          Disable compilation and testing of the ROSE/tutorial
                          directory
  --enable-memory-pool-no-reuse
                          Enable special memory pool model: no reuse of
                          deleted memory (default is to reuse memory)
  --enable-smaller-generated-files
```

```
NMI Platform (OS and Machine) Prerequisites
x86_64_deb_5.0 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3
x86_64_fedora_12-updated : "boost-1.36.0, libtool-2.2.6b"
x86_64_macos_10.5-updated : "boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.
x86_64_rhap_5 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b"
x86_64_rhap_5.2 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b"
x86_64_rhap_5.3 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b"
x86_64_rhas_4 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63"
x86_deb_5.0 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, libxml2-2.7.3"
x86_macos_10.4 : "boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3"
x86_rhap_5 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b"
x86_rhas_3 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, autoconf-2.59, automake-1.10"
x86_rhas_4 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63"
x86_sles_9 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3, t
x86_suse_10.0 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, automake-1.10, autoconf-2.63, libxml2-2.7.3
x86_suse_10.2 : "gcc-4.2.4, boost-1.36.0, libtool-2.2.6b, libxml2-2.7.3"
```

Figure 2.3: Example NMI machine preques used for nightly tests.

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

Figure 2.4: Example NMI machine configure options used for nightly tests.

Chapter 3

Writing a Source-To-Source Translator

This chapter contains information about how to build ROSE translators. Numerous specific examples are in the *ROSE Tutorial*, a separate document from this *ROSE User Manual*.

3.1 ROSE Tutorial

The ROSE Tutorial contains additional details and the steps used in examples of increasing sophistication. The ROSE Tutorial also explains a number of useful features of ROSE, including:

- AST Traversals.
  There are a number of different kinds of traversals, including a classic object-oriented visitor pattern and a more general useful traversal mechanism that supports a single `visit` function. Each traversal can operate on either just those IR nodes that have positions in the source file (non-shared), typically statements and expressions, or over all IR nodes (shared and non-shared).
- AST Queries.
  The ROSE Tutorial demonstrates the ROSE AST query mechanism and how to build more complex user-defined queries.
- PDF Output of AST.
  ROSE includes a number of ways to visualize the AST to support debugging and AST construction (i.e. how specific C++ examples map to the IR). A PDF representation of the AST permits the hierarchy of bookmarks to index the tree structure of the AST. This technique works on large-scale ASTs (typically a 300K-node AST [from a 40K-line source code] will define a 400Meg PDF file).
- DOT Output of AST.
  For smaller ASTs (less than 100K nodes) the AST can be viewed as a DOT graph. For very small ASTs, the graph can be converted to postscript files, but for larger graphs (500+ IR nodes), special dot viewers are required (e.g. *zgrviewer*).
- AST Rewrite Mechanism.
  The ROSE Tutorial shows examples of how to use a range of AST rewrite mechanisms for supporting program transformations.

Example Source-to-Source Translator

```
// Example ROSE Translator: used for testing ROSE infrastructure

#include "rose.h"

int main( int argc, char * argv[] )
   {
  // DQ (4/6/2017): This will not fail if we skip calling ROSE_INITIALIZE (but
  // any warning message using the message looging feature in ROSE will fail).
     ROSE_INITIALIZE;

  // 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);
   }
```

Figure 3.1: Example of simple translator, building and AST, unparsing it, and compiling the generated (unparsed) code.

3.2 Example Translator

This section shows an example translator that uses ROSE and how to build it. The ROSE Tutorial discusses the design of the translator in more detail; for now we need only an example translator to demonstrate the practical aspects of how to compile and link an application (translator) using ROSE.

Where is the example ction? We need to get gure closer to the text.

In this example, line 12 builds the AST (a pointer of type `SgProject`). Line 15 runs optional internal tests on the AST. These are optional because they can be expensive (several times the cost of building the AST). Look for details in the *Related Pages* of the *Programmer's Reference* for what tests are run. Line 20 generates the source code from the AST *and* compiles it using the associated vendor compiler (the backend compiler).

3.3 Compiling a Translator

We can use the following `makefile` to build this translator, which we will call `exampleMakefile` to avoid name collisions within the build system's `Makefile`.

In this case, the test code and makefile have been placed into the following directory: `{CompileTree}/ExampleTranslators/DocumentedExamples/SimpleTranslatorExamples`. The makefile `exampleMakefile` is also there.

Need to get the figure closer to the test.

To compile the test application, type `make -f exampleMakefile`. This builds an example translator and completes the demonstration of the build process, a process much like what the user can create using any directory outside of the ROSE compile tree.

3.4 Running the Processor

This section covers how to run the translator that you built in the previous section. Translators built with ROSE can be handed several options; these are covered in subsection 3.4.1. The command line required for the example translator is presented in subsection 3.4.2. Example output from a translator is presented in subsection 3.4.3.

3.4.1 Translator Options Defined by ROSE

The details of these options can be obtained by using the `--help` option on the command line when executing the translator. For example, using the example translator from the previous section, type `exampleTranslator --help`. Figure 3.4.1 shows the output from the `--help` option.

FIXME: *It appe figure reference n inc*

3.4.2 Command Line for ROSE Translators

Executing a translator built with ROSE is just like running a compiler with the compiler name changed to the name of the translator executable. All the command line arguments (except ROSE-specific and EDG-specific options) are internally handed to the backend compiler (additional command line options required for the EDG front-end are specified for the frontend along with any EDG-specific options; e.g. `--edg:no_warnings`). All ROSE and EDG specific options are stripped from the command line that is passed to the backend compiler for final compilation of the ROSE generated code; so as not to confuse the backend compiler.

Figure 3.4.2 shows the execution of a test code through an example translator.

3.4.3 Example Output from a ROSE Translator

Figure 3.4.3 shows the output of the processing through the translator.

Simple Makefile To Compile exam

```
# Example Makefile for ROSE users
# This makefile is provided as an example of how to use ROSE when ROSE is
# installed (using "make_install"). This makefile is tested as part of the
# "make_distcheck" rule (run as part of tests before any CVS checkin).
# The test of this makefile can also be run by using the "make_installcheck"
# rule (run as part of "make_distcheck").


# Location of include directory after "make_install"
ROSE_INCLUDE_DIR = /export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/install_tree/

# Location of Boost include directory
BOOST_CPPFLAGS = -pthread -I/nfs/casc/overture/ROSE/opt/rhel7/x86_64/boost/1_60_0/gcc/4.9.3/include

# Location of Dwarf include and lib (if ROSE is configured to use Dwarf)
ROSE_DWARF_INCLUDES =
ROSE_DWARF_LIBS_WITH_PATH = @DWARF_LINK@
ROSE_INCLUDE_DIR += $(ROSE_DWARF_INCLUDES)
ROSE_LIBS += $(ROSE_DWARF_LIBS_WITH_PATH)

CC                    = gcc
CXX                   = g++
CPPFLAGS              =
#CXXCPPFLAGS           = @CXXCPPFLAGS@
CXXFLAGS              =  -g -O2 -Wall -Wall
LDFLAGS               =

# Location of library directory after "make_install"
ROSE_LIB_DIR = /export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/install_tree/lib

ROSE_LIBS = $(ROSE_LIB_DIR)/librose.la

ROSE_SOURCE_DIR = /export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/exampleTransla

# Default make rule to use
all: exampleTranslator
	@if [ x$${ROSE_IN_BUILD_TREE:+present} = xpresent ]; then echo "ROSE_IN_BUILD_TREE_should_not_

# Example suffix rule for more experienced makefile users
# .C.o:
#	g++ -c -I$(ROSE_INCLUDE_DIR) -o $@ $(@:.o=.C)

# Compile the exampleTranslator using the file identityTranslator.C
exampleTranslator.lo:
	/bin/sh ../../../libtool --mode=compile $(CXX) $(CXXFLAGS)  $(CPPFLAGS) -I$(ROSE_INCLUDE_DIR)

exampleTranslator: exampleTranslator.lo
	/bin/sh ../../../libtool --mode=link $(CXX) $(CXXFLAGS) $(LDFLAGS) -o exampleTranslator examp

# Rule used by make installcheck to verify correctness of installed libraries
check:
	./exampleTranslator -c $(ROSE_SOURCE_DIR)/testCode.C
```

Figure 3.2: Example of makefile to build the example translator. Notice that we use the `identityTranslator.C` file presented in ROSE Tutorial.

`--help` Option Output
--help option output is **not** available (build ROSE, then rebuild documentation).

Figure 3.3: Example output from current version of translator build in `ROSE/src`.

Example command-line to execute `exampleTranslator`
exampleTranslator roseTestProgram.C

Figure 3.4: Example command-line for compilation of C++ source file (roseTestProgram.C).

Example Output From Execution of `exampleTranslator`
Execution output is **not** available (build ROSE, then rebuild documentation).

Figure 3.5: Example of output from execution of `exampleTranslator`.

Chapter 4

The ROSE Infrastructure

4.1 Introduction

This chapter was requested by several people who wanted to understand how ROSE was designed and implemented. ROSE supports a number of different languages and used different parsers and or frontends to address each on. For C, C99, UPC, and C++; we use the EDG frontend. While for Fortran we use the Open Fortran Parser as a parser and build the fronend end required. ROSE contains a midend, where analysis support is made available and and backend which does the code generation from the IR.

The goal of the design of the IR is to not loose any source code information. Thus ROSE is especially well suited to source-to-source translation. However then means that the IR for ROSE is quite large and this has advantages and disadvantages. The IR forms the base for an abstract syntax tree, so clearly some syntatic details are lost in the IR, but these are regenerated in the back-end (which has language specific support).

More languages could be added to ROSE, ROSE is designed to be langauge neutral, but it is implemented in C++. PHP has for example been added to ROSE, but it represnted initial work and an experiment with the general subject of run-time typed scripting language support.

4.2 Design

Fundamentally, ROSE has three parts:

1. frontend, which addresses language specific parsers/frontend issues (and the binary disassembly for the case of the binary support in ROSE);

2. midend, which addresses analysis and transformation issues;

3. backend which addresses code generation issues.

The frontend constructs an AST which saves as much as possible about the structure of the original source code (or binary for the case of the ROSE binary supposrt).

This section will cover the design goals etc. of ROSE.

4.3 Directory Structure

The top level of the ROSE directory tree has a simple design. All the source code is in `src`, all the tests are in `tests`, all the documentation is in `docs`. ROSE uses **autoconf** and **automake** so there is an autoconf generated *configure* script included. The `conf` directory contains all the *autconf macros* used in ROSE. The `projects` directory contains a collection of ongoing and past projects in ROSE that are either not large enough or mature enough to stand along as sperate projects. We use this location to incubate developing tools or technologies built on ROSE, as they are developed some are moved into the ROSE `src` directory proper. The README file contains information on how to install ROSE, and information about where information on ROSEis located.

Add more detail about each directory.

4.4 Implementation of ROSE

ROSE is implemented in C++. It supports source-to-source analysis and transformations on source code in a language neutral way (or alternativeively in a collection of langauge specific ways).

This section will be added to in the future.

4.4.1 Implementation of ROSETTA

ROSETTA is a tools built internally to generate code for ROSE so that ROSE follows simple and consistant design rules. ROSE relies heavily on code generatation as a way to automate as much as possible and permits ROSE to be maintained by as easily as possible. ROSETTA is thus used so that we can avoid spending all our time doing mainainance. ROSETTA is however not very ROSE specific and might be more generally useful, we have not pursued this line of work. We are happy to have ROSETTA be only used in ROSE, it is however separated out in the `src/ROSETTA/src` and `src/ROSETTA/Grammar` directories.

This section will be added to in the future.

4.4.2 Implementation of Fortran support

All fortran support in ROSE used the Open Fortran Parser (OFP) developed at Los Alamos and part of a community effort to define an open fortran parser that tracks the Fortran language (supports Fortran 2003 and the anticipated Fortran 2008). ROSE uses the OFP and builds from the parser the implementations of the parser actions required to construct a proper Fortran frontend. That the Fortran frontendin ROSE uses the ROSE IR means that the analysis in the midend can be used (or has been fixed up for use with Fortran). A backend is also defined in ROSE so that source-to-source support for Fortran is provide.

Chapter 5

SAGE III Intermediate Representation

There are many details that this chapter on SAGE will present.

incomplete-doc

5.1 History of SAGE

We chose to develop and use SAGE III, originally developed as SAGE++ by Dennis Gannon and others at University of Indiana, and then SAGE II by Dennis at IU and Carl Kesselman at ISI, and others. Because SAGE III is a reimplementation of the similar object-oriented IR API, their work gave us a significant head start in the development of ROSE (and an understanding of object-oriented IRs).

5.1.1 Differences Between SAGE++ and SAGE II

SAGE++ was the first version of SAGE and it provided support for C, a subset of C++ (C++ evolved quite a bit early on and was a moving target), and F90. SAGE II introduced the use of the EDG front-end, and dropped the handling of Fortran, but its work was incomplete.

5.1.2 Difference Between SAGE II and SAGE III

The SAGE III IR is now completely generated using the ROSETTA IR generator tool (a source-code generation tool) which we developed to support our work within ROSE. Initial versions of SAGE II were well done, but not complete. Numerous details were addressed in the work on SAGE II as part of its preparation for use within ROSE. We are very thankful to the initial developers of SAGE II for all their work. Sage III hopefully fulfills on a number of the goals behind their work. SAGE III continues to use the EDG frontend and has updated the versions of EDG in use (over SAGE II) and separated out the EDG work so that the connection of SAGE III to EDG is easier to maintain and update in the future with new versions of EDG.

5.1.3 Differences Between SAGE III and ROSE

ROSE uses SAGE III internally and adds numerous, more sophisticated mechanisms. For example, ROSE adds:

- Attribute mechanisms for use within traversals (ideas borrowed from attribute grammars).
- A sophisticated AST rewrite mechanism to simplify the development of transformations.

- A more sophisticated persistent attribute mechanism.
- Loop analysis and optimization (loop fusion, fission, blocking, etc.)
- Operators for conversion of AST subtrees to strings, and of strings to AST fragments.
- Database support for global analysis.
- C++ Template support.
- Fast binary AST File IO.
- An AST merge mechanism for supporting whole program analysis (across hundreds of files).
- Complete language support for C, C99, UPC, C++, Fortran 66, Fortran 77, Fortran 90/95, and Fortran 2003.
- AST visualizations (program visualization for debugging).
- ROSE User Manual and ROSE Tutorial Documentation.
- Full IR documentation via Doxygen (web pages).
- Web site with software and svn repository access.
- And lots more, ...

5.2 Comments Handling

Comments are placed into the SAGE III AST using a separate pass over the source file. EDG does not preserve comments at this time, and we felt it was important to preserve them within the unparsed (generated) output of the source-to-source mechanism that ROSE defines. Comment processing can also be addressed using the AST Rewrite Mechanism, though the order of how the comments appear in the code is determined by the order of invocation of the AST `insert()` function with a comment as the input string. Internally, the comments annotate the AST (tree decoration) so that AST queries may use the comments at will.

5.3 C Preprocessor (`cpp`) Directive Handling

The C Preprocessor (`cpp`) directives (*not* #pragma) are handled internally using the same mechanism as comments. Although they are fully expanded at compile time they are reinserted back into the unparsed source code as it is being unparsed. Internally, the directives annotate the AST (tree decoration) so that AST queries may use the directives at will. Note that pragmas are a part of the language specification (grammar) and not a CPP directive.

Note also that `extern ‘‘C’’` {} is also recognized so that it can be placed around `#include` directives and other identified blocks of declarations. Internally such declarations are explicitly marked as having extern C linkage.

5.4 Pragma Handling

The `#pragma` is special and is not really a C Preprocessor (`cpp`) directive. It is formally part of the C and C++ language grammar, and thus we are justified in putting it into the AST with the rest of the language constructs (comments and directives are open for a degree of interpretation as to where they can be attached within the AST). Details of this subject may be open to minor changes in future releases of ROSE.

Pragmas are the mechanism in which C and C++ permit language extension. Of course, some people describe this a bit differently, but `#pragma` is not interpreted by `CPP`, and it **is** interpreted by the compiler. And it has a specific semantics since it is part of the language grammar. The EDG documentation refers to them as pragma declarations, so they should be treated that way. This also is why they only really work in the grammar if they are declarations (since they are only permitted were common declarations are permitted and no where else).

Note that `#pragma pack` declarations are handled in a special normalization (see section 18). These pragmas are a bit different from other pragmas and are handled as a stack-based embedded language.

5.5 Copying IR Nodes and Subtrees

Support is provided for a policy-based copying of the AST and subtrees of the AST. Flexibility and control is provided through an independent policy mechanism that defines the copying process as shallow or deep for different types of nodes within the AST.

Each `SgNode` object has the following public virtual member function:

```
class SgNode {
     ...
     virtual SgNode* copy ( SgCopyHelp & help) const;
     ...
};
```

Here SgCopyHelp is a virtual policy class for duplicating `SgNode` objects and is defined as:

```
class SgCopyHelp {
   public:
     virtual SgNode* copyAst ( const SgNode *n ) = 0;
};
```

Two concrete classes, `SgShallowCopy` and `SgTreeCopy`, are provided as subclasses of `SgCopyHelp` to configure a shallow copy (duplicating the current `SgNode` object only) or a deep copy (duplicate the complete subtree rooted at the current `SgNode` object) respectively. The following example illustrates how to use `SgShallowCopy` and `SgTreeCopy` to duplicate SAGE nodes and sub-trees.

```
  SgNode *orig;
  ...
  SgNode *n1 = orig->copy(SgShallowCopy::static_instance());
  SgNode *n2 = orig->copy(SgTreeCopy::static_instance());
  ...
```

Here *n1* points to a duplicate of the `SgNode` object pointed to by `orig`, while *n2* points to a duplicate of the complete subtree rooted at `orig`. Therefore, the shallow copy *n1* from `orig` shares all the children of `orig`, while the deep copy *n2* from `orig` duplicates all the children of `orig` by recursively cloning the children objects. Note that the children of node `orig` are determined by the tree-traversal mechanism of ROSE. A field `fp` within `orig` is considered a child of `orig` only if `fp` is traversed by the tree-traversal mechanism. For all other fields in `orig`, only shallow copies are performed. As a result, only pointers to `SgNode`s that are part of the tree traversal rooted at `orig` can be recursively cloned.

To simplify the specification of shallow and deep cloning of `SgNode`s, two macros are further defined:

```
#define SgSHALLOW_COPY SgShallowCopy::static_instance()
#define SgTREE_COPY SgTreeCopy::static_instance()
```

The above example code, therefore, can be rewritten as:

```
  SgNode *orig;
   ...
  SgNode *n1 = orig->copy(SgSHALLOW_COPY);
  SgNode *n2 = orig->copy(SgTREE_COPY);
  ...
```

5.6 Template Handling in C++

The purpose of this section is to lay out the details of handling C++ templates. Initial template handling in SAGE III represented templates as classes and function (using generated, i.e. mangled, names) and with a flag indicating there derivation from a C++ template.

ROSE allows the transformation of templated classes and functions by generating the required specializations. This way, all details of a templated class of function (or static data member) become visible to the user in the AST and permit maximum information assumed to be required for any transformation. No transformation occurs on the template declaration unless it's done explicitly by the user (this is difficult since the text string representing the template is not formed into an AST that we can traverse). Note that this is a result of a design decision on the part of EDG to provide this as a default behavior and our decision to use it. More recent work to get the template as an AST is underway, using some of the options in EDG to support this. This later work is not robust enough to be the default in ROSE without a bit more work.

5.6.1 C++ Constructs That Can Be Made Into Templates

E: *Check on template unions.*

The concept of templates does not apply to all C++ constructs and affects only a few. The only things that can be templates are classes (including structs and likely unions), functions (including member functions), and variables (static data members). The first two are common, but the case of templated variables perhaps requires an example:

```
template<typename T>
class A
   {
     public:
      // non-template data member
         int nonTemplateDataMember;

      // template data member
         T templateDataMember;

      // template static data members
         static T staticTemplateDataMember_T;
```

```
        static float staticTemplateDataMember_float;
   };

// This is a template static data member (SgVariableDeclaration)
template<class U> U A<U>::staticTemplateDataMember_T;

// This is a template static data member (SgVariableDeclaration)
template<class U> float A<U>::staticTemplateDataMember_float;

// template specialization for variable (was originally defined to be float!)
template<> float A<double>::staticTemplateDataMember_float;

// template specialization for variable (error: this is not possible, type mismatch)
template<> float A<double>::staticTemplateDataMember_T;
```

In the case of a `SgVariableDeclaration`, the information about whether or not it is a specialization is kept with the `SgVariableDeclaration`, instead of the `SgInitializedName` objects that stand for the individual variables. Since the `get_parent()` member function returns a pointer to the `SgVariableDeclaration` from the `SgInitializedName`, this information is indirectly available from the `SgInitializedName`.

`Enums`, `typedefs`, `namespaces`, etc. cannot appear as templated declarations. As a result, only a few declarations contain template specific information (`SgClassDeclaration`, `SgFunctionDeclaration`, `SgVariableDeclaration`).

5.6.2 How Templates affects the IR

Some IR nodes are present to support the use of templates in C++. These include:

- `SgTemplateParameters`
  Derived from SgSupport.
- `SgTemplateArguments`
  Derived from SgSupport.
- `SgTemplateDeclaration`
  Derived from `SgDeclarationStatement`.
  - Holds the template string (any comments are removed)
  - Template name
  - Template parameters
- `SgTemplateInstantiationDecl` (*may be renamed to SgTemplateInstantiationClassDeclaration*)
  Derived from `SgClassDeclaration`.
  - Reference to `SgTemplateDeclaration`
  - Template arguments
- `SgTemplateInstantiationFunctionDecl`
  Derived from `SgFunctionDeclaration`.
  - Reference to `SgTemplateDeclaration`
  - Template arguments
- `SgTemplateInstantiationMemberFunctionDecl`
  Derived from `SgMemberFunctionDeclaration`.

  - Reference to `SgTemplateDeclaration`
  - Template arguments

- `SgTemplateInstantiationDirective`
  This forces the explicit instantiation of the specified template when (and where) it appears in the source code.

Nodes not added include (a judgement call for now):

- SgTemplateClassDeclaration
- SgTemplateFunctionDeclaration
- SgTemplateMemberFunctionDeclaration
- SgTemplateDataMemberDeclaration

There are many types of template declarations, at present there is an enum type which identifies each category of template declaration. The enum type is:

```
enum template_type_enum
   {
     e_template_none       = 0,
     e_template_class      = 1,
     e_template_m_class    = 2,
     e_template_function   = 3,
     e_template_m_function = 4,
     e_template_m_data     = 5
   };
```

A data member of this type is held in the `SgTemplateDeclaration`.

We might have to distinguish between template member functions and member functions of template classes, so that we can exclude instantiation of template member functions separately from member functions of template classes (which are required for the definition to appear in the generated source code). At present, this is done with a member function that computes this information (see the IR node documentation for more detail).

5.6.3 Template Specialization

Things that can be specialized include classes, structures, unions, variables (static data members of templated classes), functions, and member functions. Template and template instantiations need more information stored in the IR nodes to allow the unparser to be simplified. We currently compute this information within separate, post-processing, passes over the AST (see the source code in `ROSE/src/frontend/SageIII/astPostProcessing` for details). Interestingly, a template specialization is not an instantiation and can co-exist in each file and not cause linker problems (multiply defined symbols), it may cause generation of *weak symbols*.

5.6.4 Unparsing Templates

The general handling of templates requires a specific sorting of the template output. This order permits the generation of all template specializations, which allows each specialization to be transformed uniquely. This is important to the support of transformations on templates (based on template arguments). The order of output for template handling is as follows:

1. Output templates.
   Raw template declarations (text strings) are output at the top of the file.
2. Output template function prototypes.
   Function prototypes for all specializations that we generate are required before the use of the template forces its instantiation. The point is to allow the specialized template function to be available for transformation. It can be placed anywhere in the file (typically at the end in ROSE) as long as a prototype has been output to prevent a full instantiation of the specialized template function before any use would force its instantiation by the back-end compiler. At that point, the template specialization generated by ROSE (and perhaps transformed by the user) is not only redundant, but results in an error (since the function is defined twice – first instantiated by the vendor compiler and then seen as an explicit template specialization generated by ROSE).
3. Output template function definitions.
   All template specializations can be now output, even if they referenced templated classes for functions that would force the instantiations (the reason why all prototypes must proceed the definitions of template classes and functions). These can actually appear before or after the rest of the code, so #3 and #4 may be swapped).
4. Output the rest of the code.
   This will force template instantiations of any non-specialized template classes or functions. It may appear before the template functions definitions or mixed (interleaved) with them.
5. Output all explicit template instantiation directives at the base of each namespace where they appear. It is not clear that it is required to observe namespaces since the instantiation directive could reference fully qualified type names. This should be sufficient to resolve type ambiguity.

5.6.5 Templates Details

There are several details to enumerate:

1. Comments in templates are removed. They are saved in the SAGE III AST, but likely in incorrect positions, and not within the template (before or after the template declaration). They are not lost; since they are retrieved using a separate lex pass internally within ROSE. When template declarations appear in AST form, they will be placed into the correct positions in the generated code.
2. Options specific to templates can be classified as follows:
   - No transformations on templates.
     This is the first case to got working and it is the easiest case (to some extent). Template instantiation can be handled entirely by the vendor compiler, making life simple for ROSE. We also don't have to generate any template specializations.
   - Transformations on templates.
     This case can be separated into two separate cases. The second is harder to handle than the first).
     - Transformations on template functions (including member functions).
       This case will force transformations to happen as the templated functions are instantiated (and could not happen at any earlier phase). The instantiation can happen at an earlier stage than prelinking if we force auto-instantiation of templates (often triggered automatically if the program in represented by a single translation unit).
     - Transformation of template classes.
       This case is discussed in more detail below. It is a much harder case, and is currently incomplete.

    - Transformation of template static data members.
      This case is not handled yet, but should not be much trouble.

  - Transformation of template specializations.
    ROSE generates all template instantiations internally as template specializations. As such they are no different from any other AST subtree and all ROSE mechanism can be used for analysis and transformation of the instantiated template. Those instantiated templates that are transformed are marked for output in the code generation phase and output as template specializations. In this approach, templates instantiated for different types may be easily transformed differently.

3. Transformation of templated classes is enabled via generated specializations.
   This was discussed briefly above (under *Options specific to templates: item Transformation of template specializations* above). In general, transformations on template classes, functions, and static data members are handled through the explicit generation of specializations in place of the instantiations that would be generated by the back-end vendor compiler. All templates are explicitly generated as specializations in ROSE and, in principle, no instantiations are required by the back-end vendor compiler. It is not clear if ROSE needs to be so aggressive in eliminating template instantiations by the back-end vendor compiler, but doing so allows all template instantiations to be made available for transformation using ROSE. For simplicity, we only output (within code generation) those template instantiations that are required, due to transformations, and allow the back-end compiler to generate as many of the required template instantiations as possible.

   In order for a template to be transformed, we must save it into the SAGE III AST. If it is a class template, then we only want to unparse it into the final transformed code if it was modified. Otherwise its member functions and static members will not be defined at link time. Fundamentally, specialization of a class disqualifies the instantiation of its member functions from the original template declaration, because the newly instantiated template class becomes a distinct and separate class no longer associated with the original template. The vendor compiler can generate code for the new template class using the original template declaration or the member functions associated with the original template declaration. All the functions must be generated to go along with the new, specialized form of the templated class, which we had to specialize to permit it to be transformed.

   This potentially massive generation of all the member functions of a class applies only to transformations on class templates. Transformations on member function templates are not affected. They are instantiated in the prelink stage and seen in the SAGE III AST at that time. They can be transformed in the prelink stage, during, or immediately after the instantiation. This is the earliest possible stage where transformation on instantiated templates can be done. A transformation on a templated function is handled as a transformation on each of its instantiations.

   - Generation of code for transformed templated class.
     If a class template is modified, then we have to unparse all of the templated member functions! This is because an instantiated template cannot force its member functions to be instantiated (unless we do it explicitly, as I understand the template prelinking mechanism). Unparsing the instantiated template (with a mangled name) or as a specialization causes it to be considered as a new class and forces the construction of all member functions. This is a slightly different concept than instantiation (closer to specialization, I think, since specialization is not automated and must be handled explicitly as a result). Details are discussed earlier in this section.

     The declaration of a template class as a specialization requires declaration and definition of member functions of the class because the template mechanism would permit them to be different (even though this seems redundant, and even if we can automate the construction of the member function by au-

tomating the declaration of specializations for all member functions and static member data).
This mechanism needs to be controlled, so that we can control the amount of code generated. Options are:

- Always generate used class templates AND the member functions upon which they depend. This might seem to require that we generate all possible code, though in general it is only slightly less than all the member functions minus the template class definition. So maybe this is a suitable option, but not in the current plan.
- Generate only class template instantiations that have been transformed. Then generate all the member functions upon which they depend. This is the current design within ROSE.
- Generate only the function template instantiations that have been transformed (currently all function template instantiations are generated, since we can't know in advance which ones the user might wish to transform).

Note that if a template is never used in a given translation unit, then we will not instantiate it, and we can't even allow the user to see it for a possible transformation. This is not much different than existing vendor compilers that would not instantiate the template unless it was required by at least one translation unit. It can be argued that the ability to transform templated functions and classes that are never used by an application is inherently meaningless. As is the case for any vendor compiler, if the user wants to force instantiation of all templated classes, functions, and static data members, then he or she can do so by including a test code that forces the explicit instantiation of every class, function, static data member (or using explicit template instantiation directives).
If a class template has been modified then we need to make sure that all the class definition, member functions, and static data members are instantiated (on the next pass through the prelinker). The process should involve a call to the EDG function:

- `void set_instantiation_required_for_template_class_members (a_type_ptr class_type)`

5.6.6 Different Modes of Template Instantiation

We first supported only a single mode of template instantiation. Later we will consider supporting additional modes later. ROSE will respond to the EDG options to control automatic template instantiation using the option `-edg:t`*mode*, where the mode is either:

1. none (default)
   No template instantiation will be done.
2. used
   Only templates that are used in the translation unit will be instantiated.
3. all
   All possible templates will be instantiated.
4. local
   Only used templates will be instantiated and they will be forced to be local to the file. All instantiated functions will be declared as `static`. Note that `static` functions and member functions are only seen by the local file scope (translation unit, typically the source file).

5.7 Compiling ROSE-generated Code Using ROSE

These are a few notes about parts that might be difficult if they are encountered in code generated by ROSE (meaning that they had to first appear in an applications source code and the user wanted to run the generated code through ROSE again [I can't imagine why]). It is a rare but interesting possibility.

There are only a few cases where we generate code that might be a problem to compile using ROSE. When compiling for g++ (default), ROSE generates code that will avoid specific bugs in g++:

1. static const data members defined in the class definition (floats only) EDG accepts static and g++ supports const, and neither accepts what the other considers correct. ROSE generates code specific for the back-end and so the back-end must be specified in when running `configure` for ROSE. We don't currently support EDG as a back-end, though we support Intel C++ as a back-end and they use EDG, so this should work.

verify the details here.

5.8 Correctness of AST

When processing the AST, traversing it or rewriting it, it is useful to understand why things are the way they are in the AST's implementation. This section attempts to outline the properties that constitute the correctness of the AST.

1. Null pointers in the AST.
   In general, any null valued pointer is an error in the AST. This is a policy in SAGE III, and is dramatically different from SAGE II. Our push for this policy has been incremental over the years and remains somewhat incomplete.
   (a) Parent pointers.
   Pointers to parent nodes (available through the `SgNode::get_parent()` member function) in the AST are set/reset after construction of the AST. As a result of being set within a traversal of the AST, the parents perfectly match the traversal's concept of the AST *as a tree*. This point is important since the AST included edges that make it a directed graph, and it is the traversal of the AST that gives it its form/representation as a tree. Thus all parent pointers are valid (non-null) values, except the root of the AST, which has no parent (and has a null valued pointer returned from `SgNode::get_parent()`. There are two possible nodes that can be considered a root of the AST, either the `SgProject` or the `SgFile`; both nodes have constructors that take a translator's command line arguments.
   (b) Function declarations.
   Function declarations and function prototypes are confused in the AST, and where a function is defined, i.e. with a function body, it appears in the AST as a function declaration `SgFunctionDeclarationStatement` with a pointer to a function definition (`SgFunctionDefinitionStatement`. A function prototype can have a null valued pointer returned from its `get_definition()` member function and is marked explicitly as a function prototype (so that the null valued pointer can be error checked). If the function definition is available in the file (not always the case) then the `get_definition()` may return a valid pointer to it, even for a function prototype. Thus the explicit marking of declarations as a prototypes is critical to its interpretation as a function prototype.
   (c) Pointers to `SgBasicBlock`.
   All pointers of type SgBasicBlock should be valid pointers.
   (d) Other `NULL` pointers
   A conscious attempt is made within ROSE to not communicate information through a null-valued

pointer. Unfortunately, this has been a switch from the original design of SAGE II, which had `NULL` pointers throughout the AST. In general within the newer work, any NULL pointer is currently an error.

2. What lists can be empty.
SAGE III uses STL lists internally; children on many IR nodes are contained in such STL lists. There are nodes where the STL lists can be empty. These nodes include:
   (a) SgBasicBlock
   (b) SgGlobal
   (c) SgExpresionList
   (d) SgNamespaceDeclaration
3. Which access functions are simple and which do meaningful computation.
This question will be addressed later when we can automate queries of this sort. In general, member functions beginning with `get_xxx` and `set_xxx` get or set a private data member named `p_xxx`. Most such functions are trivial access functions, but some have more complex semantics. Given that there are over 200 IR nodes in the SAGE III IR, and that each has numerous member functions, we will defer addressing this question until we can implement a more automated mechanism on the SAGE III source code. See the Doxygen generated documentation for more details on the IR nodes and their member functions.

5.9 AST Normalization: Subtle Ways That ROSE Output Differs from the Original Source Code

In general, every attempt is made to preserve the look and feel of the original input code. Original formatting, use of C preprocessor directives (e.g. `#include<file.h>`), and comments are preserved within the AST and output in the generate code. However, there can be minor differences between the input source code and the code that is generated from ROSE translators. In all cases this difference is due to normalizations internally within the EDG front-end. Current normalizations include:

1. White space differences.
ROSE-generated code will appear somewhat different due to slightly different uses of white space within formatting of the generated code. All attempts are to preserve as much of the original formatting as possible (or practical).
2. Variable declarations are normalized to separated declarations.
Variable declarations containing multiple names (variables to be declared) are normalized within the AST to form one declaration for each name (variable). This simplifies program analysis since it avoids one of two ways of searching for a variable declaration (as a separate declaration and as a member of a list in another declaration). As an example:

```
int x,y,z;
```

appears in the AST (and in the unparsed [generated] code) as:

```
int x;
int y;
int z;
```

This *feature* could be changed at some point, but it has not been a high priority (and may be more desirable than the alternative).

3. Typedef template arguments are expressed in terms of their base type
This is not something that we can fix or change. EDG simply represents at least some and maybe all template arguments with their types normalized to strip away all typedefs. Fixing this would allow generation of code that is easier to verify visually. This may receive some attention in the future.

4. Comments within templates.
Comments within templates are ignored and not reproduced in the generated source code. This is because the template code is held in the AST as a string generated by EDG, and EDG ignores the comments. We currently output the comments at either the top or bottom of the template declaration. Later then the template declaration is represented as an AST, the comments will be folded into place where they belong.

5. Member functions of template instantiations.
Member functions of template instantiations use the same IR node as templated member functions of templated classes and templated member functions of non-templated classes. This is because the reason why a `SgTemplateInstantiationMemberFunctionDecl` exists to store the pointer to the `SgTemplateDeclaration` and there is only one of these, either because

   (a) the template declaration is of the class and the member function is declared in the class, or

   (b) the template declaration is of a member function of a templated class and is defined/declared outside of the class. In this case, the member function can be for a template or non-template member function, but not both.

6. Calls via dereferencing of function pointers.
Function calls from dereferencing pointers to functions can be represented with two different forms of syntax. For example:

```
xPtr ();
(*xPtr) ();
```

appears in the AST (and in the unparsed (generated) code) as

```
(*xPtr)();
(*xPtr)();
```

7. C++ style cast are normalized to C style casts.
EDG appears to normalize all C++ style cases to C style casts. We are working on the analysis to backout where C style casts could in fact be C++ style casts of a specific classification: const_cast, static_cast, dynamic_cast,and reinterpret_cast.

8. Floating-point literal normalization
Floating-point literals are internally represented in EDG as float, double, or long double (dependent on the type), thus the exact string representing the floating point literal is lost. We have modified EDG to save the string representation (from the token stream) the floating-point literal, this work is recent and handles all the different ways that floating point literals can be expressed (even including hexidecimal representation of floating point literals). The value as a float, double, or long double is also stored explicitly in the AST to simplify forms of analysis. Constant folded values are stored in the AST as well, with full unfolded constant expressions output in the generated code (by default), to reproduce the original source code as much as possible.

9. Normalization of member access from a pointer.
Member function access can be represented with two different forms of syntax. For example:

```
xPtr->foo();
(*xPtr).foo();
```

appears in the AST (and in the unparsed (generated) code) as

```
xPtr->foo();
xPtr->foo();
```

The following code is normalized differently (and somewhat inconsistently):

```
(*xPtrPtr)->foo();
(**xPtrPtr).foo();
```

appears in the AST (and in the unparsed (generated) code) as

```
(*(*xPtrPtr)).foo();
(*(*xPtrPtr)).foo();
```

when operators are explicitly defined by the user, as in

```
class A
   {
     public:
          A();
          A( int *x, int y);
          int & operator[](int i);
          A *operator->() const { return Aptr; }
          A& operator*() const  { return *Aptr; }
          A*  Aptr;
          A** Aptrptr;
   };
```

The following code is normalized differently (and somewhat inconsistently):

```
A a;
A* aptr = &a;
A** aptrptr = &aptr;

aptr->operator[](1);
(*aptr)[1];
(*aptrptr)->operator[](1);
(*(*aptrptr))[1];

(aptr->Aptr)->operator[](1);
(*(aptr->Aptrptr))->operator[](1);
```

and appears in the AST (and in the unparsed [generated] code) as

```
class A a;
class A *aptr = (&a);
class A **aptrptr = (&aptr);
(*aptr)[1];
(*aptr)[1];
(*(*aptrptr))[1];
(*(*aptrptr))[1];
(*aptr -> Aptr)[1];
(*(*aptr -> Aptrptr))[1];
```

10. Normalization of const ref (`const &`).
Const references, such as

```
X<A const & > x3;
```

are presently normalized to be

```
X<const A & > x3
```

11. Template arguments explicitly output.
Template types are output with template arguments. Code such as:

```
std::string var = std::string("");
```

is normalized to be

```
std::string var = std::basic_string < char , std::char_traits< char > , std::allocator< char > > ((""));
```

12. Constructor calls are really variable declarations.
C++ classes can define constructors. When they do the constructors are represented in the AST as a member function declaration and marked specifically as a constructor (conversion operators and destructors are also member function declarations and marked explicitly). However, the call to a constructor is a bit special in C++ and does not appear in the AST as a member function call. It appears as a variable declaration within the AST fragment representing the variable declaration a `SgConstructorInitializer` is used. So, where a variable of a class type `X` is written in the code as

```
X variable;
```

the form in the AST is more similar to the code represented by

```
X variable = X();
```

Semantically the two forms of code are equivalent (since the redundant constructor calls will be optimized away), and so this represents a form of normalization within the AST.

13. Redundant casts and copy constructors.
The use of redundant casts are represented as nested calls to copy constructors. Code such as:

```
std::string arg5 = (std::string) (std::string)std::string("");
```

is normalized to be

```
std::string arg5 =
     std::basic_string < char , std::char_traits< char > , std::allocator< char > >
          (std::basic_string < char , std::char_traits< char > , std::allocator< char > > ((""))));
```

14. Array indexing represented as pointer arithmetic.
Array indexing is translated by EDG into pointer arithmetic. It is not clear if this specific sort of AST normalization is desirable. Code such as:

```
void foobar ( double *d1, double *d2 );
void foo()
   {
     double **array;
     int n;
     array[n] = new double[100];
     foobar(&(array[n][n]),&array[n++][n]);
   }
```

is normalized to be

```
void foobar ( double *d1, double *d2 );
void foo()
   {
     double **array;
     int n;
     array[n] = new double[100];
     foobar (array[n] + n,array[n++] + n);
   }
```

15. Case statements always have an attached SgBasicBlock object.

16. Qualifiers are often normalized to longer names since they are computed on-the-fly as needed during unparsing. The original qualified names are lost and, as a result, the generated types can be excessively long and not at all similar to the original source code. For example, the `STL map::const_iterator` can become: $std ::_R b_tree < std :: map < int, int, std :: less < int >, std :: allocator < std :: pair < constint, int >>>:: key\_type, std :: map < int, int, std :: less < int >, std :: allocator < std :: pair < constint, int >>>:: value_type, std :: \_Select1st < std :: map < int, int, std :: less < int >, std :: allocator < std :: pair < constint, int >>>:: value\_type >, std :: map < int, int, std :: less < int >, std :: allocator < std :: pair < constint, int >>>:: key\_compare, std :: allocator < std :: pair < constint, int >>>:: const\_iterator$

    This problem could be fixed by computing a style alias table to permit the shortest type name to always be used. Either that or we should explicitly store the lists of qualified names and recompute them only where transformations have been done.

17. Unnamed typedefs of enums are normalized to enums.
    EDG appears to normalize unnamed typedefs to be enums, and the information about the origin as an unnamed typedef is lost. Since there appears to be no difference in the EDG AST, ROSE is unable to recover when the *typedef* keyword was used. This is not a real problem and the semantics of the application is the same. Without the name of the typedef, the typedef type can't be referenced except through its tag name. However, since there are subtle ways in which the tag name is not a type name in C (requires the *struct* keyword), this could be an issue for C. I have not isolated a code to demonstrate this as a problem. Thus, within ROSE, code such as:

    ```
    typedef enum enumType { zero = 0, one, two };
    ```

    is normalized to be

    ```
    enum enumType { zero = 0, one, two };
    ```

    This is demonstrated in `test2005_188.C`.

18. Packing pragmas: *#pragma pack* normalizations.
    The use of packing pragmas is handled separately from other pragmas within ROSE. Most pragmas are strings and no special processing is done internally. Packing pragmas assume a stack based semantics and allow:

    ```
    #pragma pack(n)      // Sets packing alignment to value n = 1,2,4,8,16, ... powers of 2
    #pragma pack(push,n) // Push previous packing alignment value and set new value to n
    #pragma pack(pop)    // Use previously pushed value of packing alignment
    #pragma pack(push)...#pragma pack(n)...#pragma pack(pop) // Alternative to #pragma pack(push,n) and #pragma pack(pop)
    #pragma pack()       // resets to packing alignment selected by compiler (default value)
    ```

    ROSE will normalize this to explicit packing pragmas for each structure (translating the *pack(push,n)* and *pack(pop)* to explicit values (using *pack(n)*). The reasons this is done is because this is that EDG stores the packing alignment values directly with the data structure and does not represent the pragma explicitly. Generated code using ROSE thus only uses *#pragma pack(n)* and *#pragma pack()* explicitly for each structure declaration (before and after each declaration, respectively). The specific placement of the *#pragma pack()* is also modified so that it appears immediately before and after the opening and closing parents for the class or structure definition. As an example, the following code as input:

```
#pragma pack(4)
struct A { unsigned short a; };
#pragma pack(push,8)
struct B1 { unsigned short a; };
struct B2 { unsigned short a; };
#pragma pack(pop)
struct C { unsigned short a; };
#pragma pack(push,1)
```

```
struct D { unsigned short a; };
#pragma pack(2)
struct F { unsigned short a; };
struct G { unsigned short a; };
#pragma pack(pop)
struct H { unsigned short a; };
struct I { unsigned short a; };
#pragma pack()
struct J { unsigned short a; };
```

will be translated (normalized) to

```
struct A
#pragma pack(4)
{ unsigned short a; }
#pragma pack()
;
struct B1
#pragma pack(8)
{ unsigned short a; }
#pragma pack()
;
struct B2
#pragma pack(8)
{ unsigned short a; }
#pragma pack()
;
struct C
#pragma pack(4)
{ unsigned short a; }
#pragma pack()
;
struct D
#pragma pack(1)
{ unsigned short a; }
#pragma pack()
;
struct F
#pragma pack(2)
{ unsigned short a; }
#pragma pack()
;
struct G
#pragma pack(2)
{ unsigned short a; }
#pragma pack()
;
struct H
#pragma pack(4)
{ unsigned short a; }
#pragma pack()
;
#pragma pack(4)
struct I { unsigned short a; }
#pragma pack()
;
struct J { unsigned short a; };
```

19. Expressions in C++ `typeid()` construct.

Expressions within are sometimes normalized. This is an example of input code using the typeid() operator:

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual ~A() { } };
struct B : A { };
struct C { };
struct D : C { };

void foo() {
  B bobj;
  A* ap = &bobj;
  A& ar = bobj;
  cout << "ap: " << typeid(*ap).name() << endl;
  cout << "ar: " << typeid(ar).name() << endl;
  D dobj;
  C* cp = &dobj;
  C& cr = dobj;
  cout << "cp: " << typeid(*cp).name() << endl;
  cout << "cr: " << typeid(cr).name() << endl;
  cout << "expression: " << typeid(true && false).name() << endl;
  bool t,f;
  cout << "expression: " << typeid(t && f).name() << endl;
  int less,more;
  cout << "expression: " << typeid(less < more).name() << endl;
  cout << "expression: " << typeid(less | more).name() << endl;
  cout << "expression: " << typeid(less + more).name() << endl;
```

This is the associated output code using the typeid() operator (with some reformatting)

```
#include <iostream>
#include <typeinfo>
using namespace std;

struct A { virtual inline ~A() {} };
struct B : public A {};
struct C {};
struct D : public C {};

void foo() {
  struct B bobj;
  struct A *ap = (&bobj);
  struct A &ar = bobj;
  ( *((&std::cout))<<"ap: "<<(typeid(( *ap))).name())<<std::endl;
  ( *((&std::cout))<<"ar: "<<(typeid(ar)).name())<<std::endl;
  struct D dobj;
  struct C *cp = (&dobj);
  struct C &cr = dobj;
  ( *((&std::cout))<<"cp: "<<(typeid(C )).name())<<std::endl;
  ( *((&std::cout))<<"cr: "<<(typeid(C )).name())<<std::endl;
  ( *((&std::cout))<<"expression: "<<(typeid(bool )).name())<<std::endl;
  bool t;
  bool f;
  ( *((&std::cout))<<"expression: "<<(typeid(bool )).name())<<std::endl;
  int less;
  int more;
```

```
  ( *((&std::cout))<<"expression: "<<(typeid(bool )).name())<<std::endl;
  ( *((&std::cout))<<"expression: "<<(typeid(int )).name())<<std::endl;
  ( *((&std::cout))<<"expression: "<<(typeid(int )).name())<<std::endl;
}
```

Notice that not all expressions are normalized, and that the cases which are normalized vs. those which are not is very subtle. This normalization appears to be a result of the internal working of EDG and not the Sage III IR. This test code can be found in `test2006_95.C`.

5.10 Non-Standard Features: C++ Extensions That We Are Forced to Handle

Philosophically, I don't think much of language extensions. we don't add any, we don't think we should add any, and we would not trust ourselves to add them correctly. That having been said, there are a few C++ extensions that are introduced by EDG (only one that I know of) and a fair number by g++. Because in many cases these features are implemented differently, we find them all worth avoiding. However, some applications use them, so we are somewhat forced to support them and handle the differences between how they are supported within both the EDG front-end and the back-end compiler (most often GNU g++). We list specific non-standard features of C++ that we are forced to handle (because applications we compile mistakenly use them).

One non-standard feature that requires special handling in ROSE is the in-class initialization of static const non-integer types. In-class initialization refers to code such as:

```
class X
   {
     public:
          static const int integerValueConstant  = 42;   // Legal C++ code
          static const int integerValueConstant  = 42;   // Legal C++ code
          static const bool booleanValueConstant = true; // Legal C++ code
          static const char charValueConstant    = '\0'; // Legal C++ code

       // Illegal C++ code (non-standard, does not compile with EDG, but does with g++)
          static const double doubleValueConstant1 = 3.14;
       // Illegal C++ code (non-standard, but compiles with EDG, and does not with g++)
          const double doubleValueConstant2 = 3.14;
   };
```

and it applies to integer-based types only (why such types are special while float and double are not, I don't know). However, `double` is somewhat supported as a non-standard extension by both EDG and GNU g++ (though in different ways). This is a little corner of C++ which is truly obscure, but shows up in some large applications at LLNL. Since the code that works with EDG does not work with GNU g++ (and vice versa), there is no common ground. So we assume that the code will compile using EDG (we have no choice) and then generate code that will compile with GNU g++. This means that we generate C++ code that can't be compiled with EDG, but this is the mess that application developers get themselves into when they use non-standard features.

The fix-up of the AST to force the generation of code suitable to GNU g++ is handled in the `ROSE/src/frontend/SageIII/astFixup` directory.

5.11 Notes on ROSE-specific Header Files

We borrow the header files of whatever compiler is specified as the target back-end compiler. This allows the same expansion of any macros as would be expanded without ROSE to match the expansion that would be done with ROSE. The mechanism for borrowing the header files from the target back-end compiler is somewhat messy, but fully automated. There are several steps, including translation and matching the values of the target compiler's predefined macros, to build a set of header files that can be used by ROSE (by the EDG front-end) from those used by the target back-end. The details are handled automatically, and need not be a concern for users of ROSE. We use the `--preinclude` mechanism in EDG to force a specific generated header file to be read ahead of any ROSE system header files (translated from the back-end system header files by the ROSE `configure` mechanism). This head file contains all the back-end specific macros definitions. The file name is: `rose_edg_required_macros_and_functions.h` and is placed in the install tree (`<prefix>/include/<back-end compiler name>_HEADERS/`.

5.12 Comments About Declarations (Defining Declarations vs. Non-defining Declarations)

Declarations come in two kinds: those that can have a separate definition (e.g class and function declarations) and those that cannot (e.g. enum and pragma declarations). For example, enums have to have their definition in their declaration; there is no concept of forward declarations of enums in C or C++.[1]

A class declaration, in C++, can have a forward declaration (even repeated forward declarations) before the declaration that contains the class definition (the {} part). Thus the following code is valid C++:

```
class X;    // forward declaration (declaration with NULL pointer to definition)
class X {}; // defining declaration (declaration with pointer to definition)
```

Note that multiple forward declarations can exist, as in:

```
class X;    // first forward declaration
class X;    // second forward declaration
class X {}; // defining declaration
```

The first forward declaration is the `firstNondefiningDeclaration` within ROSE. All forward declarations are marked as forward declarations (see declarations modifiers documentation, `isForward()` member function). The second forward declaration is just another declaration and should not be referenced as a firstNondefining declaration from any other declaration. Its defining declaration is set in the AST fix-up phase.

The following code is legal, but particularly bothersome (it now works in ROSE):

```
void foo (struct X *ptr);  // first declaration (but not really a forward declaration)
class X;     // first or second forward declaration (not really sure if this is the first or second)
class X {};  // defining declaration (one one of these is allowed, in the same scope)
```

In this code example, the first declaration of X appears in the function parameter list of the forward declaration of the function foo. This is not a typical forward struct declaration. We keep track of which is the defining declaration and which is the first nondefining declaration; the information about which is a forward declaration is somewhat redundant. The unparser can't just use the result of `isForward()` since declarations can be shared.

[1]This is in spite of the fact that they are implemented in many compilers. They are not part of the C or C++ language, so they are not implemented in ROSE. They are, however, one of the most common language extensions to C and C++ compilers (even certain standard following front-ends such as EDG).

This would result in unparsing the class definition multiple times. Thus, we separate the two concepts of defining and nondefining. Defining declarations are never shared (except through the `definingDeclaration` pointer); only non-defining declarations are shared (through the `firstNondefiningDeclaration` pointer).

SAGE III contains a `SgDeclarationStatement` IR node from which all declarations IR nodes are derived (e.g. `SgClassDeclaration`, `SgFunctionDeclaration`, etc.). Contained in the `SgDeclarationStatment` IR node are pointers (accessed through corresponding `get_` and `set_` member functions [access functions]) to the first declaration (called firstNondefiningDeclaration) and the defining declaration (called definingDeclaration). Both of these pointers are used internally when a pointer is required to a declaration (so that the same first declaration can be shared) and within the unparser (most importantly to output the definition where it appeared in the original code).

These pointers are initialized in the EDG/Sage interface code and are in a few cases (redundant forward declarations where only the first one is given a proper reference to the defining declaration), fixed-up in the ROSE/src/frontend/SageIII/AstFixes.C (AST fix-up phase). They are handy in transformations since they simplify how one can find a declaration and the definition if it is required.

5.13 Mangled Names and Qualified Names

Several C++ constructions (IR nodes) have qualified names. These are used to specify the location of the construct within the *space of names* (we have avoided calling the *space of names* the `namespace`, since that is a specific C++ construct) presented by the C++ program.

Note that none of the `get_mangled()` functions are called within the EDG/Sage translation (I think). At least none are called directly!

IR nodes that contain a `get_qualified_name()` member function are:

- SgEnumDeclaration
- SgTypedefDeclaration
- SgTemplateDeclaration
- SgNamespaceDeclarationStatement
- SgClassDeclaration
- SgTemplateInstantiationDecl
- SgMemberFunctionDeclaration
- SgScopeStatement
- SgGlobal
- SgBasicBlock
- SgNamespaceDefinitionStatement
- SgClassDefinition
- SgTemplateInstantiationDefn
- SgNamedType

Mangled names are a mechanism to build unique mappings to functions, classes, and any other constructs that could be identified using a non-unique string. Mangled names should include the qualified names of any scopes in which they are contained.

IR nodes that contain a `get_mangled_name()` member function are:

- SgInitializedName
- SgStatement (all derived classes)

Note that mangled names include parts that represents the qualified name. The algorithm used for name mangling is best described in the actual code where the documentation should be clear. The code for this is in the SgType IR nodes (and its derived IR nodes). The codes used for the operators is present in the function `SgType::mangledNameSupport(SgName,SgUnparse_Info)`.

5.14 Passing Options to EDG and ROSE

By default, all command line options (except EDG or ROSE-specific options) are passed to the back-end compiler. As a result the command line for the compiler can be used with any translator built using ROSE. This is particularly effective in allowing large complex `Makefiles` to be used by only changing the name of the compiler (`CC` or `CXX`).

Command line options are considered EDG-specified when prefixed with option: `-edg:xxx`, `--edg:xxx`, `-edg_parameter:xxx n`, or `--edg_parameter:xxx n`, which then translates to `-xxx`, `--xxx`, `-xxx n`, or `--xxx n` (respectively) for only the command line passed to the EDG front-end (not passed to the back-end compiler). These are required to support the different types of command line arguments used in EDG. For a complete list of the EDG options, see the EDG documentation (available only from EDG and covered under their license to use EDG).

Similarly, ROSE-specific command line options are prefixed using `-rose:xxx` and only interpreted by ROSE (not passed on to EDG or the back-end compiler). To see a complete list use any translator build using ROSE with the option `--help`.

All other options are passed to the back-end compiler with no processing.

5.15 How to Control Language Specific Modes: C++, C, C99, UPC

ROSE supports a number of different modes internally (within ROSE, the SAGE III IR, and the EDG front-end). There are five modes supported:

1. C++ mode.

   (a) C++ mode (default).
   This mode is used when compiling all files when no command line options are specified.

   (b) C++ (strict_warnings) mode `-edg:a`.
   This is the mode used when compiling with the `-edg:a`, violations are issued as warnings. Note that currently, gnu builtin functions are not properly defined in strict modes (so they modes should not be used).

   (c) C++ (strict) mode `-edg:A`.
   This is the mode used when compiling with the `-edg:A`, violations are issued as errors. Note that currently, gnu builtin functions are not properly defined in strict modes (so they modes should not be used). So these strict modes are incompatable with the use of the g++ and gcc compilers as a back-end to ROSE.

2. C mode.

(a) ANSI C (non-strict) mode.
This is the mode used when compiling with the `-rose:C_only` C89 standard (works best if files have ".c" filename extension). This implies conformance with the C89 ANSI standard. Also equivalent to `--edg:c` option.

(b) ANSI C (strict_warnings) mode `-edg:a`.
This is the mode used when compiling with the `-edg:a` in addition to the `--edg:c` or `-rose:C_only` options (file must have ".c" filename extension). This implies conformance with the C89 standard, violations are issued as warnings.

(c) ANSI C (strict) mode `-edg:A`.
This is the mode used when compiling with the `-edg:A` in addition to the `--edg:c` or `-rose:C_only` options (file must have ".c" filename extension). This implies conformance with the C89 standard, violations are issued as errors.

3. C99 mode.

(a) ANSI C99 default mode.
This is the mode used when compiling with the `--edg:c99` (file must have ".c" filename extension). This implies conformance with the C99 standard. This is the same as using `-rose:C99_only`.

(b) ANSI C99 *strict* mode.
This is the mode used when compiling with the `-edg:a` in addition to the `--edg:c99` or `-rose:C99_only` options (file must have ".c" filename extension). This implies conformance with the C89 standard, violations are issued as errors.

Note that in ANSI C99, flexible array structures can not be data members of other structures. See test2005_189.c for an example.

4. UPC mode.
This is the mode used when compiling with UPC specific modifiers, use `--edg:upc`. Note that we have modified the EDG front-end to support this mode for both C and C++ programs. The generated code does not support calls to a UPC runtime system at present, so this is just the mode required to support building the translator for C or C++ which would introduce the transformations required to call a UPC runtime system (such as has been done for OpenMP by Liao from University of Houston).

5. K&R C *strict* mode.
This is the mode used when compiling with the `--edg:old_c` (file must have ".c" filename extension). This option will not currently work with ROSE because prototyped versions of functions are used within `rose_edg_required_macros_and_functions.h` and these are not allowed in EDG's `--old_c` mode (translated from the ROSE `--edg:old_c`.

Most of the time the C++ mode is sufficient for compiling either C or C++ applications. Sometimes the C mode is required (then, typically `-rose:C_only` is sufficient). The specific K&R strict C mode does not currently work in ROSE. But K&R C will compile in both the C and often C++ modes without problem. For C99-specific codes (relatively rare), `-rose:C99_only` is sufficient. On rare occasions, a greater level of control is required and the other modes can be used.

5.15.1 Strict modes can not be used with g++ and gcc compilers as back-ends to ROSE

Note that currently, gnu builtin functions are not properly defined in strict modes (so they modes should not be used). This is a problem for strict modes for both C and C++.

5.15.2 Use `*.c` filename suffix to compile C language files

In general most C programs can be compiled using the `-rose:C_only` independent of their filename suffix. However, sometimes C program files that use a non *.c suffix cannot be handled by the `-rose:C_only` option because they contain keywords from C++ as variable names, etc. In order to compile these C language programs their files must use a *.c (lower case *c*) as a filename extension (suffix). This is an EDG issue related to the front-end parsing and the language rules that are selected (seemingly independent of the options specified to EDG and based partly on the filename suffix). Fortunately most C language programs already use the lower case *c*) as a filename extension (suffix). Test code `test2006_110.c` demonstrates an example where the **.c* suffix is required.

Chapter 6

Query Library

6.1 Introduction

This chapter presents defined techniques in ROSE to do simple queries on the AST that don't require an explicit traversal of the AST to be defined. As a result, these AST queries are only a single function call and can be composed with one another to define even composite queries (using function composition). Builtin queries are defined to return: AST IR nodes (Node Queries), strings (name queries), or numbers (number queries).

Any query can optionally execute a user-defined function on a SgNode. This makes it easier to customize a query over a large set of nodes. Internally these functions will accumulate the results from the application of the user-defined function on each IR node and return them as an STL list (std::list<SgNode*>).

There are three different types of queries in the NodeQuery mechanism:

1. queries of a sub tree of a AST from a SgNode,
2. queries of a node list, and
3. queries of the memory pool.

If the last parameter of the querySubTree has the value: *NodeQuery::ChildrenOnly* then only the IR nodes which are immediate children of the input IR node (SgNode*) in the AST are traversed, else the whole of the AST subtree will be traversed.

VariantVector objects are internally a bitvector or IR node types (from the hierarchy of IR nodes). VariantVector can be formed via masks built from variant names.

```
VariantVector ir_nodes (V_SgType);
```

For all AST queries taking a VariantVector, if no VariantVector is provided (to the function *queryMemoryPool()*) the whole memory pool will be traversed (all IR nodes from all files).

6.2 Node Queries

AST Queries can return list of IR nodes. These queries are useful as a simple way to extract subsets of the AST. Node queries can be applied to the whole of the memory pool or any subtree of the AST. The result of an AST Node query on the AST is a list of IR nodes, the same interface permits additional AST Node queries to be done of the STL list of IR nodes. This permits compositional queries using simple function composition.

6.2.1 Interface Functions

The functions supported in the AST Node Query interface are:

```
namespace NodeQuery
{
//** Functions that visits every node in a subtee of the AST and returns a
//Rose_STL_Container<SgNode*>s. It is the subtree of the first parameter of the
//interface which is traversed**//

  template<typename NodeFunctional>
   querySubTree( SgNode*, NodeFunctional,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

  querySubTree( SgNode*, TypeOfQueryTypeOneParameter,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

   querySubTree( SgNode*, roseFunctionPointerOneParameter,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

   querySubTree( SgNode*, SgNode*, roseFunctionPointerTwoParameters,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

  querySubTree( SgNode*, SgNode*, TypeOfQueryTypeTwoParameters,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

  querySubTree( SgNode*, VariantT,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

  querySubTree( SgNode*, VariantVector,
     AstQueryNamespace::QueryDepth = AstQueryNamespace::AllNodes)

//** Functions that visits every node in a Rose_STL_Container<SgNode*>'s and returns a
//Rose_STL_Container<SgNode*>s **//

  queryNodeList( NodeQuerySynthesizedAttributeType,
     TypeOfQueryTypeOneParameter elementReturnType)

  queryNodeList( Rose_STL_Container<SgNode*>, roseFunctionPointerOneParameter )

  queryNodeList( Rose_STL_Container<SgNode*>, SgNode*, roseFunctionPointerTwoParameters)

  queryNodeList( Rose_STL_Container<SgNode*>, SgNode*, TypeOfQueryTypeTwoParameters)

  queryNodeList( Rose_STL_Container<SgNode*>, VariantT)

  queryNodeList( Rose_STL_Container<SgNode*>, VariantVector*)

//** Functions that visit only the nodes in the memory pool that is
//specified in a VariantVector and returns a Rose_STL_Container<SgNode*>s **//

  template<typename NodeFunctional>
  queryMemoryPool( NodeFunctional, VariantVector* = NULL)

  queryMemoryPool( roseFunctionPointerOneParameter,
     VariantVector* = NULL)

  queryMemoryPool( SgNode*, roseFunctionPointerTwoParameters,
```

```
    VariantVector* = NULL)

  queryMemoryPool( TypeOfQueryTypeOneParameter,
     VariantVector* = NULL)

  queryMemoryPool( SgNode*, TypeOfQueryTypeTwoParameters,
     VariantVector* = NULL)

}
```

6.3 Predefined Queries

For the convenience of the user some common functions are preimplemented and can be invoked by the user through an enum variable. There are two types of preimplemented queries; a TypeOfQueryTypeOneParameter and a TypeOfQueryTypeTwoParameters.

```
enum TypeOfQueryTypeOneParameter
{
  VariableDeclarations,
  VariableTypes,
  FunctionDeclarations,
  MemberFunctionDeclarations,
  ClassDeclarations,
  StructDeclarations,
  UnionDeclarations,
  Arguments,
  ClassFields,
  StructFields,
  UnionFields,
  StructDefinitions,
  TypedefDeclarations,
  AnonymousTypedefs,
  AnonymousTypedefClassDeclarations
};
```

A TypeOfQueryTypeTwoParameters requires an extra parameter of SgNode* type like for instance the TypeOfQueryTypeTwoParameters::ClassDeclarationNames which takes a SgName* which represents the class name to look for.

```
enum TypeOfQueryTypeTwoParameters
{
  FunctionDeclarationFromDefinition,
  ClassDeclarationFromName,
  ClassDeclarationsFromTypeName,
  PragmaDeclarationFromName,
  VariableDeclarationFromName,
};
```

6.4 User-Defined Functions

Both C style functions and C++ style functionals can be used for the user-defined query functions. The C++ style functionals can be used together with powerful concepts like std::bind etc. to make the interface very flexible. An example functional is:

```
    class DefaultNodeFunctional :  public std::unary_function<SgNode*, std::list<SgNode*> >
    {
      public:
   result_type operator()(SgNode* node )
      {
result_type returnType;
returnType.push_back(node);
return returnType;
      }
    };
```

For the legacy C-Style interface there are two type of functions: typedef std::list <SgNode*>(*roseFunctionPointerOneParameter) (SgNode *); typedef std::list <SgNode*>(*roseFunctionPointerTwoParameters) (SgNode *, SgNode *); The second function allows a user-defined second parameter which can be provided to the interfaces directly. This parameter has no side-effect outside the user-defined function. For the querySubTree the second parameter to the interface will be the parameter to the user-defined function, but for the memory pool traversal and the query of a node list the first parameter will be the second parameter to the user defined function.

6.5 Name Queries

The name query provides exactly the same interfaces as the NodeQuery except for two differences; the user defined functions returns a Rose_STL_Container¡std::string¿s and the C-Style functions take a std::string as a second parameter. The predefined functions implemented in this interface are:

```
namespace NameQuery{

  enum TypeOfQueryTypeOneParameter
  {
    VariableNames,
    VariableTypeNames,
    FunctionDeclarationNames,
    MemberFunctionDeclarationNames,
    ClassDeclarationNames,
    ArgumentNames,
    ClassFieldNames,
    UnionFieldNames,
    StructFieldNames,
    FunctionReferenceNames,
    StructNames,
    UnionNames,
    TypedefDeclarationNames,
    TypeNames
  };

  enum TypeOfQueryTypeTwoParameters
  {
    VariableNamesWithTypeName
  };

}
```

6.6 Number Queries

The number query provides exactly the same interfaces as the NodeQuery except for two differences; the user defined functions returns a Rose_STL_Container¡int¿s and the C-Style functions take an 'int' as a second parameter. The predefined functions implemented in this interface are:

```
namespace NumberQuery{
     enum TypeOfQueryTypeOneParameter
{
  NumberOfArgsInConstructor,
  NumberOfOperands,
  NumberOfArgsInScalarIndexingOperator,
};

     enum TypeOfQueryTypeTwoParameters
{
  NumberOfArgsInParanthesisOperator
};
}
```

6.7 Extending the AST Query

The AST Query can be extended using the internal interface. The functions supported in the AST Query are.

The purpose of the AST Query is to provide a convenient mechanism for performing operations that can be done in each node in isolation. This operation may either be for the purpose of transforming the AST or more commonly for the purpose of returning some information about a set of nodes.

Operations on nodes are specified by a functional in the AST Query mechanism (http://www.cplusplus.com/reference/std/functional/). A functional is basically a stateless object that can be treated like a function. The benefit of using functionals over regular function pointers is that functionals can be used in conjunction with STL mechanism that extends it's capabilities. STL::bind can for instance help for extending the AST Query to take functionals of two arguments by binding the second argument to a variable. Binding the return list to the second variable tends to speed up the AST Query equivalence of the NodeQuery by up to 3000

Each return type needs a Merge(,) function to specify how to merge the result of two applications of the functional:

```
void AstQueryNamespace::Merge(Rose_STL_Container<SgNode*>& mergeWith, Rose_STL_Container<SgNode*>  mergeTo );
```

ROSE supports returning void*, Rose_STL_Container¡SgNode*¿, Rose_STL_Container¡int¿, Rose_STL_Container¡std::string¿ or Rose_STL_Container¡SgFunctionDeclaration*¿. But it is easy to extend it to support any other type by adding your own custom AstQueryNamespace::Merge function.

6.7.1 Query a Subtree of the AST

The AstQueryNamespace::querySubTree functions apply the provided functional to nodes in a subtree of the AST.

The following function let's you the subtree of 'node' for all the nodes returned by the functional 'nodeFunc' when applied by default to all nodes in the subtree using a preorder traversal. You can specify a querydepth of your own

```
  enum QueryDepth
  {
    ChildrenOnly                       = 1, // only of depth 1
    AllNodes                           = 2, // all nodes
    ExtractTypes                       = 3, // visit types
  };
```

or specify your own query order.

```
  template<typename NodeFunctional>
    typename NodeFunctional::result_type
    AstQueryNamespace::querySubTree(SgNode* node, NodeFunctional nodeFunc, AstQueryNamespace::QueryD
t_traverseOrder treeTraversalOrder = preorder)
```

The following function is similar to the function above that operates on nodes using functionals, but for convenience reasons we here support using function pointers instead of functionals. If the function pointer that you want to apply to every node is

```
std::list<SgNode*> exampleFunction(SgNode* arg1)
```

Then you have to instantiate the following template with the argument 'AstQueryNamespace::querySubTree< std::list<SgNode*> >(..)'.

```
    template <class _Result>
    _Result AstQueryNamespace::querySubTree ( SgNode * subTree,
_Result (*__x)(SgNode*),
AstQueryNamespace::QueryDepth defineQueryType = AstQueryNamespace::AllNodes )
```

We have extended the support for function pointers to function pointers with two arguments. If you want to apply the following function to every node

```
void* exampleFunction(SgNode* arg1, std::list<SgNode*> returnList)
```

then you have to instantiate the following template with the arguments 'AstQueryNamespace::querySubTree< std::list<SgNode*>, void* >(..)'. The third argument to 'querySubTree', x_arg, is always provided as the second argument to the function pointer.

```
    template <class _Arg, class _Result>
    _Result AstQueryNamespace::querySubTree ( SgNode * subTree,
_Result (*__x)(SgNode*,_Arg), _Arg x_arg,
AstQueryNamespace::QueryDepth defineQueryType = AstQueryNamespace::AllNodes )
```

6.7.2 Query an Iterator Range

The AST Query mechanism supports querying using iterators in a similar manner to STL functions in STL::Algorithms.

The following function will query the range betweeen begin and end.

```
    template <class Iterator, class NodeFunctional>
    typename NodeFunctional::result_type
    AstQueryNamespace::queryRange(Iterator begin, Iterator end,
NodeFunctional nodeFunc)
```

Similarly to querySubTree we support using function pointers instead of functionals. If you want to apply the functional

```
std::list<SgNode*> exampleFunction(SgNode* arg1)
```

to every node in the range you have to invoke the template like 'AstQueryNamespace::queryRange< std::list<SgNode*> > (..)'

```
    template <class _Result>
    _Result AstQueryNamespace::queryRange (typename _Result::iterator begin, typename _Result::iterator end,
_Result (*__x)(SgNode*))
```

We also support function pointers that take two arguments. If you for instance want to apply the pointer

```
void* exampleFunction(SgNode* arg1, std::list<SgNode*> returnList)
```

to every node in the range you have to invoke the template like 'AstQueryNamespace::queryRange< std::list<SgNode*>, void* > (..)'. the last argument of queryrange, x_arg, is always provided as the second argument to the function pointer.

```
    template <class _Arg, class _Result>
    _Result AstQueryNamespace::queryRange ( typename _Result::iterator begin, const typename _Result::iterator
_Result (*__x)(SgNode*,_Arg), _Arg x_arg)
```

6.7.3 Query the ROSE Memory Pool

The AST Query provides a mechanism for querying the node types in the memory pool without having to travers the whole AST. This can potentially give a large speedup if you only need to traverse certain types.

The basic version of querying the memory pool takes a functional 'nodeFunc' and a VariantVector that specifies which types should be traversed as arguments.

```
    template<typename NodeFunctional>
    typename NodeFunctional::result_type
    AstQueryNamespace::queryMemoryPool(NodeFunctional nodeFunc , VariantVector* targetVariantVector = NULL)
```

For convenience reasons we support using function pointers instead of functionals. If you want to apply the following function

```
std::list<SgNode*> exampleFunction(SgNode* arg1)
```

to every node of the types in the VariantVector you need to instantiate the following template as 'AstQueryNamespace::queryMemoryPool< std::list<SgNode*> >()'

```
template <class _Result>
_Result AstQueryNamespace::queryMemoryPool ( _Result (*__x)(SgNode*), VariantVector* targetVariantVe
}
```

We also support function pointers that take two arguments. If you want to apply the following function

```
void* exampleFunction(SgNode* arg1, std::list<SgNode*> returnList)
```

to every node type in the VariantVector you need to instantiate the following template as 'AstQueryNamespace::queryMemoryPool< std::list<SgNode*>, void* >()'. The second argument, x_arg, is always provided as the second argument to the function pointer.

```
template <class _Arg, class _Result>
_Result AstQueryNamespace::queryMemoryPool (
    _Result (*__x)(SgNode*,_Arg), _Arg x_arg,
    VariantVector* targetVariantVector = NULL)
```

Chapter 7

AST Traversal

7.1 Introduction

FIXME: *This cha cover both Object-Oriented Visi Traversal (which h built, but which I Markus is implement) and t traversals based on ite the memory pools built in the latest inte of ROSE in December three of th implemented are pr the ROSE Tutorial wh a section for the uni classic Object-Orier Pattern Traversal n memory pools as a p (only the code need b*

FIXME: *Does RO breadth-firs*

Tree traversal is a fundamental operation used to visit each node in a tree data structure in a systematic way. Such traversals are categorized by the order in which the nodes are visited. A traversal can start at the root and explore as far as possible along each branch before backtracking, which is referred to as a depth-first traversal. It can also be breadth-first traversal which begins at the root node and explores all nodes at depth level 1, then nodes at level 2, and so on. For depth-first traversal of a binary tree, it can in turn have three variants: preorder (parent, left-child, right-child), inorder (left-child, parent, right-child), or postorder (left-child, right-child, parent). Inorder traversal does not make too much sense for tree nodes with varying number of children since the "in" position of a parent among its children is not well defined. Please note that postorder (left-child, right-child, parent) is **NOT** equal to the reversed order of preorder (right-child, left-child, parent).

ROSE aids the library writer by providing a traversal mechanism that visits all the nodes of the AST in a predefined order and to compute attributes. Based on a fixed traversal order, we provide inherited attributes for passing information down the AST (top-down processing) and synthesized attributes for passing information up the AST (bottom-up processing). Inherited attributes can be used to propagate context information along the edges of the AST, whereas synthesized attributes can be used to compute values based on the information of the subtree. One function for computing inherited attributes and one function for computing synthesized attributes must be implemented when attributes are used. We provide different interfaces that allow both, one, or no attribute to be used; in the latter case it is a simple traversal with a visit method called at each node.

The AST processing mechanism can be used to gather information about the AST, or to "query" the AST. Only the functions that are invoked by the AST processing mechanism need to be implemented by the user of AstProcessing classes; no traversal code must be implemented.

7.2 Common Interface of the Processing Classes

All five Ast*Processing classes provide three different functions for invoking a traversal on the AST:

T traverse(SgNode* node, ...): traverse full AST (including nodes that represent code from include files)

T traverseInputFiles(SgProject* projectNode, ...): traverse the subtree of the AST that represents the file(s) specified on the command line to a translator; files that are the *input* to the translator.

T traverseWithinFile(SgNode* node, ...): traverse only those nodes that represent code of the same file where the traversal started. The traversal stays *within* the file.

The return type `T` and the other parameters are discussed for each `Ast*Processing` class in the following sections.

Further, the following virtual methods can be defined by the user (the default implementations are empty):

void atTraversalStart(): called by the traversal code to signal to the processing class that a traversal is about to start

void atTraversalEnd(): called by the traversal code to signal that a traversal has terminated (all nodes have been visited)

As these methods are the same for all processing classes, they are not repeated in the class descriptions below.

7.3 AstSimpleProcessing

This class is called *Simple* because, in contrast to three of the other processing classes, it does not provide the computation of attributes. It implements a traversal of the AST and calls a visit function at each node of the AST. This can be done as a preorder or postorder traversal.

```
typedef {preorder,postorder} t_traversalOrder;

class AstSimpleProcessing {
   public:
      void traverse(SgNode* node, t_traversalOrder treeTraversalOrder);
      void traverseWithinFile(SgNode* node, t_traversalOrder treeTraversalOrder);
      void traverseInputFiles(SgProject* projectNode, t_traversalOrder treeTraversalOrder);
   protected:
      void virtual visit(SgNode* astNode)=0;
};
```

To use the class AstSimpleProcessing the user needs to implement the function `visit` for a user-defined class that inherits from class AstSimpleProcessing. To invoke a traversal, one of the three traverse functions needs to be called.

7.3.1 Example

In this example, we traverse the AST in preorder and print the name of each node in the order in which they are visited.

The following steps are necessary:

Interface: Create a class, *MyVisitor*, that inherits from `AstSimpleProcessing`.

Implementation: Implement the function `visit(SgNode* astNode)` for class *MyVisitor*.

Usage: Create an object of type *MyVisitor* and invoke the function `traverse (SgNode* node, t_traverseOrder treeTraversalOrder)`.

Figure 7.1 presents the interface.
Figure 7.2 presents the implementation.
Figure 7.3 presents the useage.

```
class MyVisitor : public AstSimpleProcessing {
 protected:
  void virtual visit(SgNode* astNode);
}
```

Figure 7.1: Headerfile *MyVisitor.h*.

```
#include "rose.h"
#include "MyVisitor.h"

MyVisitor::visit(SgNode* node) {
  cout << node->get_class_name() << endl;
}
```

Figure 7.2: Implementation file *MyVisitor.C*.

```
#include "rose.h"
#include "MyVisitor.h"

int main (int argc, char* argv[]) {
  SgProject* astNode=frontend(argc,argv);
  MyVisitor v;
  v.traverseInputFiles(astNode,preorder);
}
```

Figure 7.3: Example main program *MyVisitorMain.C*.

7.4 AstPrePostProcessing

The `AstPrePostProcessing` class is another traversal class that does not use attributes. In contrast to the `AstSimpleProcessing` class, which performs either a preorder or a postorder traversal, `AstPrePostProcessing` has both a preorder and a postorder component. Two different visit methods must be implemented, one of which is invoked in preorder (before the child nodes are visited), while the other is invoked in postorder (after all child nodes have been visited). This traversal is therefore well-suited for applications that require actions to be triggered when 'entering' or 'leaving' certain subtrees of the AST.

```
class AstPrePostProcessing {
   public:
      void traverse(SgNode* node);
      void traverseWithinFile(SgNode* node);
      void traverseInputFiles(SgProject* projectNode);
   protected:
      virtual void preOrderVisit(SgNode *node) = 0;
      virtual void postOrderVisit(SgNode *node) = 0;
};
```

The user needs to implement the `preOrderVisit` and `postOrderVisit` methods which are called before and after visiting child nodes, respectively.

7.5 AstTopDownProcessing

This class allows the user to use a restricted form of inherited attributes to be computed for the AST. The user needs to implement the function evaluateInheritedAttribute. This function is called for each node when the AST is traversed. The inherited attributes are restricted such that a single attribute of a parent node is inherited by all its child nodes (i.e., the return value computed by the function `evaluateInheritedValue` at the parent node is the input value to the function `evaluateInheritedValue` at all child nodes).

```
template<InheritedAttributeType>
class AstTopDownProcessing {
   public:
      void traverse(SgNode* node, InheritedAttributeType initialInheritedAttribute);
      void traverseWithinFile(SgNode* node, InheritedAttributeType initialInheritedAttribute);
      void traverseInputFiles(SgProject* projectNode, InheritedAttributeType initialInheritedAttribute);
   protected:
      InheritedAttributeType
      virtual evaluateInheritedAttribute(SgNode* astNode, InheritedAttributeType inheritedValue)=0;
      void virtual destroyInheritedValue(SgNode* astNode, InheritedAttributeType inheritedValue);
  };
```

The function `evaluateInheritedAttribute` is called at each node. The traversal is a preorder traversal.

In certain rare cases, the inherited attribute computed at a node may involve resources that must be freed; for instance, the attribute may be a pointer to dynamically-allocated memory that is no longer needed after the traversal of the child nodes has been completed. (Dynamically allocated attributes are only recommended for very large attributes where copying would be prohibitively expensive.) In such cases the `destroyInheritedValue` method may be implemented. This method is invoked with the inherited attribute computed at this node after all child nodes have been visited. It can free any resources necessary. An empty default implementation of this method is provided, so the method can be ignored if it is not needed.

```
class MyIndentLevel {
 public:
  MyIndentLevel():level(0) {
  }
  unsigned int level;
};

class MyIndenting : public AstTopDownProcessing<MyIndentLevel> {
 protected:
  void virtual evaluateInheritedAttribute(SgNode* astNode);
 private:
  unsigned int tabSize;
};
```

Figure 7.4: Headerfile *MyIndenting.h*.

7.5.1 Example

In this example, we traverse the AST and print the node names with proper indentation, according to the nesting level of C++ basic blocks. The function `evaluateInheritedAttribute` is implemented and an inherited attribute is used to compute the nesting level.

The following steps are necessary:

Interface: Create a class, *MyIndenting*, which inherits from AstTopDownProcessing, and a class `MyIndentLevel`. The latter will be used for attributes. Note that the constructor of the class `MyIndentLevel` initializes the attribute value.

Implementation: Implement the function `evaluateInheritedAttribute(SgNode* astNode)` for class `MyIndenting`.

Usage: Create an object of type *MyIndenting* and invoke the function traverse(SgNode* node, t_traverseOrder treeTraversalOrder);

Figure 7.4 presents the interface.

Figure 7.5 presents the implementation.

Figure 7.6 presents the useage.

Note that we could also use `unsigned int` as attribute type in this simple example. But in general, the use of objects as attributes is more flexible and necessary, if you need to compute more than one attribute value (in the same traversal).

7.6 AstBottomUpProcessing

This class allows to use synthesized attributes. The user needs to implement the function evaluateSynthesizedAttribute to compute from a list of synthesized attributes a single return value. Each element in the list is the result computed at one of the child nodes in the AST. The return value is the synthesized attribute value computed at this node and passed upwards in the AST.

```
#include "rose.h"
#include "MyIndenting.h"

MyIndenting::MyIndenting():tabSize(4) {
}
MyIndenting::MyIndenting(unsigned int ts):tabSize(ts) {
}

MyIndentLevel
MyIndenting::evaluateInheritedAttribute(SgNode* node, MyIndentLevel inh) {
  if(dynamic_cast<SgBasicBlock*>(node)) {
    inh.level=inh.level+1;
  }
  //printspaces(inh.level*tabSize);
  cout << node->get_class_name() << endl;
  return inh;
}
```

Figure 7.5: Implementation file *MyIndenting.C*.

```
#include "rose.h"
#include "MyVisitor.h"

int main (int argc, char* argv[]) {
  SgProject* astNode=frontend(argc,argv);
  MyVisitor v;
  v.traverseInputFiles(astNode,preorder);
}
```

Figure 7.6: Example main program *MyIndentingMain.C*.

```
template<SynthesizedAttributeType>
class AstBottomUpProcessing {
   public:
      SynthesizedAttributeType traverse(SgNode* node);
      SynthesizedAttributeType traverseWithinFile(SgNode* node);
      void traverseInputFiles(SgProject* projectNode);
      typedef ... SynthesizedAttributesList;
   protected:
      SynthesizedAttributeType
      virtual evaluateSynthesizedAttribute(SgNode* astNode, SynthesizedAttributesList synList)=0;
      SynthesizedAttributeType
      virtual defaultSynthesizedAttribute();
};
```

The type SynthesizedAttributesList is an opaque typedef that in most cases behaves like a Standard Template Library (STL) vector of objects of type SynthesizedAttributeType; in particular, it provides iterators and can be indexed like a vector. The main difference to vectors is that no operations for inserting or deleting elements or otherwise resizing the container are provided. These should not be necessary as the list of synthesized attributes is only meant to be read, not modified.

Using an iterator to operate on the list is necessary when the number of child nodes is arbitrary. For example, in a SgBasicBlock, the number of SgStatement nodes that are child nodes ranges from 0 to n, where n = synList.size(). For AST nodes with a fixed number of child nodes these values can be accessed by name, using enums defined for each AST node class. The naming scheme for attribute access is <CLASSNAME>_<MEMBERVARIABLENAME>.

The method defaultSynthesizedAttribute must be used to initialize attributes of primitive type (such as int, bool, etc.). This method is called when a synthesized attribute needs to be created for a non-existing subtree (i.e. when a node-pointer is null). A null pointer is never passed to an evaluate function. If a class is used to represent a synthesized attribute, this method does not need to be implemented because the default constructor is called. In order to define an default value for attributes of primitive type, this method must be used.

Two cases exist when a default value is used for a synthesized attribute and the defaultSynthesizedAttribute method is called:

- When the traversal encounters a null-pointer it will not call an evaluate method but instead calls default-SynthesizedAttribute.
- When the traversal skips over specific IR nodes. For example, traverseInputFiles() only calls the evaluate method on nodes which represent the input-file(s) but skips all other nodes (of header files for example).

7.6.1 Example: Access of Synthesized Attribute by Name

The enum definition used to access the synthesized attributes by name at a SgForStatement node is:

```
enum E_SgForStatement {SgForStatement_init_stmt, SgForStatement_test_expr_root, SgForStatement_increment_expr_root, SgForStatement_
```

The definitions of the enums for all AST nodes can be found in the generated file <COMPILETREE>/SAGE/Cxx_GrammarTreeTraversalAccessEnums.h.

For example, to access the synthesized attribute value of the SgForStatement's test-expression the synthesized attributes list is accessed using the enum definition for the test-expr. In the example we assign the pointer to a child node to a variable myTestExprSynValue:

```
SgNode* myTestExprSynValue=synList[SgForStatement_test_expr_root].node;
```

For each node with a fixed number of child nodes, the size of the synthesized attributes value list is always the same size, independent of whether the children exist or not. For example, for the SgForStatement it is always of size 4. If a child does not exist, the synthesized attribute value is the default value of the respective type used for the synthesized attribute (as template parameter).

7.7 AstTopDownBottomUpProcessing

This class combines all features from the two classes that were previously presented. It allows the user to use inherited and synthesized attributes. Therefore, the user needs to provide an implementation for two virtual functions, for evaluateInheritedAttribute and evaluateSynthesizedAttribute. The signature for evaluateSynthesizedAttribute has an inherited attribute as an additional parameter. This allows the results of inherited and synthesized attributes to be combined. You can use the inherited attribute that is computed at a node A by the evaluateInheritedAttribute method in the evaluateSynthesizedAttribute method at node A. But you cannot use synthesized attributes for computing inherited attributes (which is obvious from the method signatures). If such a data dependence needs to be represented, member variables of the traversal object can be used to *simulate* such a behavior to some degree. Essentially, this allows for the implementation of a pattern, also called *accumulation.* For example, building a list of all nodes of the AST can be implemented using this technique.

```
template<InheritedAttributeType, SynthesizedAttributeType>
class AstTopDownBottomUpProcessing {
   public:
      SynthesizedAttributeType traverse(SgNode* node,
            InheritedAttributeType initialInheritedAttribute);

      SynthesizedAttributeType traverseWithinFile(SgNode* node,
            InheritedAttributeType initialInheritedAttribute);

      void traverseInputFiles(SgProject* projectNode,
            InheritedAttributeType initialInheritedAttribute);

      typedef ... SynthesizedAttributesList;
   protected:
      InheritedAttributeType
      virtual evaluateInheritedAttribute(SgNode* astNode, InheritedAttributeType inheritedValue);
      SynthesizedAttributeType
      virtual evaluateSynthesizedAttribute(SgNode* astNode, InheritedAttributeType inh,
                       SynthesizedAttributesList synList)=0;
      SynthesizedAttributeType
      virtual defaultSynthesizedAttribute();
};
```

7.8 Combined Processing Classes

Running many read-only traversals on a single unchanged AST is an inefficient operation because every node is visited many times. ROSE therefore provides *combined* traversals that make it possible to run several traversals of the same base type in a single traversal, reducing the overhead considerably. Processing classes need not be adapted for use with the combined processing framework, so existing traversals can be reused; new traversals can be developed and tested independently and combined at any time.

To make sure that combined traversals work correctly, they should not change the AST or any other shared data. Terminal output from combined processing classes will be interleaved. No assumptions should be make about the order in which the individual traversals will be executed on any node.

For each `Ast*Processing` class there is a corresponding `AstCombined*Processing` class that behaves similarly. The interfaces for two of these classes are presented below, the others are analogous.

```
typedef {preorder,postorder} t_traversalOrder;

class AstCombinedSimpleProcessing {
   public:
      void traverse(SgNode* node, t_traversalOrder treeTraversalOrder);
      void traverseWithinFile(SgNode* node, t_traversalOrder treeTraversalOrder);
      void traverseInputFiles(SgProject* projectNode, t_traversalOrder treeTraversalOrder);

      void addTraversal(AstSimpleProcessing* traversal);
      vector<AstSimpleProcessing*>& get_traversalPtrListRef();
};


template<InheritedAttributeType, SynthesizedAttributeType>
class AstCombinedTopDownBottomUpProcessing {
   public:
      vector<SynthesizedAttributeType>
        traverse(SgNode* node, vector<InheritedAttributeType> initialInheritedAttributes);

      vector<SynthesizedAttributeType>
         traverseWithinFile(SgNode* node, vector<InheritedAttributeType> initialInheritedAttributes);

     void traverseInputFiles(SgProject* projectNode, vector<InheritedAttributeType> initialInheritedAttributes);

     typedef ... SynthesizedAttributesList;

      void addTraversal(AstTopDownBottomUpProcessing<InheritedAttributeType, SynthesizedAttributeType>* traversal);
      vector<AstTopDownBottomUpProcessing<InheritedAttributeType, SynthesizedAttributeType>*>
        & get_traversalPtrListRef();
};
```

Note that these classes do not contain virtual functions for the user to override. They are meant to be used through explicit instances, not as base classes. Instead of calling one of the traverse methods on the individual processing classes, they are combined within an instance of the `AstCombined*Processing` class and started collectively using one of its traverse methods. Inherited and synthesized attributes are passed in and back through STL vectors.

Two methods for managing the list of traversals are provided: The `addTraversal` method simply adds the given traversal to its list, while `get_traversalPtrListRef` returns a reference to its internal list that allows any other operations such as insertion using iterators, deletion of elements, etc.

7.9 AST Node Attributes

To each node in the AST user-defined attributes can have an attribute attached to it *by name* (by defining a unique name, string, for the attribute). The user needs to implement a class that inherits from `AstAttribute`. Instances of this class can be attached to an AST node by using member functions of `SgNode::attribute`.

Example: let node be a pointer to an object of type SgNode:

```
int main() {
  int n=10;
  while(n>0) {
    n=n-1;
  }
  return n;
}
```

Figure 7.7: Example program used as running example

```
class MyAstAttribute : public AstAttribute {
   public:
   MyAstAttribute(int v):value(v) {}
           virtual AstAttribute* copy() const override {
               return new MyAstAttribute(*this);
           }
           virtual std::string attribute_class_name() const override {
               return "MyAstAttribute";
           }
      ...
   private:
      int value;
      ...
};

node->attribute.setAttribute("mynewattribute",new MyAstAttribute(5));
```

Using this expression, an attribute with name `mynewattribute` can be attached to the AST node pointed to by node. Similarly, the same attribute can be accessed *by name* using the member function getAttribute:

```
MyAstAttribute* myattribute=node->attribute.getAttribute("mynewattribute");
```

AST attributes can be used to combine the results of different processing phases. Different traversals that are performed in sequence can store and read results to and from each node of the AST. For example, the first traversal may attach its results for each node as attributes to the AST, and the second traversal can read and use these results.

7.10 Visualization

7.10.1 Example Graphs

The graph shown in figure 7.8 is the AST of the program in figure 7.7. Such an output can be generated for an AST with:

```
AstDOTGeneration dotgen;
dotgen.generateInputFiles(projectNode, AstDOTGeneration::PREORDER);
```

where `projectNode` is a node of type `SgProjectNode` and the order in which the AST is traversed is specified to be `AstDOTGeneration::PREORDER` (or `AstDOTGeneration::POSTORDER`).

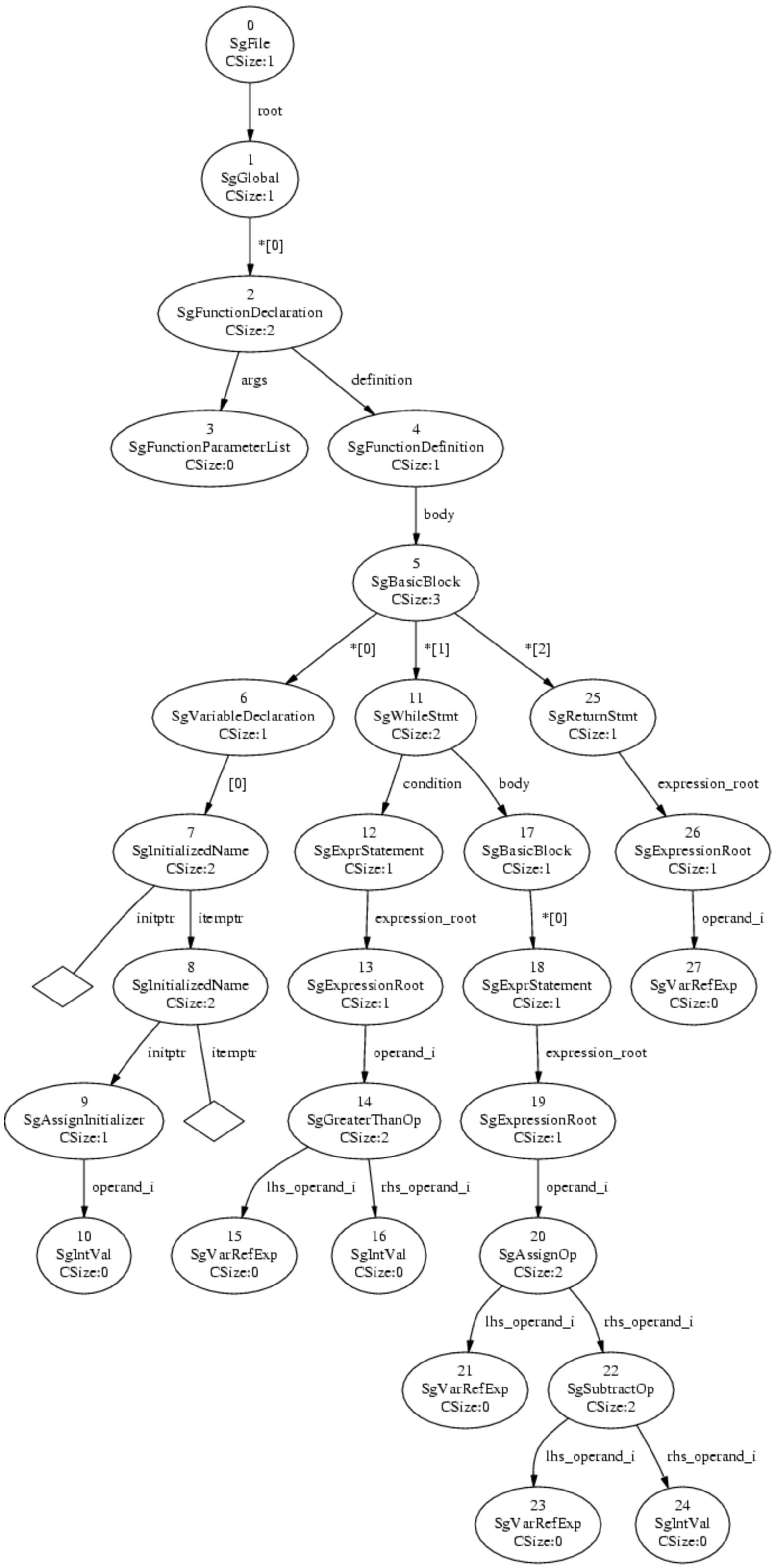

Figure 7.8: Numbers at nodes show the order in which the visit function is called in a preorder traversal

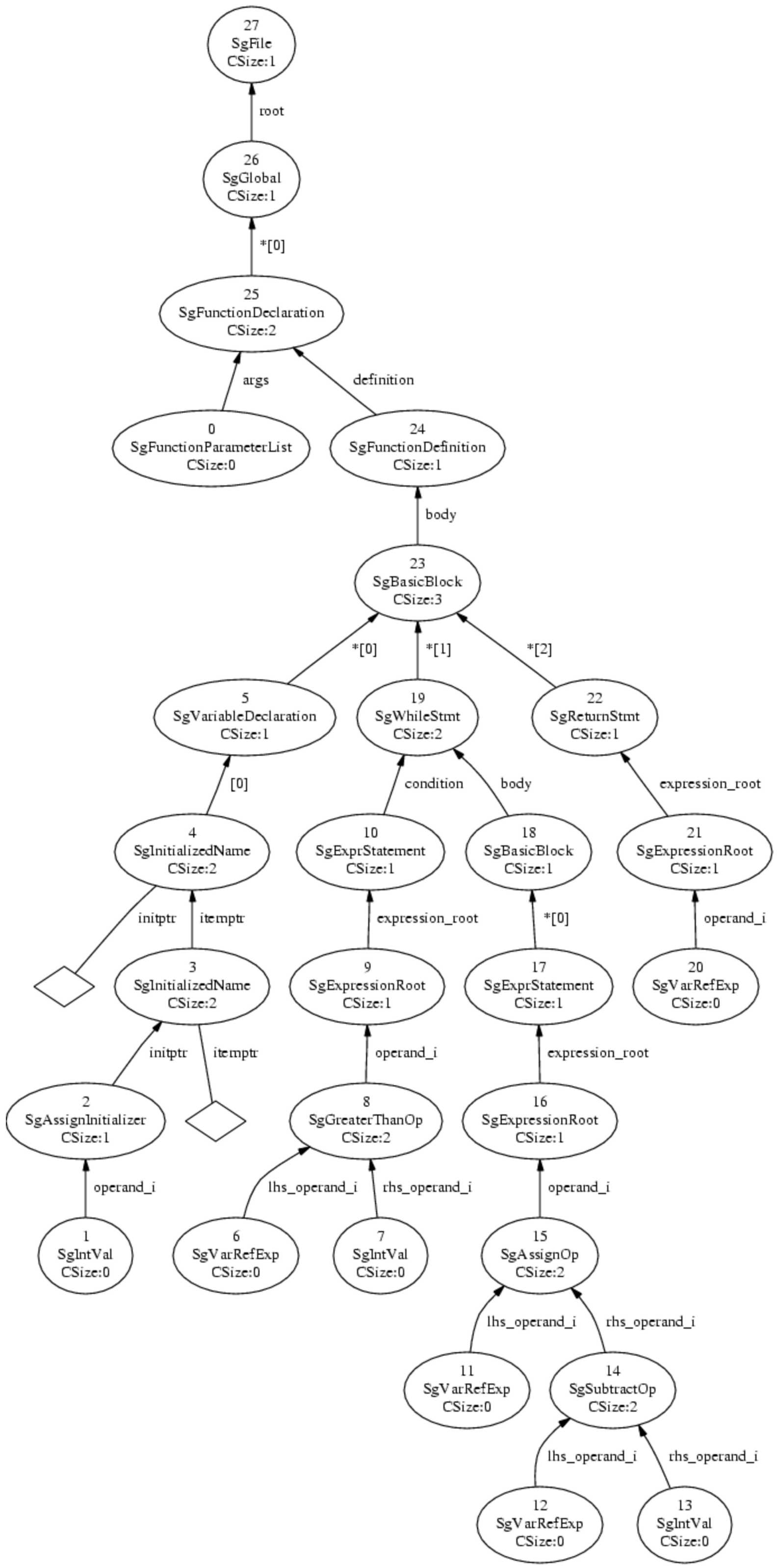

Figure 7.9: Numbers at nodes show the order in which the visit function is called in a postorder traversal

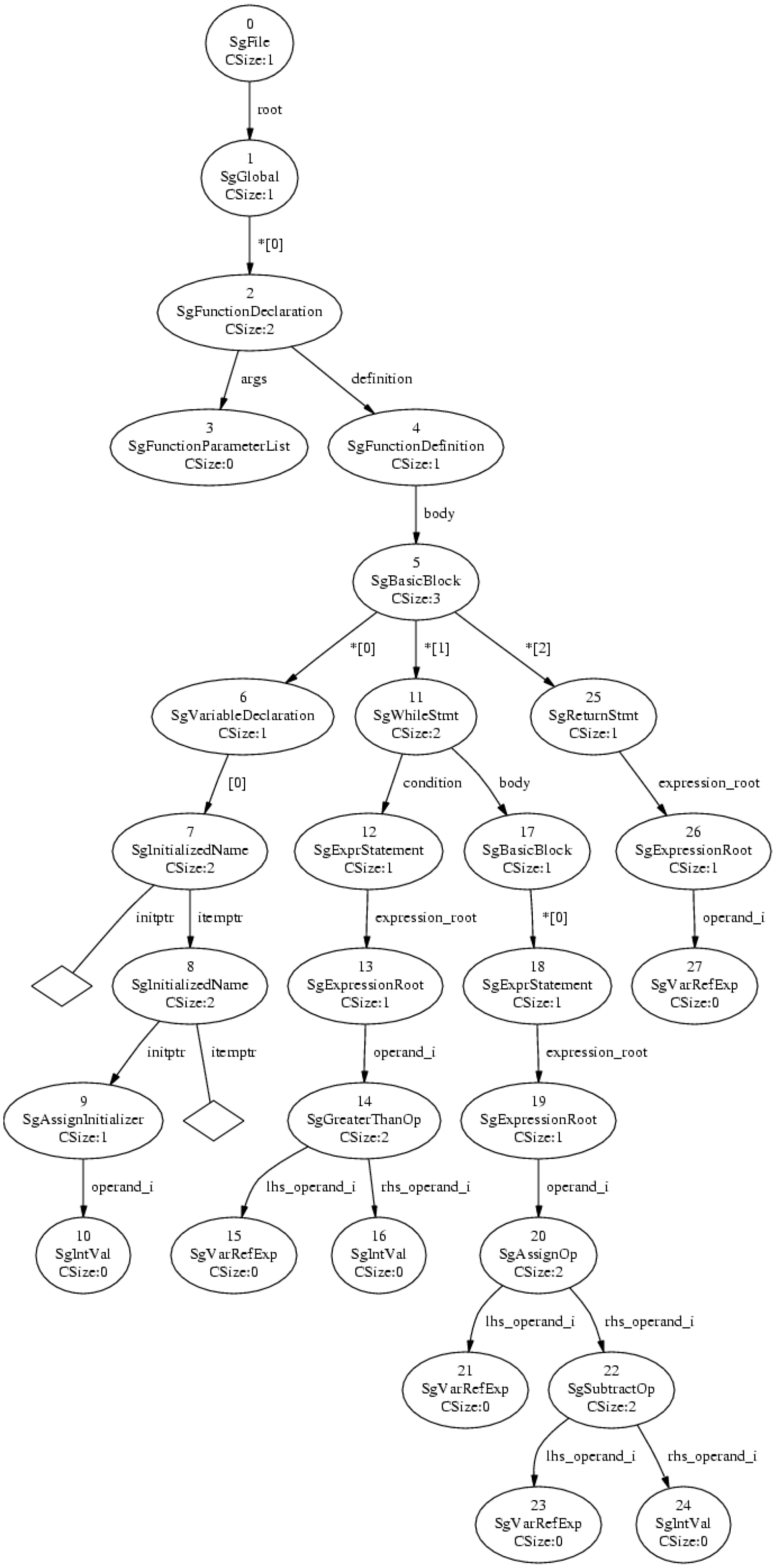

Figure 7.10: Numbers at nodes show the order in which the function evaluateInheritedAttribute is called in a top-down processing

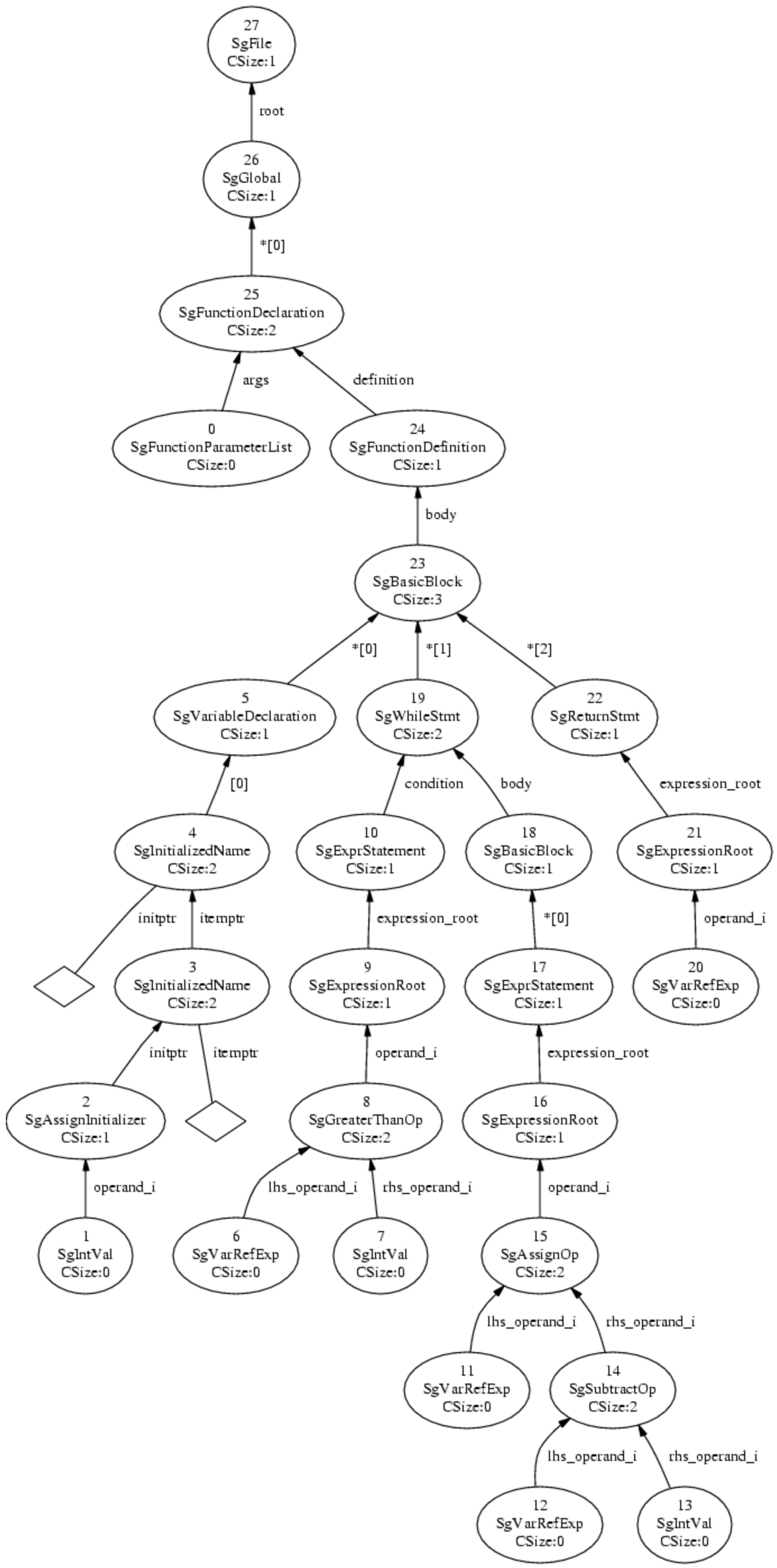

Figure 7.11: Numbers at nodes show the order in which the function evaluateSynthesizedAttribute is called in a bottom up processing

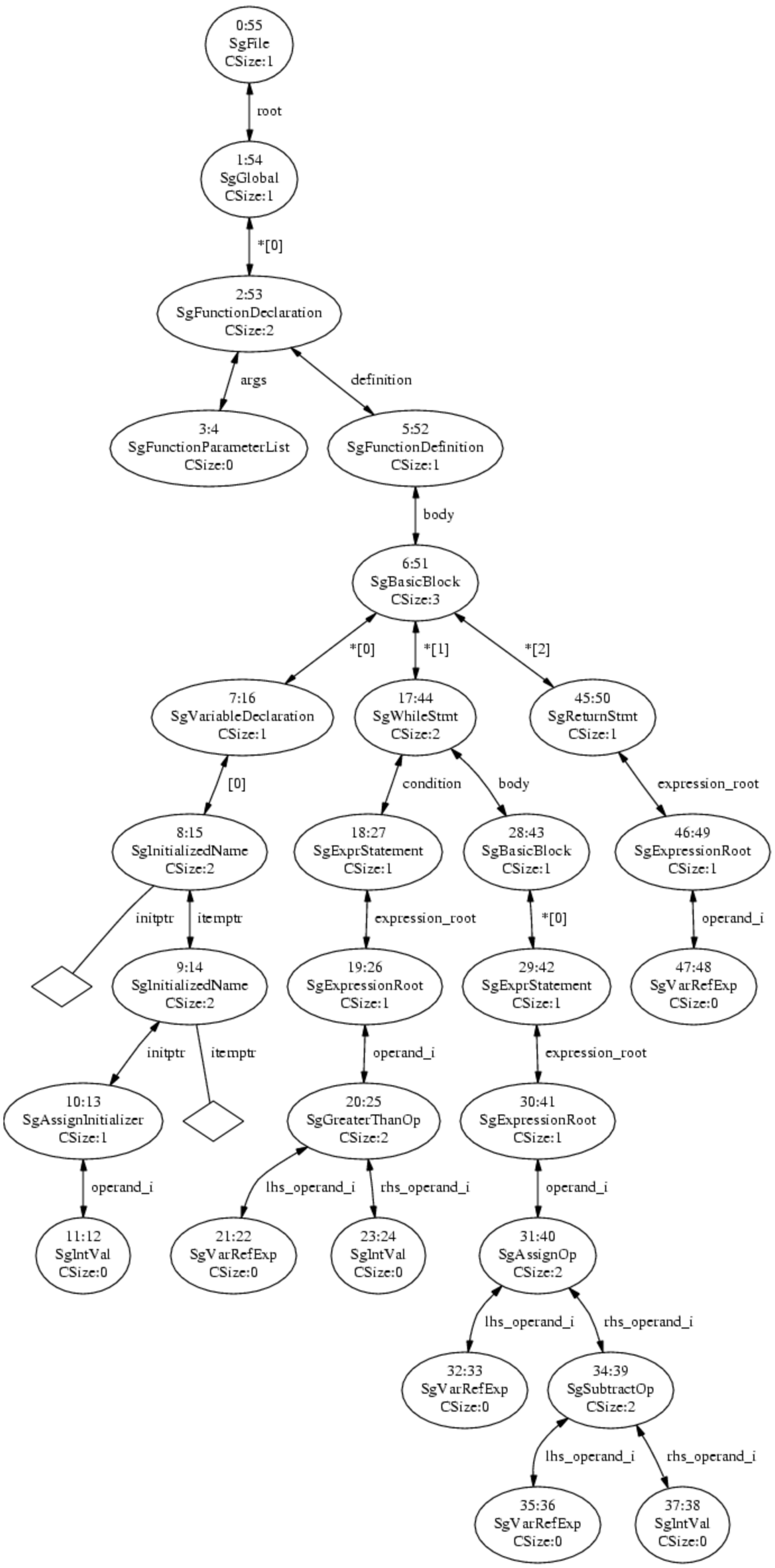

Figure 7.12: The pair of numbers at nodes shows the order in which the function evaluateInheritedAttribute (first number) and evaluateSynthesizedAttribute (second number) is called in a top-down-bottom-up processing.

7.11 Conclusions

All AST*Processing classes provide similar interfaces that differ only by the attributes used. AST node attributes can be used to attach data to each AST node and to share information between different traversals.

Additional examples for traversal, attributes, pdf, and dot output can be found in

- `ROSE/exampleTranslators/documentedExamples/astProcessingExamples`.

Table 7.1 summaries and compares the traversal and query support in ROSE. It shows the types of SgNode visited, context, performance, order of nodes being visited, source files being visited, and suggested use for each of the choices.

Regular AST Traversal						
	Node Types	**Context**	**Performance**	**Order**	**File Visited**	**Suggested Use**
AstSimpleProcessing		No		pre or post order		simple analysis
AstTopDownProcessing		attributes from parent nodes		preorder		inherited attributes
AstBottomUpProcessing	a predefined	attributes from child nodes	regular	postorder	configurable	synthesized attribute
AstTopDownBottomUpProcessing	subset	attributes from both parent and children		pre and post order		mixed inherited and synthesized attributes
AstPrePostProcessing		No		pre and post order		mixed topdown and bottomup w/o attributes
Combined AST Traversal						
AstCombinedSimpleProcessing	Similar to the corresponding traversals above but fuse multiple traversals into one					
AstCombinedTopDownProcessing						
AstcombinedBottomUpProcessing						
AstCombinedTopDownBottomUpProcessing						
AstCombinedPrePostProcessing						
Reverse AST Traversal						
AstReversePrefixSimpleProcessing	????					
AstReversePrefixInhProcessing						
AstReversePrefixSynProcessing						
AstReversePrefixInhSynProcessing						
AstReverseBranchSimpleProcessing	???					
AstReverseBranchInhProcessing						
AstReverseBranchSynProcessing						
AstReverseBranchInhSynProcessing						
Parallel Traversal						
AstSharedMemoryParallelSimpleProcessing	Similar to AstCombined*Processing if traverse() is called, but do parallel traversal via Pthreads if traverseInParallel() is called.					
AstSharedMemoryParallelTopDownProcessing						
AstSharedMemoryParallelBottomUpProcessing						
AstSharedMemoryParallelTopDownBottomUpProcessing						
AstSharedMemoryParallelPrePostProcessing						
AstSharedMemoryParallelizableSimpleProcessing	Internal use only, representing a traversal that that can run in parallel without interfering others.					
AstSharedMemoryParallelizableTopDownProcessing						
AstSharedMemoryParallelizableBottomUpProcessing						
AstSharedMemoryParallelizableTopDownBottomUpProcessing						
AstSharedMemoryParallelizablePrePostProcessing						
Memory Pool Traversal						
ROSE_VisitTraversal	all	No	better	sequential	all	simple analysis
ROSE_VisitorPattern						classic support
AST Query						
NodeQuery::querySubTree	subset	No	regular	pre order?	??	simple analysis

Table 7.1: Summary and Comparison of AST Traversals and Queries

Chapter 8

AST Rewrite Mechanism

WARNING: AST Rewrite Mechanism is not robust. Please use SageInterface functions instead! See ROSE tutorial's AST construction Chapter for details.

The Abstract Syntax Tree (AST) Rewrite Mechanism permits modifications to the AST. To effect changes to the input source code, modifications to the AST are done by a ROSE translator; and new version of the source code is produced. Although analysis is possible by only reading the AST, transformations (and changes in the output code from the input code) can only be accomplished by rewriting portions of the AST. The AST is the single intermediate form manipulated by the preprocessor. All changes are eventually output as modifications to the input source code after being processed through the intermediate form.

The material in this chapter builds on material presented in the previous two chapters; Writing a Source-to-Source Preprocessor (chapter **??**) and AST Processing (chapter 7). This chapter presents the required AST Rewrite Traversal and the simple interface functions to the `AST_Rewrite` class. A section is included that demonstrates code that rewrites the AST for any input code. More complex examples are possible but each uses the AST Rewrite Mechanism in a similar way. The ROSE Tutorial documents a few more interesting examples.

8.1 Introduction

The rewrite mechanism in ROSE contains four different levels of interface within its design. Table 8.1 shows the different levels of the interface design for the ROSE rewrite mechanism. Each level consists of simple tree editing operations (`insert()`, `replace()`, and `remove()`) that can operate on statements within the AST.

8.2 Multiple Interfaces to Rewrite Mechanism

There are four different levels of interfaces in the rewrite mechanism because there are many different program transformations requirements. Each level builds on the lower level, and the highest level interface is the most sophisticated internally. Each interface has only three functions: `insert()`, `replace()`, and `remove()`.

8.2.1 SAGE III Rewrite Interface

This lowest possible level of interface is implemented as member functions on the SgNode objects. It is used internally to implement the higher level interfaces (including the Low Level Rewrite Interface. Uniformly, oper-

Relative Positioning (contains state)	String-Based	High Level Interface (level 4)	insert(SgNode*,string,scope,location) replace(SgNode*,string,scope,location) remove(SgNode*)
Absolute Positioning (contains no state)	String-Based	Mid Level Interface (level 3)	insert(SgNode*,string,location) replace(SgNode*,string,location) remove(SgNode*)
	SgNode*	Low Level Interface (level 2)	insert(SgNode*,SgNode*) replace() remove(SgNode*)
		SAGE III Interface (level 1)	insert(SgNode*,SgNode*) replace(SgNode*,SgNode*) remove(SgNode*)

Table 8.1: Different levels of the ROSE Rewrite mechanism.

ations of `insert()`, `replace()`, and `remove()` apply only to SAGE III objects representing containers (SAGE III objects that have containers internally, such as SgGlobal, SgBasicBlock, etc.). Strings cannot be specified at this level of interface; only subtrees of the AST may be specified. New AST fragments must be built separately and may be inserted or used to replace existing AST subtrees in the AST. Operations using this interface have the following properties:

- Operations performed on collections only.
- Operations are immediate executed.
- Operations are local on the specified node of the AST.
- Operations do not take attached comments or preprocessor directives into account.
  This can lead to unexpected results (e.g. removing or moving `#include` directives by accident).

8.2.2 Low Level Rewrite Interface

This interface is similar to the SAGE III Rewrite Interface except that operations are performed on any statement and not on the containers that store the statement lists. The domain of the operations – on the statements instead of on the parent nodes of the statements – is the most significant difference between the two interfaces. An additional feature includes support for repositioning attached comments/directives from removed nodes to their surrounding nodes to preserve them within `replace()` and `remove()` operations. Additional support is provided for marking inserted statements as transformations within the Sg_File_Info objects. Operations using this interface have the following properties:

- Attached comments/directives are relocated.
- Inserted AST fragments are marked within the Sg_File_Info objects.
- Operations are immediate.
- Operations are local on the specified node of the AST.

8.2.3 Mid Level Rewrite Interface

This interface builds on the low-level interface and adds the string interface, which permits simpler specification of transformations. Operations using this interface have the following properties:

- Strings used to specify transformations.
- Operations are immediate.
- Operations are local on the specified node of the AST.

8.2.4 High Level Rewrite Interface

This interface presents the same string based rewrite mechanism as the mid-level interface but adds additional capabilities. This interface is the most flexible rewrite interface within ROSE. Although it must be used within a traversal to operate on the AST, it provides a mechanism to express more sophisticated transformations with less complexity due to its support of relative positioning of transformation strings within the AST (relative to the current node within a traversal).

The high-level rewrite mechanism uses the same three functions as the other rewrite interfaces, but with an expanded range of enum values to specify the intended scope and the location in that scope. The scope is specified using the ScopeIdentifierEnum type defined in the HighLevelCollectionTypedefs class. These enum values are:

- unknownScope
- LocalScope
- ParentScope
- NestedLoopScope
- NestedConditionalScope
- FunctionScope
- FileScope
- GlobalScope
- Preamble

The position in any scope is specified by the PlacementPositionEnum type, which is defined in the HighLevelCollectionTypedefs class. These enum values are:

- PreamblePositionInScope
- TopOfScope
- TopOfIncludeRegion
- BottomOfIncludeRegion
- BeforeCurrentPosition
- ReplaceCurrentPosition
- AfterCurrentPosition
- BottomOfScope

Function prototypes of interface functions:

```
void insert (SgNode*, string ,HighLevelCollectionTypedefs::ScopeIdentifierEnum,HighLevelCollectionTypedefs::PlacementPositionEnum);
```

Example of how to use specific insertion of transformation into the AST (required traversal not shown):

```
insert (astNode, ``int x;'' ,HighLevelCollectionTypedefs::FunctionScope,HighLevelCollectionTypedefs::TopOfScope);
```

Operations using this interface have the following properties:

- Adds relative positioning to the specification of transformations.
- Requires traversal for operation on the AST.
- Operations are delayed and occur durring the required traversal, all operations are completed by the end of the traversal.
- Operations occur on AST nodes along a path defined by the chain from the current input node to the operator to the root node of the AST (SgProject).

8.2.5 Advantages and Disadvantages of Rewrite Interfaces

Each interface builds upon the lower level interfaces and each has some advantages and disadvantages. Table 8.2 lists the major features and requirements associated with each. The high-level interface (Level 4) presents the most sophisticated features, but only works as part of a traversal of the AST. The mid-level interface is the lowest level interface that permits the specification of transformations as strings. The low-level interface is useful when AST fragments are built directly using the SAGE III classes through their constructors (a somewhat tedious process). The low level interface preserves the original interfaces adopted from SAGE II.

Interface:Features	Contains State	Positioning	String	Traversal
Level 1	No State	Absolute	AST Subtree	Not Used
Level 2	No State	Absolute	AST Subtree	Not Used
Level 3	No State	Absolute	String	Not Used
Level 4	State	Relative	String	Required

Table 8.2: Advantages and disadvantages of different level interfaces within the ROSE Rewrite Mechanism.

8.3 Generation of Input for Transformation Operators

Providing operators to `insert()`, `replace()`, `remove()` solves only part of the problem of simplifying transformations. The other part of the problem is generating the input to the transformation operators. Both `insert()` and `replace()` require input, either as an AST fragment or as a string containing source code. This section presents the pros and cons of the specification of transformations as strings.

8.3.1 Use of Strings to Specify Transformations

The mid-level and high-level rewrite interfaces introduce the use of strings to specify transformations. Using strings to specify transformations attempts to define a simple mechanism for a non-compiler audience to express moderately complex transformations. The alternative is to build the AST fragments to be inserted directly using SAGE III and the constructors for its objects. In general, the direct construction of AST fragments is exceedingly tedious, and while aspects can be automated, the most extreme example of this automation is the AST constructions from source code strings. A disadvantage is that the generation of the AST fragment from strings is slower, but it is only a compile-time issue.

8.3.2 Using SAGE III Directly to Specify Transformations

It is possible to build AST fragments directly using SAGE III and insert these into the AST. This alternative to the use of strings is more complex and is only briefly referenced in this section.

The constructors for each of the SAGE III objects form the user interface required to build up the AST fragments. The documentation for these appear in the reference chapter of this manual.

A few notes:

1. Use the `Sg_File_Info* Sg_File_Info::generateDefaultFileInfoForTransformationNode();` static member function to generate the Sg_File_Info object required for each of the constructor calls. This marks the IR nodes as being part of a transformation and signals that they should be output within code generation (unparsing).

8.4 AST Rewrite Traversal of the High-Level Interface

The AST Rewrite Mechanism uses a traversal of the AST, similar to the design of a traversal using the AST Processing (Chapter 7) part of ROSE. The example code **??** specifically shows an **AstTopDownBottomUp-Processing** 7.7 traversal of the AST. Using conditional compilation, the example code shows the somewhat trivial changes required to convert a read-only AST traversal into a read-write AST rewrite operation. In this example the AST traversal is converted to be ready for rewrite operations, but no rewrite operations are shown. The purpose of this example is only to show the modifications to an existing traversal that are required to use the AST rewrite mechanism.

The specialized AST rewrite traversal is internally derived from the ASTProcessing **TopDownBottomUp** traversal (processing) but adds additional operations in recording the local context of source code position (in the inherited attribute) and performs additional operations on the way back up the AST (on the synthesized attribute).

```
#include "rose.h"
#include "rewrite.h"

// Extra headers for customizing the rewrite mechanism
#include "rewriteTreeTraversalImpl.h"
#include "rewriteSynthesizedAttributeTemplatesImpl.h"
#include "ASTFragmentCollectorTraversalImpl.h"
#include "prefixGenerationImpl.h"
#include "rewriteASTFragementStringTemplatesImpl.h"
#include "nodeCollectionTemplatesImpl.h"
#include "rewriteDebuggingSupportTemplatesImpl.h"

// Use new rewrite mechanism
#define USE_REWRITE_MECHANISM 1

// Notice that only the names of the evaluate functions change
// along with the derivation of the attributes from an AST_Rewrite nested class
#if USE_REWRITE_MECHANISM
```

```
#define EVALUATE_INHERITED_ATTRIBUTE_FUNCTION evaluateRewriteInheritedAttribute
#define EVALUATE_SYNTHESIZED_ATTRIBUTE_FUNCTION evaluateRewriteSynthesizedAttribute
#else
#define EVALUATE_INHERITED_ATTRIBUTE_FUNCTION evaluateInheritedAttribute
#define EVALUATE_SYNTHESIZED_ATTRIBUTE_FUNCTION evaluateSynthesizedAttribute
#endif

// Build an inherited attribute for the tree traversal to test the rewrite mechanism
class MyInheritedAttribute
   {
     public:
       // Note that any constructor is allowed
          MyInheritedAttribute () {};
   };

// Build a synthesized attribute for the tree traversal to test the rewrite mechanism
class MySynthesizedAttribute
#if USE_REWRITE_MECHANISM
   : public HighLevelRewrite::SynthesizedAttribute
#endif
   {
     public:
          MySynthesizedAttribute() {};
   };

// tree traversal to test the rewrite mechanism
#if USE_REWRITE_MECHANISM
 /*! A specific AST processing class is used (built from SgTopDownBottomUpProcessing)
  */
class MyTraversal
   : public HighLevelRewrite::RewriteTreeTraversal<MyInheritedAttribute,MySynthesizedAttribute>
#else
 /*! Any AST processing class may be used but the conversion
     is trivial if SgTopDownBottomUpProcessing is used.
  */
class MyTraversal
   : public SgTopDownBottomUpProcessing<MyInheritedAttribute,MySynthesizedAttribute>
#endif
   {
     public:
          MyTraversal () {};

       // Functions required by the tree traversal mechanism
          MyInheritedAttribute EVALUATE_INHERITED_ATTRIBUTE_FUNCTION (
             SgNode* astNode,
             MyInheritedAttribute inheritedAttribute );

          MySynthesizedAttribute EVALUATE_SYNTHESIZED_ATTRIBUTE_FUNCTION (
             SgNode* astNode,
             MyInheritedAttribute inheritedAttribute,
             SubTreeSynthesizedAttributes synthesizedAttributeList );
   };

// Functions required by the tree traversal mechanism
MyInheritedAttribute
MyTraversal::EVALUATE_INHERITED_ATTRIBUTE_FUNCTION (
     SgNode* astNode,
```

```
      MyInheritedAttribute inheritedAttribute )
   {
  // Note that any constructor will do
      MyInheritedAttribute returnAttribute;

      return returnAttribute;
   }

MySynthesizedAttribute
MyTraversal::EVALUATE_SYNTHESIZED_ATTRIBUTE_FUNCTION (
      SgNode* astNode,
      MyInheritedAttribute inheritedAttribute,
      SubTreeSynthesizedAttributes synthesizedAttributeList )
   {
  // Note that any constructor will do
      MySynthesizedAttribute returnAttribute;

      return returnAttribute;
   }

int
main ( int argc, char** argv )
   {
  // Main Function for default example ROSE Preprocessor
  // This is an example of a preprocessor that can be built with ROSE
  // This example can be used to test the ROSE infrastructure

  // DQ (4/6/2017): This will not fail if we skip calling ROSE_INITIALIZE (but
  // any warning message using the message looging feature in ROSE will fail).
      ROSE_INITIALIZE;

      SgProject* project = frontend(argc,argv);

      MyTraversal treeTraversal;

      MyInheritedAttribute inheritedAttribute;

  // Ignore the return value since we don't need it
      treeTraversal.traverseInputFiles(project,inheritedAttribute);

      return backend(project);
   }
```

FIXME: *This should that fits onto a*

This example shows the setup required to use the AST Rewrite Mechanism. The next section shows how to add new code to the AST. The `main()` function is as in example of how to use a traversal (see chapter **??**).

Note that the differences between the traversal required for use with the AST Rewrite Mechanism is different from the traversals associated with 7.7. The exact differences are enabled and disabled in the example 8.4 by setting the macro `USE_REWRITE_MECHANISM` to zero (0) or one (1).

The differences between traversals using `AstTopDownBottomUpProcessing<InheritedAttrbute,SynthesizedAttribute>` and traversals using the AST Rewrite Mechanism (`AST_Rewrite::RewriteTreeTraversal<InheritedAttrbute,SynthesizedAttrib` are both required to use the AST Rewrite Mechanism. They are:

1. InheritedAttributes must derive from `AST_Rewrite::InheritedAttribute`.
2. Must define constructor `InheritedAttribute::InheritedAttribute(SgNode* astNode)`.
3. Must define copy constructor:

InheritedAttribute::InheritedAttribute(const InheritedAttribute & X, SgNode* astNode).

4. SynthesizedAttribute must derive from AST_Rewrite::SynthesizedAttribute
5. Must derive new traversal from
AST_Rewrite::RewriteTreeTraversal<InheritedAttrbute,SynthesizedAttribute>) instead of
AstTopDownBottomUpProcessing<InheritedAttrbute,SynthesizedAttribute>.

8.5 Examples

This section presents several examples using the different interfaces to specify simple transformations.

8.5.1 String Specification of Source Code

Both the mid-level and high-level interfaces use strings to specify source code. The examples below show how to specify the strings.

Specification of Source Code

Specification of source code is straight forward. However, quoted strings must be escaped and strings spanning more then one line must use the string continuation character ("\").

- MiddleLevelRewrite::insert(statement,"int newVariable;",locationInScope);
- MiddleLevelRewrite::insert(statement,"timer(\"functionName\");",locationInScope);
- MiddleLevelRewrite::insert(statement,
"/* Starting Comment */ \n \
int y; int y; for (y=0; y < 10; y++)z = 1; \n \
/* Ending Comment */\n",locationInScope);

Specification of CPP Directives

Specification of CPP directives as strings is as one would expect except that where quotes ("") appear in the string they must be escaped (\"\") to remain persistent in the input string.

- MiddleLevelRewrite::insert(statement,"#define TEST",locationInScope);
- MiddleLevelRewrite::insert(statement,"#include<foo.h>",locationInScope);
- MiddleLevelRewrite::insert(statement,"#include \"foo.h\"",locationInScope);

Specification of Comments

Specification of comments are similar.

- MiddleLevelRewrite::insert(statement,"/* C style comment test */",locationInScope);
- MiddleLevelRewrite::insert(statement,"// C++ comment test ",locationInScope);

Specification of Macros

The specification of macros is similar to CPP directives except that longer macros often have line continuation and formatting. We show how to preserve this in the example macro definition below. Transformation involving the use of a macro is more complex if the macro call is to be preserved in the final transformation (left unexpanded in the generation of the AST fragment with the rewrite mechanism).

Macro Definition: A macro definition is similar to a CPP directive. The long example is taken from the Tuning Analysis Utilities (TAU) project which instruments code with similar macro calls.

- `MiddleLevelRewrite::insert(statement,"#include<foo.h>",locationInScope);`
- `MiddleLevelRewrite::insert(statement,"#include \"foo.h\"",locationInScope);`
- `MiddleLevelRewrite::insert(statement,"#define PRINT_MACRO(name) name;",locationInScope);`
-
```
MiddleLevelRewrite::insert(statement,
"\n\
#ifdef USE_ROSE\n\
// If using a translator built using ROSE process the simpler tauProtos.h header \n\
// file instead of the more complex TAU.h header file (until ROSE is more robust) \n\
#include \"tauProtos.h\"\n\n
// This macro definition could be placed into the tauProtos.h header file \n\
#define TAU_PROFILE(name, type, group) \\\n\
static TauGroup_t tau_gr = group; \\\n\
static FunctionInfo tauFI(name, type, tau_gr, #group); \\\n\
Profiler tauFP(&tauFI, tau_gr); \n\
#else\n\
#include \"TAU.h\"\n\
#endif"\\
,locationInScope);
```

Macro Use: This example of macro use shows how to leave the macro unexpanded in the AST fragment (which is generated to be patched into the application's AST).

-
```
MiddleLevelRewrite::insert(statement,
MiddleLevelRewrite::postponeMacroExpansion("PRINT_MACRO(\"Hello World!\")"),locationInScope);
```

-
```
MiddleLevelRewrite::insert(statement,
MiddleLevelRewrite::postponeMacroExpansion("TAU_PROFILE(\"main\",
\"\",TAU_USER)"),locationInScope);
```

8.6 Example Using AST Rewrite

This section demonstrates a simple example using the AST Rewrite Mechanism. The input code 8.6 contains the variable declaration statement `int x;` which example preprocessor `testRewritePermutations` (a testcode in the **ROSE/tests/nonsmoke/functional/roseTests/astRewriteTests** directory) will use to place additional variable declarations in all possible relative/absolute positions.

```
#include<stdio.h>

int main()
   {
     for (int i=0; i < 1; i++)
        int x;
     return 0;
   }
```

The new variable declarations contain, as a substring of the variable name, the relative scope and location in that scope (relative to the target declaration `int x;`. The output of processing this input file is a new code 8.6 with many added declarations, one for each possible relative/absolute position possible (relative to the declaration: `int x;`).

```
int y_GlobalScope_TopOfIncludeRegion;
int y_FileScope_TopOfIncludeRegion;
int y_FunctionScope_TopOfIncludeRegion;
int y_NestedConditionalScope_TopOfIncludeRegion;
int y_NestedLoopScope_TopOfIncludeRegion;
int y_ParentScope_TopOfIncludeRegion;
int y_LocalScope_TopOfIncludeRegion;

#include<stdio.h>

int y_LocalScope_BottomOfIncludeRegion;
int y_ParentScope_BottomOfIncludeRegion;
int y_NestedLoopScope_BottomOfIncludeRegion;
int y_NestedConditionalScope_BottomOfIncludeRegion;
int y_FunctionScope_BottomOfIncludeRegion;
int y_FileScope_TopOfScope;
int y_FileScope_BottomOfIncludeRegion;
int y_FileScope_BeforeCurrentPosition;
int y_GlobalScope_TopOfScope;
int y_GlobalScope_BottomOfIncludeRegion;
int y_GlobalScope_BeforeCurrentPosition;

int main()
   {
     int y_FunctionScope_TopOfScope;
     int y_NestedConditionalScope_TopOfScope;
     int y_NestedLoopScope_TopOfScope;
     int y_ParentScope_TopOfScope;
     for (int i = 0; i < 1; i++)
        {
          int y_LocalScope_TopOfScope;
          int y_LocalScope_BeforeCurrentPosition;

          int x;

          int y_LocalScope_AfterCurrentPosition;
          int y_LocalScope_BottomOfScope;
        }
     int y_ParentScope_BottomOfScope;
     int y_NestedLoopScope_BottomOfScope;
     int y_NestedConditionalScope_BottomOfScope;
```

```
     int y_FunctionScope_BottomOfScope;

     return 0;
   }

int y_FileScope_AfterCurrentPosition;
int y_FileScope_BottomOfScope;
int y_GlobalScope_AfterCurrentPosition;
int y_GlobalScope_BottomOfScope;
```

8.7 Limitations (Known Bugs)

There are several types of statements the AST rewrite mechanism can not currently process. This section enumerates these and explains why each is difficult or not currently possible. Note that some appear unable to be handled, while others will only require special handling that is not yet implemented.

1. Why we have to skip SgCaseOptionStmt statements.
   Example of code in generated intermediate file for a SgCaseOptionStmt:

   ```
   int GlobalScopePreambleStart;
   int GlobalScopePreambleEnd;
   int CurrentLocationTopOfScopeStart;
   int CurrentLocationTopOfScopeEnd;
   int CurrentLocationBeforeStart;
   int CurrentLocationBeforeEnd;
   int CurrentLocationReplaceStart;
   case 0:{y++;break;}
   int CurrentLocationReplaceEnd;
   int CurrentLocationAfterStart;
   int CurrentLocationAfterEnd;
   int CurrentLocationBottomOfScopeStart;
   int CurrentLocationBottomOfScopeEnd;
   ```

   The problem is that marker declarations that appear after the SgCaseOptionStmt are included in the scope of the SgCaseOptionStmt while those that appear before it are not in the same scope.
2. SgDefaultOptionStmt (see reason #1 above).
3. SgCtorInitializerList
   This case would require special handling to be generated in the intermediate file, and it would require special handling isolated from the AST. This case can probably be handled in the future with extra work.
4. SgFunctionParameterList (see reason #3 above).
5. SgClassDefinition
   Since the SgClassDefinition is so structurally tied to the SgClassDeclaration, it makes more sense to process the SgClassDeclaration associated with the SgClassDefinition instead of the SgClassDefinition directly. Presently the processing of the SgClassDefinition is not supported through any indirect processing of the SgClassDeclaration, this could be implemented in the future.
6. SgGlobal
   This case is not implemented. It would require special handling, but it might be implemented in the future.
7. SgBasicBlock used in a SgForStatement
   Because of the declaration of the `for` loop (C language construct) index variable, this case would require special handling. This case could be implemented in the future.

8. SgBasicBlock used in a SgFunctionDefinition
   Because of the declaration of the function parameter variable, this case would require special handling. This case could be implemented in the future.
9. SgBasicBlock used in a SgSwitchStatement
   Example of code in generated intermediate file for a SgBasicBlock used in SgSwitchStatement:

```
int main()
{ /* local stack #0 */
int x;
int y;
switch(x)
{ /* local stack #1 */
int GlobalScopePreambleStart;
int GlobalScopePreambleEnd;
int CurrentLocationTopOfScopeStart;
int CurrentLocationTopOfScopeEnd;
int CurrentLocationBeforeStart;
int CurrentLocationBeforeEnd;
int CurrentLocationReplaceStart;
{case 0:{y++;break;}default:{y++;break;}}
int CurrentLocationReplaceEnd;
int CurrentLocationAfterStart;
int CurrentLocationAfterEnd;
int CurrentLocationBottomOfScopeStart;
int CurrentLocationBottomOfScopeEnd;
/* Reference marker variables to avoid compiler warnings */
   };     };
```

   This is more difficult because the declaration markers must appear after the `"{ /* local stack #1 */"` but then the statement `"case 0:y++;break;default:y++;break;"` cannot appear after a switch. It is probably impossible to fix this case due to the design and constraints of the C++ language (design and limitations of the switch statement). This is not a serious problem; it just means that the whole switch statement must be operated upon instead of the block within the switch statement separately (not a serious limitation).

Chapter 9

Program Analysis

Program analysis is an important part of required support for sophisticated transformations. This work is currently incomplete and is the subject of significant current research work. Specific support for global analysis is provided via a database mechanism provided within ROSE and as part of work in merging multiple ASTs from different files to hold the AST from a whole project (many files) in memory at one time.

Note that ROSE supports binary analysis, chapter 14 contains details that are specific to program analysis on binaries (executables, libraries, object files, etc.). A goal of ROSE is to unify much of the program analysis for source code and binaries, but there are details that are presented separately.

9.1 General Program Analysis

General program analysis is a critical piece of the work to provide optimization capabilities to ROSE to support DOE applications. This work generally lags behind the compiler construction issues and robustness required to handle large scale DOE applications.

9.1.1 Call Graph Analysis

Global *call graphs* are available, examples are in the ROSE Tutorial. ROSE supports two modes of building the call graph, with and without SQLite database support (controled from the ROSE configure commandline). The use of SQLite database support permits information to be accumulated into a named database file over the compilation of many files as required to support large multi-file projects (even separated over many directories).

The call graph uses the new graph IR nodes to support handling large scale graphs in ROSE. *Need to comment more on this.*

The mechanisms for filtering functions from the call graph is a subject of ongoing work to reduce the size of the call graph for more useful analysis and presentation.

Note: The Call Graph currently has a mechanism for inclusion/exclusion of paths/files locations of functions so that functions can be filtered from the Call Graph. This mechanism is currently hard coded in the test code and need to be controled from the command line in the future. ROSe included commandline support for specification of (see `--help` option for details):

- include paths
- exclude paths

- include files
- exclude files

At present we can include paths for the locations of functions to be included in the Call Graph. The specification of paths and files to include/exclude should be controled from the commandline.

9.1.2 C++ Class Hierarchy Graph Analysis

Class hierarchy graphs are available, examples are in the ROSE Tutorial.

9.1.3 Control Flow Graphs

Control graphs exist in two forms, one that is closely tied to the AST and another that is separate from the AST. See the ROSE Tutorial for examples of how to use these.

9.1.4 Dependence Analysis

Complete use-def chains are available, the ROSE Tutorial shows examples of how to access this information.

9.1.5 Alias Analysis

A linear alias analysis is provided, we need to write more about this.

9.1.6 Open Analysis

The Open Analysis project provides a connection to ROSE and permits the use of their pointer analysis with ROSE. More details on Open Analysis (and a reference) later.

9.1.7 More Program Analysis

Current work and collaborations will hopefully support an significant expansion of the program analysis supported within ROSE. We are working with a number of groups on pointer analysis, abstract interpretation, etc.

9.2 Database Support for Global Analysis

The purpose of database support in ROSE is to provide a persistent place for the accumulation of analysis results. The database used within ROSE is the publicly available SQLite relational database. Work has been done to provide a simple and extensible interface to SQLite. The demonstration and testing of the ROSE database mechanism has been supported through the construction of the call graph and class hierarchy graphs. These are discussed in subsequent subsections.

See chapter on *Getting Started* **??** for details of SQLite installation and configuration. Previous work supported MySQL, but this was overly complex.

9.2.1 Making a Connection To the Database and Table Creation

Figure 9.1 shows the listing of a program that connects to the ROSE database, creates a custom table, and performs some simple SQL queries. In the main function, at line 12, a *GlobalDatabaseConnection* object is created and is used to connect to the database in line 13. When the initialization succeeds, the database connection and ROSE database are ready for use.

Line 16 creates a *TableAccess* object. This object can be used to perform SQL queries like SELECT, INSERT or MODIFY on a given table in the database. The TableAccess object is templated by a *RowdataInterface* object that defines the structure of the table. For this example program, a RowdataInterface object for a test table is created in line 6 and 7. Here, two macros are called that handle the definition of the RowdataInterface class and all standard member functions. The general syntax is `CREATE_TABLE_[n]( [tablename], [column-1-datatype], [column-1-name], [column-2-datatype], [column-2-name], ... [column-n-datatype], [column-n-name] );`, where the "[...]" represents values to be filled in, such as the name of the table. As column datatype, all standard C-datatypes as *bool,char,short,long,float,double etc.* are valid. The resulting RowdataInterface class will contain standard functions to retrieve information about the table or its columns. An instance of this class has all private member variables to store the data of a single row of the table. Furthermore it has `get_[column-X-name]()` functions together with the corresponding `set_[column-X-name]([value])` functions to modify the values. By convention, tables used in ROSE will have one column more than specified, hence, $n + 1$ in total. The first column, which is always added, is a column of type `int` with the name *id*. This is used to easily identify all rows of a table. RowdataInterface classes used as template argument with a TableAccess class are required to have an *id*-column.

The class created by *CREATE_TABLE* will be called "[tablename]Rowdata," where "[tablename]" is the first argument for the *CREATE_TABLE* macro-call. The *DEFINE_TABLE* call is necessary to define global and static member variables of the RowdataInterface class. It has to be called once in a project, e.g. in the source file containing the main function, with exactly the same parameters as the *CREATE_TABLE* call. Thus, lines 6 and 7 together with lines 16 and 17 define the test table as having three columns: an integer "id" column, a "name" column storing a string and finally a third column "number" storing a double precision floating point number. The `initialize` call in line 17 will ensure the table exists and create it if necessary.

The next two statements at line 20 and 21 create a rowdata object that stores all fields of a single row of the test table. The constructor has initial arguments for all of the fields of a row. In this case "name" and 1.0 are used to initialize the field's name and number, which were specified in lines 6 and 7. The first argument *UNKNOWNID* is used to set the value of the row-id to the default value 0, which means that the id is not yet properly initialized. 0 is never used as an id for table rows; the lowest possible valid id is 1. Note that the insert function initializes the id of the row, as insert will create a new row in table that has a valid id.

In lines 24 and 25 a SQL query is performed, which selects all rows where the number column is equal to 1.0. The string passed to the *select* member function call contains the conditional expression (the *WHERE* clause) of an SQL statement. Hence, the single equals sign is an SQL equality test, and not, for example, an assignment. The selected rows of the table are returned as a vector of rowdata objects. As in line 21, a row matching the select condition was inserted into the table, so at least one row should be returned (if the example program was executed multiple times without deleting the test table, entries from previous runs may be returned as well). Assuming that the example program is run for the first time, the SQL query should return the inserted row, and the first object in the results vector should be identical to the inserted one. Lines 27 and 28 modify the name and number fields in memory. The modify call in line 29 then updates the database, by changing the existing row in the table and making the changes persistent. Line 32 is an exemplary call to delete a row of the table – the deletion uses the id of a row, so all other fields do not have to contain the same values as the row stored in the database.

The insert statement in line 35 simply inserts the row just deleted into the table again, leaving the test table in a different state. Hence, executing the example program multiple times should fill the test table with multiple rows. In line 37, the connection to the database is closed. Try to add a call to `GlobalDatabase::DEBUG_dump()` before the shutdown function call, and run it multiple times to see how the automatic id assignment works.

9.2.2 Working With the Predefined Tables

While the first database example worked on a self-defined table, this tutorial will explain how to use one of the tables that are predefined for usage within ROSE. Its source code is shown in Figure 9.2. These tables are easier to use because their structure is already defined in the `TableDefinition.h` file. Lines 6 and 7 define the tables used for storing information about projects and files in ROSE using the macros *DEFINE_TABLE_PROJECTS* and *DEFINE_TABLE_FILES*. These macros call the corresponding macros from the previous example to define the structure of these tables.

The easiest way to use these tables is the *CREATE_TABLE* macro. The first parameter is a GlobalDatabaseConnection object, the second one is the name of the table. Hence, line 17 will initialize the projects table, and create an instance of the projectsTableAccess object having the same name as the table, "projects." Line 18 initializes the files table in the same way. Now two instances of the TableAccess class for the projectsRowdata and the filesRowdata objects are declared in the main scope, and are ready to be used.

The example program performs an initialization to retrieve the ids for the project and the file currently processed, which is usually needed for a traversal. Lines 21 and 22 set values for project and file name, although these values might normally be retrieved from the corresponding SgProject and SgFile nodes . As all projects work on the single ROSE database and share the same tables for function and data, each of these tables has a *projectId* column to specify to which project each row belongs. Thus, one of the first tasks a preprocessor using the database will do is to enable these ids to select or insert rows.

XME: *project name? parameter?*

The *TableAccess::retrieveCreateByColumn* function is used for this purpose. It tries to identify an entry using a unique name, and creates that entry if it does not yet exist, or retrieves the id of the existing entry otherwise. The function takes a pointer to a rowdata object, the name of the column to use, and the unique name of the row as arguments (see line 25). So in this case the "name" column and the string "testProject" are used. As with the normal insert function from the first example, the retrieveCreateByColumn function sets the id field of the rowdata to the correct value. A new variable storing this project id is created in line 27. For the file id, the procedure is almost the same – with the exception that the project id is also passed to the function call in line 32. For most other ids other than the project id, the project id is used to retrieve the row for the desired project. If a project id is passed to the retrieveCreateByColumn function, it assumes the table has a "projectId" column, which has to match the given value.

Instead of working with these ids, the example program just prints these values to stdout, and quits. The ids will remain the same over multiple runs of the program. Try changing the file or project ids, to force new entries to be created.

9.2.3 Working With Database Graphs

The following tutorial program will use the ROSE tables to build a graph for a user-defined table. Each execution of the program will enlarge the test graph by adding three nodes and edges to them from a random node in the graph.

9.2.4 A Simple Callgraph Traversal

The last database example tutorial will show how to use the database graph features explained in the previous example in combination with a AST-traversal to build a simple callgraph.

Database Connection Example

```
// database access
#include "GlobalDatabaseConnectionMYSQL.h"
#include "TableAccess.h"

CREATE_TABLE_2( testtable, string,name, double,number );
DEFINE_TABLE_2( testtable, string,name, double,number );

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

  // DQ (4/6/2017): This will not fail if we skip calling ROSE_INITIALIZE (but
  // any warning message using the message looging feature in ROSE will fail).
     ROSE_INITIALIZE;

     GlobalDatabaseConnection db;
     int initOk = db.initialize();
     assert( initOk==0 );

     TableAccess< testtableRowdata > testtable( &db );
     testtable.initialize();

  // add a row
     testtableRowdata testrow( UNKNOWNID, "name", 1.0 );
     testtable.insert( &testrow );

  // select & modify
     vector<testtableRowdata> results = testtable.select(" number=1.0 ");
     assert( results.size() > 0 );
     results[0].set_name( string("newname") );
     results[0].set_number( 2.0 );
     testtable.modify( &results[0] ); // this uses the ID of the row

  // remove entry
     testtable.remove( &results[0] );

  // add again for next run...
     testtable.insert( &results[0] );

     db.shutdown();
     return( 0 );
   }
```

Figure 9.1: Source code for the database connection example.

Table Creation Example

```
#include <iostream>

// database access
#include "GlobalDatabaseConnectionMYSQL.h"
#include "TableDefinitions.h"
DEFINE_TABLE_PROJECTS();
DEFINE_TABLE_FILES();

//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
   {
  // DQ (4/6/2017): This will not fail if we skip calling ROSE_INITIALIZE (but
  // any warning message using the message looging feature in ROSE will fail).
     ROSE_INITIALIZE;

     GlobalDatabaseConnection db;
     int initOk =  db.initialize();
     assert( initOk==0 );

     CREATE_TABLE(db, projects);
     CREATE_TABLE(db, files);

  // initialize project
     string projectName = "testProject";  // this should be given at the command line
     string fileName    = "testFile.C";   // this should be retrieved from a SgFile node

     projectsRowdata prow( UNKNOWNID ,projectName, UNKNOWNID );
     projects.retrieveCreateByColumn( &prow, "name", projectName );
     long projectId      = prow.get_id();

  // get id of this file
     filesRowdata frow( UNKNOWNID, projectId , fileName );
     files.retrieveCreateByColumn( &frow, "fileName", fileName, frow.get_projectId() );
     long fileId         = frow.get_id();

  // do some work...
     std::cout << "Project ID:"<< projectId <<" , File ID:" << fileId << std::endl;

     db.shutdown();
     return( 0 );
   }
```

Figure 9.2: Source code for the predefined tables example.

Database Graph Example

```
#include <iostream>
#include "GlobalDatabaseConnectionMYSQL.h"
#include "TableDefinitions.h"
DEFINE_TABLE_PROJECTS();
DEFINE_TABLE_FILES();
DEFINE_TABLE_GRAPHDATA();
DEFINE_TABLE_GRAPHNODE();
DEFINE_TABLE_GRAPHEDGE();
CREATE_TABLE_2( testtable, string,name, double,number );
DEFINE_TABLE_2( testtable, string,name, double,number );
#define TABLES_DEFINED 1

#include "DatabaseGraph.h"


//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
   {
  // DQ (4/6/2017): This will not fail if we skip calling ROSE_INITIALIZE (but
  // any warning message using the message looging feature in ROSE will fail).
     ROSE_INITIALIZE;

     GlobalDatabaseConnection db;
     int initOk =  db.initialize();
     assert( initOk==0 );

     CREATE_TABLE(db, projects);
     CREATE_TABLE(db, files);
     CREATE_TABLE(db, graphdata);
     CREATE_TABLE(db, graphnode);
     CREATE_TABLE(db, graphedge);

     TableAccess< testtableRowdata > testtable( &db );
     testtable.initialize();

  // initialize project
     string projectName = "testProject";  // this should be given at the command line
     string fileName    = "testFile.C";   // this should be retrieved from a SgFile node

     projectsRowdata prow( UNKNOWNID ,projectName, UNKNOWNID );
     projects.retrieveCreateByColumn( &prow, "name", projectName );
     long projectId      = prow.get_id();

  // get id of this file
     filesRowdata frow( UNKNOWNID, projectId , fileName );
     files.retrieveCreateByColumn( &frow, "fileName", fileName, frow.get_projectId() );
     long fileId         = frow.get_id();

  // init graph
     DatabaseGraph<testtableRowdata, EdgeTypeEmpty> *graph = new DatabaseGraph<testtableRowdata, EdgeT
     graph->loadFromDatabase( );

  // the graph is now ready for use...
  // add some example nodes and edges

     testtableRowdata testrow1( UNKNOWNID, "dgb_exrow1", 0.1 );
     ostringstream whereClause;
     whereClause << " graphId='" << graph->getGraphId() <<"' ";
     vector<graphnodeRowdata> result = graphnode.select( whereClause.str() );
     if(result.size() > 0)
        {
       // select a random node from the graph
          int index = rand() % result.size();
       // the node will be identified by the id, so the other fields
       // dont have to be inited correctly
          testrow1.set_id( result[index].get_nodeId() );
        }
       else
```

Simple Callgaph Example

```
#include <iostream>
#include "GlobalDatabaseConnectionMYSQL.h"
#include "TableDefinitions.h"
DEFINE_TABLE_PROJECTS();
DEFINE_TABLE_FILES();
DEFINE_TABLE_GRAPHDATA();
DEFINE_TABLE_GRAPHNODE();
DEFINE_TABLE_GRAPHEDGE();
CREATE_TABLE_2( testtable,  string,name,  double,number );
DEFINE_TABLE_2( testtable,  string,name,  double,number );
#define TABLES_DEFINED 1

#include "DatabaseGraph.h"


//-----------------------------------------------------------------------------
// define traversal classes NYI
//-----------------------------------------------------------------------------


//-----------------------------------------------------------------------------
int main(int argc, char *argv[])
   {
  // DQ (4/6/2017): This will not fail if we skip calling ROSE_INITIALIZE (but
  // any warning message using the message looging feature in ROSE will fail).
     ROSE_INITIALIZE;

     GlobalDatabaseConnection db;
     int initOk =  db.initialize();
     assert( initOk==0 );

     CREATE_TABLE(db, projects);
     CREATE_TABLE(db, files);
     CREATE_TABLE(db, graphdata);
     CREATE_TABLE(db, graphnode);
     CREATE_TABLE(db, graphedge);

     TableAccess< testtableRowdata > testtable( &db );
     testtable.initialize();

  // initialize project
     string projectName = "testProject";  // this should be given at the command line
     string fileName    = "testFile.C";   // this should be retrieved from a SgFile node

     projectsRowdata prow( UNKNOWNID ,projectName, UNKNOWNID );
     projects.retrieveCreateByColumn( &prow, "name", projectName );
     long projectId      = prow.get_id();

  // get id of this file
     filesRowdata frow( UNKNOWNID, projectId , fileName );
     files.retrieveCreateByColumn( &frow, "fileName", fileName, frow.get_projectId() );
     long fileId         = frow.get_id();

  // init graph
     DatabaseGraph<testtableRowdata, EdgeTypeEmpty> *callgraph = new DatabaseGraph<testtableRowdata, EdgeTypeEmp
     callgraph->loadFromDatabase( );

  // traverse... NYI

  // save graph to dot file, and to database
     graph->writeToDOTFile( "simplecallgraph_example.dot" );
     graph->writeToDatabase( );
     delete callgraph;

     db.shutdown();
     return( 0 );
   }
```

Chapter 10

Loop Transformations

10.1 Introduction

The loop transformation package implements all the algorithms published by Yi and Kennedy [34, 40, 36], including the transitive dependence analysis algorithm by Yi, Adve, and Kennedy [35].[1] These algorithms automatically optimize the loop structures of applications for better performance. For now, the implementation aims only to improve the cache locality of applications running on a single-processor machine. In the future, it can be expanded to optimize parallel applications by maximizing the parallelism and minimizing the communication cost of loop structures [32, 31, 28].

To optimize applications for better cache locality, this package applies the following loop transformations: interchange, fusion, fission(or distribution), and blocking (or tiling). The implementation can successfully optimize arbitrary loop structures, including complex, non-perfect loop nests such as the one from LU factorization with no pivoting in Figure 10.2. The following examples illustrate the effect of applying the transformations.

Figure 10.1 uses a pseudo code of *matrix multiplication* to illustrate the effect of applying the package to optimize perfect loop nests. The original code is in (a). After performing dependence analysis on this loop nest, the package applies loop interchange transformation to improve the data reuse in caches (note that in C/C++ language, the matrix is stored in row-major order). The transformed code is shown in (b). The cache locality of this code can be further improved by loop blocking, and the result is shown in (c).

Figure 10.2 uses the pseudo code of *LU factorization without pivoting* to illustrates the effect of applying

[1]The package does not include the recursion transformation algorithm in this publication.

for $(i = 0; i <= n - 1; i+ = 1)$
  for $(j = 0; j <= n - 1; j+ = 1)$
    for $(k = 0; k <= N - 1; k+ = 1)$
      $c[i][j] = c[i][j] + a[i][k] * b[k][j];$

(a) original code

for $(i = 0; i <= n - 1; i+ = 1)$
  for $(k = 0; k <= n - 1; k+ = 1)$
    for $(j = 0; j <= N - 1; j+ = 1)$
      $c[i][j] = c[i][j] + a[i][k] * b[k][j];$

(b) after loop interchange

for $(x_k = 0; x_k <= n - 1; x_k+ = b)$
  for $(x_j = 0; x_j <= n - 1; x_j+ = b)$
    for $(i = 0; i <= n - 1; i+ = 1)$
      for $(k = x_k; k <= min(n - 1, x_k + b - 1); k+ = 1)$
        for $(j = x_j; j <= min(n - 1, x_j + b - 1); j+ = 1)$
          $c[i][j] = c[i][j] + a[i][k] * b[k][j];$

(c) after loop blocking

Figure 10.1: Optimizing matrix multiplication, first applying loop interchange to arrange the best nesting order in (b), then applying blocking to exploit data reuses carried by k and j loops in (c).

```
    for (k = 0; k <= n - 2; k+ = 1)
    {
      for (i = k + 1; i <= n - 1; i+ = 1)
s1:     a[k][i] = a[k][i]/a[k][k];
      for (j = k + 1; j <= n - 1; j+ = 1)
        for (i = k + 1; i <= n - 1; i+ = 1)
s2:       a[j][i] = a[j][i] - a[j][k] * a[k][i];
    }
```

(a) original code

```
    for (j = 0; j <= n - 1; j+ = 1)
    {
      for (k = 0; k <= j - 1; k+ = 1)
        for (i = k + 1; i <= n - 1; i+ = 1)
s2:       a[j][i] = a[j][i] - a[j][k] * a[k][i];
      for (i = j + 1; i <= n - 1; i+ = 1)
s1        a[j][i] = a[j][i]/a[j][j];
    }
```

(b) after loop interchange

```
    for (x_k = 0; x_k <= n - 2; x_k+ = b)
      for (j = x_k; j <= n - 1; j+ = 1)
      {
        for (k = x_k; k <= min(x_k + b - 1, j - 1); k+ = 1)
          for (i = k + 1; i <= n - 1; i+ = 1)
s2:         a[j][i] = a[j][i] - a[j][k] * a[k][i];
        for (i = j + 1; i <= n - 1; i+ = 1)
s1:       a[j][i] = a[j][i]/a[j][j];
      }
```

(c) after blocking row dimension

Figure 10.2: Optimizing non-pivoting LU. In (b), the $k(s_1)$ loop is fused with the $j(s_2)$ loop and the fused loop is then put at the outermost position, achieving a combined interchange and fusion transformation; the code in (c) achieves blocking in the row dimension of the matrix through combined interchange, fusion and tiling transformations.

the package to optimize complex, non-perfectly nested loop structures. Although the original loops in (a) are not perfectly nested, the package recognizes that the $k(s_1)$ loop (k loop surrounding statement s_1) can be recombined with the loop $j(s_2)$ and that the recombined loop can then be placed outside of the original $k(s_2)$ loop. The transformed code in (b) simultaneously achieves two effects: the fusion of $k(s_1)$ with $j(s_2)$ loop, and the interchange of $k(s_2)$ with $j(s_2)$ loop. Section 10.3.2 explains this combined-interchange-and-fusion transformation in more detail. The code in (b) can further be blocked, and the result is shown in (c).

Figure 10.3 illustrates the effect of applying loop fusion to a sequence of loop nests in the subroutine *tridvpk* of the application benchmark *Erlebacher* from ICASE. The original code in (a) contains four separate loop nests, all of which can be fused into a single one. The package performs multiple levels of loop fusion simultaneously using a combined-interchange-and-fusion transformation(see Section 10.3.2), and the optimized code is shown in (b).

10.2 Interface for End-Users and Compiler Developers

This package is written in C++ language in a object-oriented style. It utilizes traditional techniques developed to optimize loop nests in Fortran programs. When optimizing C or C++ applications, this package only recognizes and optimizes a particular for-loop that corresponds to the *DO* loop construct in Fortran programs. Within the ROSE source-to-source compiler infrastructure, such a loop is defined to have the following formats:

$$for\ (i = lb; i <= ub; i+ = positiveStep)\quad or\quad for\ (i = ub; i >= lb; i+ = negativeStep) \tag{10.1}$$

Here i is an arbitrary integer variable, lb and ub are arbitrary integer expressions, and *positiveStep* and *negativeStep* are positive and negative integer expressions respectively. To expand this definition, the user can rewrite the $LoopTransformInterface$ class within the package distribution (see Section 10.2.2) or use a preprocessor within ROSE to translate all the non-Fortran loops into the aforementioned formats. Such a loop-normalization preprocessor will be provided within ROSE.

The package distribution within ROSE also includes a loop optimization tool called $LoopProcessor$, which automatically transforms the Fortran loops in C/C++ applications for better performance. In addition, the package also provides two levels of internal user interfaces: one for end users that intend to apply this package

```
for (j = 0; j <= n − 1; j+ = 1)
  for (i = 0; i <= n − 1; i+ = 1)
    duz[i][j][1] = duz[i][j][1] ∗ b[1];
for (k = 1; k <= n − 2; k+ = 1)
  for (j = 0; j <= n − 1; j+ = 1)
    for (i = 0; i <= n − 1; i+ = 1)
      duz[i][j][k] = (duz[i][j][k] − a[k] ∗ duz[i][j][k − 1]) ∗ b[k];
for (j = 0; j <= n − 1; j+ = 1)
  for (i = 0; i <= n − 1; i+ = 1)
  tot[i][j] = 0;
for (k = 0; k <= n − 2; k+ = 1)
  for (j = 0; j <= n − 1; j+ = 1)
    for (i = 0; i <= n − 1; i+ = 1)
      tot[i][j] = tot[i][j] + d[k] ∗ duz[i][j][k];
for (j = 0; j <= n − 1; j+ = 1)
  for (i = 0; i <= n − 1; i+ = 1)
    duz[i][j][n − 1] = (duz[i][j][n − 1] − tot[i][j]) ∗ b[n − 1];
for (j = 0; j <= n − 1; j+ = 1)
  for (i = 0; i <= n − 1; i+ = 1)
    duz[i][j][n − 2] = duz[i][j][n − 2] − e[n − 2] ∗ duz[i][j][n − 1];
for (k = n − 3; k >= 0; k+ = −1)
  for (j = 0; j <= n − 1; j+ = 1)
    for (i = 0; i <= n − 1; i+ = 1)
      duz[i][j][k] = duz[i][j][k] − c[k] ∗ duz[i][j][k + 1]
                     −e[k] ∗ duz[i][j][n − 1];
```

(a) original code

```
for (i = 0; i <= n − 1; i+ = 1)
{
  for (j = 0; j <= n − 1; j+ = 1)
  {
    tot[i][j] = 0.0;
    duz[i][j][1] = duz[i][j][1] ∗ b[1];
    for (k = 1; k <= n − 2; k+ = 1)
    {
      duz[i][j][k] = (duz[i][j][k] − a[k] ∗ duz[i][j][k − 1]) ∗ b[k];
      tot[i][j] = tot[i][j] + d[k] ∗ duz[i][j][k];
    }
    duz[i][j][n − 1] = (duz[i][j][n − 1] − tot[i][j]) ∗ b[n − 1];
    duz[i][j][n − 2] = duz[i][j][n − 2] − e[n − 2] ∗ duz[i][j][n − 1];
    for (k = n − 3; k >= 0; k+ = −1)
      duz[i][j][k] = (duz[i][j][k] − c[k] ∗ duz[i][j])[k + 1])
                     −e[k] ∗ ((duz[i])[j])[n − 1];
  }
}
```

(b) after fusion

Figure 10.3: Optimizing *tridvpk* from Erlebacher: combining loop interchange and fusion, thus fusing multiple levels of loops simultaneously

to optimize their applications, and one for compiler developers that intend to extend this package for various purposes.

10.2.1 End-User Interface

The following function comprises the package interface for end users of the ROSE source-to-source infrastructure, which applies various traversal and rewrite mechanisms to transform C++ applications using the SAGE intermediate representation.

$$Boolean\ \ SageLoopTransformation(unsigned\ \ argc, char**\ \ argv, SgGlobal*\ \ r, SgNode*\ \ n); \tag{10.2}$$

Here both *SgGlobal* and *SgNode* are classes defined by the SAGE intermediate representation: the *SgGlobal* pointer r represents the global root of an input program, and the *SgNode* pointer n represents the root of the input code fragment to be transformed by the package. The parameters *argc* and *argv* represent command-line arguments that instruct the package to adopt specific optimization strategies: *argc* contains the number of arguments, and *argv* contains the vector of *string* arguments.

The package currently recognizes the following arguments:

- -bk1 <blocksize> : apply outer-loop blocking for better data reuse
- -bk2 <blocksize> : apply inner-loop blocking for better data reuse
- -ic1 : apply loop interchange for better data reuse
- -fs0 : perform maximum loop distribution with no fusion afterwards
- -fs1 : apply hierarchical single-level loop fusion for better data reuse
- -fs2 : apply simultaneous multi-level loop fusion for better data reuse
- -tm : report timing information for each phase of the transformation package
- -ta <int> : set the maximum number of split nodes when performing transitive dependence analysis
- -clsize <int> : set cache-line size for spatial reuse analysis

The loop transformation tool *LoopProcessor* within ROSE recognizes these command-line arguments and then automatically selects the corresponding optimization strategies. When invoked with no argument, *LoopProcessor* prints out usage information of this package.

10.2.2 Developer Interface

Utilizing the available internal interface, compiler developers can easily extend this package in two aspects. First, they can rewrite the outside-interface classes of the implementation to port it to a different compiler infrastructure (other than ROSE). Second, they can provide their own profitability analysis to expand the transformation policy classes of the implementation.

Porting to a different compiler infrastructure The package provides the following infrastructure-independent interface to compiler developers.

$$AstNodePtr\ \ LoopTransformation(LoopTransformInterface\ \&interface, const\ AstNodePtr\ \&head); \tag{10.3}$$

Here the class *LoopTransformInterface* provides the interface for accessing the intermediate representation of an arbitrary compiler, and the pointer reference *AstNodePtr* represents an arbitrary code fragment to be transformed. Both classes, $AstNodePtr$ and $LoopTransformInterface$, need to be defined at location $outsideInterface/LoopTransformInterface.h$, which currently contains the ROSE implementation of these two classes. By rewriting this file, a compiler developer can port the package to a completely different infrastructure (this package already works under two compiler infrastructures: the ROSE C++ infrastructure and the DSystem Fortran infrastructure at Rice University [29]).

Plugging in different profitability analysis algorithms This package provides a static configuration class, $LoopTransformOptions$ (defined in the location $driver/LoopTransformOptions.h$ of the package distribution), for plugging in different loop optimization policies. This configuration class uses a set of policy classes (automatically selected from the command-line arguments, as described in Section 10.2.1) to control the application of three loop transformations: interchange, fusion and blocking. The currently available policy classes are defined in the locations $driver/InterchangeAnal.h$, $driver/FusionAnal.h$ and $driver/BlockingAnal.h$ respectively. To plug in different optimization strategies, the developer can write new profitability policy classes and then configure $LoopTransformOptions$ to use the new algorithms. The command-line configurations are automatically extended when the developer registers these new policy classes.

10.3 Analysis and Transformation Techniques

This package implements the following techniques to optimize applications for better cache locality. This section provides only brief introductions to the algorithms without going into any detail. Most algorithms are described in detail in Qing Yi's Ph.D. thesis [34].

10.3.1 Dependence and Transitive Dependence Analysis

Similar to most of the existing loop optimizing compilers, this package models the safety requirement of loop transformations using a dependence graph. The dependence graph includes all the statements of the input code segment as vertices, and a dependence edge is put from statement s_1 to s_2 in the graph if s_1 must be executed before s_2. If a statement reordering transformation does not reverse the direction of any dependence edge, the transformation is guaranteed to preserve the original semantics of the program. If two statements, s_1 and s_2, are both surrounded by loops, for each dependence edge between s_1 and s_2, the dependence graph also defines a condition that must hold between the iterations of these loops. The compiler then uses the dependence relations to determine the safety of transforming these loops.

In traditional unimodular and single loop transformation systems, the dependence relation between each pair of statements s_1 to s_2 is defined using a vector of direction or distance entries, where each direction or distance entry defines the relation between the iterations of a common loop surrounding both s_1 and s_2. The compiler then uses these dependence vectors to determine the safety of transforming a set of common loops that are perfectly nested.

In order to effectively transform arbitrary, non-perfectly nested loop structures, this package extends the traditional dependence model with a new dependence representation, *Extended Direction Matrix(EDM).* Given two statements, s_1 and s_2, a dependence EDM from s_1 to s_2 defines a direction or distance entry for each pair of loops (ℓ_1, ℓ_2) s.t. ℓ_1 surrounds s_1 and ℓ_s surrounds s_2. This new dependence representation thus defines dependence conditions for not only common loops surrounding both s_1 and s_2, but also non-common loops that surround only one of the two statements.

To compute the EDM representation of dependences, this package uses an adapted Gaussian elimination algorithm to solve a set of integer linear equations of loop induction variables. For each array access in the original input program, the algorithm first constructs a set of linear equations based on the index expressions of the array access. If no loop induction variable has a symbolic coefficient in the array access expressions, such as the ones in the *Matrix Multiplication* code in Figure 10.1 and the *non-pivoting LU* in Figure 10.2, the algorithm is at least as powerful as the combined ZIV, SIV, and Delta dependence tests described by Allen and Kennedy [30, 33]. However, when loop induction variables do have symbolic coefficients, the algorithm assumes a conservative solution and is less precise than the symbolic solution algorithms described in [30, 33].

This package also extends the traditional dependence model by implementing the transitive dependence analysis algorithm published by Yi, Adve, and Kennedy [35]. Note that although the algorithm is quite efficient in summarizing the complete transitive dependence information between statements, this package applies transitive dependence analysis only when transforming complex loop structures that cannot be translated into sequences of perfectly nested loops. Because the safety of transforming perfect loop nests can be determined based on individual dependence edges alone, it is often more economic to do without the extra cost of transitive dependence analysis. This package examines the original loop structures of programs and performs transitive dependence analysis only when required.

10.3.2 Dependence Hoisting Transformation

As the base technique for loop interchange, fusion and blocking, this package implements a novel loop transformation, *dependence hoisting* (first introduced by Yi and Kennedy [40]), that facilitates a combined fusion and interchange transformation for a group of arbitrarily nested loops. Applying the dependence and transitive dependence analysis algorithms, this transformation first selects a group of arbitrarily nested loops, such as the $k(s_1)$ (k loop surrounding s_1) and the $j(s_2)$ loops in the non-pivoting LU code in Figure 10.2(a), that can be legally fused and then placed at the outermost position of a code segment. It then performs the transformation through a compound sequence of traditional transformations on single loops and perfectly nested loops. A combined interchange and fusion transformation is established on an arbitrary loop structure as a result. An example of the transformation result is shown for the non-pivoting LU code in Figure 10.2(b) (here the transformation is applied to the $k(s_1)$ and $j(s_2)$ loops in (a)).

Given a group of loops as input for a dependence hoisting transformation, the safety of fusing and shifting these loops is determined from the dependence constraints on iterations of these loops. If the group is a single loop in the original code, such as the i, j or k loop in the matrix multiplication code in Figure 10.1, traditional loop interchange analysis for perfect loop nests would suffice; however, if the group includes non-common loops surrounding different statements, such as the $k(s_1)$ and $j(s_2)$ loops in the non-pivoting LU code in Figure 10.2(a), transitive dependence analysis is performed on the dependence graph and the transitive dependences are used to determine the safety of fusing and shifting these loops.

Because dependence hoisting is realized by combining a sequence of traditional loop distribution, interchange and index set splitting transformations on single or perfectly nested loops, the complexity of applying dependence hoisting is equivalent to that of the corresponding sequence of sub-transformations. In the worst case, applying dependence hoisting to a loop nest takes time proportional to $N^2 + L^2 D$, where N is the number of statements in the nest, L is the depth of the nest, and D is the size of the dependence graph for the nest. In an average case, however, dependence hoisting requires much less time to finish. For a perfect loop nest, dependence hoisting is equivalent to a standard loop interchange on perfect loop nests followed by a single-loop distribution, in which case the required complexity is $O(N + D)$.

10.3.3 Transformation Framework

To optimize applications for better locality, this package uses *dependence hoisting* to achieve three loop transformations: loop fusion, interchange and blocking. It uses a construct, *computation slice* (or simply *slice*), to encode the input information necessary to perform each dependence hoisting transformation. For example, for the dependence hoisting transformation on the non-pivoting LU code from Figure 10.2(a) to (b), the computation slice contains two loops: $k(s_1)$ and $j(s_2)$. Each computation slice must be valid in that the corresponding dependence hoisting transformation does not reverse any dependence direction of the original program.

To model the memory performance of applications, this package associates each computation slice with a floating point number, which defines the number of array references that can be reused at each iteration of the slice, that is, the number of references that can be reused when the loops in the slice are placed at the innermost position of a loop structure [30]. Here the floating point number is necessary to model the spatial reuses resulted from references residing in the same cache line, where in average less than one reference could be reused at each iteration of the *slicing loops* (loops in the computation slice). These floating point numbers provide the data reuse information of computation slices to the transformation framework, which then uses the information to guide loop interchange, fusion and blocking transformations.

Using the data reuse information of computation slices, the transformation framework optimizes a code segment in the following steps. First, it applies dependence analysis and constructs all the legal computation slices for an input code segment. It then treats all the valid computation slices as if they form a sequence of loop nests and rearranges these slices to achieve better cache locality. For each set of computation slices that forms a single loop nest, the package first selects a nesting order so that the loops that are associated with more reuses are nested inside. It then fuses each pair of disjunct computation slices (slices that contain disjunct sets of statements) when their statements access a common set of data. After fusion, if some non-innermost slices carry data reuses, the package marks the corresponding slice nest to be tiled later. Finally, the framework uses the rearranged computation slices to perform a sequence of dependence hoisting transformations to achieve the desired transformation result. Note that all the transformations are applied only when legal, that is, no semantics of the original program is violated by the transformations.

The following briefly describes the optimization strategies implemented in this package. For more details of the optimization algorithms, see [34].

Loop Interchange and Blocking To achieve loop interchange, the package carefully arranges the order of applying dependence hoisting transformations using different computation slices. Because each slice represents a set of loops that can be fused into a single loop, interchanging the nesting order of two slices corresponds directly to the interchange of the two sets of slicing loops. The effects of applying loop interchange is shown for *matrix multiplication* in Figure 10.1(b) and for *non-pivoting LU factorization* in Figure 10.2(b).

Because this package implements loop interchange using dependence hoisting, it achieves loop blocking by combining a sequence of dependence hoisting with loop strip-mining transformation. Given an input loop nest C, the algorithm takes the computation slices constructed for C in the reverse of their desired nesting order and then uses each slice to perform a dependence hoisting transformation. After each dependence hoisting transformation, if the new outermost loop ℓ_f should be blocked, the algorithm strip-mines ℓ_f into a strip-counting loop ℓ_c and a strip-enumerating loop ℓ_t. It then uses loop ℓ_t as the input loop nest for further dependence hoisting transformations, which in turn will shift a new set of loops outside loop ℓ_t but inside loop ℓ_c, thus blocking loop ℓ_f. The effects of applying loop blocking is shown for *matrix multiplication* in Figure 10.1(c) and for *non-pivoting LU factorization* in Figure 10.2(c).

Loop Fusion and Distribution (Fission) To achieve an aggressive multi-level loop fusion effect, the package merges multiple computation slices and then uses the merged slices to transform the original code. Given two disjunct computation slices (two slices that contain disjunct sets of statements), because each computation slice fuses a set of loops that can be shifted to the same loop level, fusing these two slices automatically achieves the fusion of the loops in both slices. For example, in Figure 10.3, after transformation analysis, the package constructs a computation slice for each of the loops in the original code in (a). It then performs fusion analysis and realizes that all the j slices (and thus all the j loops) can be legally fused into a single loop. After merging these slices, it uses a single j slice to perform a dependence hoisting transformation and thus automatically achieves the fusion of all the j loops in (a). Similarly, all the i loops are also fused into a single loop, and two of the k loops are fused.

Because the original loop structure may need to be distributed to achieve better performance, before applying loop fusion analysis, this package first performs maximum loop fission to distribute all the loop nests in the original code. The distributed loop nests are then recombined during the loop fusion phase. This strategy ensures that both loop fission and fusion optimizations are applied and that the final result of the optimization does not depend on the original loop structure of the application.

Combined Loop Interchange and Fusion This package optimizes applications to improve the memory performance of applications through a combined loop interchange and multi-level fusion strategy [36]. Since loop fusion is implemented in terms of merging computation slices, given a code segment C to optimize, the package first constructs all the valid computation slices. It then applies loop interchange analysis to these slices to arrange the best nesting order for each loop nest in C. When applying fusion analysis to merge the disjunct computation slices, it performs data reuse analysis and performs the actual fusion only when loop fusion does not interfere with loop interchange or when fusion is more favorable even if it interferes with loop interchange. Because multiple computation slices are constructed for each loop nests, and all of these slices participate in the fusion analysis simultaneously, multiple loops may be fused for each loop nest in a single pass of fusion analysis. As the result, this package achieves a combined loop interchange and multi-level fusion optimization for a collection of loop nests. For example, in Figure 10.3, even though the j and i loops are nested at different levels in the original code in (a), the package successfully achieves the fusion of these loops because all the loops are collected as computation slices in a single pass and together they participate in the fusion analysis.

10.3.4 Profitability Analysis

This package separates the profitability analysis of loop transformations from the actual transformations by encoding profitability analysis algorithms in a set of policy classes and then using these policy classes to control the application of loop transformations. A flexible internal interface is provided for compiler writers to plug in their own performance model for various optimization purposes (see Section 10.2.2).

The currently available performance model includes only the counting of array references being reused, including both temporary and spatial cache reuses. Because the package has not yet implemented the calculation of the working set size of each loop body, it cannot automatically decide the tile size for each blocked loop nest. Similarly, because the current data reuse analysis is insufficient in calculating the trade-off between outer-loop blocking and inner-loop blocking, the package asks the user to specify the desired strategy. It then applies the specified strategy uniformly for all the loop nests.

The profitability analysis algorithms within this package are not yet complete and will incorporate more sophisticated algorithms in the future. These algorithms include not only various strategies to automate the decision of blocking parameters, but also runtime tuning strategies that execute applications on a specific machine and then use the collected performance information to automatically select the best overall transformations.

Chapter 11

AST Merge: Whole Program Analysis Support

11.1 Introduction

The AST merge support in ROSE is a mechanism to generate a single binary file representing the AST for a whole program that could consist of thousands of files. A focus in this work has been on the scaling required to handle realistic large scale laboratory applications.

11.2 Usage

See tutorial for an example.

Chapter 12

OpenMP Support

12.1 Introduction

ROSE supports the OpenMP 3.0 specifications [42]. OpenMP is a popular shared-memory parallel programming model which extends serial programming languages like C/C++ and Fortran 77/90 to include additional parallel semantics. The extensions OpenMP provides contain compiler directives, user level runtime routines and environment variables. Depending on the languages (C/C++ or Fortran 77-2003), the OpenMP support in ROSE includes parsing OpenMP directive, generating dedicated AST nodes for them, and finally translating OpenMP programs into multithreaded programs targeting the GCC GOMP OpenMP runtime library or the Omni OpenMP runtime library. We also have an implementation of automatic parallelization using OpenMP, which automatically introduces OpenMP directives into sequential C/C++ applications.

12.2 Command Line Options

Like most other OpenMP implementations, ROSE's OpenMP support has to be explicitly turned on via command line options. By running any ROSE-based translator (e.g. identityTranslator) with *-help*, the OpenMP-related command line options can be displayed as follows:

```
-rose:OpenMP, -rose:openmp
                        follow OpenMP 3.0 specification for C/C++ and Fortran, perform one of the following actions:
-rose:OpenMP:parse_only, -rose:openmp:parse_only
                        parse OpenMP directives to OmpAttributes, no further actions (default behavior now)
-rose:OpenMP:ast_only, -rose:openmp:ast_only
                        on top of -rose:openmp:parse_only, build OpenMP AST nodes from OmpAttributes, no further actions
-rose:OpenMP:lowering, -rose:openmp:lowering
                        on top of -rose:openmp:ast_only, transform AST with OpenMP nodes into multithreaded code
                        targeting GCC GOMP runtime library
```

The options above work with any ROSE-based translators. The OpenMP support is optionally invoked inside of SgProject * frontend(). The AST generated by frontend() will reflect changes caused by OpenMP parsing and lowering depending on the option used.

12.3 Entry Point and Top Level Function

ROSE processes OpenMP directives right after preprocessing information is processed within SgFile::callFrontEnd(). This order is necessary since preprocessing information may exist within OpenMP directives, such as macro calls. ROSE needs to be aware of such preprocessing information when processing OpenMP.

The top level function for processing OpenMP is named processOpenMP() (defined in *rose/src/frontend/ompAstConstruction.cpp*), as shown below:

```
void processOpenMP(SgSourceFile *sageFilePtr)
{
  ROSE_ASSERT(sageFilePtr != NULL);
  // skip if no OpenMP directives
  if (sageFilePtr->get_openmp() == false)
    return;

  // parse OpenMP directives and attach OmpAttributeList to relevant SgNode
  attachOmpAttributeInfo(sageFilePtr);

  // stop here if only OpenMP parsing is requested
  if (sageFilePtr->get_openmp_parse_only())
    return;

  //Build OpenMP AST nodes based on parsing results
  build_OpenMP_AST(sageFilePtr);

  // Stop here if only OpenMP AST construction is requested
  if (sageFilePtr->get_openmp_ast_only())
    return;

  // Translate to multithreaded code targeting GOMP
  lower_omp(sageFilePtr);
}
```

12.4 Parsing OpenMP Directives

Since the EDG C/C++ frontend (as of version 4.0 and earlier versions) does not support OpenMP, ROSE has its own OpenMP directive parsers. The parsers recognize all OpenMP directives as defined in OpenMP 3.0. The source files of the OpenMP directive parser for C/C++ are located in *rose/src/frontend/SageIII/* . They include *omplexer.ll* (for Lex) and *ompparser.yy* (for Yacc). The Fortran OpenMP parser is hand crafted and has only one source file, *src/frontend/SageIII/ompFortranParser.C*.

Persistent AST attributes (*OmpAttribute* as defined in *src/frontend/SageIII/OmpAttribute.h*) are generated as the results of parsing. The attributes are attached to relevant AST nodes and serve as a light-weight representation for OpenMP directives since they only incur minimum changes to existing ROSE AST.

A set of C/C++/Fortran OpenMP tests is located in *src/tests/nonsmoke/functional/CompileTests/OpenMP_tests* to test the ROSE OpenMP parsers.

12.5 Generating AST with OpenMP Nodes

This phase converts the ROSE AST annotated with OmpAttribute instances into AST with OpenMP-specific nodes. The introduction of OpenMP-specific ROSE AST nodes can help reuse all existing ROSE AST traversal, query, and other manipulation interfaces, thus facilitating analysis and translation of OpenMP programs.

All OpenMP AST nodes have names starting with *SgOmp*. Directives are treated as statement-like nodes, which in turn can have a list of clause nodes. A list of some example OpenMP constructs and their corresponding ROSE AST nodes are given below:

```
omp atomic             SgOmpAtomicStatement
omp barrier            SgOmpBarrierStatement
omp critical           SgOmpCriticalStatement
omp parallel           SgOmpParallelStatement
omp for                SgOmpForStatement
omp task               SgOmpTaskStatement
omp sections           SgOmpSectionsStatement
omp flush              SgOmpFlushStatement
omp taskwait           SgOmpTaskwaitStatement
omp threadprivate      SgOmpThreadprivateStatement

reduction              SgOmpReductionClause
schedule               SgOmpScheduleClause
private                SgOmpPrivateClause
firstprivate           SgOmpFirstprivateClause
lastprivate            SgOmpLastprivateClause
nowait                 SgOmpNowaitClause
copyin                 SgOmpCopyinClause
collapse               SgOmpCollapseClause
untied                 SgOmpUntiedClause
ordered                SgOmpOrderedClause
```

Please refer to the ROSE Web Reference[1] for details about these nodes and their class hierarchy.

More details about the implementation: the conversion was first implemented for C/C++ to translate pragmas annotated with OmpAttribute to OpenMP-specific AST nodes. We later on added implementation for Fortran. But Fortran does not have the concept of pragmas. In order to reuse the conversion code we developed for C/C++, we temporarily generate pragma nodes from Fortran OpenMP directive first and then call the conversion function (as shown below).

```
void build_OpenMP_AST(SgSourceFile *sageFilePtr)
{
  if (is_Fortran_language())
  {
    convert_Fortran_OMP_Comments_to_Pragmas (sageFilePtr);
  }
  convert_OpenMP_pragma_to_AST( sageFilePtr);
}
```

12.6 Translating OpenMP Directives

Using a command line like *identityTranslator -rose:openmp:lower inputcode.c*, ROSE can translate OpenMP 3.0 programs into multithreaded code targeting the XOMP runtime library. XOMP is a thin layer of runtime which is designed to accommodate minor differences among popular OpenMP runtime libraries, such as the GCC GOMP OpenMP runtime library and the Omni OpenMP compiler runtime library. The motivation is to enable ROSE to support more than one runtime libraries, with minimum changes at the XOMP layer if all possible. More details about the design and implementation of XOMP can be found in a paper [39].

If the path to the underlying runtime, such as GOMP (preferably the one shipped with GCC 4.3 or above for OpenMP task support), is specified (using a configure option *–with-gomp_omp_runtime_library=/nfs/apps/gcc/4.4.1/lib64/*), the generated multithreaded code can be automatically linked to the specified library to generate final executables after compilation.

[1] http://www.rosecompiler.org/ROSE_HTML_Reference/index.html

The major source file of the OpenMP translation is *src/midend/ompLowering/omp_lowering.cpp*. Basically, it applies the following algorithm to each input source file using OpenMP:

1. Use a top-down AST traversal to make implicit data-sharing attribute explicit, including implicit private variables for loop constructs and implicit firstprivate variables for task constructs.
2. Uses a bottom-up AST traversal to locate OpenMP nodes and performance necessary translation for each type of them.
   (a) Handle OpenMP-specific variables, such as private, firstprivate, lastprivate and reduction variables used by the target construct, if any.
   (b) For parallel (omp parallel) and task (omp task) constructs, call the general-purpose source-to-source AST outliner [37] to generate tasks and replace the original code block with GOMP runtime calls.
   (c) For loop constructs, normalize target loops and generate code to calculate iteration chunks for each thread.
   (d) Translation for other constructs are relatively simpler cases and details are omitted here.

Many translation tests (from *ROSE_SOURCE_TREE/src/tests/nonsmoke/functional/CompileTests/OpenMP tests*) are provided within the ROSE release, you can type *make check* under *ROSE_BUILD_TREE/tests/nonsmoke/functional/roseTests/ompLoweringTests* to see the OpenMP translation in action.

12.6.1 Variable Handling

The separation of OpenMP variable handling from the rest of translation is the key to the successful reuse of a general-purpose outliner within an OpenMP 3.0 implementation. After OpenMP variable handling, a structured code block of a parallel or task construct becomes much easier to be handled by the outliner.

Variable handling is implemented in OmpSupport::transOmpVariables(). It translates most OpenMP clauses with variable lists, such as private, firstprivate, lastprivate, reduction, etc. (The threadprivate clause is not handled here and will be explained later.) Assume bb is a structured block affected by the variable clauses, the algorithm for handling OpenMP variables is given below:

1. Collect all variables used in clauses with variable lists
2. For each variable, do the following:
   (a) Prepend a local declaration statement for the variable to the beginning of bb.
   (b) Insert an assignment statement after the declaration statement to initialize the local copy (e.g. for firstprivate and reduction variables).
   (c) Replace all references to the variable within bb with references to its local copy.
   (d) Append an assignment statement to save the value of the local copy to its global counter part (e.g. for reduction and lastprivate variables)

Please note that a variable can be associated with more than one clauses, such as firstprivate and lastprivate.

12.6.2 Parallel Regions

Translation of a simple OpenMP parallel region (**#pragma** omp parallel) without any variable uses is demonstrated in Figure 12.2 for an input code shown in Figure 12.1.

Similarly, translation of a simple Fortran OpenMP parallel region is demonstrated in Figure 12.4 for an input code shown in Figure 12.3.

```
/*
 * test the simplest case, no variable handling
By C. Liao
*/
#include <stdio.h>

#ifdef _OPENMP
#include <omp.h>
#endif

int main(void)
{
#pragma omp parallel
  {
    printf("Hello,world!");
  }
  return 0;
}
```

Figure 12.1: Example of a simple parallel region

```
/*
 * test the simplest case, no variable handling
By C. Liao
*/
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#include "libxomp.h"
static void OUT__1__12649__(void *__out_argv);

int main(int argc,char **argv)
{
  int status = 0;
  XOMP_init(argc,argv);
  XOMP_parallel_start(OUT__1__12649__,0,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/functional/Com
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__12649__(void *__out_argv)
{
  printf("Hello,world!");
}
```

Figure 12.2: Translation of a simple parallel region

```
C*************************************************
C test the simplest helloworld in OpenMP Fortan
C  note the different sentinels and lower/upper cases
C Liao, 5/24/2009
C*************************************************
      PROGRAM HELLO
!$OMP PARALLEL
      PRINT *, 'Hello World!'
c$omp end parallel
      PRINT *, 'the end'
      END
```

Figure 12.3: Example of a simple parallel region in Fortran

```
C*************************************************
C test the simplest helloworld in OpenMP Fortan
C  note the different sentinels and lower/upper cases
C Liao, 5/24/2009
C*************************************************
      PROGRAM HELLO
      INTEGER :: status = 0
      external :: OUT__1__13921__
      CALL XOMP_init()
      CALL XOMP_parallel_start(OUT__1__13921__,1,0,0)
      CALL XOMP_parallel_end()
      PRINT *, 'the end'
      CALL XOMP_terminate(status)
      END PROGRAM

      SUBROUTINE OUT__1__13921__(out_argv)
      include "omp_lib.h"
      INTEGER :: out_argv
      PRINT *, 'Hello World!'
      END SUBROUTINE
```

Figure 12.4: Translation of a simple parallel region in Fortran

Translation of a relatively complex OpenMP parallel region with variable references is demonstrated in Figure 12.6 for an input code shown in Figure 12.5. Note the handling of shared, and reduction variables during the translation.

```
#include<assert.h>
#include<omp.h>
#include<stdio.h>

int main(void)
{
  int i =100, sum=100;
  int thread_num;
#pragma omp parallel reduction(+:sum)
  {
#pragma omp single
   {
     thread_num = omp_get_num_threads();
   }
    sum += i;
  }
  printf("thread_num=%d sum =%d\n", thread_num, sum);
  assert(sum == (i*thread_num + 100));
  return 0;
}
```

Figure 12.5: Example of a complex parallel region

Similarly, translation of a Fortran OpenMP parallel region with shared and private variables is demonstrated in Figure 12.8 for an input code shown in Figure 12.7.

```c
#include<assert.h>
#include<omp.h>
#include<stdio.h>
#include "libxomp.h"

struct OUT__1__13886___data
{
  void *i_p;
  void *sum_p;
  void *thread_num_p;
}
;
static void OUT__1__13886__(void *__out_argv);

int main(int argc,char **argv)
{
  int status = 0;
  XOMP_init(argc,argv);
  int i = 100;
  int sum = 100;
  int thread_num;
  struct OUT__1__13886___data __out_argv1__13886__;
  __out_argv1__13886__ . thread_num_p = ((void *)(&thread_num));
  __out_argv1__13886__ . sum_p = ((void *)(&sum));
  __out_argv1__13886__ . i_p = ((void *)(&i));
  XOMP_parallel_start(OUT__1__13886__,&__out_argv1__13886__,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
  printf("thread num=%d sum =%d\n",thread_num,sum);
  sum == i * thread_num + 100?((void )0) : __assert_fail("sum == (i*thread_num + 100)","parallel-reduction.c",
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__13886__(void *__out_argv)
{
  int *i = (int *)(((struct OUT__1__13886___data *)__out_argv) -> i_p);
  int *sum = (int *)(((struct OUT__1__13886___data *)__out_argv) -> sum_p);
  int *thread_num = (int *)(((struct OUT__1__13886___data *)__out_argv) -> thread_num_p);
  int _p_sum;
  _p_sum = 0;
  if (XOMP_single()) {
    *thread_num = omp_get_num_threads();
  }
  XOMP_barrier();
  _p_sum +=  *i;
  XOMP_atomic_start();
   *sum =  *sum + _p_sum;
  XOMP_atomic_end();
}
```

Figure 12.6: Translation of a complex parallel region

```
C*****************************************************
C test the simplest helloworld in OpenMP Fortan
C Liao, 12/7/2010
C*****************************************************
      PROGRAM HELLO
      include 'omp_lib.h'

      INTEGER TID
      INTEGER S
      PARAMETER (S=888)
      REAL K
      K = 0.5
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World by thread ', TID, ' shared variables ', S, K
c$omp end parallel
      PRINT *, 'the end'
      END
```

Figure 12.7: Example of a parallel region with private and shared variables in Fortran

```
C*****************************************************
C test the simplest helloworld in OpenMP Fortan
C Liao, 12/7/2010
C*****************************************************
      PROGRAM HELLO
      include "/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/src/frontend/SageIII/omp_lib.h"
      INTEGER :: TID
      INTEGER :: S
      parameter(S = 888)
      REAL :: K
      INTEGER :: status = 0
      external :: OUT__1__13468__
      CALL XOMP_init()
      K = 0.5
      CALL XOMP_parallel_start(OUT__1__13468__,1,0,2,S,K)
      CALL XOMP_parallel_end()
      PRINT *, 'the end'
      CALL XOMP_terminate(status)
      END PROGRAM

      SUBROUTINE OUT__1__13468__(S,K)
      include "omp_lib.h"
      INTEGER :: S
      REAL :: K
      INTEGER :: i_TID_1
      i_TID_1 = omp_get_thread_num()
      PRINT *, 'Hello World by thread ',i_TID_1,' shared variables ',S,K
      END SUBROUTINE
```

Figure 12.8: Translation of a parallel region with variable accesses in Fortran

Translation of an OpenMP parallel region within a C++ class member function is demonstrated in Figure 12.10 for an input code shown in Figure 12.9. Note the generation of an outlined function using C-bindings and the declaration as a friend function within the original class.

```
#include <iostream>
using namespace std;
class A
{
  private:
    int i;
  public:
    void pararun()
    {
#pragma omp parallel
      {
#pragma omp critical
        cout<<"i= "<< i <<endl;
      }
    }
};
```

Figure 12.9: Example of a parallel region within a class member function

```
static void *xomp_critical_user_;
#include <iostream>
#include "libxomp.h"
using namespace std;

struct OUT__1__13717___data
{
  void *this__ptr___p;
}
;
static void OUT__1__13717__(void *__out_argv);

class A
{
  public: friend void ::OUT__1__13717__(void *__out_argv);
  private: int i;


  public: inline void pararun()
{
// //A declaration for this pointer
    class A *this__ptr__ = this;
    struct OUT__1__13717___data __out_argv1__13717__;
    __out_argv1__13717__ . this__ptr___p = ((void *)(&this__ptr__));
    XOMP_parallel_start(OUT__1__13717__,&__out_argv1__13717__,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-R
    XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/functional/Co
  }
}
;

static void OUT__1__13717__(void *__out_argv)
{
  class A **this__ptr__ = (class A **)(((struct OUT__1__13717___data *)__out_argv) -> this__ptr___p);
  XOMP_critical_start(&xomp_critical_user_);
  ((cout<<"i= ") << (( *this__ptr__) -> i)) << std::endl;
  XOMP_critical_end(&xomp_critical_user_);
}
```

Figure 12.10: Translation of a parallel region within a class member function

12.6.3 Loop Constructs

Translation of a loop construct is given in Figure 12.12 for an input code shown in Figure 12.11. Note that GOMP does not provide a runtime function to calculate iteration chunks for the default schedule policy. Compilers usually have to generate code to calculate the chunks for each thread if they directly target GOMP. We provide a runtime function in the XOMP runtime layer to simplify the translation.

```c
/*
* default loop scheduling
*/
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
int a[20];
int main(void)
{
  int i;
#pragma omp parallel
  {
#pragma omp single
    printf ("Using %d threads.\n",omp_get_num_threads());
#pragma omp for nowait
    for (i=0;i<2;i+=1)
    //for (i=0;i<20;i+=3)
    {
      a[i]=i*2;
      printf("Iteration %2d is carried out by thread %2d\n",\
          i, omp_get_thread_num());
    }
  }
  return 0;
}
```

Figure 12.11: Example of a loop construct

```
/*
* default loop scheduling
*/
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#include "libxomp.h"
int a[20];
static void OUT__1__12682__(void *__out_argv);

int main(int argc,char **argv)
{
  int status = 0;
  XOMP_init(argc,argv);
  int i;
  XOMP_parallel_start(OUT__1__12682__,0,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/functional/Com
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__12682__(void *__out_argv)
{
  if (XOMP_single()) {
    printf("Using %d threads.\n",(omp_get_num_threads()));
  }
  XOMP_barrier();
{
    int _p_i;
    long p_index_;
    long p_lower_;
    long p_upper_;
    XOMP_loop_default(0,1,1,&p_lower_,&p_upper_);
    for (p_index_ = p_lower_; p_index_ <= p_upper_; p_index_ += 1)
//for (i=0;i<20;i+=3)
{
      a[p_index_] = p_index_ * 2;
      printf("Iteration %2d is carried out by thread %2d\n",p_index_,(omp_get_thread_num()));
    }
  }
}
```

Figure 12.12: Translation of a loop construct

Similarly, translation of a Fortran OpenMP DO loop is demonstrated in Figure 12.14 for an input code shown in Figure 12.13.

```
       program  main
       implicit none

       include 'omp_lib.h'

       integer i, j;
!$omp  parallel do private(j)
       do i = 1, 10
         j = omp_get_thread_num();
         print *, "Iteration␣", i, "␣by␣thread:", j
       enddo

       end
```

Figure 12.13: Example of a Fortran Do Loop

```
      PROGRAM main
      IMPLICIT NONE
      include "/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/src/frontend/SageIII/omp
      INTEGER :: i, j
      INTEGER :: status = 0
      external :: OUT__1__14166__
      CALL XOMP_init()
      CALL XOMP_parallel_start(OUT__1__14166__,1,0,0)
      CALL XOMP_parallel_end()
      CALL XOMP_terminate(status)
      END PROGRAM

      SUBROUTINE OUT__1__14166__(out_argv)
      include "omp_lib.h"
      INTEGER :: out_argv
      INTEGER :: p_index_1
      INTEGER :: p_lower_1
      INTEGER :: p_upper_1
      INTEGER :: i_i_2
      INTEGER :: i_j_3
      CALL XOMP_loop_default(1,10,1,p_lower_1,p_upper_1)
      DO p_index_1 = p_lower_1, p_upper_1, 1
      i_j_3 = omp_get_thread_num()
      PRINT *, "Iteration␣",p_index_1,"␣by␣thread:",i_j_3
      END DO
      CALL XOMP_barrier()
      END SUBROUTINE
```

Figure 12.14: Translation of a Fortran Do Loop

A loop with a decremental iteration space is shown in Figure 12.16 for an input code given in Figure 12.15.

XOMP wraps loop scheduling functions from GOMP/Omni for loop constructs with readily known chunk sizes (no calculation is needed) or with an ordered clause. Figure 12.18 shows the translation of a dynamic scheduled loop, for an input code given in Figure 12.17.

```
/*
 * test decremental loop iteration space
 * Liao 9/22/2009
*/
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
void foo(int iend, int ist)
{
  int i;
#pragma omp parallel
  {
#pragma omp single
    printf ("Using %d threads.\n",omp_get_num_threads());

#pragma omp for nowait schedule(static)
    for (i=iend;i>=ist;i--)
    {
      printf("Iteration %d is carried out by thread %d\n",i, omp_get_thread_num());
    }
  }
}
```

Figure 12.15: Example of an OpenMP loop with a decremental iteration space

```
/*
 * test decremental loop iteration space
 * Liao 9/22/2009
*/
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#include "libxomp.h"

struct OUT__1__12735___data
{
  void *iend_p;
  void *ist_p;
}
;
static void OUT__1__12735__(void *__out_argv);

void foo(int iend,int ist)
{
  int i;
  struct OUT__1__12735___data __out_argv1__12735__;
  __out_argv1__12735__.ist_p = ((void *)(&ist));
  __out_argv1__12735__.iend_p = ((void *)(&iend));
  XOMP_parallel_start(OUT__1__12735__,&__out_argv1__12735__,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
}

static void OUT__1__12735__(void *__out_argv)
{
  int *iend = (int *)(((struct OUT__1__12735___data *)__out_argv) -> iend_p);
  int *ist = (int *)(((struct OUT__1__12735___data *)__out_argv) -> ist_p);
  if (XOMP_single()) {
    printf("Using %d threads.\n",(omp_get_num_threads()));
  }
  XOMP_barrier();
{
    int _p_i;
    long p_index_;
    long p_lower_;
    long p_upper_;
    XOMP_loop_default( *iend, *ist,-1,&p_lower_,&p_upper_);
    for (p_index_ = p_lower_; p_index_ >= p_upper_; p_index_ += -1) {
      printf("Iteration %d is carried out by thread %d\n",p_index_,(omp_get_thread_num()));
    }
  }
}
```

Figure 12.16: Translation of the loop with a decremental iteration space

```c
/*
 * Dynamic schedule
 */
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
int a[20];

void foo(int lower, int upper, int stride)
{
  int i;
#pragma omp for schedule(dynamic)
  for (i=lower;i>upper;i-=stride)
  {
    a[i]=i*2;
    printf("Iteration %2d is carried out by thread %2d\n",\
        i, omp_get_thread_num());
  }
}

int main(void)
{
#pragma omp parallel
  {
#pragma omp single
    printf ("Using %d threads.\n",omp_get_num_threads());
   foo(0,20,3);
  }
  return 0;
}
```

Figure 12.17: Example of an OpenMP loop with a dynamic schedule

```c
/*
 * Dynamic schedule
*/
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#include "libxomp.h"
int a[20];

void foo(int lower,int upper,int stride)
{
  int i;
{
    int _p_i;
    long p_index_;
    long p_lower_;
    long p_upper_;
    XOMP_loop_dynamic_init(lower,upper + 1,-1 * stride,1);
    if (XOMP_loop_dynamic_start(lower,upper + 1,-1 * stride,1,&p_lower_,&p_upper_)) {
      do {
        for (p_index_ = p_lower_; p_index_ >= p_upper_; p_index_ += -1 * stride) {
          a[p_index_] = p_index_ * 2;
          printf("Iteration %2d is carried out by thread %2d\n",p_index_,(omp_get_thread_num()));
        }
      }while (XOMP_loop_dynamic_next(&p_lower_,&p_upper_));
    }
    XOMP_loop_end();
  }
}
static void OUT__1__12734__(void *__out_argv);

int main(int argc,char **argv)
{
  int status = 0;
  XOMP_init(argc,argv);
  XOMP_parallel_start(OUT__1__12734__,0,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-we
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__12734__(void *__out_argv)
{
  if (XOMP_single()) {
    printf("Using %d threads.\n",(omp_get_num_threads()));
  }
  XOMP_barrier();
  foo(0,20,3);
}
```

Figure 12.18: Translation of the loop with a dynamic schedule

Finally, translation of multiple Fortran OpenMP DO loop is demonstrated in Figure 12.20 for an input code shown in Figure 12.19. Note that translation-generated loop variables have to be grouped together in the beginning of the Fortran subroutine body, which is not necessary for C or C++.

```fortran
! multiple omp do loops
      program main
      implicit none

      include 'omp_lib.h'

      integer i, j;
!$omp parallel


!$omp do private(j) schedule(static,2)
      do i = 1, 10
        j = omp_get_thread_num();
        print *, "Iteration ", i, " by thread:", j
      enddo

!$omp do private(j) schedule(dynamic,3)
      do i = 1, 10
        j = omp_get_thread_num();
        print *, "Iteration ", i, " by thread:", j
      enddo

!$omp end parallel

      end
```

Figure 12.19: Example of Multiple Fortran Do Loops

```
! multiple omp do loops
      PROGRAM main
       IMPLICIT NONE
       include "/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/src/frontend/SageIII/omp
       INTEGER :: i, j
       INTEGER :: status = 0
       external :: OUT__1__14301__
       CALL XOMP_init()
       CALL XOMP_parallel_start(OUT__1__14301__,1,0,0)
       CALL XOMP_parallel_end()
       CALL XOMP_terminate(status)
       END PROGRAM

       SUBROUTINE OUT__1__14301__(out_argv)
       include "omp_lib.h"
       INTEGER :: out_argv
       INTEGER :: p_index_1
       INTEGER :: p_lower_1
       INTEGER :: p_upper_1
       include "libxompf.h"
       INTEGER :: i_i_2
       INTEGER :: i_j_3
       INTEGER :: p_index_4
       INTEGER :: p_lower_4
       INTEGER :: p_upper_4
       INTEGER :: i_i_5
       INTEGER :: i_j_6
       CALL XOMP_loop_static_init(1,10,1,2)
       IF (XOMP_loop_static_start(1,10,1,2,p_lower_4,p_upper_4) .EQ. 1) THEN
   20  CONTINUE
       DO p_index_4 = p_lower_4, p_upper_4, 1
       i_j_6 = omp_get_thread_num()
       PRINT *, "Iteration ",p_index_4," by thread:",i_j_6
       END DO
       IF (XOMP_loop_static_next(p_lower_4,p_upper_4) .EQ. 1) THEN
       GOTO 20
       ELSE
       END IF
       END IF
       CALL XOMP_loop_end()
       CALL XOMP_loop_dynamic_init(1,10,1,3)
       IF (XOMP_loop_dynamic_start(1,10,1,3,p_lower_1,p_upper_1) .EQ. 1) THEN
   10  CONTINUE
       DO p_index_1 = p_lower_1, p_upper_1, 1
       i_j_3 = omp_get_thread_num()
       PRINT *, "Iteration ",p_index_1," by thread:",i_j_3
       END DO
       IF (XOMP_loop_dynamic_next(p_lower_1,p_upper_1) .EQ. 1) THEN
       GOTO 10
       ELSE
       END IF
       END IF
       CALL XOMP_loop_end()
       END SUBROUTINE
```

Figure 12.20: Translation of Multiple Fortran Do Loops

12.6.4 Threadprivate

GCC uses thread local storage (TLS) to implement OpenMP threadprivate variables. No additional support is needed from the runtime library's point of view. The translation is very simple: add the keyword __thread in front of the original declaration for a variable declared as threadprivate and then remove the OpenMP pragma.

Figure 12.22 shows the translation result for a test input code (Figure 12.21). It also demonstrates the handling of loop constructs using the ordered clause.

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
int counter=0;
#pragma omp threadprivate(counter)
int main(void)
{
 int i;
#pragma omp parallel for ordered
 for(i=0;i<100;i++)
     counter++;
#pragma omp parallel
   printf("counter=%d\n",counter);
   return 0;
}
```

Figure 12.21: Example using threadprivate

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif
#include "libxomp.h"
__thread int counter = 0;
static void OUT__1__13418__(void *__out_argv);
static void OUT__2__13418__(void *__out_argv);

int main(int argc,char **argv)
{
  int status = 0;
  XOMP_init(argc,argv);
  int i;
  XOMP_parallel_start(OUT__2__13418__,0,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-we
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
  XOMP_parallel_start(OUT__1__13418__,0,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-we
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__13418__(void *__out_argv)
{
  printf("counter=%d\n",counter);
}

static void OUT__2__13418__(void *__out_argv)
{
  int _p_i;
  long p_index_;
  long p_lower_;
  long p_upper_;
  XOMP_loop_ordered_static_init(0,99,1,0);
  if (XOMP_loop_ordered_static_start(0,99,1,0,&p_lower_,&p_upper_)) {
    do {
      for (p_index_ = p_lower_; p_index_ <= p_upper_; p_index_ += 1) {
        counter++;
      }
    }while (XOMP_loop_ordered_static_next(&p_lower_,&p_upper_));
  }
  XOMP_loop_end();
}
```

Figure 12.22: Translation of threadprivate

12.6.5 Task Constructs

The translation of task constructs is similar to the translation of parallel constructs. They share the same ROSE AST outliner to generate outlined functions for explicit or implicit tasks.

Figure 12.24 shows the translation of untied task constructs(input code given in Figure 12.23).

```c
#include <stdio.h>
#include <omp.h>

#define LARGE_NUMBER 10
//#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
void process (double input)
{
  printf("processing %f by thread %d\n",input, omp_get_thread_num());
}
int main ()
{
#pragma omp parallel
  {
#pragma omp single
    {
      int i;
      printf("Using %d threads.\n",omp_get_num_threads());
#pragma omp task untied
// i is firstprivate according to implicit rules
      {
        for (i = 0; i < LARGE_NUMBER; i++)
        {
#pragma omp task if(1)
          process (item[i]);
        }
      }
    }
  }
  return 0;
}
```

Figure 12.23: Example of untied tasks

Figure 12.26 shows the translation of task constructs used with taskwait(an input code given in Figure 12.25).

```c
#include <stdio.h>
#include <omp.h>
#define LARGE_NUMBER 10
//#define LARGE_NUMBER 10000000
#include "libxomp.h"
double item[10];

void process(double input)
{
  printf("processing %f by thread %d\n",input,(omp_get_thread_num()));
}

struct OUT__1__13253___data
{
  int i;
}
;
static void OUT__1__13253__(void *__out_argv);

struct OUT__2__13253___data
{
  int i;
}
;
static void OUT__2__13253__(void *__out_argv);
static void OUT__3__13253__(void *__out_argv);

int main(argc,argv)
int argc;
char **argv;
{
  int status = 0;
  XOMP_init(argc,argv);
  XOMP_parallel_start(OUT__3__13253__,0,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-we
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__13253__(void *__out_argv)
{
  int i = (int )(((struct OUT__1__13253___data *)__out_argv) -> i);
  int _p_i = i;
  process(item[_p_i]);
}

static void OUT__2__13253__(void *__out_argv)
{
  int i = (int )(((struct OUT__2__13253___data *)__out_argv) -> i);
  int _p_i = i;
  for (_p_i = 0; _p_i < 10; _p_i++) {
    struct OUT__1__13253___data __out_argv1__13253__;
    __out_argv1__13253__ . i = _p_i;
    XOMP_task(OUT__1__13253__,&__out_argv1__13253__,0,sizeof(struct OUT__1__13253___data ),4,1,0);
  }
}

static void OUT__3__13253__(void *__out_argv)
{
  if (XOMP_single()) {
    int i;
    printf("Using %d threads.\n",(omp_get_num_threads()));
    struct OUT__2__13253___data __out_argv2__13253__;
    __out_argv2__13253__ . i = i;
    XOMP_task(OUT__2__13253__,&__out_argv2__13253__,0,sizeof(struct OUT__2__13253___data ),4,1,1);
  }
  XOMP_barrier();
}
```

Figure 12.24: Translation of the untied tasks

```c
/* Based on A.13.4c, p182 of OMP 3.0 spec.
 * Liao, 9/15/2008
 */
#include <stdio.h>
#include <assert.h>
unsigned long int input = 40;
unsigned long int fib(unsigned long int n)
{
  unsigned long int i, j;
  if (n<2)
    return n;
  else
  {
#pragma omp task shared(i)
    i=fib(n-1);
#pragma omp task shared(j)
    j=fib(n-2);
#pragma omp taskwait
    return i+j;
  }
}
int main()
{
  unsigned long int result = 0;
#pragma omp parallel
  {
   #pragma omp single
    {
      result = fib(input);
    }
  }
  return 0;
}
```

Figure 12.25: Example of tasks with taskwait

```
/* Based on A.13.4c, p182 of OMP 3.0 spec.
 * Liao, 9/15/2008
 */
#include <stdio.h>
#include <assert.h>
#include "libxomp.h"
unsigned long input = 40;

struct OUT__2__12990___data
{
  unsigned long n;
  void *j_p;
}
;
static void OUT__2__12990__(void *__out_argv);

struct OUT__3__12990___data
{
  unsigned long n;
  void *i_p;
}
;
static void OUT__3__12990__(void *__out_argv);

unsigned long fib(unsigned long n)
{
  unsigned long i;
  unsigned long j;
  if (n < 2)
    return n;
   else {
    struct OUT__3__12990___data __out_argv3__12990__;
    __out_argv3__12990__ . i_p = ((void *)(&i));
    __out_argv3__12990__ . n = n;
    XOMP_task(OUT__3__12990__,&__out_argv3__12990__,0,sizeof(struct OUT__3__12990___data ),4,1,0);
    struct OUT__2__12990___data __out_argv2__12990__;
    __out_argv2__12990__ . j_p = ((void *)(&j));
    __out_argv2__12990__ . n = n;
    XOMP_task(OUT__2__12990__,&__out_argv2__12990__,0,sizeof(struct OUT__2__12990___data ),4,1,0);
    XOMP_taskwait();
    return i + j;
  }
}

struct OUT__1__12990___data
{
  void *result_p;
}
;
static void OUT__1__12990__(void *__out_argv);

int main(argc,argv)
int argc;
char **argv;
{
  int status = 0;
  XOMP_init(argc,argv);
  unsigned long result = 0;
  struct OUT__1__12990___data __out_argv1__12990__;
  __out_argv1__12990__ . result_p = ((void *)(&result));
  XOMP_parallel_start(OUT__1__12990__,&__out_argv1__12990__,1,0,"/export/tmp.rose-mgr/jenkins/edg4x/workspace/
  XOMP_parallel_end("/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/tests/nonsmoke/func
  XOMP_terminate(status);
  return 0;
}

static void OUT__1__12990__(void *__out_argv)
{
  unsigned long *result = (unsigned long *)(((struct OUT__1__12990___data *)__out_argv) -> result_p);
  if (XOMP_single()) {
     *result = fib(input);
  }
  XOMP_barrier();
}

static void OUT__2__12990__(void *__out_argv)
{
  unsigned long n = (unsigned long )(((struct OUT__2__12990___data *)__out_argv) -> n);
  unsigned long *j = (unsigned long *)(((struct OUT__2__12990___data *)__out_argv) -> j_p);
  unsigned long _p_n = n;
   *j = fib(_p_n - 2);
}

```

Finally, translation of Fortran OpenMP tasks is demonstrated in Figure 12.28 for an input code shown in Figure 12.27.

```
****************************************
* Simplest OMP task example
* Liao 1/21/2010
****************************************
      program main
      include 'omp_lib.h'
      integer i
      integer item(10)
      external process
      do i = 1, 10
        item(i) = 11-i
      enddo

!$omp parallel
!$omp single
      print *, 'using ', omp_get_num_threads(), ' threads'
      do i = 1, 10
!$omp task
      call process(i,item)
!$omp end task
      enddo
!$omp end single
!$omp end parallel
      end

****************************************
      subroutine process(input,item)
      include 'omp_lib.h'
      integer input
      integer item(10)
      print *, 'idx ', input,'-> ', item(input)
      print *, 'by thread ', omp_get_thread_num()
      end
```

Figure 12.27: Example of Fortran OpenMP Tasks

```
*****************************************
* Simplest OMP task example
* Liao 1/21/2010
*****************************************
      PROGRAM main
      include "/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/src/frontend/SageIII/omp
      INTEGER :: i
      INTEGER, DIMENSION(10) :: item
      external :: process
      INTEGER :: status = 0
      external :: OUT__2__14539__
      CALL XOMP_init()
      DO i = 1, 10
      item(i) = 11 - i
      END DO
      CALL XOMP_parallel_start(OUT__2__14539__,1,0,2,i,item)
      CALL XOMP_parallel_end()
      CALL XOMP_terminate(status)
*****************************************
      END PROGRAM

      SUBROUTINE process(input,item)
      include "/export/tmp.rose-mgr/jenkins/edg4x/workspace/release-ROSE-docs-weekly/src/frontend/SageIII/omp
      INTEGER :: input
      INTEGER, DIMENSION(10) :: item
      PRINT *, 'idx ',input,'-> ',item(input)
      PRINT *, 'by thread ',omp_get_thread_num()
      END SUBROUTINE

      SUBROUTINE OUT__1__14539__(i,item)
      include "omp_lib.h"
      INTEGER :: i
      INTEGER, DIMENSION(10) :: item
      CALL process(i,item)
      END SUBROUTINE

      SUBROUTINE OUT__2__14539__(i,item)
      include "omp_lib.h"
      INTEGER :: i
      INTEGER, DIMENSION(10) :: item
      external :: OUT__1__14539__
      include "libxompf.h"
      IF (XOMP_single() .EQ. 1) THEN
      PRINT *, 'using ',omp_get_num_threads(),' threads'
      DO i = 1, 10
      CALL XOMP_task(OUT__1__14539__,0,16,4,1,0,6,1,4,i,0,8,item)
      END DO
      END IF
      CALL XOMP_barrier()
      END SUBROUTINE
```

Figure 12.28: Translation of Fortran OpenMP Tasks

12.7 Automatic Parallelization

ROSE has an implementation of automatic parallelization using OpenMP. The implementation is also being used to explore semantics-aware automatic parallelization, as described in one of our paper [38]. Out goal is to handle both traditional C/Fortran and modern C++ applications.

As part of an ongoing and evolving work, the automatic parallelization implementation (referred to as the ROSE parallelizer) is not yet integrated into the main source tree of ROSE. The source files are currently located in *rose/projects/autoParallelization*. A standalone executable program (named *autoPar*) is generated and installed to the installation tree of ROSE (under *ROSE_INS/bin*). The program will take in sequential C (or some C++) code and automatically insert OpenMP pragmas into it, if possible.

12.7.1 Algorithm

The ROSE parallelizer is designed to handle both conventional loops operating on primitive arrays and modern applications using high-level abstractions. The parallelizer uses the following algorithm: The loops may contain variables of either primitive data types or STL container types, or both.

1. Preparation and Preprocessing
   (a) Read a specification file for known abstractions and semantics.
   (b) Apply optional custom transformations based on input code semantics, such as converting tree traversals to loop iterations on memory pools.
   (c) Normalize loops, including those using iterators.
   (d) Find candidate array computation loops with canonical forms (for omp for) or loops and functions operating on individual elements (for omp task).
2. For each candidate:
   (a) Skip the target if there are function calls without known semantics or side effects.
   (b) Call dependence analysis and liveness analysis.
   (c) Classify OpenMP variables (autoscoping), recognize references to the current element, and find order-independent write accesses.
   (d) Eliminate dependencies associated with autoscoped variables, those involving only the current elements, and output dependencies caused by order-independent write accesses.
   (e) Insert the corresponding OpenMP constructs if no dependencies remain.

The key idea of the algorithm is to capture dependencies within a target and eliminate them later on as much as possible based on various rules. Parallelization is safe if there are no remaining dependencies. Please refer to one of our papers [38] for the details.

12.7.2 Dependence Analysis

Dependence analysis is the basis for the parallelizer to decide whether a loop is parallelizable. The ROSE parallelizer invokes the dependence analysis from the loop optimizer, which implements algorithms proposed in [35, 36] to effectively transform both perfectly nested loops and non-perfectly nested loops. An extended direction matrix (EDM) dependence representation is used to cover non-common loop nests that surround only one of the two statements in order to handle non-perfectly nested loops. For array accesses within loops, a Gaussian elimination algorithm is used to solve a set of linear integer equations of loop induction variables.

Figure 12.29 gives an example dependence graph dump for an input code, in which a statement is surrounded by two loops ($commonlevel = 2$). Two true dependence relations exist, caused by two pairs of array references and carried in both loop levels ($CarryLevel = 0$ and $CarryLevel = 1$). The extended direction matrices give the dependence directions(one of $=$, $\leq$,$\geq$, and $*$) and alignment factors. The details of the dependence analysis and corresponding graph can be found in [35, 36]. It is clear from the dependence analysis that the example code in Fig. 12.29 cannot be parallelized because of loop-carried dependences in both loop levels.

```
  for (i=1;i<n;i++)
    for (j=1;j<n;j++)
      a[i][j]=a[i][j-1]+a[i-1][j];
/*
dep SgExprStatement @3--> SgExprStatement @3
    2*2 TRUE_DEP; commonlevel = 2 CarryLevel = 1
    SgPntrArrRefExp:a[i])[j] @3:14->SgPntrArrRefExp:a[i][j - 1] @3:19
     == 0; *   0;
     *  0; == -1;

dep SgExprStatement @3--> SgExprStatement @3
    2*2 TRUE_DEP; commonlevel = 2 CarryLevel = 0
    SgPntrArrRefExp:a[i][j]@3:14->SgPntrArrRefExp:a[i - 1][j]@3:31
    == -1;  * 0;
    *  0 ; == 0;
*/
```

Figure 12.29: An example output of ROSE's dependence graph

12.7.3 Variable Classification

Table 12.1 shows the categories of data-sharing attributes for variables based on their live-in (before the execution of a loop) and live-out (after the execution of a loop) analysis results. For instance, a private variable inside a loop is neither live-in nor live-out of the loop, which means the variable is immediately killed (redefined) inside the loop and then used inside the loop somehow, but is never going to be used anywhere after the loop. All loop index variables are also classified as OpenMP private variables to a avoid possible race condition. On the other hand, shared variables are live at both the beginning and the end of the loop. Firstprivate and lastprivate variables are live at either only the beginning or only the end of the loop, respectively.

Table 12.1: OpenMP variable classification based on liveness analysis

Data-sharing attribute	Live-in	Live-out	Written
private	No	No	Yes
firstprivate	Yes	No	No
lastprivate	No	Yes	Yes

Reduction variables are handled specially to maximize the opportunities for parallelization. A typical reduction operation inside a loop, such as **sum** = **sum** + a[i], causes a loop-carried output dependence, a loop-carried anti-dependence, and a loop independent anti-dependence. We use an idiom recognition analysis to capture such typical operations and exclude the associated loop-carried dependences when deciding if a loop is parallelizable.

12.7.4 Examples

Test input codes of the ROSE parallelizer are located in *src/rojects/autoParallelization/tests*. We show a few representative examples here.

Figure 12.31 shows the auto parallelization result generated from an input code given in Figure 12.30). As we can see, the ROSE parallelizer detects all parallelizable loop levels and make private loop index variables explicit. It also reports the parallelized loops and their line numbers during the execution (shown in Figure 12.32).

```
int i,j;
int a[100][100];
void foo()
{
  for (i=0;i<100;i++)
    for (j=0;j<100;j++)
      a[i][j]=a[i][j]+1;
}
```

Figure 12.30: Example of a simple loop

```
#include <omp.h>
int i;
int j;
int a[100][100];

void foo()
{

#pragma omp parallel for private (i,j)
  for (i = 0; i <= 99; i += 1) {

#pragma omp parallel for private (j)
    for (j = 0; j <= 99; j += 1) {
      a[i][j] = a[i][j] + 1;
    }
  }
}
```

Figure 12.31: Parallelized code

```
WARNING: Command line option -rose:C99 is deprecated!
=====================================================
Automatically parallelized a loop at line:5
=====================================================
Automatically parallelized a loop at line:6
```

Figure 12.32: Screen output during the execution of the ROSE parallelizer

Figure 12.34 shows the auto parallelization result generated from an input code given in Figure 12.33). Again, the relevant information is reported during the execution (shown in Figure 12.35). Reasons are given for loops which cannot be automatically parallelized.

Reduction recognition is demonstrated in Figure 12.37 generated from an input code given in Figure 12.36). The ROSE parallelizer is able to recognize reduction variables based on the idiom recognition analysis.

```c
/* Only the inner loop can be parallelized
 */
void foo()
{
  int n=100, m=100;
  double b[n][m];
  int i,j;
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
      b[i][j]=b[i-1][j-1];
}
```

Figure 12.33: Example of a simple loop

```c
/* Only the inner loop can be parallelized
 */
#include <omp.h>

void foo()
{
  int n = 100;
  int m = 100;
  double b[n][m];
  int i;
  int j;
  for (i = 0; i <= n - 1; i += 1) {

#pragma omp parallel for private (j)
    for (j = 0; j <= m - 1; j += 1) {
      b[i][j] = b[i - 1][j - 1];
    }
  }
}
```

Figure 12.34: Parallelized code

```
====================================================
Unparallelizable loop at line:8 due to the following dependencies:
 Distance Matrix size:2*2 TRUE_DEP; commonlevel = 2 CarryLevel = (0,0)
 Is precise SgPntrArrRefExp:b[i][j]@10:11->SgPntrArrRefExp:b[i - 1][j - 1]@10:21
=-1;*0;||*0;=-1;||::
====================================================
Automatically parallelized a loop at line:9
```

Figure 12.35: Screen output during the execution of the ROSE parallelizer

```
/*
 *  Test for automatic recognition of reduction variables
 * */
int a[100], sum;
void foo()
{
  int i,sum2,xx,yy,zz;
  sum = 0;
  for (i=0;i<100;i++)
  {
    a[i]=i;
    sum = a[i]+ sum;
    xx++;
    yy--;
    zz*=a[i];
  }
  sum2=sum+xx+yy+zz;
  a[1]=1;
}
```

Figure 12.36: Example of a simple loop

```
/*
 *  Test for automatic recognition of reduction variables
 * */
#include <omp.h>
int a[100];
int sum;

void foo()
{
  int i;
  int sum2;
  int xx;
  int yy;
  int zz;
  sum = 0;

#pragma omp parallel for private (i) reduction (+:sum,xx) reduction (-:yy) reduction (*:zz)
  for (i = 0; i <= 99; i += 1) {
    a[i] = i;
    sum = a[i] + sum;
    xx++;
    yy--;
    zz *= a[i];
  }
  sum2 = sum + xx + yy + zz;
  a[1] = 1;
}
```

Figure 12.37: Parallelized code

With known semantics of STL vectors, loops operating on vectors can also be parallelized(shown in Figure 12.39 generated from an input code given in Figure 12.38).

```
//test mixed element access member functions
#include <vector>
int main(void)
{
  int i;
  std::vector <int> v1(100);
  for (i=1; i< v1.size(); i++)
    v1.at(i) = v1[i] +1;
  return 0;
}
```

Figure 12.38: Example of a simple loop operating on a vector

```
//test mixed element access member functions
#include <vector>

int main()
{
  int i;
  class std::vector< int  , class std::allocator< int  >  > v1(100);
  for (i = 1; ((unsigned long )i) <= v1 . size() - 1; i += 1) {
    v1 . at(i) = v1[i] + 1;
  }
  return 0;
}
```

Figure 12.39: Parallelized code

12.7.5 More Instructions

An executable file *autoPar* is created during the process of building ROSE. Users can use it to parallelize their codes. *autoPar* will be installed to the path specified by *–prefix=ROSE_INSTALLATION_PATH*. Here are some instructions.

During configuration, you might want to specify a prefix path, such as *–prefix=/home/youraccount/opt/rose-rev-8345*. Then, follow the ROSE installation guide to build and install ROSE, often by typing *make* and *make install*.

The next step is to set environment variables for search paths for ROSE executables (including *autoPar*) and shared libraries. Assume you are using bash, put the following lines into a file (e.g. set.rose) and source it (typing *source set.rose*)

```
ROSE_INS=/home/youraccount/opt/rose-rev-8345
export ROSE_INS
PATH=$ROSE_INS/bin:$PATH
export PATH
LD_LIBRARY_PATH=$ROSE_INS/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Now you can use *autoPar* to parallelize your code. Assume you have a sequential file: *file.c*, just type *autoPar -c file.c*. An output file *rose_file.c* will be generated. If *autoPar* can find anything parallelizable, the output file should have OpenMP pragmas generated.

Chapter 13

UPC Support

13.1 Introduction

ROSE supports Unified Parallel C (UPC) programs. UPC [41] is a famous extension of the C99 programming language to support high performance computing using a partitioned global address space (PGAS) memory model. ROSE leverages the EDG frontend to parse input UPC programs and generate EDG IR with UPC extensions. It then converts the EDG IR into ROSE's internal AST and provides unparsing support for the AST. An example UPC-to-C translator is also provided to demonstrate how one can use ROSE to translate UPC programs into C programs with runtime calls to the Berkeley UPC (BUPC) runtime system V. 2.6.0 [43]. More information on UPC is availalbe from the language specification at: UPC Language Specifications (pdf).

13.2 Supported UPC Constructs

ROSE currently supports all UPC constructs as defined in UPC 1.2, using all the standard header files: *upc.h*, *upc_relaxed.h*, *upc_strict.h*, *upc_collective.h*, *upc_io.h*. Additional Non-standard Berkeley UPC extensions are supported using the UCB *bupc_extensions.h* header file.

[1]. A list of those UPC constructs and their corresponding ROSE AST representations are given below:

```
MYTHREAD              SgUpcMythread
THREADS               SgUpcThreads
upc_barrier           SgUpcBarrierStatement
upc_blocksizeof       SgUpcBlocksizeofExpression
upc_elemsizeof        SgUpcElemsizeofExpression
upc_fence             SgUpcFenceStatement
upc_forall            SgUpcForAllStatement
upc_localsizeof       SgUpcLocalsizeofExpression
upc_notify            SgUpcNotifyStatement
upc_wait              SgUpcWaitStatement
```

[1]The supported version is limited by the EDG frontend, which only supports UPC 1.1.1 (__UPC_VERSION__ string is defined as 200310L). ROSE uses EDG 3.3 currently and it originally only supported UPC 1.0. We merged the UPC 1.1.1 support from EDG 3.10 into our EDG 3.3 frontend. We have also added the required wotk to support UPC 1.2. Please let us know if anything is left out and we will fix it.

```
strict/relaxed/shared   SgUPC_AccessModifier
UPC_MAX_BLOCKSIZE       1073741823 (~1GB)
```

As we can see, most UPC constructs are represented by their corresponding dedicated ROSE AST nodes. A few others, such as strict, relaxed and shared, are represented as instances of SgUPC_AccessModifier. UPC_MAX_BLOCKSIZE is treated as a macro and is expanded to a predefined integer constant value.

Much of the UPC language is supported via UPC specific language runtime libraries. These calls are simply normal functions and not represented as special UPC specific IR nodes. Such runtime library function calls are, in general, not a property of the UPC grammar and thus not represented in the AST as specialized IR nodes. The function names are those from the UPC Langauge Specifications and so must be detected based on the function name. No specialized support in currently present in ROSE to classify these function or identify them as UPC secific, but such functionality could be easily written. Such support could be added to the SageInteface. If users write such specific support they are welcme to contribute it back to ROSE.

Alternatively ROSE provides function identification based on header files where the declarations occured and this would be used to automate the recognition of UPC specific runtime support functions. See the use of the *classifyFileName()* function in *tests/nonsmoke/functional/roseTests/fileLocation_tests* directory for examples of how file names can be classified. Alternatively, just the infomation in the source position reports the file name of the original declaration and this can be used to identify specific functions as defined in there associated header files, e.g. *upc_collective.h*, *upc_io.h*, or *bupc_extensions.h*.

13.3 Command Line Options

ROSE can automatically recognize a source file with a suffix of *.upc* as a UPC input and turn on its UPC support. For other UPC files without the *.upc* extension, a command line option (*-rose:UPC_only or -rose:UPC*) is available to turn on the UPC support explicitly. In addition, *-rose:upc_threads n* can be used to enable ROSE's support for UPC static threads compilation with n threads.

13.4 Example UPC Code Acceptable for ROSE

We give some output after ROSE's source-to-source translation of some example UPC input. These UPC input are actually some of ROSE's daily regression test input available from *ROSE/tests/nonsmoke/functional/CompileTests/UPC_tests*.

Figure 13.1 shows the output of the ROSE identityTranslator handling a hello program in UPC.

```
#include <stdio.h>

int main()
{
  printf("Hello World from thread %d of %d threads\n",MYTHREAD ,1);
  upc_barrier ;
  return 0;
}
```

Figure 13.1: Output of an UPC hello program

Figure 13.2 shows the handling of UPC language constructs related to memory consistency.

Figure 13.3 shows the use of upc_forall with **continue** and Figure 13.4 shows the use of upc_forall with affinity .

```
#include "upc_strict.h"
relaxed shared[1] int array1[100];
strict shared[1] int array2[100];

int main()
{

#pragma upc strict
  return 0;
}
```

Figure 13.2: Output for UPC strict

```
#include <stdio.h>

int main()
{
  int i;
  upc_forall (i = 0;i < 10; i++; continue)
    printf("i=%d\n",i);
  return 0;
}
```

Figure 13.3: Output for upc_forall with continue

```
#include<stdio.h>
#include "upc.h"
shared[2] int arr[10];

int main()
{
  int i;
  upc_forall (i = 0;i < 10; i++; &arr[i])
{
    printf("thread %d of %d threads performing %d iteration.\n",MYTHREAD ,1,i);
  }
/*there is no implicit barrier after upc_forall*/
  upc_barrier ;
/* chunksize is 2 now for loop iteration scheduling */
  for (i = 0; i < 10; i++)
    if ((MYTHREAD ) == upc_threadof((&arr[i])))
      printf("2. thread %d of %d threads performing %d iteration.\n",MYTHREAD ,1,i);
  return 0;
}
```

Figure 13.4: Output for upc_forall with affinity

ROSE's support for various uses of shared and unshared UPC variables is given in Figure 13.5 and Figure 13.6. All kinds of shared, shared to shared, shared to private, and private to shared variables can be correctly parsed and unparsed.

```
/*
examples for shared and unshared, global and static data in UPC
Liao, 7/7/2008
*/
/* ---------- unshared data (TLD)---------------------------*/
/*Unshared global variables , with extern */
extern int quux;
/*unshared global variables: scalar, array, w or w/o initializer */
int counter;
int counter2 = 100;
double myarray[10];
double myarray2[5] = {(0.0), (1.1), (2.2), (3.3), (4.4)};
/*special case: private to shared */
shared[4] int *p2s_p1;
shared[4] int *p2s_p2 = 0;
/*------------shared data (SSD)--------------------------------
  shared scalar, array, initializer
*/
shared[1] int global_counter;
#if 1
// DQ (2/17/2011): Remove the initializaion since EDG reports it as an error with EDG 4.0.
shared[1] int global_counter2;
#else
#endif
/* shared arrays */
shared[5] double array[100];
/* Berkeley UPC compiler does not yet fully implement this. See their bug 36
*/
#if 1
// DQ (2/17/2011): Remove the initializaion since EDG reports it as an error with EDG 4.0.
shared[5] double array2[10];
#else
#endif
/* shared pointers */
/*shared to shared */
shared[1] int *shared[10] s2s_p4;
```

Figure 13.5: Output for UPC shared: part A

```
/*shared to shared */
shared[10] int *shared[1] s2s_p44;
/*shared to shared */
shared[5] int *shared[8] s2s_p444;
/*shared to private */
int *shared[1] s2p_p3;
/*shared to private */
int *shared[5] s2p_p33;

int foo()
{
/* --------unshared static data -------------*/
/* static scalar */
  static int counter;
/* static scalar with initializer */
  static int counter2 = 0;
/* static array */
  static double fooArray[2];
/* static array */
  static double fooArray2[2] = {(3.1), (1.3)};
/* --------shared static data -------------*/
/* static shared scalar */
  static shared[1] int scounter;
#if 1
// DQ (2/17/2011): Remove the initializaion since EDG reports it as an error with EDG 4.0.
/* static shared scalar with initializer */
  static shared[1] int scounter2;
#else
/* static shared scalar with initializer */
#endif
/*static shared array */
  static shared[1] int sfooArray3[5];
#if 1
// DQ (2/17/2011): Remove the initializaion since EDG reports it as an error with EDG 4.0.
  static shared[1] int sfooArray5[5];
#else
```

Figure 13.6: Output for UPC shared: part B

Support for UPC locks is demonstrated in Figure 13.7.

```
/* from UPC Manual Example 2.2.5
*/
#include "upc_relaxed.h"
#include <stdio.h>
/* DQ (1/9/2010): This is a problem for the Intel compiler! */
#ifndef __INTEL_COMPILER
#include <stdlib.h> /*for srand() etc.*/
#endif
#include <math.h>
#define N 1000
shared[5] int arr[1];
upc_lock_t *lock;

int main()
{
  int i = 0;
  int index;
/* DQ (1/9/2010): This is a problem for the Intel compiler! */
#ifndef __INTEL_COMPILER
  srand((MYTHREAD ));
#endif
  if ((lock = upc_all_lock_alloc()) == ((void *)0))
    upc_global_exit(1);
  upc_forall (i = 0;i < 1000; i++; i)
{
/* DQ (1/9/2010): This is a problem for the Intel compiler! */
#ifndef __INTEL_COMPILER
    index = rand() % 1;
#endif
    upc_lock(lock);
    arr[index] += 1;
    upc_unlock(lock);
  }
  upc_barrier ;
  if ((MYTHREAD ) == 0) {
    for (i = 0; i < 1; i++)
      printf("TH%2d: # of arr is %d\n",i,arr[i]);
    upc_lock_free(lock);
  }
  return 0;
}
```

Figure 13.7: Output for UPC Locks

13.5 An Example UPC-to-C Translator Using ROSE

An example UPC-to-C translator, namely roseupcc, is provided to demonstrate how one can use ROSE to build a translator translating UPC programs into C programs with runtime calls to the Berkeley UPC (BUPC) runtime system V. 2.6.0. The source files of roseupcc are located in *ROSE/projects/UpcTranslation*. Please be advised that only a subset of UPC (including variable handling) is translated currently since the translator is meant to be a starting example. Also variable handling is arguably the most difficult part of a UPC implementation.

Translation result for the UPC hello program (shown in Figure 13.1) is given in Figure 13.8. Mostly, high level SageInterface functions are used to easily translate the ROSE AST. BUPC-specified preprocessing directives, such as **#include** "upcr.h" and **#define** UPCR_WANT_MAJOR 3, are inserted. The original user main function is rewritten to user_main with runtime initialization (UPCR_BEGIN_FUNCTION) and termination (UPCR_EXIT_FUNCTION) functions. upc_barrier is simply replaced with a call to its corresponding runtime function upcr_barrier(). UPCRI_ALLOC_filename_xxx() handles per file UPCRI allocation of possible shared variables (none here). UPCRI_INIT_filename_xxx() function is used for per-file initialization of shared and unshared (thread local) data.

Implementing various UPC variable accesses is complex due to the two level memory scheme used by the partitioned global memory address space. The translation has to handle variable declaration, memory allocation, value initialization, variable access, and so on.

UPC variables can be roughly categorized as shared and unshared (or thread local) variables. For shared variables, they can be divided into two categories: statically allocated shared variables (including global shared variables and local static shared variables) and dynamically allocated shared variables. We focus on statically allocated shared variables here as an example. Translation of statically allocated shared variables is demonstrated in Figure 13.10 and Figure 13.11 for an input code shown in Figure 13.9.

The BUPC runtime distinguishes between phaseless (blocksize ==1 or 0 or unspecified) and phased (all other cases) shared pointers for better performance. So two types of global scoped proxy pointers (upcr_pshared_ptr_t and upcr_shared_ptr_t) are used to represent static shared variables. Global static shared variables directly use their names as the proxy pointer names (such as global_counter). Local static variables use their mangled name (e.g. _lsscounter3769188422_) for those pointers to avoid name collision. Accesses to shared variables are implemented by a set of runtime library calls, such as UPCR_GET_SHARED() and UPCR_PUT_SHARED_VAL. Again, as shown in Figure 13.11, UPCRI_ALLOC_filename_xxx() handles per file UPCRI allocation of shared variables and UPCRI_INIT_filename_xxx() function is used for per-file initialization of those data.

ROSE provides a set of AST interface functions to help developers handle UPC-specific types and facilitate translation. Those functions include bool isUpcSharedType(), bool isUpcSharedArrayType(), bool isUpcPhaseLessSharedType(), bool isUpcPrivateToSharedType(), etc. A type mangling function (mangleType()) is also provided to implement the Itanium C++ ABI specification [45].

Unshared variables in UPC (also called Thread local data, or TLD) consists of local auto variables and global (and static local) variables. Local auto variables do not need special translation. But global and static local variables do. Implementation details for them vary depending on their scope, internal or external linkage, if they are scalar or array types, if they point to shared data, if they have initializers, and so on. But the basic scheme for variable declaration, allocation/initialization, and accesses is similar to the handling of shared UPC variables. Please refer to the BUPC runtime interface specification 3.10 [44] for details. We only provide a translation example in Figure 13.13 and Figure 13.14 for an input code shown in Figure 13.12.

```
#define UPCR_WANT_MAJOR 3
#define UPCR_WANT_MINOR 6
#define UPCR_SHARED_SIZE_ 8
#define UPCR_PSHARED_SIZE_ 8
#include <stdio.h>
#include "upcr.h"
#include "upcr_proxy.h"
extern int upcrt_gcd(int a,int b);
extern int _upcrt_forall_start(int start_thread, int step, int lo UPCRI_PT_ARG);
#define upcrt_forall_start(start_thread, step, lo)  \
        _upcrt_forall_start(start_thread, step, lo UPCRI_PT_PASS)
int32_t UPCR_TLD_DEFINE_TENTATIVE(upcrt_forall_control, 4, 4);
#define upcr_forall_control upcrt_forall_control
#ifndef UPCR_EXIT_FUNCTION
#define UPCR_EXIT_FUNCTION() ((void)0)
#endif
#if UPCR_RUNTIME_SPEC_MAJOR > 3 || (UPCR_RUNTIME_SPEC_MAJOR == 3 && UPCR_RUNTIME_SPEC_MINOR >= 8)
#define UPCRT_STARTUP_SHALLOC(sptr, blockbytes, numblocks, mult_by_threads, elemsz, typestr) \
        { &(sptr), (blockbytes), (numblocks), (mult_by_threads), (elemsz), #sptr, (typestr) }
#else
#define UPCRT_STARTUP_SHALLOC(sptr, blockbytes, numblocks, mult_by_threads, elemsz, typestr) \
        { &(sptr), (blockbytes), (numblocks), (mult_by_threads) }
#endif
#define UPCRT_STARTUP_PSHALLOC UPCRT_STARTUP_SHALLOC

extern int user_main(int argc,char **argv)
{
  UPCR_BEGIN_FUNCTION();
  printf("Hello World from thread %d of %d threads\n",(upcr_mythread()),(upcr_threads()));
  upcr_barrier(,0);
  UPCR_EXIT_FUNCTION();
  return 0;
}

void UPCRI_ALLOC_hello_15483911821992209560()
{
  UPCR_BEGIN_FUNCTION();
  UPCR_SET_SRCPOS("_hello_15483911821992209560_ALLOC",0);
  UPCR_EXIT_FUNCTION();
}

void UPCRI_INIT_hello_15483911821992209560()
{
  UPCR_BEGIN_FUNCTION();
  UPCR_SET_SRCPOS("_hello_15483911821992209560_INIT",0);
  UPCR_EXIT_FUNCTION();
}
```

Figure 13.8: Translation of upc hello

```
/*------------global shared data (SSD)--------------------------------*/
/*  shared scalar */
shared int global_counter;
shared int global_counter1 = 0;
shared int global_counter2 = 2;

/* shared arrays */
shared[5] double garray[100*THREADS];
/* Berkeley UPC compiler does not yet fully implement this: their bug 36
shared[5] double garray2[10*THREADS]={1.1, 2.2};
*/

/* shared pointers */
shared int* shared[10] s2s_p4; /*shared to shared */
shared[10] int* shared s2s_p44;
shared[5] int* shared[8] s2s_p444;

int *shared s2p_p3; /*shared to private */
int *shared[5] s2p_p33; /*shared to private */

int foo()
{
/* -------local shared static data -----------*/
  static shared int lsscounter; /* static shared scalar */
  static shared int lsscounter1 =0; /* static shared scalar with initializer */
  static shared int lsscounter2 =77; /* static shared scalar with initializer */

  /*static shared array */
  static shared int lssfooArray3[5*THREADS];
 /* The translation is not implemented by the Berkeley UPC
  static shared int lssfooArray5[5*THREADS] = {1,2,3,4,5};
  */

/*Write reference to a shared variable */
  lsscounter=99;
  return 0;
}
int main()
{
 return 0;
}
```

Figure 13.9: Example input for shared variables

```
#define UPCR_WANT_MAJOR 3
#define UPCR_WANT_MINOR 6
#define UPCR_SHARED_SIZE_ 8
#define UPCR_PSHARED_SIZE_ 8
/*------------global shared data (SSD)--------------------------------*/
/*  shared scalar */
#include "upcr.h"
#include "upcr_proxy.h"
extern int upcrt_gcd(int a,int b);
extern int _upcrt_forall_start(int start_thread, int step, int lo UPCRI_PT_ARG);
#define upcrt_forall_start(start_thread, step, lo)  \
        _upcrt_forall_start(start_thread, step, lo UPCRI_PT_PASS)
int32_t UPCR_TLD_DEFINE_TENTATIVE(upcrt_forall_control, 4, 4);
#define upcr_forall_control upcrt_forall_control
#ifndef UPCR_EXIT_FUNCTION
#define UPCR_EXIT_FUNCTION() ((void)0)
#endif
#if UPCR_RUNTIME_SPEC_MAJOR > 3 || (UPCR_RUNTIME_SPEC_MAJOR == 3 && UPCR_RUNTIME
_SPEC_MINOR >= 8)
#define UPCRT_STARTUP_SHALLOC(sptr, blockbytes, numblocks, mult_by_threads, elem
sz, typestr) \
        { &(sptr), (blockbytes), (numblocks), (mult_by_threads), (elemsz), #sptr
, (typestr) }
#else
#define UPCRT_STARTUP_SHALLOC(sptr, blockbytes, numblocks, mult_by_threads, elem
sz, typestr) \
        { &(sptr), (blockbytes), (numblocks), (mult_by_threads) }
#endif
#define UPCRT_STARTUP_PSHALLOC UPCRT_STARTUP_SHALLOC
upcr_pshared_ptr_t global_counter;
upcr_pshared_ptr_t global_counter1 = UPCR_INITIALIZED_PSHARED;
upcr_pshared_ptr_t global_counter2 = UPCR_INITIALIZED_PSHARED;
/* shared arrays */
upcr_shared_ptr_t garray;
/* Berkeley UPC compiler does not yet fully implement this: their bug 36
shared[5] double garray2[10*THREADS]={1.1, 2.2};
*/
/* shared pointers */
/*shared to shared */
upcr_shared_ptr_t s2s_p4;
upcr_pshared_ptr_t s2s_p44;
upcr_shared_ptr_t s2s_p444;
/*shared to private */
upcr_pshared_ptr_t s2p_p3;
/*shared to private */
upcr_shared_ptr_t s2p_p33;
upcr_pshared_ptr_t _lsscounter_10146142690091286598_;
upcr_pshared_ptr_t _lsscounter1_15710525892080606013_ = UPCR_INITIALIZED_PSHARED
;
upcr_pshared_ptr_t _lsscounter2_15710525892080606010_ = UPCR_INITIALIZED_PSHARED
;
upcr_pshared_ptr_t _lssfooArray3_7126625236460005300_;

int foo()
{
  UPCR_BEGIN_FUNCTION();
/* The translation is not implemented by the Berkeley UPC
  static shared int lssfooArray5[5*THREADS] = {1,2,3,4,5};
  */
/*Write reference to a shared variable */
  UPCR_PUT_PSHARED_VAL(_lsscounter_10146142690091286598_,0,99,8);
  UPCR_EXIT_FUNCTION();
  return 0;
}
/* --------local shared static data ------------*/
/* static shared scalar */
/* static shared scalar with initializer */
/* static shared scalar with initializer */
/*static shared array */

extern int user_main()
{
  UPCR_BEGIN_FUNCTION();
```

Figure 13.10: Translation of UPC shared variables, part A

```
  UPCR_EXIT_FUNCTION();
  return 0;
}

void UPCRI_ALLOC_shared_1_14391433639748249774()
{
  UPCR_BEGIN_FUNCTION();
  upcr_startup_shalloc_t info[] = {UPCRT_STARTUP_SHALLOC(garray,40,20,1,8,"A100H
_Invalid.mtype"), UPCRT_STARTUP_SHALLOC(s2s_p4,8,1,0,8,"R10_PInvalid.mtype"), UP
CRT_STARTUP_SHALLOC(s2s_p444,8,1,0,8,"R8_PInvalid.mtype"), UPCRT_STARTUP_SHALLOC
(s2p_p33,4,1,0,4,"R5_Pi")};
  upcr_startup_pshalloc_t pinfo[] = {UPCRT_STARTUP_PSHALLOC(_lsscounter_10146142
690091286598_,4,1,0,4,"Invalid.mtype"), UPCRT_STARTUP_PSHALLOC(_lsscounter1_1571
0525892080606013_,4,1,0,4,"Invalid.mtype"), UPCRT_STARTUP_PSHALLOC(_lsscounter2_
15710525892080606010_,4,1,0,4,"Invalid.mtype"), UPCRT_STARTUP_PSHALLOC(_lssfooAr
ray3_7126625236460005300_,4,5,1,4,"A5H_Invalid.mtype"), UPCRT_STARTUP_PSHALLOC(g
lobal_counter,4,1,0,4,"Invalid.mtype"), UPCRT_STARTUP_PSHALLOC(global_counter1,4
,1,0,4,"Invalid.mtype"), UPCRT_STARTUP_PSHALLOC(global_counter2,4,1,0,4,"Invalid
.mtype"), UPCRT_STARTUP_PSHALLOC(s2s_p44,8,1,0,8,"R1_PInvalid.mtype"), UPCRT_STA
RTUP_PSHALLOC(s2p_p3,4,1,0,4,"R1_Pi")};
  UPCR_SET_SRCPOS("_shared_1_14391433639748249774_ALLOC",0);
  upcr_startup_shalloc(info,sizeof(info) / sizeof(upcr_startup_shalloc_t ));
  upcr_startup_pshalloc(pinfo,sizeof(pinfo) / sizeof(upcr_startup_pshalloc_t ));
  UPCR_EXIT_FUNCTION();
}

void UPCRI_INIT_shared_1_14391433639748249774()
{
  UPCR_BEGIN_FUNCTION();
  UPCR_SET_SRCPOS("_shared_1_14391433639748249774_INIT",0);
  int global_counter1_val = 0;
  if (upcr_mythread() == 0) {
    upcr_put_pshared(global_counter1,0,&global_counter1_val,8);
  }
  int global_counter2_val = 2;
  if (upcr_mythread() == 0) {
    upcr_put_pshared(global_counter2,0,&global_counter2_val,8);
  }
  int _lsscounter1_15710525892080606013__val = 0;
  if (upcr_mythread() == 0) {
    upcr_put_pshared(_lsscounter1_15710525892080606013_,0,&_lsscounter1_15710525
892080606013__val,8);
  }
  int _lsscounter2_15710525892080606010__val = 77;
  if (upcr_mythread() == 0) {
    upcr_put_pshared(_lsscounter2_15710525892080606010_,0,&_lsscounter2_15710525
892080606010__val,8);
  }
  UPCR_EXIT_FUNCTION();
}
```

Figure 13.11: Translation of UPC shared variables, part B

```c
/*
examples for unshared data in UPC
Liao, 9/3/2008
*/
/*shared one*/
shared int gsj;

/* ---------- unshared data (TLD)--------------------------*/
/*Unshared global variables , with extern */
 extern int quux;

/*unshared global variables: scalar */
int counter;
int counter1 = 0;
int counter2 = 100;

/*unshared global arrays, w or w/o initializer */
double myarray[10];
double myarray2[5]={0.0, 1.1, 2.2,3.3,4.4};

/* structure */
struct mystruct
        {
        char name[50];
        float calibre;
        };
struct mystruct ms1;

/* union , with embedded declaration*/
union item_u
{
  int i;
  float f;
  char c;
} item;

int foo()
{
/* -------unshared static data -----------*/
  static int lscounter; /* static scalar */
  static int lscounter1 =0; /* static scalar with initializer */
  static int lscounter2 =77; /* static scalar with initializer */

  static double lsfooArray [2]; /* static array */
  static double lsifooArray2 [2] = {3.1, 1.3}; /* static array */

/*reference to local static unshared variable*/
  lscounter =99;
  return 0;
}
int bar()
{
  return 0;
 }
int main()
{
/* references to global unshared variables*/
  counter ++;

  item.i = 9;
  ms1.calibre = 0.7;
  return 0;
}
```

Figure 13.12: Example input for non-shared variables

```
#define UPCR_WANT_MAJOR 3
#define UPCR_WANT_MINOR 6
#define UPCR_SHARED_SIZE_ 8
#define UPCR_PSHARED_SIZE_ 8
/*
examples for unshared data in UPC
Liao, 9/3/2008
*/
/*shared one*/
#include "upcr.h"
#include "upcr_proxy.h"
extern int upcrt_gcd(int a,int b);
extern int _upcrt_forall_start(int start_thread, int step, int lo UPCRI_PT_ARG);
#define upcrt_forall_start(start_thread, step, lo)  \
        _upcrt_forall_start(start_thread, step, lo UPCRI_PT_PASS)
int32_t UPCR_TLD_DEFINE_TENTATIVE(upcrt_forall_control, 4, 4);
#define upcr_forall_control upcrt_forall_control
#ifndef UPCR_EXIT_FUNCTION
#define UPCR_EXIT_FUNCTION() ((void)0)
#endif
#if UPCR_RUNTIME_SPEC_MAJOR > 3 || (UPCR_RUNTIME_SPEC_MAJOR == 3 && UPCR_RUNTIME
_SPEC_MINOR >= 8)
#define UPCRT_STARTUP_SHALLOC(sptr, blockbytes, numblocks, mult_by_threads, elem
sz, typestr) \
        { &(sptr), (blockbytes), (numblocks), (mult_by_threads), (elemsz), #sptr
, (typestr) }
#else
#define UPCRT_STARTUP_SHALLOC(sptr, blockbytes, numblocks, mult_by_threads, elem
sz, typestr) \
        { &(sptr), (blockbytes), (numblocks), (mult_by_threads) }
#endif
#define UPCRT_STARTUP_PSHALLOC UPCRT_STARTUP_SHALLOC
upcr_pshared_ptr_t gsj;
/* ---------- unshared data (TLD)---------------------------*/
/*Unshared global variables , with extern */
extern int quux;
/*unshared global variables: scalar */
upcr_shared_ptr_t counter;
/* int UPCR_TLD_DEFINE_TENTATIVE (counter, 4, 4 ); */
int counter1 = 0;
/* int UPCR_TLD_DEFINE (counter1, 4, 4 ) = 0; */
int counter2 = 100;
/* int UPCR_TLD_DEFINE (counter2, 4, 4 ) = 100; */
/*unshared global arrays, w or w/o initializer */
typedef double _type_myarray[10];
_type_myarray myarray;
/* _type_myarray UPCR_TLD_DEFINE_TENTATIVE (myarray, 80, 4 ); */
typedef double _type_myarray2[5];
_type_myarray2 myarray2 = {(0.0), (1.1), (2.2), (3.3), (4.4)};
/* _type_myarray2 UPCR_TLD_DEFINE (myarray2, 40, 4 ) = {(0.0),(1.1),(2.2),(3.3),
(4.4)}; */
/* structure */

struct mystruct
{
  char name[50];
  float calibre;
}
;
upcr_shared_ptr_t ms1;
/* struct mystruct UPCR_TLD_DEFINE_TENTATIVE (ms1, 0, 0 ); */
/* union , with embedded declaration*/

union item_u
{
```

Figure 13.13: Translation of UPC unshared variables, part A

```
  int i;
  float f;
  char c;
}
;
upcr_shared_ptr_t item;
/* union item_u {int i;float f;char c;} UPCR_TLD_DEFINE_TENTATIVE (item, 0, 0 );
 */
upcr_shared_ptr_t _lscounter_15598860026086995447_;
/* int UPCR_TLD_DEFINE_TENTATIVE (_lscounter_15598860026086995447_, 4, 4 ); */
int _lscounter1_10145199976516849872_ = 0;
/* int UPCR_TLD_DEFINE (_lscounter1_10145199976516849872_, 4, 4 ) = 0; */
int _lscounter2_10145199976516849887_ = 77;
/* int UPCR_TLD_DEFINE (_lscounter2_10145199976516849887_, 4, 4 ) = 77; */
typedef double _type__lsfooArray_10145424818072310501_[2];
_type__lsfooArray_10145424818072310501_ _lsfooArray_10145424818072310501_;
/* _type__lsfooArray_10145424818072310501_ UPCR_TLD_DEFINE_TENTATIVE (_lsfooArra
y_10145424818072310501_, 16, 4 ); */
typedef double _type__lsifooArray2_6727348945159008206_[2];
_type__lsifooArray2_6727348945159008206_ _lsifooArray2_6727348945159008206_ = {(
3.1), (1.3)};
/* _type__lsifooArray2_6727348945159008206_ UPCR_TLD_DEFINE (_lsifooArray2_67273
48945159008206_, 16, 4 ) = {(3.1),(1.3)}; */

int foo()
{
  UPCR_BEGIN_FUNCTION();
/*reference to local static unshared variable*/
   *((int *)(UPCR_TLD_ADDR(_lscounter_15598860026086995447_))) = 99;
  UPCR_EXIT_FUNCTION();
  return 0;
}
/* -------unshared static data -----------*/
/* static scalar */
/* static scalar with initializer */
/* static scalar with initializer */
/* static array */
/* static array */

int bar()
{
  UPCR_BEGIN_FUNCTION();
  UPCR_EXIT_FUNCTION();
  return 0;
}

extern int user_main()
{
  UPCR_BEGIN_FUNCTION();
/* references to global unshared variables*/
  ( *((int *)(UPCR_TLD_ADDR(counter))))++;
  ( *((union item_u *)(UPCR_TLD_ADDR(item)))) . i = 9;
  ( *((struct mystruct *)(UPCR_TLD_ADDR(ms1)))) . calibre = 0.7;
  UPCR_EXIT_FUNCTION();
  return 0;
}

void UPCRI_ALLOC_unshared_1_16481913609346533702()
{
  UPCR_BEGIN_FUNCTION();
  upcr_startup_pshalloc_t pinfo[] = {UPCRT_STARTUP_PSHALLOC(gsj,4,1,0,4,"Invalid
.mtype")};
  UPCR_SET_SRCPOS("_unshared_1_16481913609346533702_ALLOC",0);
  upcr_startup_pshalloc(pinfo,sizeof(pinfo) / sizeof(upcr_startup_pshalloc_t ));
  UPCR_EXIT_FUNCTION();
```

Figure 13.14: Translation of UPC unshared variables, part B

Chapter 14

Binary Analysis: Support for the Analysis of Binary Executables

14.1 Introduction

ROSE supports the disassembly and analysis of binary executables for x86, PowerPC, and AMR instruction sets. ROSE implements this support as part of general research work to support combining analysis for source code and analysis for binaries and supporting performance analysis and optimization. Through this support ROSE addresses the requirements for the analysis and transformation of software in a general context useful to as wide a group of users as possible.

ROSE handles a number of binary executable file formats and also reads Dwarf information into the AST to support additional analysis.

Recent work in ROSE has added support for dynamic analysis and for mixing of dynamic and static analysis using the Intel Pin framework. Intel Pin support in ROSE is presented in section 14.6.

14.2 The Binary AST

14.2.1 The Binary Executable Format

ROSE handles Linux and Windows binary formats; thus ELF format for Linux and PE, NE, LE, DOS formats for Windows. The details of each format are represented in IR nodes in the AST (using structures common to the representation of such low level data). About 60 IR nodes have been added to ROSE to support the binary executable formats; this support allows the analysis of any Linux, Windows, OS2, or DOS binary executable.

The binary executable file format can be analyzed separately from the instructions using the command line option: `-rose:read_executable_file_format_only`. This allows graphs generated using the ROSE visualization mechanisms (and even some analysis) to be easily restricted (in size) to the just the IR nodes specific to the binary executable file format.

Figure 14.1 shows the class design of the IR nodes for the binary file format address the requirements of ELF (Linux, and others), PE (MS Windows), NE (older MS Windows), LE (OS2), and DOS (MS Dos) executable formats. The colors represent different executable formats, brown classes are used as base classes for more than one format. Dark colors represent principle IR nodes in the AST, lighter color IR nodes represent supporting

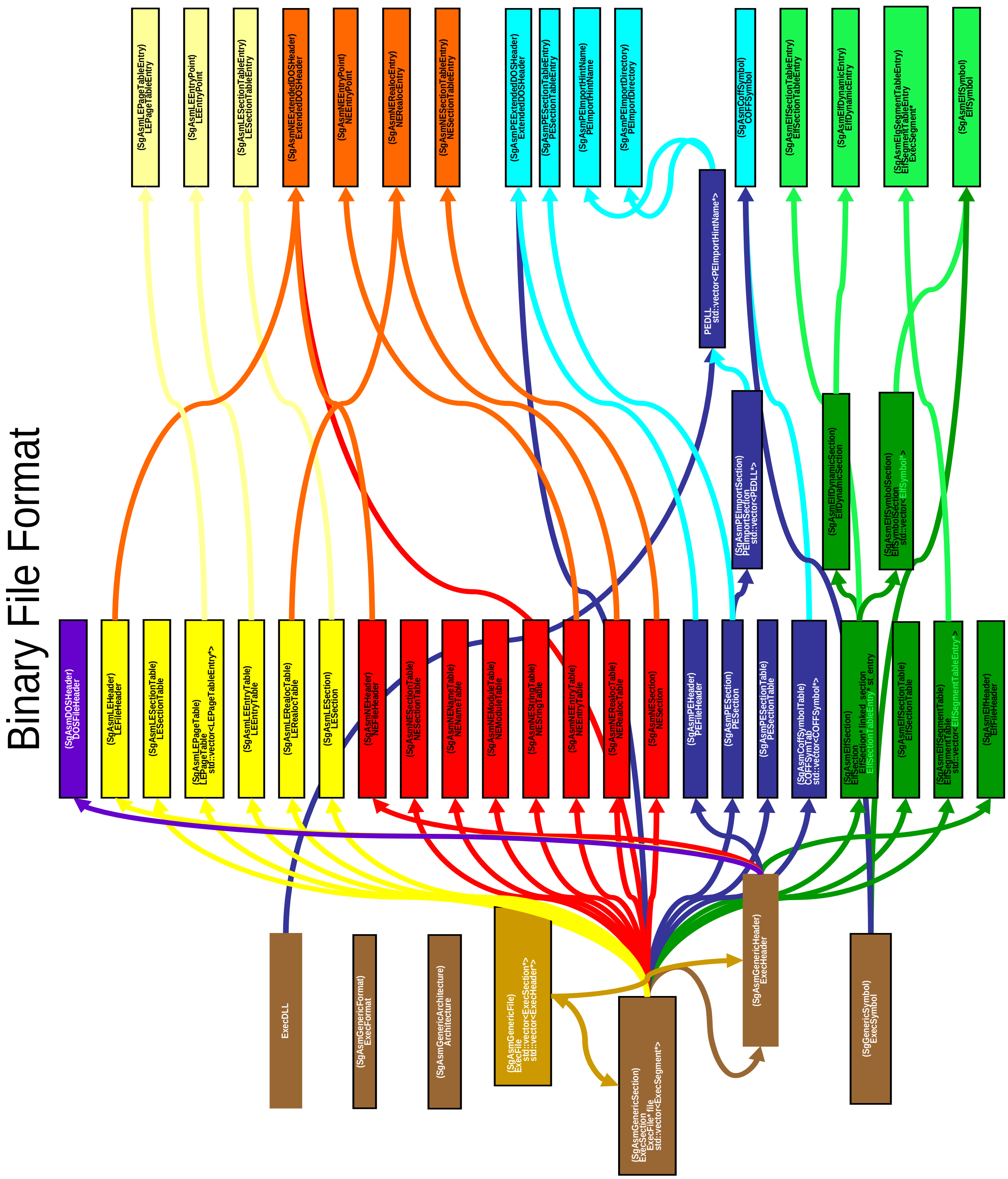

Figure 14.1: The class design of the IR nodes for the binary file format.

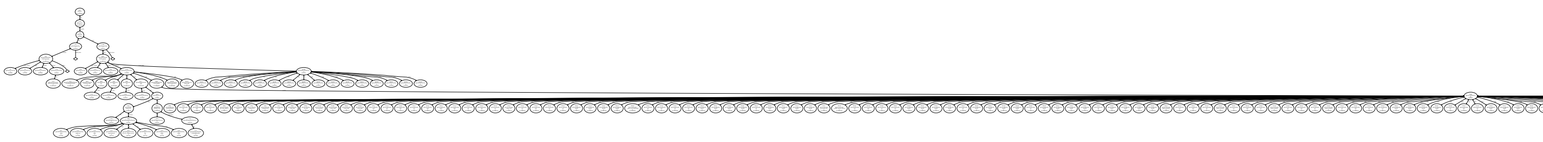

Figure 14.2: The AST for a PE (Windows) binary executable (binary file format only), with long list of symbols (half of which are clipped on the right side of the image).

infrastructure in the AST. Arrows are either dark colored or light colored; dark colors represent class derivation, and light colors represent member relationships.

Figure 14.2 shows the graph of the AST formed by just the binary file format (sections, symbols, etc.). This figure shows the different IR nodes used to represent the binary file format (without the disassembled instructions, in this case) and the large list of symbols within the symbol table (the long list that is partially truncated on the right edge of the figure). This graph is quite large and does not fit well on the page, the next figure 14.3 shows a clipped image with more detail.

Figure 14.3 shows a cropped view of the graph of the AST formed by just the binary file format (sections, symbols, etc.). This figure shows two *SgAsmInterpretation* IR nodes; this is a Windows PE binary and all windows PE binaries contain both a 16-bit MS-DOS header (and some 16-bit code) and a 32-bit Windows PE header (with sections that have 32-bit code); thus there are two SgAsmInterpretation IR nodes (one for the 16-bit interpretation of the instructions and one for the 32-bit interpretation of the instructions). Note that ELF format files will have only one SgAsmInterpretation IR nodes (because there is only a single interpretation possible), but other file formats can contain many interpretations; formed form a composite of code to support wide ranges of portability.

*Note: The actually file used is in these figures is: `ROSE/docs/Rose/asm_code_samples_gcc.pdf`.

14.2.2 Instruction Disassembly

ROSE has its own disassembler (for x86, ARM, and PowerPC); a recursive disassembler that is well suited to details of variable length instruction set handling and data stored in the instruction stream. All details of the instructions, and the operands and operator expression trees, etc. are stored in the binary AST as separate IR nodes. The SgAsmInstruction class and its architecture-specific subclasses represent individual instructions. The arguments for those instructions are represented by the SgAsmExpression class and subclasses thereof.

FIXME: *We need an the AST for a few in*

Disassembly normally happens automatically unless the `-rose:read_executable_file_format_only` switch is specified. Alternatively, this Disassembler class can be used to explicitly disassemble parts of a file. The Disassembler class handles all non-architecture-specific details of disassembly, such as where to search for instructions in the address space and how instructions are concatenated into basic blocks. The Disassembler has a pure virtual method, disassembleOne, that is implemented by architecture-specific subclasses and whose purpose is to disassemble one instruction.

New architectures can be added to ROSE without modifying any ROSE source code. One does this by subclassing an existing disassembler, overriding any necessary virtual methods, and registering an instance of this subclass with the `Disassembler::register_subclass` method. If the new subclass can handle multiple architectures then a disassembler is registered for each of those architectures.

When ROSE needs to disassemble something, it calls `Disassembler::lookup`, which in turn calls the `can_disassemble` method for all registered disassemblers. The first disassembler whose `can_disassemble` re-

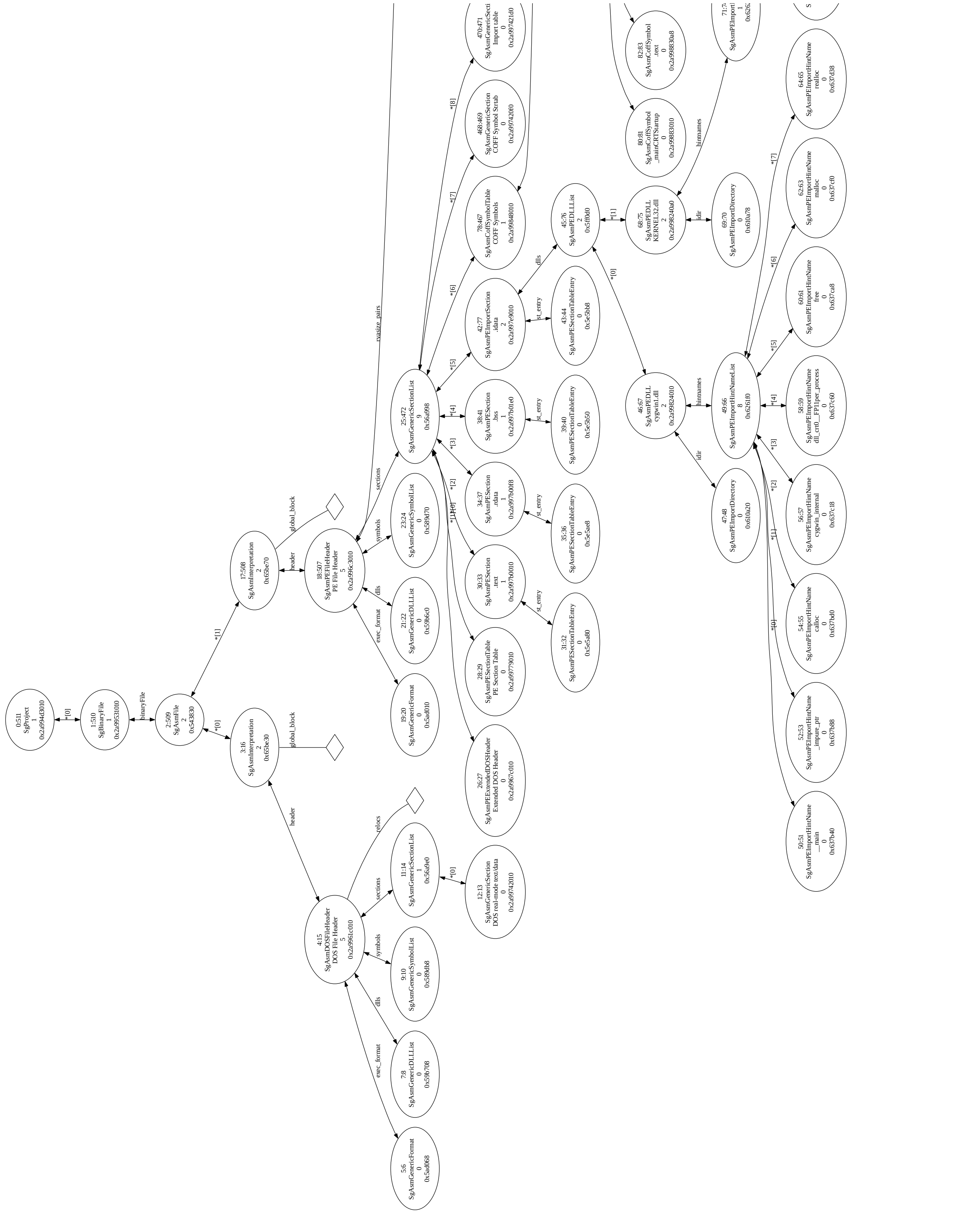

Figure 14.3: The CROPPED AST for a PE (Windows) binary executable (binary file format only).

turns true is used for the disassemby.

If an error occurs during the disassembly of a single instruction, the disassembler will throw an exception. When disassembling multiple instructions the exceptions are saved in a map, by virtual address, and the map is returned to the caller along with the instructions that were successfully disassembled.

The main interface to the disassembler is the `disassembleBuffer` method. It searches for instructions based on the heuristics specified in the `set_search` method, reading instruction bytes from a supplied buffer. A MemoryMap object is supplied in order to specify a mapping from virtual address space to offsets in the supplied buffer. The `disassembleBuffer` method is used by methods that disassemble whole sections, whole interpretations, or whole files; in turn, it calls `disassembleBlock` which disassembles sequential instructions until a control flow branch is encountered.

A MemoryMap object can be built that describes the entire virtual address space and how it relates to offsets in the executable file. This object, together with the entire contents of the file, can be passed to the `disassembleBuffer` method in order to disassemble the entire executable in one call. However, if the executable contains multiple independent interpretations (like a PE file that contains a Windows executable and a DOS executable) then the best practice is to disassemble each interpretation individually. The `disassemble` method is convenient for this.

While the main purpose of the Disassembler class is to disassemble instructions, it also needs to be able to group those instructions into basic blocks (SgAsmBlock) and functions (SgAsmFunction). It uses an instance of the Partitioner class to do so. The user can supply a partitioner to the disassembler or have the disassembler create a default partitioner on the fly. The user is also free to call the partitioner directly on the InstructionMap object returned by most of the disassembler methods.

14.2.3 Instruction Partitioning

While the main purpose of the Disassembler class is to disassemble instructions, it also needs to be able to group those instructions into basic blocks (SgAsmBlock) and functions (SgAsmFunction). It uses an instance of the Partitioner class to do so. The user can supply a partitioner to the disassembler or have the disassembler create a default partitioner on the fly. The user is also free to call the partitioner directly on the InstructionMap object returned by most of the disassembler methods.

When grouping instructions into basic blocks, the partitioner looks at the instruction type and known successor addresses. A known successor address is a virtual address where the processor will disassemble and execute the next instruction. Unconditional branch instructions typically have a single known successor (the branch target); conditional branches usually have two successors (the following, or fall-through, address and the branch target); data processing and testing instructions have one successor (the following address); and interrupt-causing instructions have no known successors. A branch to a calculated (non-immediate) address does not qualify as a known successor. The SgAsmInstruction's terminatesBasicBlock virtual method is used to make this determination.

Once instructions are assigned to basic blocks, the partitioner assigns the basic blocks to functions using a variety of heuristics, the set of which is determined by the values specified in the Partioner's `set_heuristics` method. These are documented in the SgAsmFunction class (see the FunctionReason enumeration). When a function is created, its `reason` attribute will contain a bit vector describing which heuristics detected this function.

14.2.4 Dwarf Debug Support

ROSE can now read the Dwarf debug information stored into binary executables (ELF only at this point). This information is represented as Dwarf specific IR nodes in the AST and thus can be optionally used (when it is available in the binary) as part of any binary analysis. Only a few sections are supported at present: *.debug_info*, *.debug_line*, etc. The dwarf support in ROSE uses *libdwarf* and is enabled in ROSE using a configuration option: `(configure --with-dwarf=<path to libdwarf>`. Note that *libdwarf* is optional, must be separately installed by the user, and thus is obviously not distributed within ROSE. We anticipate the Dwarf information in the AST to be useful for performance tools that operate on the binary executable when the binary executable has been generated to include Dwarf debug information.

14.3 Binary Analysis

A number of binary analysis passes are provided, most are a part of the Compass framework for software analysis. See the *Compass User Manual* for more details on supported binary analysis.

The ROSE tutorial shows a number of binary analysis passes over both the binary instructions and the executable file format.

14.4 Compass as a Binary Analysis Tool

Compass is a tool framework for building software analysis tools using rules (on source code and alternatively directly on binary executables). Compass reports violations of the rules in the evaluation of the software. Compass is a relatively simple application built on top of ROSE. Most of the complexity and code within Compass is that it includes a large collection to rules, each rule has its own implementation of an arbitrary test over the source code or the binary. Rules (checkers) may be defined over the AST or any other graph built within ROSE to store program analysis information. See the Compass manual for more details on supported binary analysis. The ability to perform analysis of binary executables using Compass makes no assumptions that it is compiled with any specific options or that it contains debug information, symbols, etc.

14.5 Static Binary Rewriting

As part of general research on transformations of binaries (separate from analysis) a number of techniques have been developed to support classes of transformations. This static rewriting of the binary permits the development of performance tools that could support the analysis and rewriting of binaries for support of High Performance Computing (HPC). A principal focus is on IBM BGL and Cray XT support (DOE Office of Science supercomputers).

14.5.1 Generic Section/Segment Modifications

1a. Section/Segment file address shifting (low-level)

The low-level movement of an ELF Section or Segment within the file address space is performed with SgAsmGenericSection::set_offset. It changes the location of the section in the file and updates all relative virtual addresses (RVAs) that were primarily associated with the moved section.

The main problems with this function are that it doesn't take into account the file locations of other sections, the file alignment constraints of the moved section, or the memory mapping. Specifically, after calling this function to move **.text** one byte later in the file:

- **.text** might not satisfy its file alignment constraint.

- The end of **.text** might overlap with the following section. The ELF unparser has undefined behavior when two sections overlap without storing identical bytes at the overlapping regions.
- **.text**, if memory mapped (which it surely is), might not be consistent with the mapping of other adjacent or overlapping sections. For instance, **.text** is contained in "ELF Load Segment 2" both in the file address space and in the mapped memory space. The offset from ELF Load Segment 2 to **.text** must be identical in both file and memory.
- RVAs that point to instructions in **.text** can be associated with the **.text** section or with ELF Load Segment 2, depending on how they were parsed. Normally it doesn't matter which since the relationship between file address space and memory address space is consistent. But if you change the file addresses without changing memory addresses then the byte to which the RVA points could be ambiguous.

Changes to ELF Section or Segment file addresses are reflected in the ELF Section Table and/or ELF Segment Table. If the particular SgAsmGenericSection is present in both tables then modifying its file address will result in updates to both tables.

NOTE: Do not modify section offsets and sizes by modifying the section table entries. Changes to these values will be overwritten with actual, current section offsets and sizes when the section table is unparsed:

- SgAsmElfSectionTableEntry::set_sh_offset
- SgAsmElfSectionTableEntry::set_sh_size
- SgAsmElfSectionTableEntry::set_sh_addr

NOTE: Do not modify segment offsets and sizes by modifying the segment table entries. Changes to these values will be overwritten with actual, current segment offsets and sizes when the segment table is unparsed:

- SgAsmElfSegmentTableEntry::set_offset
- SgAsmElfSegmentTableEntry::set_filesz
- SgAsmElfSegmentTableEntry::set_vaddr
- SgAsmElfSegmentTableEntry::set_memsz

1b. Section/Segment file address shifting (high-level)

The SgAsmGenericFile::shift_extend method is the preferred way to make minor offset and/or size adjustments to an ELF Section or Segment. It is able to shift a section to a high file and/or memory address and/or extend the segment:

- It takes into account all sections in the file, adjusting their offsets and/or sizes accordingly.
- Sections to the right of the the section in question (Sq) are shifted upward to make room and prevent overlaps.
- Sections overlapping with Sq are extended to contain all of what they previously contained.
- The shift amounts are adjusted to satisfy alignment constraints of all affected sections.
- Unreferenced areas of the file can optionally be utilized as unused address space.
- Adjusting file address spaces also adjusts the memory address spaces in a compatible manner.

NOTE: Do not modify section offsets and sizes by modifying the section table entries. Changes to these values will be overwritten with actual, current section offsets and sizes when the section table is unparsed:

- SgAsmElfSectionTableEntry::set_sh_offset

- SgAsmElfSectionTableEntry::set_sh_size
- SgAsmElfSectionTableEntry::set_sh_addr

NOTE: Do not modify segment offsets and sizes by modifying the segment table entries. Changes to these values will be overwritten with actual, current segment offsets and sizes when the segment table is unparsed:

- SgAsmElfSegmentTableEntry::set_offset
- SgAsmElfSegmentTableEntry::set_filesz
- SgAsmElfSegmentTableEntry::set_vaddr
- SgAsmElfSegmentTableEntry::set_memsz

2a. Section/Segment resizing (low-level)

The size of an ELF Section or Segment can be modified by calling SgAsmGenericSection::set_size (for file size) and set_mapped_size (for mapped memory). However, this is a low-level interface that doesn't take into account other sections in the same file. The preferred way to resize a section is with SgAsmGenericFile::shift_extend.

NOTE: For many kinds of sections, making the section larger will create an unreferenced area ("internal hole") at the end of the section. Other sections will automatically do something with the new address space (e.g., SgAsmElfStringSection will add the new address space to its free list).

2b. Section/Segment resizing (high-level)

The preferred way to extend a section is to call SgAsmGenericFile::shift_extend, which extends sections that contain the resized-section and shifts sections that are right (higher address) of the resized-section. This function also takes into account alignment constraints, memory address space, and (optionally) holes in the address space.

14.5.2 Modifications to the ELF File Header

1. Entry Point RVA

The entry RVA stored in the ELF File Header is adjusted whenever the section into which it points is moved in memory. It is also possible to adjust this address explicitly by modifying the first (and only) entry in SgAsmGenericHeader::entry_rvas.

NOTE: An RVA (rose_rva_t) is an offset with respect to the beginning of some section. If the section starting memory address changes then the RVA implicitly changes (RVA's are virtual addresses relative to some format-wide base address). Multiple sections can be mapped to the same memory (e.g., **.text** and **ELF Load Segment 2** are typically overlap in memory), but since an RVA is associated with only one section, modifying the other section(s) has no effect on the RVA even if the RVA happens to be inside the other sections as well.

NOTE: The binding between an RVA and a section can be modified with rose_rva_t::set_section. In fact, the link can be completely broken by passing a null section pointer, in which case the RVA is not relative to any section.

2. File Format Byte order

File byte order can be changed by modifying the SgAsmGenericFormat object pointed to by the file header:

SgAsmGenericHeader *fhdr = ...; fhdr-¿get_exec_format()-¿set_sex(ORDER_MSB);

NOTE: Modifying the byte order affects only those sections that are actually parsed. If the ELF file contains a section whose purpose we don't recognize then the original section data is written to the new file.

f the byte order is not ied in the ELF header ta_encoding other than n the parser will make ted guess and assign a The unparsed file will om the original in this e sixth byte of the file.

3. ELF Word Size

File word size can be changed between 4 bytes and 8 bytes by modifying the SgAsmGenericFormat object pointed to by the file header:

SgAsmGenericHeader *fhdr = ...; fhdr-¿get_exec_format()-¿set_word_size(4);

When changing word sizes, any fields that have values too large to represent in the new word size will cause the unparser to abort.

NOTE: Modifying the word size affects only those sections that are actually parsed. If the ELF file contains a section whose purpose we don't recognize then the original section data is written to the new file.

FIXME: *Increasin probably requires allo space for many of t Vice versa for de*

4. ELF Header Magic Number

An ELF header has a four-byte magic number, usually 0x7f, 'E', 'L', 'F'. The magic number can be modified by changing the string from SgAsmGenericHeader::get_magic. It must be exactly four characters in length.

5. ELF File Purpose (lib, executable, core, etc.)

The file purpose should be modified by setting two fields, using

1. SgAsmElfFileHeader::set_p_e_type
2. SgAsmGenericFormat::set_purpose

Both members should be set to compatible values. The former is the value from the ELF specification and the latter is a constant: PURPOSE_UNSPECIFIED, PURPOSE_LIBRARY, PURPOSE_EXECUTABLE, PURPOSE_CORE_DUMP, PURPOSE_PROC_SPECIFIC, PURPOSE_OTHER.

FIXME: *set_p_e_ probably call set_purp can't go the oth because the map*

6. ELF Version

To change the ELF version assign a new value by calling set_version on the object returned by SgAsmGeneric-Header::get_exec_format. This doesn't have any effect on the code generated by the unparser since the parser only knows about ELF format 1.

7. ELF Target Architecture

Modify the target architecture by calling two functions:

SgAsmElfHeader::set_e_machine – sets the ELF specific value SgAsmGenericHeader::set_isa – sets the generic value

You should call both with consistent values.

8. ELF Section or Segment Table location

The SgAsmElfFileHeader::set_e_shoff and set_e_phoff methods have been removed since calling them had no lasting effect anyway. Instead, if you want to change one of these values for unparsing, then modify the actual SgAsmGenericSection that holds the table (e.g., calling SgAsmGenericFile::shift_extend).

9. ELF Section or Segment Table size

The number of entries in the section or segment table cannot be modified by calling set_e_shnum or set_e_phnum on the SgAsmElfFileHeader . Rather, the sizes are obtained by looking at what sections and segments are currently defined and writing an entry to the file for each one.

10. ELF Section Names

Elf section names can be modified. Doing so may cause extensive changes to the executable due to reallocation of the section holding the string table.

Do not call SgAsmElfSectionTableEntry::set_sh_name since that value will be overwritten based on the actual, current location of the name in the associated string table.

11. ELF Segment Names

ELF segment names are often parser-generated based on constants in the ELF Segment Table. However, if the segment corresponds to an actual ELF Section defined in the ELF Section Table then the segment and section share the same SgAsmGenericSection object and changing the name causes the ELF Section name to change with no effect on the segment table.

12. ELF Section Name Table

The section that holds the section names is identified in the ELF File Header (get_e_shstrndx). Although it is possible to change this value, doing so will have no effect on the currently-defined sections: they will continue to use the original string table for their names.

14.5.3 Modifications to ELF String Tables and their Containing Sections

1. Move/Extend

See SgGenericFile::shift_extend. When a string table is extended the new address space is added to the table's free list.

2. New String

A new string can be created by calling the SgAsmStoredString allocator and passing a string table (something derived from SgAsmGenericStrtab) and the initial string value. The string is not actually allocated space in the file until the new file is unparsed or until someone calls SgAsmStoredString::get_offset.

3. Value modification

A string can be modified by assigning a new value via SgAsmStoredString::set_string. Storage is not allocated for the new value until the AST is unparsed or someone calls SgAsmStoredString::get_offset. The previous value is freed.

4. Shared strings

Three forms of sharing are supported:

1. Two objects (section names, symbol names, etc) share the same string and changing one string causes the other to change as well. This kind of sharing is not typically encountered in ELF although the underlying string table classes support it.
2. Two objects have independent strings that happen to have the same value and point to the same offset in the string table. In this case, changing one string doesn't change the other. This kind of sharing is often encountered in ELF.
3. Two objects have independent strings and one is an ending substring of another (e.g., "main" and "domain"). Changing one string does not affect the other. This kind of sharing is also common in ELF.

5. String table internal holes

If a sequence of bytes in a string table is not referenced by anything known to the parser, then those bytes are marked as internal holes and are prevented from moving with respect to the beginning of the string table. Internal holes are not placed on the string table free list (because something we didn't parse might be pointing to them). The internal holes are available with SgAsmGenericSection::congeal.

6. Reallocation of all strings

A string table can be repacked by freeing all it's strings and then reallocating. We can reallocate around the internal holes or through the internal holes.

strtab.free_all_strings(); /* free_all_strings(true) blows away internal holes */ strtab.reallocate();

The ELF allocator will do its best to overlap storage (e.g., "domain" overlaps with "main").

7. Deletion of a string

A string is deleted by changing its value to the empty string.

8. Stored strings vs. non-stored strings.

If a string value has storage space in a file (such as an ELF Section name), then it's an instance of SgAsmStoredString. Otherwise the string is either an std::string or SgAsmBasicString. SgAsmBasicString and SgAsmStoredString both derive from SgAsmGenericString. Changing the value of an SgAsmBasicString has no effect on the unparsed file.

14.5.4 Modifications ELF Section Table Entries

Every ELF Section defined by the ELF Section Table is parsed as an SgAsmElfSection, which is derived from SgAsmGenericSection. The SgAsmElfSection::get_section_entry returns a pointer to the ELF Section Table Entry (SgAsmElfSectionTableEntry). Some members of these objects can be modified and some can't.

1. These functions should not be called since their values are overwritten during the unparse phase:

- SgAsmElfSectionTableEntry::set_sh_name – see SgAsmGenericSection::set_name
- SgAsmElfSectionTableEntry::set_sh_addr – see SgAsmGenericFile::shift_extend
- SgAsmElfSectionTableEntry::set_sh_offset – see SgAsmGenericFile::shift_extend
- SgAsmElfSectionTableEntry::set_sh_size – see SgAsmGenericFile::shift_extend
- SgAsmElfSectionTableEntry::set_sh_link – don't call (no alternative yet)

2. Can modify

E: *What text should go here?*

- SgAsmElfSectionTableEntry::set_sh_type
- SgAsmElfSectionTableEntry::set_sh_flags, although the Write and Execute bits are ignored
- **ME:** *Is this complete?* SgAsmElfSectionTableEntry::set_sh_info
- **ME:** *Is this complete?* SgAsmElfSectionTableEntry::set_sh_addralign
- **ME:** *Is this complete?* SgAsmElfSectionTableEntry::set_sh_entsize

14.6 Dynamic Analysis Support

Recent work in ROSE has added support for dynamic analysis and for mixing of dynamic and static analysis using the Intel Pin framework. This optional support in ROSE requires a configure option (`--with-IntelPin=`< *path*>. The `path` in the configure option is the path to the top level directory of the location of the Intel Pin distribution. This support for Intel Pin has only been tested on a 64bit Linux system using the most recent distribution of Intel Pin (version 2.6).

Note: The dwarf support in ROSE is currently incompatible with the dwarf support in Intel Pin. A message in the configuration of ROSE will detect if both support for Dwarf and Intel Pin are both specified and exit with an error message that they are incompatible options.

14.7 Usage

See the ROSE Tutorial for examples.

Chapter 15

RTED: Runtime Error Detection

15.1 Overview

RTED[1] is a test suite developed by Iowa State University's High Performance Computing Group[2]. It illustrates a number of runtime errors, many of which result from undefined behavior in the C and C++ specifications.

The RTED project in ROSE is a program that instruments its input to use a supplied runtime system with the aim that when the instrumented program runs, any undefined behavior is immediately caught, as well as certain other runtime errors such as certain memory leaks. For example, in the following code:

```
int foo() {
    int a[3];
    int n;

    int* p = &a[ 2 ];
    ++p;
}
```

After `p` is incremented, its address is undefined. Many compilers might well organize the call frame such that after the increment, `p == &n`, but relying on this is unsafe.

When the runtime system detects unspecified behavior, a violation is raised and handled. The default policy for most violations is to output a message and immediately terminate the program.

15.1.1 Current State

At the time of this writing, the project is incomplete. Presently, only the RTED test suite is targeted, although a large subset of its C and C++ tests' errors are caught.

15.1.2 Organization

The project is organized into two sections:

[1] http://rted.public.iastate.edu/
[2] http://www.it.iastate.edu/research/hpcg/

1. The runtime system is a library that provides an API that, in principle, can be used for other projects that require tracking variables, the stack, “typed” memory, and so on.
2. The transformation code is compiled into a binary that transforms input by injecting calls to the runtime system. For convenience, the actual injected calls are to very small wrapper functions that provide a simpler abstraction over the runtime system’s API. However, if users wish to adapt the runtime system, it is recommended that they view the wrapper functions as an example usage of the runtime system’s API, and not the API itself.

The transformation only injects function calls that are valid C and C++ code. The runtime system is written in C++ but its API is compiled with `extern \C"`

15.2 How to use RTED in ROSE

To instrument a program, one does the following:

1. Compile `runtimeCheck` in `projects/RTED` of the ROSE compile tree.
2. `#include RuntimeSystem.h` in your source files[3].
3. Run runtimeCheck -n <number-of-source-files> <sourcefile1> <sourcefile2> ... <sourcefilen> to produce `sourcefile1_rose, sourcefile2_rose, ....`
4. Compile the `*_rose` instrumented source files.

For example, the following code:

```
#include <stdlib.h>


int main( int argc, char** argv ) {

    int z[ 2 ];
    int* x;

    int i = 0;
    for( ; i >= 0; --i ) {
        int* y;
        y = (int*) malloc( sizeof( int ));
        x = y;
    }

    // error, y is out of scope so assigning to x leaks the malloc in the
    // for loop
    x = z;

    return 0;
}
```

[3]You may notice that in the ROSE makefile, `#include RuntimeSystem_ParsingWorkaround.h` is sometimes included instead. You will probably not need to do this, but if you have errors resulting from `std::endl` being undefined, you can try using `RuntimeSystem_ParsingWorkaround.h` instead.

is transformed into:

```
#include "RuntimeSystem.h"
#include <stdlib.h>

extern int RuntimeSystem_original_main(int argc,char **argv,char **envp)
{
  RuntimeSystem_roseConfirmFunctionSignature("RuntimeSystem_original_main",3,"SgTypeInt","",0,"SgTypeInt","",0

  RuntimeSystem_roseCreateVariable("envp","envp","SgPointerType","SgTypeChar",2,((unsigned long long )(&envp))

  RuntimeSystem_roseCreateVariable("argv","argv","SgPointerType","SgTypeChar",2,((unsigned long long )(&argv))

  RuntimeSystem_roseCreateVariable("argc","argc","SgTypeInt","",0,((unsigned long long )(&argc)),sizeof(argc),
  int z[2UL];

  RuntimeSystem_roseCreateVariable("z","z","SgArrayType","SgTypeInt",1,((unsigned long long )(&z)),sizeof(z),0

  RuntimeSystem_roseCreateHeap("z","L14R__scope____SgSS2____scope__z","SgArrayType","SgTypeInt",1,((unsigned lo
  int *x;

  RuntimeSystem_roseCreateVariable("x","x","SgPointerType","SgTypeInt",1,((unsigned long long )(&x)),sizeof(x)
  int i = 0;

  RuntimeSystem_roseCreateVariable("i","i","SgTypeInt","",0,((unsigned long long )(&i)),sizeof(i),1,"","0","tra

  RuntimeSystem_roseInitVariable("SgTypeInt","",0,"",((unsigned long long )(&i)),sizeof(i),0,0,"assignment_scop
  RuntimeSystem_roseEnterScope("for:12");

  RuntimeSystem_roseAccessVariable(((unsigned long long )(&i)),sizeof(i),((unsigned long long )(&i)),sizeof(i)

  RuntimeSystem_roseAccessVariable(((unsigned long long )(&i)),sizeof(i),((unsigned long long )(&i)),sizeof(i)
  for (; i >= 0; --i) {
    int *y;

    RuntimeSystem_roseCreateVariable("y","y","SgPointerType","SgTypeInt",1,((unsigned long long )(&y)),sizeof(y
    y = ((int *)(malloc(((sizeof(int ))))));

    RuntimeSystem_roseCreateHeap("y","L14R__scope____SgSS2____scope____SgSS3____scope____SgSS4____scope__y","Sg

    RuntimeSystem_roseInitVariable("SgPointerType","SgTypeInt",1,"",((unsigned long long )(&y)),sizeof(y),1,1,"

    RuntimeSystem_roseAccessVariable(((unsigned long long )(&y)),sizeof(y),((unsigned long long )(&y)),sizeof(y
    x = y;

    RuntimeSystem_roseInitVariable("SgPointerType","SgTypeInt",1,"",((unsigned long long )(&x)),sizeof(x),0,1,"
```

```
  }

  RuntimeSystem_roseExitScope("assignment_scope_for.c","16","64","for(;i >= 0;--i) {int *y;y =((int
// error, y is out of scope so assigning to x leaks the malloc in the
// for loop
  x = z;

  RuntimeSystem_roseInitVariable("SgPointerType","SgTypeInt",1,"",((unsigned long long )(&x)),sizeof
  int rstmt = 0;
  RuntimeSystem_roseCheckpoint("assignment_scope_for.c","22","72");
  return rstmt;
}


int main(int argc,char **argv,char **envp)
{
  int exit_code = RuntimeSystem_original_main(argc,argv,envp);
  RuntimeSystem_roseRtedClose("RuntimeSystem.cpp:main");
  return exit_code;
}
```

which, in the call to `RuntimeSystem_roseInitVariable` that immediately follows `x = z` will detect the memory leak and terminate the program.

15.2.1 Configuration

At present, configuration is very limited. When the runtime system is initialized, it searches for `RTED.cfg` in the current directory. If such a file exists, it is read as a newline separated list of key value pairs. If the runtime system was compiled with QT enabled, then the qt debugger can be enabled so that the GUI debugger is opened when the first violation is found. Also, invidual policies for errors types can be configured. See the RTED doxygen documentation for more information.

15.2.2 Partial Compilation

At present the transformation does not properly support partial compilation. This is not a fundamental challenge to the current design, it merely reflects the incomplete state of the project. See also section 15.4.

15.3 Extending the Runtime System

The runtime system is separate from the transformations. Users who wish to use the runtime system for other purposes, perhaps other sorts of runtime checks, can do so by compiling the runtime system (in `<rose-dir>/projects/RTED/CppRuntimeSystem`) and using it separately. The doxygen documentation has more information on the organization of the various components of the runtime system, especially the documentation for the `Runtimesystem` class.

15.4 Known Limitations

Apart from being incomplete, there are some known limitations.

1. The runtime system presently assumes all pointers have a fixed size. This size is determined at the compile time *of the runtime system* via `sizeof( void* )`.
2. Similarly, the size of basic types is determined when the runtime system is compiled.
3. Separate compilation is not properly handled yet. At present, globals and types in the global namespace are reported to the runtime system via calls injected into `main`. This obviously doesn't work when compiling a file that doesn't define `main`.
4. There is some support for handling C I/O usage, and although there is some code related to C++ I/O support, it is extremely preliminary.
5. There is no support for C++ object initializers.
6. Typedefs are not resolved by the instrumentation. Although this is not difficult, it does mean that the runtime system treats all typedefs as a single type, and distinct from their actual type.

15.4.1 A Note on RTED Scoring

The RTED suite includes scoring tests. Presently the RTED support in ROSE is checked by ensuring that an error is found on the expected line. The reference messages for RTED include information that is not currently output by instrumented inputs (such as variable names). However, much of the missing information is actually collected by the runtime system and is simply not output.

15.5 Running a specific RTED test

To run one specific test run:

make run/tests/C/memoryleaks/assignment_scope_dowhile

The different steps involved in running the test are:

1. Insert #include "RuntimeSystem.h" into file
2. Test if files compile with gcc (file_rose.c.compile.out)
3. Run transformation (file_rose.c.runtimeCheck.out)
4. Add line information
5. Compile instrumented code (file.bin.compile.out)
6. Execute instrumented code

15.6 Handling warnings

It is wise to enforce all warnings to be handled as errors. This will ensure a clean implementation. We enforced this using Eclipse. In addition we added the following flags to the compilation to skip certain warnings:

```
-Wno-non-virtual-dtor -Wno-invalid-offsetof
```

15.7 Using Eclipse for RTED development

It is possible to use Eclipse just for RTED and link against librose.so.

To use Eclipse, download the C++ Eclipse version and create new project RTED. Then include the original RTED directory. The following are the additional includes necessary in C/C++ Build Settings for the GNU C++ Compiler:

```
-I/home/panas2/programs/qt/install/qt/include/QtGui
-I/home/panas2/rose/ROSE-git/projects/RTED/CppRuntimeSystem/DebuggerQt
-I/home/panas2/rose/ROSE-git/projects/RTED/CppRuntimeSystem/
-I/home/panas2/rose/ROSE-git/projects/RTED/
-I/home/panas2/rose/ROSE-git/src/roseExtensions/qtWidgets/QCodeEditWidget
-I/home/panas2/programs/qt/install/qt/include/QtCore
-I/home/panas2/rose/build/projects/RTED/CppRuntimeSystem/DebuggerQt
-I/home/panas2/programs/qt/install/qt/include
-I/home/panas2/programs/qt/install/qt/include/Qt
-I/home/panas2/rose/ROSE-git/src/roseSupport
-I/home/panas2/rose/ROSE-git/src/midend/astProcessing/
-I/home/panas2/rose/ROSE\_includes
-I/home/panas2/programs/boost/install/include/
-I/home/panas2/rose/build/src/frontend/SageIII
-I/home/panas2/rose/build
```

A simpler way to handle all required .h files in ROSE is to copy all includes from ROSE into a separate ROSE_includes directory or just to install ROSE. Thereafter add the rose library to the shared libararies using the path: build/src/.libs/ and the file: rose. For this to work ROSE must be compiled into the build directory as files from the build directory are needed.

To run RTED, you can type the command (in bash or Eclipse):

```
/home/panas2/rose/ROSE-git/projects/RTED/Debug/RTED 1
/home/panas2/rose/ROSE-git/projects/RTED/Debug/tests2/C/memoryleaks/
assignment\_scope\_while.c
-I/home/panas2/rose/ROSE-git/projects/RTED/ -c
```

This should run this example. If you use the default tests, you will get the following error message:

```
RTED: ../RtedTransformation.cpp:67: void RtedTransformation::
transform(SgProject*, std::set<std::basic\_string<char, std::char\_
traits<char>, std::allocator<char> >, std::less<std::basic\_string
<char, std::char\_traits<char>, std::allocator<char> > >,
std::allocator<std::basic\_string<char, std::char\_traits<char>,
std::allocator<char> > > > ): Assertion roseCreateHeap failed.
```

The problem is that the test file needs to include RuntimeSystem.h

After that we compile the transformed file:

```
gcc -g -Wall -o assignment\_scope\_while.bin rose\_assignment\_
```

```
scope\_while.c ../RuntimeSystem.cpp -isystem /home/panas2/programs/
boost/install//include -I/home/panas2/rose/ROSE\_includes/
-I../ /home/panas2/rose/build/projects/RTED/CppRuntimeSystem/.libs/
libCppRuntimeSystem.so.0 -L/home/panas2/programs/qt/install/qt//lib
-L/home/panas2/programs/qt/install/qt/lib -L/usr/X11R6/lib
/home/panas2/programs/qt/install/qt/lib/libQt3Support.so
/home/panas2/programs/qt/install/qt/lib/libQtSql.so
/home/panas2/programs/qt/install/qt/lib/libQtGui.so
/home/panas2/programs/qt/install/qt/lib/libQtNetwork.so
/home/panas2/programs/qt/install/qt/lib/libQtXml.so
/home/panas2/programs/qt/install/qt/lib/libQtCore.so -lpthread
-lQtUiTools -lgcrypt  -Wl,--rpath -Wl,/home/panas2/rose/install/lib
-Wl,--rpath -Wl,/home/panas2/programs/qt/install/qt/lib -Wl,--rpath
-Wl,/home/panas2/rose/build/projects/RTED/CppRuntimeSystem/.libs/
-lCppRuntimeSystem -L/home/panas2/rose/build/projects/RTED/
CppRuntimeSystem/.libs/
```

OR when creating a static library under ROSE for CppRuntimeSystem, we can do

```
gcc -g -Wall -o assignment\_scope\_while.bin rose\_assignment\_
scope\_while.c ../RuntimeSystem.cpp -isystem /home/panas2/programs/
boost/install//include -I/home/panas2/rose/ROSE\_includes/ -I../
-L/home/panas2/programs/qt/install/qt//lib -L/home/panas2/programs/qt/
install/qt/lib -L/usr/X11R6/lib /home/panas2/programs/qt/install/
qt/lib/libQt3Support.so /home/panas2/programs/qt/install/qt/lib/
libQtSql.so /home/panas2/programs/qt/install/qt/lib/libQtGui.so
/home/panas2/programs/qt/install/qt/lib/libQtNetwork.so
/home/panas2/programs/qt/install/qt/lib/libQtXml.so
/home/panas2/programs/qt/install/qt/lib/libQtCore.so -lpthread
-lQtUiTools -lgcrypt  -Wl,--rpath -Wl,/home/panas2/rose/install/lib
-Wl,--rpath -Wl,/home/panas2/programs/qt/install/qt/lib -Wl,--rpath
-Wl,/home/panas2/workspace/CppRuntimeSystemLibarary/Debug
-lCppRuntimeSystemLibarary -L/home/panas2/workspace/
CppRuntimeSystemLibarary/Debug
```

To create a static library: Create new project in Eclipse. Select Shared Library. Thereafter import the source files, add the includes and the necessary libraries.

Includes:

```
-I/home/panas2/programs/qt/install/qt/include/QtGui
-I/home/panas2/rose/ROSE-git/projects/RTED/CppRuntimeSystem/DebuggerQt
-I/home/panas2/rose/ROSE-git/projects/RTED/CppRuntimeSystem/
-I/home/panas2/rose/ROSE-git/projects/RTED/
-I/home/panas2/rose/ROSE-git/src/roseExtensions/qtWidgets/QCodeEditWidget
-I/home/panas2/programs/qt/install/qt/include/QtCore
-I/home/panas2/rose/build/projects/RTED/CppRuntimeSystem/DebuggerQt
-I/home/panas2/programs/qt/install/qt/include
```

```
-I/home/panas2/programs/qt/install/qt/include/Qt
-I/home/panas2/rose/ROSE-git/src/roseSupport
-I/home/panas2/rose/ROSE-git/src/midend/astProcessing/
-I/home/panas2/rose/ROSE\_includes
-I/home/panas2/programs/boost/install/include/
-I/home/panas2/rose/build/src/frontend/SageIII -I/home/panas2/rose/build
```

Libraries:

```
-L/home/panas2/rose/build/src/.libs/ -lrose
```

Optionally create the CppRuntimeSystem as a separate Library. It is easy to set up as a new project and then include the .so into the RuntimeSystem project. One thing to remember is to add the CppRuntimeSystem.so into the LD_LIBRARY_PATH so that it is found when the project is run.

Chapter 16

CUDA and OpenCL

Current Status (May 2011) *: ROSE presently supports the generation of both CUDA and OpenCl. Thus ROSE translators may be built that read C/C++/Fortran and generate CUDA or OpenCL (several have been built by external research groups). We are now able to read CUDA and still working on OpenCL (which is not distribute yet).*

This chapter details the support of C for CUDA [1] and OpenCL [2].

16.1 How to use CUDA and OpenCL in ROSE

Both CUDA and OpenCL are enable in ROSE from two configuration options.

- *–enable-(cuda/opencl)*: to enable the internal support of these languages.
- *–enable-edg-(cuda/opencl)*: to enable the frontend support (parsing)

As the support of CUDA/OpenCL parsing have been developt in EDG 4.0, you also need to use the option *–enable-edg-version=4.0*.

If you use the binary version of EDG (most common case for external users), you do not need the option *–enable-edg-(cuda/opencl)* as to avoid multiple versions of the binaries we directly distribute EDG 4.0 with CUDA/OpenCL support (the option may still be needed to build some sub-project using CUDA/OpenCL support).

To build and test the CUDA/OpenCL support with your version of ROSE:
$ ROSE_SRC_TREE/configure –enable-edg-version=4.0 –enable-cuda –enable-edg-cuda –enable-opencl –enable-edg-opencl [...]
$ make
$ make -Ctests/nonsmoke/functional/CompileTests/NewEDGInterface_C_tests check # tests EDG 4.0
$ make -Ctests/nonsmoke/functional/CompileTests/CudaTests check # tests CUDA support

[1] Compute Unified Device Architecture, NVidia architecture model for their devices

[2] Open Compute Language, an open standard to program heterogeneous system, managed by the Khronos Group

$ make -Ctests/nonsmoke/functional/CompileTests/OpenCLTests check # tests OpenCL support

You can look on the test directories to see the current state of CUDA/OpenCL support.

16.2 IR adaptations

Both CUDA and OpenCL use extensions of C ANSI 99, these extensions need to be catch by SageIII. We will describe theses adaptations below.

16.2.1 C for CUDA

The C for CUDA extensions of C ANSI 99 is describe in the CUDA Programming Guide Version 3.0, appendix B and C.
CUDA programs are usually in .cu files.

Function and storage modifiers

Function Modifier As CUDA target multiple kind of device, it provide a way to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device. For this, it use some function modifiers listed below, with caller and target:

- __host__: from CPU to CPU;
- __device__: from GPU to GPU;
- __global__: from CPU to GPU, it define a *kernel*.

We have modified *SgFunctionModifier* by adding to *function_modifier_enum*:

- e_cuda_device
- e_cuda_kernel
- e_cuda_host

Storage Modifier For variables, *SgStorageModifier* is a good candidate but it can only address one modifier. That's a problem in the case of dynamically allocated shared memory. We handled it by considering following cases for *storage_modifier_enum*:

- __device__ → e_cuda_global
- (__device__) __constant__ → e_cuda_constant
- (__device__) __shared__ → e_cuda_shared
- extern (__device__) __shared__ → e_cuda_dynamic_shared

Execution configuration associated at a kernel call

CUDA introduce too a special way to call kernels (__global__ functions):

```
kernel<<<Dg, Db, Ns, S>>>(parameter);
```

Where:

- Dg is the grid dimensions
- Db is blocks dimensions
- Ns is shared memory size (optional)
- S is a stream (optional)

We catched it by adding two new nodes in the IR:

- *SgCudaKernelCallExp* at the same level than *SgFunctionCallExp*
- *SCudagKernelExecConfig*, an attribute of *SgCudaKernelCallExp*, catch the execution configuration (grid, blocks, shared memory dynamic allocation and stream).

Build-in Types, Variables and Functions

All built-in CUDA objects are defined in ROSE/config/rose_edg_required_macros_and_functions.h.in .

Types

Vectors CUDA use vectors types extended from common types.

Table 16.1: Vector types defined by CUDA

char1	char2	char3	char4
uchar1	uchar2	uchar3	uchar4
short1	short2	short3	short4
ushort1	ushort2	ushort3	ushort4
int1	int2	int3	int4
uint1	uint2	uint3	uint4
long1	long2	long3	long4
ulong1	ulong2	ulong3	ulong4
longlong1	longlong2		
float1	float2	float3	float4
double1	double2		

For examples, *int2* is equivalent to:

```
struct int2
{
int x, y;
};
```

3rd and 4th dimensions are access by z and w fields.

A constructor is associated with all build-in type: make_*type*

```
int2 make_int2(int x, int y);
```

Others Based on *uint3*, *dim3* is used to specify dimensions. Any unspecified components of a *dim3* instance is initialized to 1.

Some others types, like cudaError_t, are not documented as built-in types but *nvcc*[3] don't need that the API header was explicitly included.

Variables Some variables that can't be nor affected nor addressed by pointers:

Table 16.2: CUDA's built-in variables

Variable	Type	Purpose
gridDim	dim3	dimension of the grid
blockIdx	uint3	block index within the grid
blockDim	dim3	dimension of the block
threadIdx	uint3	thread index within the block
warpSize	int	number of threads by warp

Functions

Synchronization functions

- void __threadfence_block();
- void __threadfence();
- void __threadfence_system();
- void __syncthreads();
- int __syncthreads_count(int predicate);
- int __syncthreads_and(int predicate);
- int __syncthreads_or(int predicate);

Mathematical functions

- float __fdividef(float x,float y);
- float __sinf(float x);
- float __cosf(float x);
- float __tanf(float x);
- float __sincosf(float x, float * sin, float * cos);
- float __logf(float x);
- float __log2f(float x);
- float __log10f(float x);

[3] NVidia compileur for CUDA

- float __expf(float x);
- float __exp10f(float x);
- float __powf(float x,float y);

Texture functions

template<class Type>

- Type tex1Dfetch(texture¡Type, 1, cudaReadModeElementType¿ texRef, int x);
- float$n$ tex1Dfetch(texture¡Type$n$, 1, cudaReadModeNormalizedFloat¿ texRef, int x);

template<class Type, enum cudaTextureReadMode readMode>

- Type tex1D(texture¡Type, 1, readMode¿ texRef, float x);
- Type tex2D(texture¡Type, 1, readMode¿ texRef, float x, float y);
- Type tex3D(texture¡Type, 1, readMode¿ texRef, float x, float y, float z);

Time functions

- clock_t clock();

Atomic functions

- *type* atomicAdd(*type* * address, *type* val); with *type*: int, unsigned int, unsigned long long int and float.
- *type* atomicSub(*type* * address, *type* val); with *type*: int and unsigned int.
- *type* atomicExch(*type* * address, *type* val); with *type*: int, unsigned int, unsigned long long int and float.
- *type* atomicMin(*type* * address, *type* val); with *type*: int and unsigned int.
- *type* atomicMax(*type* * address, *type* val); with *type*: int and unsigned int.
- *type* atomicInc(*type* * address, *type* val); with *type*: unsigned int.
- *type* atomicDec(*type* * address, *type* val); with *type*: unsigned int.
- *type* atomicCAS(*type* * address, *type* compare, *type* val); with *type*: int, unsigned int and unsigned long long int
- *type* atomicAnd(*type* * address, *type* val); with *type*: int and unsigned int.
- *type* atomicOr(*type* * address, *type* val); with *type*: int and unsigned int.
- *type* atomicXor(*type* * address, *type* val); with *type*: int and unsigned int.

Warp Vote functions

- int __all(int predicate);
- int __any(int predicate);
- unsigned int __ballot(int predicate);

Profiler Counter functions

- void __prof_trigger(int counter);

More Maths functions See Appendix C - Mathematical Functions of CUDA Programming Guide Version 3.0.

16.2.2 OpenCL

OpenCL language only address devices, the host (general purpose CPU) code is design in C/C++ and use OpenCL API. OpenCL programs are compiled on the fly and stored on C characters strings, that may be a problem but usually theses string are load from a separate .cl file. For more convenience, we do the assumption that all OpenCL programs are stored in separate files and never dynamically modified.

Function and storage modifiers

Function Qualifiers

- __kernel
- __attribute__((vec_type_hint(*type*)))
- __attribute__((work_group_size_hint(X, Y, Z)))
- __attribute__((reqd_work_group_size(X, Y, Z)))

__kernel qualifier can be catch by adding *e_ocl_kernel* in the *function_modifier_enum* of *SgFunctionModifier*.

The __attribute__(...) are linked to __kernel qualifier. In OpenCL, the *work group* is a set of *work items* (instance of the kernel[4]) that provide a more coarse-grained decomposition of the index space associate to the *global work*.
We had two ways to implement it: a new modifier or some modification on *SgFunctionModifier*.
We choosed to add in *SgFunctionModifier* three flags: *e_ocl_vec_type_hint*, *e_ocl_work_group_hint*, *e_ocl_work_group_req*; and two fields: *SgType * ocl_vec_type* and *struct* { *unsigned int x, y, z;* } *ocl_work_group_size*.

Address Space Qualifier They are used to specify where a variable is stored in the device.

- __global accessible from all the device
- __local accessible for a work-group only
- __constant read-only variable accessible from all the device
- __private accessible only from one "thread"

We used *SgStorageModifier* to store this information by adding to *storage_modifier_enum*:

- _e_opencl_global
- _e_opencl_local
- _e_opencl_constant
- _e_opencl_private

[4]see: OpenCL Specification v1.0, section 3.2 Execution Model

Image Access Qualifiers

- __read_only
- __write_only
- __read_write

SgAccessModifier fill good by the name but is dedicated to class...

Build-in Types

Many Built-in types: scalars, vectors, matrix and others, like image, event or complex.

Scalars OpenCL introduce alias for unsigned types (uint, uchar, ...) and some scalars types: half, size_t, ptrdiff_t, intptr_t, uintptr_t.

Vectors OpenCL define vector from scalars: char, uchar, short, ushort, int, uint, long, ulong and float. These vector types are noted type*n* where n can be 2, 4, 8 or 16. For this type, we use the same mechanism than for CUDA vector types. We may have a problem with vector components (section 6.1.7 of OpenCL Specification v1.0)

```
float4 a    = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
float4 swiz = a.wzyx;     // swiz = (4.0f, 3.0f, 2.0f, 1.0f)
float2 low  = a.lo;       // low  = (4.0f, 3.0f)
float8 b;
b.hi = a;
b.lo = swiz;              // b    = (4.0f, 3.0f, 2.0f, 1.0f, 1.0f, 2.0f, 3.0f, 4.0f)
```

Others Other built-in types are image2d_t, image3d_t, sampler_t and event_t.

Reserved See section 6.1.4 of OpenCL Specification v1.0

Build-in Functions

Work-Item Functions

- uint get_work_dim ()
- size_t get_global_size (uint dimindx)
- size_t get_global_id (uint dimindx)
- size_t get_local_size (uint dimindx)
- size_t get_local_id (uint dimindx)
- size_t get_num_groups (uint dimindx)
- size_t get_group_id (uint dimindx)

Image Functions

- float4 read_imagef (image2d_t image, sampler_t sampler, int2 coord)
- float4 read_imagef (image2d_t image, sampler_t sampler, float2 coord)
- int4 read_imagei (image2d_t image, sampler_t sampler, int2 coord)
- int4 read_imagei (image2d_t image, sampler_t sampler, float2 coord)
- unsigned int4 read_imageui (image2d_t image, sampler_t sampler, int2 coord)
- unsigned int4 read_imageui (image2d_t image, sampler_t sampler, float2 coord)
- void write_imagef (image2d_t image, int2 coord, float4 color)
- void write_imagei (image2d_t image, int2 coord, int4 color)
- void write_imageui (image2d_t image, int2 coord, unsigned int4 color)
- float4 read_imagef (image3d_t image, sampler_t sampler, int4 coord)
- float4 read_imagef (image3d_t image, sampler_t sampler, float4 coord)
- int4 read_imagei (image3d_t image, sampler_t sampler, int4 coord)
- int4 read_imagei (image3d_t image, sampler_t sampler, float4 coord)
- unsigned int4 read_imageui (image3d_t image, sampler_t sampler, int4 coord)
- unsigned int4 read_imageui (image3d_t image, sampler_t sampler, float4 coord)
- int get_image_width (image2d_t image)
- int get_image_width (image3d_t image)
- int get_image_height (image2d_t image)
- int get_image_height (image3d_t image)
- int get_image_depht (image3d_t image)
- int get_image_channel_data_type (image2d_t image)
- int get_image_channel_data_type (image3d_t image)
- int get_image_channel_order (image2d_t image)
- int get_image_channel_order (image3d_t image)
- int2 get_image_dim (image2d_t image)
- int4 get_image_dim (image3d_t image)

Synchronization Functions

- void barrier (cl_mem_fence_flags flags)

Explicit Memory Fence Functions

- void mem_fence (cl_mem_fence_flags flags)
- void read_mem_fence (cl_mem_fence_flags flags)
- void write_mem_fence (cl_mem_fence_flags flags)

Miscellaneous Functions

- event_t async_work_group_copy (__local *gentype* *dst, const __global *gentype* *src, size_t num_elements, event_t event)
- event_t async_work_group_copy (__global *gentype* *dst, const __local *gentype* *src, size_t num_elements, event_t event)
- void wait_group_events(int num_events, event_t * event_list)
- void prefetch (const __global *gentype* *p, size_t num_elements)

Where *gentype* can be scalar or vector of char, uchar, short, ushort, int, uint, long, ulong or float.

Maths Functions

Chapter 17

Polyhedral Model

This chapter reffers to the project PolyhedralModel.

17.1 What can I find in this project ?

The project come with a doxygen documentation:
$ cd projects/PolyhedralModel/docs
$ doxygen doxy.conf

17.2 What I need to compile this project with Rose ?

To enable the Polyhedral Model project, you need PPL (Parma Polyhedra Library) (May 2011: version 0.11.2). The configure options are:
–enable-ppl –with-ppl=[path to PPL]

Code generation is one of the main future of this project. It needs the library Cloog (Chunky Loop Generator) (May 2011: version 0.16.2). The configure options are:
–enable-cloog –with-cloog=[path to Cloog]

Some other works depend on ScopLib and Candl, they can be enable by the same kind of options.

17.3 And now a few maths !

More about "polyhedral representation of dependences".

17.3.1 Definitions and Notations

Integer Polyhedron

A integer polyhedron (which would be the only polyhedron that we will consider) is a set of integers points.
$\forall n \in \mathbb{N}, \mathbb{P} = \{\bar{z} \in \mathbb{Z}^n | \mathcal{P}(\bar{z}) \geq 0\}$ where:

- n is the dimension of the polyhedron
- $\mathcal{P} : \mathbb{Z}^n \rightarrow \mathbb{Z}^m$ is an affine application
- m is the number of intersected half-space

($\bar{z} \geq 0$ means $\forall i, z_i \geq 0$)

Notation

- $\Box$ is the vector concatenation operator;
- 1_i is a vector with all component equal to zero but the i^{th};
- $Row_i^{\mathbb{P}}(\bar{z}) \geq 0$: the i^{th} inequalitie that define the polyhedron $\mathbb{P}$.
  $Row_i^{\mathbb{P}}(\bar{z}) = \mathcal{P}(\bar{z}).1_i$ where $\mathcal{P}$ is the affine application associate to $\mathbb{P}$.

17.3.2 Program

All considered programs are on the scope of Affine Control Loop (ACL), follow some notation:

- $\mathbb{G}$ the set of Global variables: scalars from global scope (ex: array size).
  We will note $\bar{g}$ an element of $\mathbb{G}$.
- A set of statement $\mathbb{S}$, define by:
  - An iteration domain: $\mathbb{D} = \{\bar{z} | f(\bar{z}, \bar{g}) \geq 0\}$ where f is an affine form $f : \mathbb{Z}^n \times \mathbb{G} \rightarrow \mathbb{Z}$ where n is the number of iterators.
  - Two set of variables: read and write.
    A variable can be a scalar or an array element. In second case, the access vector to this element need to be an affine application $f : \mathbb{D} \times \mathbb{G} \rightarrow \mathbb{Z}^{dim(array)}$.
    NB: for a pseudo-multi-dimensionnal array, index needs to be $\sum_{i=1}^{dim(array)} f_i \times \prod_{j=1}^{i} size(dim_j)$

17.3.3 Dependence Graph

We consider the graph of dependence Γ, in this directed graph, nodes are statement and vertex are dependence. Vertex are qualified with the conditions of the dependence.
Given two statements S_1 and S_2 with $\mathbb{D}_1$ and $\mathbb{D}_2$ theirs iterations domains. A vertex from S_1 to S_2 will be qualified by:

- a condition $q_{1\rightarrow 2}(\bar{z}, \bar{g}) \geq 0$ where $q_{1\rightarrow 2}$ is an affine application $q_{1\rightarrow 2} : \mathbb{D}_1 \times \mathbb{G} \rightarrow \mathbb{Z}^n$ where is the number of sub-conditions.
- an affine application $f_{1\rightarrow 2} : \mathbb{D}_1 \rightarrow \mathbb{D}_2$

This mean that the statement S_1 at iteration point $\bar{z} \in \mathbb{D}_1$ depends on S_2 at iteration point $f_{1\rightarrow 2}(\bar{z}) \in \mathbb{D}_1$ if $q_{1\rightarrow 2}(\bar{z}, \bar{g}) \geq 0$.

17.3.4 Polyhedron associate to a dependence

An affine application $f : \mathbb{E}_1 \to \mathbb{E}_2$ can be represent as a polyhedron $\mathbb{P} \subset \mathbb{E}_1 \times \mathbb{E}_2$.
Given $\mathbb{P}_{1\to2} \subset \mathbb{Z}^{n_1+n_2+n_g}$ where n_1 and n_2 and n_g are respectively the number of iterators of S_1 and S_2, and the number of globals.

$\mathbb{P}_{1\to2}$ will be the polyhedron associate to the dependence between S_1 and S_2 iff:

$$\forall(\bar{z_1}, \bar{z_2}, \bar{g}) \in \mathbb{Z}^{n_1+n_2+n_g}$$
$$(\bar{z_1} \in \mathbb{D}_1 \wedge \bar{z_2} \in \mathbb{D}_2 \wedge \bar{g} \in \mathbb{G} \wedge q_{1\to2}(\bar{z_1}, \bar{g}) \geq 0 \wedge \bar{z_2} = f_{1\to2}(\bar{z_1})\ )$$
$$\Leftrightarrow$$
$$\bar{z_1} \,\square\, \bar{z_2} \,\square\, \bar{g} \in \mathbb{P}_{1\to2}$$

17.3.5 Example

```
1 for (i = 0; i < n; i++) {
2       s[i] = 0
3       for (j = 0; j < n; j++) {
4             s[i] = s[i] + a[i][j] * x[j]
5       }
6 }
```

Figure 17.1: A sample program

Program

Consider the small program of figure 17.1. It contains a reference to one global variable $\mathbb{G} = \{n\}$ and we have two statement S_1 and S_2, respectively at line 2 and 4, with associated domains:
$\mathbb{D}_1 = \{i | 0 \leq i \leq n-1\}$
$\mathbb{D}_2 = \{i, j | 0 \leq i, j \leq n-1\}$

Dependences

A dependence analysis (from FADAlib for example) would give us:

- S_1 has no dependence (it never read a variable)
- S_2 has dependence at iteration point (i, j):
  - on S_1 at iteration point (i): if $j = 0$
  - on S_2 at iteration point $(i, j-1)$: if $j \geq 1$

So, the dependence graph is composed from two nodes $\{S_1, S_2\}$ and two edges:

- $S_2 \to S_1$
  - $q_{S_2\to S_1}(i, j, n) = (j, -j)$
  - $f_{S_2\to S_1}(i, j) = (i)$
- $S_2 \to S_2$
  - $q_{S_2\to S_2}(i, j, n) = j - 1$
  - $f_{S_2\to S_2}(i, j) = (i, j-1)$

Polyhedron

We have two dependances, then two polyhedrons

- $\mathbb{P}_{S_2 \to S_1}$ associate with the following affine application:
  $\mathcal{P}_{S_2 \to S_1} : \mathbb{Z}^{2+1+1} \to \mathbb{Z}^{4+2+2+2}$

$$\mathcal{P}_{S_2 \to S_1}(i, j, i', n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ i' \\ n \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- $\mathbb{P}_{S_2 \to S_2}$ associate with the following affine application:
  $\mathcal{P}_{S_2 \to S_2} : \mathbb{Z}^{2+2+1} \to \mathbb{Z}^{4+4+1+4}$

$$\mathcal{P}_{S_2 \to S_2}(i, j, i', j', n) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ n \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ 0 \\ -1 \\ 0 \\ -1 \\ 0 \\ -1 \\ -1 \\ 0 \\ 0 \\ -1 \\ 1 \end{pmatrix}$$

17.4 Generating affine schedule with polyhedric model

17.4.1 What is this ?

Says that an operation is the execution of statement $S \in \mathbb{S}$ at an iteration point $\bar{z} \in \mathbb{D}_S$, we will note it $<S, \bar{z}>$. A schedule Θ will associate a date to an operation $<S, \bar{z}>$. An affine schedule is a set (one by statement) of affine applications $\Theta_S : \mathbb{D}_S \times \mathbb{G} \to \mathbb{Z}^n$ where n is the dimension of the schedule (date).

We look for 1-dimensional schedules only. In this case:

$$\Theta_S(\bar{z}, \bar{g}) = \bar{\alpha_S}.\bar{z} + \bar{\beta_S}.\bar{g} + \kappa_S$$

17.4.2 Generating the set of Valid Schedule

Valid Schedule ?

A valid schedule is a schedule that respect the dependences of a program. So, one of such schedule need to respect:

$$\forall (S_1, S_2) \in \mathbb{S},\ \mathbb{P}_{S_2 \to S_1} \neq \emptyset$$

$$\Longrightarrow$$

$$\forall\ \bar{z_2} \,\square\, \bar{z_1} \,\square\, \bar{g} \in \mathbb{P}_{S_2 \to S_1},\ \Theta_{S_1}(\bar{z_1}, \bar{g}) + \Delta_{S_1} \leq \Theta_{S_2}(\bar{z_2}, \bar{g})$$

$$\Longrightarrow$$

$$\forall\ \bar{z_2} \,\square\, \bar{z_1} \,\square\, \bar{g} \in \mathbb{P}_{S_2 \to S_1},\ \Theta_{S_2}(\bar{z_2}, \bar{g}) - \Theta_{S_1}(\bar{z_1}, \bar{g}) - \Delta_{S_1} \geq 0$$

where:

- $\mathbb{P}_{S_2 \to S_1}$ is the polyhedron associated to the dependence from S_2 to S_1, empty if this dependence do not exist;
- Δ_{S_1} is the latence of the S_1 statement execution, considered equal to 1 in following devellopement.

Farkas algorithm

Theorem: Affine Form of Farkas Lemma

Let $\mathbb{P}$ be a nonempty polyhedron defined by p affine inequalities:

$$\bar{z} \in \mathbb{P} \iff \forall k \in [\![1, p]\!], a_k.\bar{z} + b_k \geq 0$$

Then an affine form ψ is non negative everywhere in $\mathbb{P}$ iff it is a positive affine combination:

$$\forall \bar{z} \in \mathbb{P}, \psi(\bar{z}) \geq 0$$

$$\iff$$

$$\forall k \in [\![0, p]\!],\ \exists \lambda_k \geq 0 \text{ such as } \forall \bar{z} \in \mathbb{P},\ \psi(\bar{z}) = \lambda_0 + \sum_{k=1}^{p} \lambda_k.(a_k.\bar{z} + b_k)$$

With the valid schedule definition and the above theorem, we have:

$$\forall (S_1, S_2) \in \mathbb{S},\ \mathbb{P}_{S_2 \to S_1} \neq \emptyset$$

$$\Longrightarrow$$

$$\forall k \in [\![0, p]\!],\ \exists \lambda_k \geq 0 \text{ such as } \forall\ \bar{z_2} \,\square\, \bar{z_1} \,\square\, \bar{g} \in \mathbb{P}_{S_2 \to S_1},$$

$$\Theta_{S_2}(\bar{z_2}, \bar{g}) - \Theta_{S_1}(\bar{z_1}, \bar{g}) - 1 = \lambda_0 + \sum_{k=1}^{p} \lambda_k.Row_k^{\mathbb{P}}(\bar{z_2} \,\square\, \bar{z_1} \,\square\, \bar{g})$$

Example

We still consider the small program of figure 17.1.
Firstly, we need to remove from polyhedron some redondante constraints:

$$\mathcal{P}_{S_2\to S_1}(i,j,i',n) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ i' \\ n \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\mathcal{P}_{S_2\to S_2}(i,j,i',j',n) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} i \\ j \\ i' \\ j' \\ n \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \\ -1 \\ -1 \\ 0 \\ 0 \\ -1 \\ 1 \end{pmatrix}$$

We also consider the following expressions for Θ_{S_1} and Θ_{S_2}:

$$\Theta_{S_1}(i,n) = \alpha^1_{S_1} \times i + \beta^1_{S_1} \times n + \kappa_{S_1}$$

$$\Theta_{S_2}(i,j,n) = \alpha^1_{S_2} \times i + \alpha^2_{S_2} \times j + \beta^1_{S_2} \times n + \kappa_{S_2}$$

The application of Farkas Lemma give us:

- for $S_2 \to S_1$:

$$\Theta_{S_2}(i,j,n) - \Theta_{S_1}(i',n) - 1 = \lambda_0^{S_2\to S_1} + \sum_{k=1}^{6} \lambda_k^{S_2\to S_1}.Row_k^{\mathbb{P}}(i,j,i',n)$$

$$\left\{\begin{array}{rcl} \alpha^1_{S_2} & = & \lambda_1^{S_2\to S_1} - \lambda_2^{S_2\to S_1} + \lambda_5^{S_2\to S_1} - \lambda_6^{S_2\to S_1} \\ \alpha^2_{S_2} & = & \lambda_3^{S_2\to S_1} - \lambda_4^{S_2\to S_1} \\ -\alpha^1_{S_1} & = & \lambda_6^{S_2\to S_1} - \lambda_5^{S_2\to S_1} \\ \beta^1_{S_2} - \beta^1_{S_1} & = & \lambda_2^{S_2\to S_1} \\ \kappa_{S_1} - \kappa_{S_2} - 1 & = & \lambda_0^{S_2\to S_1} - \lambda_2^{S_2\to S_1} \\ \forall k \in [\![0,6]\!], & & \lambda_k^{S_2\to S_1} \geq 0 \end{array}\right.$$

- for $S_2 \to S_2$:

$$\Theta_{S_2}(i,j,n) - \Theta_{S_2}(i',j',n) - 1 = \lambda_0^{S_2\to S_2} + \sum_{k=1}^{6} \lambda_k^{S_2\to S_2}.Row_k^{\mathbb{P}}(i,j,i',j',n)$$

$$\left\{\begin{array}{rcl} \alpha^1_{S_2} & = & \lambda_1^{S_2\to S_2} - \lambda_2^{S_2\to S_2} + \lambda_5^{S_2\to S_2} - \lambda_6^{S_2\to S_2} \\ \alpha^2_{S_2} & = & -\lambda_3^{S_2\to S_2} + \lambda_4^{S_2\to S_2} + \lambda_7^{S_2\to S_2} - \lambda_8^{S_2\to S_2} \\ -\alpha^1_{S_2} & = & -\lambda_5^{S_2\to S_2} + \lambda_6^{S_2\to S_2} \\ -\alpha^2_{S_2} & = & -\lambda_7^{S_2\to S_2} + \lambda_8^{S_2\to S_2} \\ 0 & = & \lambda_2^{S_2\to S_2} + \lambda_3^{S_2\to S_2} \\ -1 & = & \lambda_0^{S_2\to S_2} - \lambda_2^{S_2\to S_2} - \lambda_3^{S_2\to S_2} - \lambda_4^{S_2\to S_2} - \lambda_7^{S_2\to S_2} + \lambda_8^{S_2\to S_2} \\ \forall k \in [\![0,8]\!], & & \lambda_k^{S_2\to S_2} \geq 0 \end{array}\right.$$

Dimension of the Valid Schedule space

Chapter 18

ROSE Tests

18.1 How We Test

ROSE includes a number of test codes. These test codes test:

1. Robustness of translators built using ROSE.
   A test translator (`testTranslator`) is built and it is used to process a number of test codes (both the compilation of the test code and the compilation of the generated source code it tested to make sure that they both compile properly). No execution of the generated code is attempted after compilation. These tests are used to verify the proper operation of ROSE as part of the standard SVN check-in process for all developers.
2. Execution of the code generated by the translator built using ROSE.
   Here tests are done to verify that the translator generated correct code that resulted in the same result as the original code.
3. Robustness of the internal mechanisms within ROSE.
   Here tests are done on separately developed features within the ROSE infrastructure (e.g. the AST Rewrite Mechanism, Loop Optimizations, etc.).

Specific directories of tests include:

- CompileTests
  This directory contains code fragments that test the internal compiler mechanisms. Many code fragments or whole codes are present either have previously or continue to present problems in the compilation (demonstrate bugs). The CompileTests directory consists of several directories. The `README` file in the CompileTests directory gives more specific information.

  The test codes developed here are intended to be a small test of ROSE (a much larger regression test suit will be available separately; and is used separately). These tests are divided into caegories:

  1. C_tests
     These are tests of the differences between the C subset of C++ and C. Specifically these are typically C codes that will not compile with a C++ compiler, even under the subset of C language rules used to invoke the subset of C (-rose:C or -rose:C_only) UNLESS the source files have the ".c" suffix, as opposed to any other suffix (e.g. ".C"). These are all specific to the C89 standard, which is what is typically assumed when refering to the C language (C99 is covered separately).

2. C99_tests
   These are tests specific to C99 (new features not in C).
3. UPC_tests
   These are tests that are specific to UPC modifiers, recognized by EDG and handled in the Sage III AST. This support for UPC does not constitute a UPC compiler, a UPC specific runtime system would be required for that.
4. Cxx_tests
   These are the C++ test files (there are more tests here than elsewhere).
5. C_subset_of_Cxx_tests
   This is the subset of C++ represented by the C language rules (it is not all of C). There are test codes here which contain `#if __cplusplus` to represent some differences between the syntax of C and C++ (typically `enum` and `struct` specifiers are requird for C where they are not required for C++.
6. RoseExample_tests
   These are examples of ROSE project source code, and testing using ROSE to compile examples of ROSE source code.
7. PythonExample_tests
   These tests use the `Python.h` header file and are part of tests of code generated by SWIG.
8. ExpressionTemplateExample_tests
   These are a number of tests demonstrating the use of expression templates. They are separated out because they take a long time to compile using ROSE. This is part of work to understand why expression templates take so long to compile generally.

- roseTests
  This directory tests the internal ROSE infrastructure. It contains separate subdirectories for individual parts of ROSE. See ROSE/tests/nonsmoke/functional/roseTests/README for details.

E: *Complete the list of ries that hold tests (in ROSE/tests directory).*

Chapter 19

Testing Within ROSE

19.1 Introduction

Testing is a particularly important part of the ROSE project. We credit our particularly high level of automated testing for why we are able to maintain a compiler project such as ROSE with so few people. All testing is automated and the results of the testing drives higher levesl of testing (from developer testing, to automated internal daily testing, to external nightly testing). Different levels of testing with different types of feedback provide a basis for the health of the project to be visible to either the individual developer, the rest of the team, or to an external audience.

Out testing exists on three levels, and uses external tools (Hudson: http://wiki.hudson-ci.org/display/HUDSON/Meet+Hudson).

1. Hudson test each "git" branch before it gets checked into the internal SVN repository. For this we use an internal compile farm of about 7 machines and test different versions of configure options (including different versions of compilers).
2. Test the internal SVN repository on different architectures, and if it passes push it out to the external repository.
3. Checkout the external SVN repository and have it be tested on the external compile farm of about 25 machines (with different OS's) at NMI (University of Wisc, Madison).

We test so that we can detect errors as quickly as possible and provide as much transparency as possible to our external users. Our goal is to communicate honestly so that we can avoid any misunderstandings, it is in our best interest to have this policy.

ROSE uses `autoconf`, `automake`, and `libtool`. As a result our generated *Makefiles* follow the GNU standards for makefiles and contain the associated makefile rules:

- `make` : used to compile all of ROSE.
- `make check`: used to run tests in each directory.
- `make distcheck`: used to build a GNU standard distribution and run `check` on on that distribution.

Additional stepts such a `build` and `configure` are covered in the ROSE Installation guide.

This chapter is organized by laying out the different ways in which ROSE is tests. All testing scripts and the information required for our approach to be used with other projects is made available in the ROSE distribution.

Section 19.2 explains why we maintain two different SVN repositories internally and externally. Section 19.3 explains what tests are done by developers, before checkin to SVN. Section 19.4 explains what daily internal tests are done. Section 19.5 explains what daily external tests are done.

19.2 Tail of Two SVN Repositories

We maintain two SVN repositories, one internal and the other external. The internal SVN repository is only available to the ROSE core development team. After passing specific tests, parts of the internal SVN repository as submitted to the external SVN repository (hosted by SciDAC at Lawrence Berkley Laboratory). The external SVN repository has public visability. It is always a snapshot of the internal SVN repository (for the revision that passed the required tests to be made externally available). Importantly, the SVN repository is always trying to be a working copy of ROSE. Non-compilable code should neve be checked in, all of ROSE as it is represented in the SVN repository should pass `make distcheck`. Atuomated tests verify this or identify developers who are at fault. Peer pressure is generally enough so that everyone fixes their code in the morning as a result of results of nightly tests.

The external SVN repository also has a number of branches, these support external collaborations and represent a community of people who contribute work to the ROSE project (which is a community open source project).

19.3 Developer Tests

Developer tests are required to run and pass `make distcheck` before checking in work to the internal SVN repository. It is at their discression if they want to only run `make check`, which is faster and less through. In general all developers know who is careful and who is not and interally we can deal with people who break the ROSE builds too often. If the developers test pass then the work can be checked into the internal SVN repository.

19.4 Daily Internal Tests

There are daily internal tests at LLNL (twice per day) that use the internal SVN repository and checkout versions of automated tests on three different platforms:

- 32-bit Linux (Red Hat system 5)
- 64-bit Linux (Red Hat system 4)
- Max OSX version 10.5 (x86)

There three system are used to evaluate and report the status of ROSE. Crontabs are used to initiate the tests. The crontabs call scripts in the ROSE/scripts .

We should provide an ...e of the crontabs used ...put of 'crontab -l', for example)

19.5 Daily External Tests

There are nightly external tests on the NSF Middleware Initiative compile farm at the University of Wisconson. The results of these tests are made public via link on the main ROSE web page. This provides transparency to our external collaborators and that is is public help keep us at our best.

/home/yuan5/qmsh/qmalt.sh

```
# Usage

if (( $# < 4 )); then
  echo "Usage: qm.sh <f|o> <QMTest test class> <ROSE translator> <Backend Compiler|NULL> {compiler arguments...} {Test arguments (-testopt:<>)...}"
  exit 1
fi

# Functions

includeFullPath () {
  local BACK=`pwd`

  ARG=`echo $ARG | sed -e "s/-I//g"`
  cd $ARG
  ARG=-I`pwd`

  cd $BACK
  return 0
} # get the absolute path of all include directories

################################################################################

# Globals

declare -i COUNT=0
declare -i FLAG=0

TEST=BADTEST.qmt# error in test creation
MODE=$1# The naming mode of the script
TEST_CLASS=$2# QMTest class
PROGRAM=$3# executable name
BACKEND=$4# The execution string with backend compiler
ARGUMENTS="['-I$PWD',"# argument stub general
OFILE=""# The original object file

###############################################################################

for ARG in $@
do
  ((COUNT++))

  if ((COUNT > 4)); then

    if [[ ${ARG:0:9} == "-testopt:" ]]; then
      ARGUMENTS="${ARGUMENTS} '`echo $ARG | sed -e 's/-testopt://g'`', "
      continue
    fi # parse out specific options to test only and not to backend

    BACKEND="${BACKEND} $ARG" # build original compile-line

    #case#######################################################################


    case $ARG in
-I*) includeFullPath;;

*.c | *.cpp | *.C | *.[cC]* )
          if [[ ${ARG:0:1} != '/' ]]; then
            ARG="`pwd`/$ARG"
          fi # take care of absolute paths

  # rename the QMTest output test file. Replace space, period, and plus
  # with their equivalents and change all chars to lower case.
  if [[ $MODE = 'f' ]]; then
            TEST=`echo $ARG | sed -e 's/\//_/g' | sed -e 's/\./_dot_/g' | \
  sed -e 's/+/plus/g' | gawk '{print tolower($0)}'`.qmt
```

```
  fi
;; # case C/C++ source files

-o)
  if [[ $MODE = 'o' ]]; then
    FLAG=1
  elif [[ $MODE = 'f' ]]; then
    FLAG=2
  fi # spike out the object flag

  continue
;; # case -o

*) ;; # default
    esac

    #esac#fi####################################################################


    if ((FLAG > 0)); then
      OFILE=$ARG
    fi # name the object file after -o declaration

    if ((FLAG == 1)); then
      if [[ ${ARG:0:1} != '/' ]]; then
        ARG="`pwd`/$ARG"
      fi # if argument not specified with absolute path then append PWD

      # rename the QMTest output test file, replace space, period, and plus
      # with equivalent symbols and change all chars to lower case.
      TEST=`echo $ARG | sed -e 's/\//_/g' | sed -e 's/\./_dot_/g' | \
    sed -e 's/+/plus/g' | gawk '{print tolower($0)}'`.qmt

      FLAG=0# reset FLAG
      continue
    elif ((FLAG == 2)); then
      FLAG=0# reset FLAG
      continue
    fi # if the -o flag used; create the object name and TEST name from object

    #fi#if######################################################################


    ARGUMENTS="${ARGUMENTS} '$ARG', "
  fi # if argument is not qm.sh argument

  #fi#done######################################################################


done # for all command-line arguments to qm.sh

OBJECT=${TEST%%.*}.o# name the object after the test file name
ARGUMENTS="${ARGUMENTS} '-o', '$OBJECT']"

#done#case######################################################################


case $TEST_CLASS in
  strings.SubStringTest) qmtest create -o $TEST -a program="$PROGRAM" -a substring="ERROR SUMMARY: 0 errors from 0 contexts"
-a arguments="$ARGUMENTS" test $TEST_CLASS;;

  *) qmtest create -o "$TEST" -a program="$PROGRAM" -a arguments="$ARGUMENTS" test $TEST_CLASS;;
esac # create qmtest test file with test class

#esac#main######################################################################


if [[ ${BACKEND:0:4} == "NULL" ]]; then
  touch $OFILE >& /dev/null # create dummy file and pipe error to NULL
  exit 0    # always exit 0
fi # skip backend compilation

# Execute backend compilation with original compile-line
$BACKEND
exit $?
```

19.6 QMTest: Introduction

`qm.sh` is a wrapper for the compile-line in an arbitrary project build system that creates qmtest test files that test ROSE. The basic assumption is that it is possible to isolate and modify the compile-line command in most project build systems. For example, Makefile systems using `make` specify compile-line commands after labels delimited by a colon. One example of this may be:

```
gcc -c -g -Wall hello.c
```

From this line `qm.sh` would create a qmtest test file that executes a ROSE translator in the place of `gcc` but with the exact same arguments `-c -g -Wall` and more if the user of `qm.sh` should specify them. `qm.sh` also mimics the compile-line process of the project's build system so that all dependencies are built as normal by the backend compiler.

19.6.1 Usage

`qm.sh <f|o> <QMTest test class> <ROSE translator> <Backend compiler> {compiler arguments...} {ROSE arguments...}`

`<f|o>` : The output file naming mode. Option "f" specifies `qm.sh` to use source file names in naming output `.qmt` files. Option "o" specifies `qm.sh` to use object file names, as specified by the compile-line `-o` flag to the backend compiler, for naming `.qmt` output files.

<QMTest test class> : The test class of the created test file, i.e rose.RoseTest or command.ExecTest.

<ROSE translator> : The full path specifying a ROSE translator.

<Backend compiler> : The name of the backend compiler used in the normal compilation of the project build system. Specify "NULL" as the <Backend compiler> if you want to skip backend compilation.

{compiler arguments...} : The arguments specified on the command-line of the project build system.

{ROSE arguments...} : The arguments to the ROSE translator prefixed with -rose:<ROSE argument>, e.x. `-rose:--edg:no_warnings`. Note, these may be placed anywhere after the <Backend Compiler> argument.

19.6.2 Variables

COUNT : The for loop counter, keeps track of number of `qm.sh` arguments.

FLAG : Logical flag variable used in naming output `.qmt` files.

TEST : The name of the QMTest test file created.

TEST_CLASS : The QMTest class specified on command-line.

PROGRAM : The ROSE translator specified on command-line.

BACKEND : The original command-line of the project build system with the backend compiler.

ARGUMENTS : The compile-line arguments specified on the command-line with any script user specified arguments for the ROSE translator such as `--edg:no_warnings` bound for the QMTest test file.

LAST_ARG : The closing stub to the QMTest arguments format along with the `-o <object file name>` argument.

ARG : The current compile-line argument place holder used in constructing the argument format to QMTest arguments `ARGUMENTS="['arg1','arg2',...,'argN']"`.

19.6.3 Execution Walkthrough

`qm.sh` is broken into code blocks which each perform some procedure. These blocks are delimited with a solid line of 80 `#` characters.

```
1 for ARG in $@
2 do
3   ((COUNT++))

4   if ((COUNT > 3)); then

5     if [[ ${ARG:0:6} == "-rose:" ]]; then
6       ARGUMENTS="${ARGUMENTS} '`echo $ARG | sed -e 's/-rose://g'`', "
7       continue
8     fi

9     BACKEND="${BACKEND} $ARG" # build original compile-line
```

Figure 19.1: Backend and ROSE argument construction block

19.6.4 Backend and ROSE arguments

This block of code builds the original compile line of the project's build system along with the arguments passed specifically to the ROSE compiler. In the `for` loop all the arguments passed to `qm.sh` are looped through, however, the first three arguments are skipped due to the `if` statement on line 4. All other arguments after the third are considered arguments of either ROSE or the original project's build system. ROSE arguments must be prefixed with `-rose:<ROSE argument>` when specified on the compile-line. Each argument with this prefix is stripped of the prefix `-rose:` and added to the `ARGUMENT` list of the QMTest test file. ROSE arguments are not carried over to the `BACKEND` compile-line variable but all other arguments are appended without change with the exception of the `-o <Object file Name>` flag.

19.6.5 Relative Path Compile-line Arguments

```
case $ARG in
    -I*) includeFullPath;;

    [!/]*.c | [!/]*.cpp | [!/]*.C | [!/]*.[cC]* ) ARG="`pwd`/$ARG";;

    -o) BOOL=1 ; continue;; # spike out -o outputfilename

    *) ;;
esac
```

Figure 19.2: Relative to Absolute Paths in Arguments

This block of code handles all compile-line arguments containing relative file or include paths. The `case...esac` switch statement compares against patterns indicative of C/C++ source files or an include directive. All source files without absolute paths stemming from root are simply appended with their present working directory. Directories specified by the `-I` include directive call the function `includeFullPath` which changed relative paths to absolute paths.

19.6.6 Naming QMTest Files

At this block of code it is assumed that `ARG` contains name of either the source or object file specified by the command-line. This name is must first contain its absolute path to prevent name collisions which is handled by the `if` construct on lines 1-3. The `TEST` name is then created on line 4 by replacing any `'/'`(forward slashes) or

```
if [[ ${ARG:0:1} != ’/’ ]]; then
  ARG=‘pwd‘/$ARG
fi

TEST=‘echo $ARG | sed -e ’s/\//_/g’ | sed -e ’s/\./_/g’ | gawk ’{print tolower($0)}’‘.qmt

OBJECT=${TEST%%.*}.o
```

Figure 19.3: Naming procedure for QMTest Files

`'.'`(dots) in `ARG` with underscores. The `OBJECT` name is simply the `TEST` name value with the `.o` extension. The object file name argument held in `OBJECT` is appended to the end of the QMTest argument list along with the `-o` flag. Note that QMTest does not allow capital alphabetic letters or periods in the names of individual tests.

19.6.7 Create QMTest test and Execute Backend

```
1 qmtest create -o "$TEST" -a program="$PROGRAM" -a arguments="$ARGUMENTS" test $TEST_CLASS;;

2 $BACKEND # Execute the old command-line to fake the makefile
3 exit $?
```

Figure 19.4: Create .qmt and Execute Backend

Line 1 creates a `.qmt` QMTest file with the name `TEST` that executes `PROGRAM` with arguments `ARGUMENTS` using the class `TEST_CLASS`. The `.qmt` test file is created in the present working directory of the project's build system file structure under the "make" process. Lines 2-3 execute the reconstructed original backend compile-line of project's build system. The script `qm.sh` exits with the same code as the exit status of the backend process.

19.6.8 Example

The following example edits a trivial makefile and builds QMTest files with `qm.sh` by editing the makefile.

```
CXX = g++
CFLAGS = -g -Wall

CPU.out : main.o registers.o reader.o decoder.o
        $(CXX) $(CFLAGS) -o CPU.out reader.o registers.o decoder.o main.o

main.o : main.c registers.h reader.h decoder.h instruction.h
        $(CXX) $(CFLAGS) -c main.c -o main.o

registers.o : registers.c registers.h main.h
        $(CXX) $(CFLAGS) -c registers.c -o registers.o

reader.o : reader.c reader.h instruction.h
        $(CXX) $(CFLAGS) -c reader.c -o reader.o

decoder.o : decoder.c decoder.h
        $(CXX) $(CFLAGS) -c decoder.c -o decoder.o
```

Figure 19.5: makefile before editing

By inserting the `qm.sh` wrapper before each instance of `g++` in this case it is possible to generate `.qmt` test files. The modified makefile is shown below:

```
QM = /home/yuan5/RoseQMTest/scripts/qm.sh
ROSE = /home/yuan5/bin/identityTranslator
MYCC = $(QM) rose.RoseTest $(ROSE)
CXX = $(MYCC) g++
ROSEFLAGS = -rose:--edg:no_warnings
CFLAGS = $(ROSEFLAGS) -g -Wall

CPU.out : main.o registers.o reader.o decoder.o
        $(CXX) $(CFLAGS) -o CPU.out reader.o registers.o decoder.o main.o

main.o : main.c registers.h reader.h decoder.h instruction.h
        $(CXX) $(CFLAGS) -c main.c -o main.o

registers.o : registers.c registers.h main.h
        $(CXX) $(CFLAGS) -c registers.c -o registers.o

reader.o : reader.c reader.h instruction.h
        $(CXX) $(CFLAGS) -c reader.c -o reader.o

decoder.o : decoder.c decoder.h
        $(CXX) $(CFLAGS) -c decoder.c -o decoder.o
```

Figure 19.6: makefile after editing

After the edits have taken place it is evident that `qm.sh` wraps around each compile-line of the makefile. The arguments to `qm.sh` are themselves encompassed by the variable `MYCC` leaving minimal edits to the makefile itself. The makefile may now be run with `make` and the project will be made along with all the QMTest `.qmt` files.

```
bash-2.05b$ make
/home/yuan5/RoseQMTest/scripts/qm_file.sh rose.RoseTest /home/yuan5/bin/identityTranslator
g++ -rose:--edg:no_warnings -g -Wall -c main.c
/home/yuan5/RoseQMTest/scripts/qm_file.sh rose.RoseTest /home/yuan5/bin/identityTranslator
g++ -rose:--edg:no_warnings -g -Wall -c registers.c
/home/yuan5/RoseQMTest/scripts/qm_file.sh rose.RoseTest /home/yuan5/bin/identityTranslator
g++ -rose:--edg:no_warnings -g -Wall -c reader.c
/home/yuan5/RoseQMTest/scripts/qm_file.sh rose.RoseTest /home/yuan5/bin/identityTranslator
g++ -rose:--edg:no_warnings -g -Wall -c decoder.c
/home/yuan5/RoseQMTest/scripts/qm_file.sh rose.RoseTest /home/yuan5/bin/identityTranslator
g++ -rose:--edg:no_warnings -g -Wall -o CPU.out reader.o registers.o decoder.o main.o
```

Figure 19.7: `make` output

```
bash-2.05b$ find . -name "*.qmt"
_home_yuan5_roseqmtest_project_p2_cpu_out.qmt
_home_yuan5_roseqmtest_project_p2_decoder_c.qmt
_home_yuan5_roseqmtest_project_p2_main_c.qmt
_home_yuan5_roseqmtest_project_p2_reader_c.qmt
_home_yuan5_roseqmtest_project_p2_registers_c.qmt
```

Figure 19.8: `find . -name "*.qmt"` output

19.6.9 Running the Tests

This section describes how to collect and run the test created by `qm.sh` after building the project with an edited build system. When the project has completed building, the QMTest files will most likely be scattered across all the local directories containing their object file counterparts. Thus it's necessary to collect them all into one directory which will serve as a QMTest database. From the directory where `make` or the project's build system was launched type the command:

```
find . -name "*.qmt" -exec mv {} test_database \;
```

This will recursively find all files with extensions `.qmt` and move them to the directory test_database which was created by the user. Change directory to test_database and type the command:

```
qmtest -D`pwd` create-tdb
```

This command will allow QMTest to access the test files by creating a test database. Once this test database has been created by QMTest it is possible to run tests from the command-line or GUI with the respective commands:

```
qmtest run -o results.qmr
# runs command-line and writes QMTest output to results.qmr

qmtest gui
# runs the QMTest GUI by which the user may read results stored in results.qmr
# or run additional tests.
```

Chapter 20

Appendix

This appendix covers a number of relevant topics to the use of ROSE which have not been worked into the main body of text in the ROSE User Manual.

FIXME: *The sect this Appendix are while we figure out belong in the ROSE U (or*

20.1 Error Messages

The user will mostly only see error messages from EDG, these will appear like normal C++ compiler error messages.

These can be turned off using the EDG option:
–edg:no_warnings
or
-edg:w
on the command-line of any translator built using ROSE.

20.2 Specifying EDG options

The EDG options are specified using –edg:<edg option> for EDG options starting with "–" or -edg:<edg option> for EDG options starting with "-".

The details of the EDG specific options are available at:
http://www.edg.com/docs/edg_cpp.pdf available from the EDG web page at:
http://www.edg.com/cpp.html

20.3 Easy Mistakes to Make: How to Ruin Your Day as a ROSE Developer

There are a few ways in which you can make mistakes within the development of the ROSE project:

1. Never run `configure` in your source tree. If you do, then never run `make distclean`, since this will remove many things required to develop ROSE. Things removed by `make distclean` are:
   (a) documentation (including several of the directories in `ROSE/docs/Rose`)

20.4 Handling of source-filename extensions in ROSE

On case-sensitive systems, ROSE handles .c as the (only) valid filename extension for c-language and .cc, .cp, .c++, .cpp, .cxx, as the valid filename extensions for C++ language. On case-insensitive systems, ROSE handles .c and .C as valid filename extensions for c-language, and .cc, .cp, .c++, .cpp, .cxx, .CC, .CP, .C++, .CPP, .CXX as valid filename extensions for C++.

There are some inconsistencies in the filename handler such as: (1) not recognizing .CC, .CP, .C++, .CPP, .CXX as valid filename extensions for C++ language on case-sensitive systems and (2) not recognizing .CxX, .cPp, etc. as valid filename extensions for C++ language on case-sensitive systems. The sole reason for the inconsistency is that of compatibility with GNU (as well as EDG).

20.5 IR Memory Consumption

The Internal Representation is used to build the AST and, for large programs, it can translate into a large number of IR nodes. Typically the total number of IR nodes is about seven times the number of lines of codes (seems to be a general rule, perhaps a bit more when templates are used more dominantly). The memory consumption of any one file is not very significant, but within support for whole program analysis, the size of the AST can be expected to be quite large. Significant sharing of declarations is made possible via the AST merge mechanisms. C and C++ have a One-time Definition Rule (ODR) that requires definitions be the same across separate compilations of files intended to be linked into a single application. ODR is significantly leveraged within the AST merge mechanism to share all declarations that appear across multiple merged files. Still, a one-million line C++ application making significant use of templates can be expected to translate into 10-20 million IR nodes in the AST, so memory space is worth considering.

The following is a snapshot of current IR node frequency and memory consumption for a moderate 40,000 line source code file (one file calling a number of header files). Note that the Sg_File_Info IR nodes are most frequent and consume the greatest amount of memory. This reflects our bias toward preserving significant information about the mapping of language constructs back to the positions in the source file to support a rich set of source-to-source functionality.

```
AST Memory Pool Statistics: numberOfNodes = 114081 memory consumption = 5019564 node = Sg_File_Info
AST Memory Pool Statistics: numberOfNodes =  31403 memory consumption =  628060 node = SgTypedefSeq
AST Memory Pool Statistics: numberOfNodes =  14254 memory consumption =  285080 node = SgStorageModifier
AST Memory Pool Statistics: numberOfNodes =  14254 memory consumption = 1140320 node = SgInitializedName
AST Memory Pool Statistics: numberOfNodes =   8458 memory consumption =  169160 node = SgFunctionParameterTypeList
AST Memory Pool Statistics: numberOfNodes =   7868 memory consumption = 1101520 node = SgModifierType
AST Memory Pool Statistics: numberOfNodes =   7657 memory consumption =  398164 node = SgClassType
AST Memory Pool Statistics: numberOfNodes =   7507 memory consumption = 2071932 node = SgClassDeclaration
AST Memory Pool Statistics: numberOfNodes =   7060 memory consumption =  282400 node = SgTemplateArgument
AST Memory Pool Statistics: numberOfNodes =   6024 memory consumption =  385536 node = SgPartialFunctionType
AST Memory Pool Statistics: numberOfNodes =   5985 memory consumption = 1388520 node = SgFunctionParameterList
AST Memory Pool Statistics: numberOfNodes =   4505 memory consumption = 1477640 node = SgTemplateInstantiationDecl
AST Memory Pool Statistics: numberOfNodes =   3697 memory consumption =  162668 node = SgReferenceType
AST Memory Pool Statistics: numberOfNodes =   3270 memory consumption =  758640 node = SgCtorInitializerList
AST Memory Pool Statistics: numberOfNodes =   3178 memory consumption =   76272 node = SgMemberFunctionSymbol
AST Memory Pool Statistics: numberOfNodes =   2713 memory consumption =  119372 node = SgPointerType
AST Memory Pool Statistics: numberOfNodes =   2688 memory consumption =  161280 node = SgThrowOp
AST Memory Pool Statistics: numberOfNodes =   2503 memory consumption =   60072 node = SgFunctionSymbol
AST Memory Pool Statistics: numberOfNodes =   2434 memory consumption =  107096 node = SgFunctionTypeSymbol
AST Memory Pool Statistics: numberOfNodes =   2418 memory consumption =  831792 node = SgFunctionDeclaration
AST Memory Pool Statistics: numberOfNodes =   2304 memory consumption =   55296 node = SgVariableSymbol
AST Memory Pool Statistics: numberOfNodes =   2298 memory consumption =  101112 node = SgVarRefExp
AST Memory Pool Statistics: numberOfNodes =   2195 memory consumption =  114140 node = SgSymbolTable
AST Memory Pool Statistics: numberOfNodes =   2072 memory consumption =  721056 node = SgMemberFunctionDeclaration
AST Memory Pool Statistics: numberOfNodes =   1668 memory consumption =  400320 node = SgVariableDeclaration
AST Memory Pool Statistics: numberOfNodes =   1667 memory consumption =  393412 node = SgVariableDefinition
AST Memory Pool Statistics: numberOfNodes =   1579 memory consumption =  101056 node = SgMemberFunctionType
AST Memory Pool Statistics: numberOfNodes =   1301 memory consumption =   31224 node = SgTemplateSymbol
AST Memory Pool Statistics: numberOfNodes =   1300 memory consumption =  364000 node = SgTemplateDeclaration
AST Memory Pool Statistics: numberOfNodes =   1198 memory consumption =  455240 node = SgTemplateInstantiationMemberFunctionDecl
AST Memory Pool Statistics: numberOfNodes =   1129 memory consumption =   54192 node = SgIntVal
AST Memory Pool Statistics: numberOfNodes =   1092 memory consumption =   56784 node = SgAssignInitializer
AST Memory Pool Statistics: numberOfNodes =   1006 memory consumption =   52312 node = SgExpressionRoot
```

```
AST Memory Pool Statistics: numberOfNodes =     922 memory consumption =   36880 node = SgBasicBlock
AST Memory Pool Statistics: numberOfNodes =     861 memory consumption =   27552 node = SgNullStatement
AST Memory Pool Statistics: numberOfNodes =     855 memory consumption =   47880 node = SgFunctionType
AST Memory Pool Statistics: numberOfNodes =     837 memory consumption =   40176 node = SgThisExp
AST Memory Pool Statistics: numberOfNodes =     817 memory consumption =   42484 node = SgArrowExp
AST Memory Pool Statistics: numberOfNodes =     784 memory consumption =   31360 node = SgFunctionDefinition
AST Memory Pool Statistics: numberOfNodes =     781 memory consumption =  212432 node = SgTypedefDeclaration
AST Memory Pool Statistics: numberOfNodes =     764 memory consumption =   18336 node = SgTypedefSymbol
AST Memory Pool Statistics: numberOfNodes =     762 memory consumption =   42672 node = SgTypedefType
AST Memory Pool Statistics: numberOfNodes =     753 memory consumption =   18072 node = SgEnumFieldSymbol
AST Memory Pool Statistics: numberOfNodes =     643 memory consumption =   33436 node = SgDotExp
AST Memory Pool Statistics: numberOfNodes =     630 memory consumption =   22680 node = SgReturnStmt
AST Memory Pool Statistics: numberOfNodes =     605 memory consumption =   26620 node = SgExprListExp
AST Memory Pool Statistics: numberOfNodes =     601 memory consumption =   33656 node = SgCastExp
AST Memory Pool Statistics: numberOfNodes =     548 memory consumption =   28496 node = SgFunctionCallExp
AST Memory Pool Statistics: numberOfNodes =     399 memory consumption =   19152 node = SgBoolValExp
AST Memory Pool Statistics: numberOfNodes =     371 memory consumption =   13356 node = SgExprStatement
AST Memory Pool Statistics: numberOfNodes =     351 memory consumption =    8424 node = SgClassSymbol
AST Memory Pool Statistics: numberOfNodes =     325 memory consumption =   18200 node = SgMemberFunctionRefExp
AST Memory Pool Statistics: numberOfNodes =     291 memory consumption =   68676 node = SgUsingDeclarationStatement
AST Memory Pool Statistics: numberOfNodes =     290 memory consumption =   15080 node = SgPntrArrRefExp
AST Memory Pool Statistics: numberOfNodes =     223 memory consumption =   10704 node = SgFunctionRefExp
AST Memory Pool Statistics: numberOfNodes =     209 memory consumption =   78584 node = SgTemplateInstantiationFunctionDecl
AST Memory Pool Statistics: numberOfNodes =     201 memory consumption =    8844 node = SgClassDefinition
AST Memory Pool Statistics: numberOfNodes =     193 memory consumption =   10036 node = SgMultiplyOp
AST Memory Pool Statistics: numberOfNodes =     181 memory consumption =    8688 node = SgStringVal
AST Memory Pool Statistics: numberOfNodes =     168 memory consumption =    8064 node = SgArrayType
AST Memory Pool Statistics: numberOfNodes =     157 memory consumption =    7536 node = SgUnsignedLongVal
AST Memory Pool Statistics: numberOfNodes =     151 memory consumption =   35032 node = SgTemplateInstantiationDirectiveStatement
AST Memory Pool Statistics: numberOfNodes =     150 memory consumption =    6600 node = SgTemplateInstantiationDefn
AST Memory Pool Statistics: numberOfNodes =     126 memory consumption =    6048 node = SgUnsignedIntVal
AST Memory Pool Statistics: numberOfNodes =     118 memory consumption =    6136 node = SgAssignOp
AST Memory Pool Statistics: numberOfNodes =     115 memory consumption =    5980 node = SgAddOp
AST Memory Pool Statistics: numberOfNodes =     101 memory consumption =    4040 node = SgBaseClassModifier
AST Memory Pool Statistics: numberOfNodes =     101 memory consumption =    2828 node = SgBaseClass
AST Memory Pool Statistics: numberOfNodes =      82 memory consumption =    4592 node = SgConditionalExp
AST Memory Pool Statistics: numberOfNodes =      77 memory consumption =    3388 node = SgNamespaceDefinitionStatement
AST Memory Pool Statistics: numberOfNodes =      77 memory consumption =   19712 node = SgNamespaceDeclarationStatement
AST Memory Pool Statistics: numberOfNodes =      72 memory consumption =    3744 node = SgEqualityOp
AST Memory Pool Statistics: numberOfNodes =      61 memory consumption =    3172 node = SgCommaOpExp
AST Memory Pool Statistics: numberOfNodes =      53 memory consumption =    3180 node = SgConstructorInitializer
AST Memory Pool Statistics: numberOfNodes =      49 memory consumption =    1568 node = SgPragma
AST Memory Pool Statistics: numberOfNodes =      49 memory consumption =   11368 node = SgPragmaDeclaration
AST Memory Pool Statistics: numberOfNodes =      46 memory consumption =    3312 node = SgEnumVal
AST Memory Pool Statistics: numberOfNodes =      46 memory consumption =    2208 node = SgIfStmt
AST Memory Pool Statistics: numberOfNodes =      42 memory consumption =    2184 node = SgEnumType
AST Memory Pool Statistics: numberOfNodes =      42 memory consumption =   11088 node = SgEnumDeclaration
AST Memory Pool Statistics: numberOfNodes =      42 memory consumption =    1008 node = SgEnumSymbol
AST Memory Pool Statistics: numberOfNodes =      36 memory consumption =    1872 node = SgPointerDerefExp
AST Memory Pool Statistics: numberOfNodes =      35 memory consumption =    1680 node = SgShortVal
AST Memory Pool Statistics: numberOfNodes =      32 memory consumption =    1664 node = SgSubtractOp
AST Memory Pool Statistics: numberOfNodes =      28 memory consumption =     560 node = SgQualifiedName
AST Memory Pool Statistics: numberOfNodes =      26 memory consumption =    1352 node = SgAddressOfOp
AST Memory Pool Statistics: numberOfNodes =      24 memory consumption =    1152 node = SgCharVal
AST Memory Pool Statistics: numberOfNodes =      23 memory consumption =    1196 node = SgLessThanOp
AST Memory Pool Statistics: numberOfNodes =      22 memory consumption =    1144 node = SgGreaterOrEqualOp
AST Memory Pool Statistics: numberOfNodes =      21 memory consumption =    1092 node = SgPlusPlusOp
AST Memory Pool Statistics: numberOfNodes =      19 memory consumption =     988 node = SgNotEqualOp
AST Memory Pool Statistics: numberOfNodes =      19 memory consumption =     912 node = SgUnsignedShortVal
AST Memory Pool Statistics: numberOfNodes =      18 memory consumption =     936 node = SgAndOp
AST Memory Pool Statistics: numberOfNodes =      18 memory consumption =     864 node = SgPointerMemberType
AST Memory Pool Statistics: numberOfNodes =      18 memory consumption =     864 node = SgLongIntVal
AST Memory Pool Statistics: numberOfNodes =      15 memory consumption =     780 node = SgDivideOp
AST Memory Pool Statistics: numberOfNodes =      14 memory consumption =     728 node = SgBitAndOp
AST Memory Pool Statistics: numberOfNodes =      12 memory consumption =     624 node = SgMinusMinusOp
AST Memory Pool Statistics: numberOfNodes =      11 memory consumption =     616 node = SgDoubleVal
AST Memory Pool Statistics: numberOfNodes =      11 memory consumption =     572 node = SgFloatVal
AST Memory Pool Statistics: numberOfNodes =      10 memory consumption =     520 node = SgUnsignedLongLongIntVal
AST Memory Pool Statistics: numberOfNodes =      10 memory consumption =     520 node = SgModOp
AST Memory Pool Statistics: numberOfNodes =      10 memory consumption =     520 node = SgLongLongIntVal
AST Memory Pool Statistics: numberOfNodes =       9 memory consumption =     540 node = SgLongDoubleVal
AST Memory Pool Statistics: numberOfNodes =       9 memory consumption =     468 node = SgNotOp
AST Memory Pool Statistics: numberOfNodes =       8 memory consumption =     416 node = SgBitOrOp
AST Memory Pool Statistics: numberOfNodes =       7 memory consumption =     364 node = SgMinusOp
AST Memory Pool Statistics: numberOfNodes =       7 memory consumption =     308 node = SgWhileStmt
AST Memory Pool Statistics: numberOfNodes =       5 memory consumption =     260 node = SgForStatement
AST Memory Pool Statistics: numberOfNodes =       4 memory consumption =     208 node = SgOrOp
AST Memory Pool Statistics: numberOfNodes =       4 memory consumption =     208 node = SgGreaterThanOp
AST Memory Pool Statistics: numberOfNodes =       4 memory consumption =     192 node = SgDeleteExp
AST Memory Pool Statistics: numberOfNodes =       4 memory consumption =     192 node = SgAggregateInitializer
AST Memory Pool Statistics: numberOfNodes =       4 memory consumption =     176 node = SgNamespaceSymbol
AST Memory Pool Statistics: numberOfNodes =       4 memory consumption =     144 node = SgForInitStatement
AST Memory Pool Statistics: numberOfNodes =       3 memory consumption =     156 node = SgRshiftOp
AST Memory Pool Statistics: numberOfNodes =       3 memory consumption =     156 node = SgRshiftAssignOp
AST Memory Pool Statistics: numberOfNodes =       3 memory consumption =     156 node = SgPlusAssignOp
AST Memory Pool Statistics: numberOfNodes =       3 memory consumption =     156 node = SgLshiftOp
AST Memory Pool Statistics: numberOfNodes =       3 memory consumption =     156 node = SgBitXorOp
AST Memory Pool Statistics: numberOfNodes =       3 memory consumption =     156 node = SgBitComplementOp
AST Memory Pool Statistics: numberOfNodes =       2 memory consumption =     104 node = SgDivAssignOp
AST Memory Pool Statistics: numberOfNodes =       2 memory consumption =     104 node = SgAndAssignOp
AST Memory Pool Statistics: numberOfNodes =       1 memory consumption =      96 node = SgFile
```

```
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      84 node = SgProject
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      48 node = SgCatchOptionStmt
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      44 node = SgTypeInt
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeWchar
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeVoid
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeUnsignedShort
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeUnsignedLongLong
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeUnsignedLong
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeUnsignedInt
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeUnsignedChar
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeString
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeSignedChar
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeShort
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeLongLong
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeLongDouble
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeLong
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeFloat
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeEllipse
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeDouble
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeDefault
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeChar
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTypeBool
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgTryStmt
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      40 node = SgGlobal
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      36 node = SgFunctionTypeTable
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =      36 node = SgCatchStatementSeq
AST Memory Pool Statistics: numberOfNodes =      1 memory consumption =     232 node = SgUsingDirectiveStatement
```

20.6 Compilation Performance Timings

An initial snapshot of the performance for the previous 40,000 line single file is included so that it is clear that the performance code of the source-to-source is a small multiple of the cost of the compilation using g++ (when g++ is used at its fastest, with no optimization).

```
Performance Report (resolution = 0.010000, number of IR nodes = 289439, memory used = 20144 Kilobytes):
     AST (SgProject::parse(argc,argv)): time (sec) =  18.382917
          AST (SgProject::parse()): time (sec) =  18.381067
               AST SgFile Constructor: time (sec) =  18.380805
                    AST Front End Processing (SgFile): time (sec) =  4.846442
                         AST Constrution (included Sage III Translation): time (sec) =  4.840888
                              EDG AST Constrution: time (sec) =  0.807095
                              AST EDG/Sage III Translation: time (sec) =  3.926241
                    AST post-processing: time (sec) =  13.513127
                         (fixup function definitions - missing body) time (sec) =  0.379914
                         (fixup template declarations) time (sec) =  0.435447
                         (reset parent pointers) time (sec) =  2.468755
                         (subTemporaryAstFixes) time (sec) =  1.303070
                         (initialize IR nodes containing explicit scope data member) time (sec) =  0.122380
                         (reset template names) time (sec) =  1.433229
                         (fixup class data member initialization) time (sec) =  0.575695
                         (fixup for generation of GNU compatable code) time (sec) =  0.580172
                         (testing declarations (no side-effects to AST))) time (sec) =  0.638836
                         (fixup storage access of forward template declarations (EDG bug)) time (sec) =  0.542976
                         (fixup template specializations) time (sec) =  0.860818
                         (mark template specializations for output) time (sec) =  0.595816
                         (mark template instantiations for output) time (sec) =  0.567450
                         (fixup defining and non-defining declarations) time (sec) =  0.686581
                         (fixup symbol tables) time (sec) =  0.547633
                              (fixup global symbol table) time (sec) =  0.000000
                              (fixup local symbol tables) time (sec) =  0.547604
                         (fixup templateHandlingOptions) time (sec) =  0.546708
                         (mark transformations for output) time (sec) =  0.529240
                         (check the isModifiedFlag in each IR node) time (sec) =  0.130703
                    AST Comment Processing: time (sec) =  0.020377
     AST Consistancy Tests: time (sec) =  9.429836
     AST Object Code Generation (backend): time (sec) =  0.756793
          AST Code Generation (unparsing): time (sec) =  0.009177
          AST Backend Compilation (SgProject): time (sec) =  0.744890
               AST Object Code Generation (compile output): time (sec) =  0.743146
```

Chapter 21

Developer's Appendix

21.1 Adding Contributions to ROSE

We will be happy to work with you if you want to add new features to ROSE. We have an external git repository at `https://github.com/rose-compiler/rose`. You can publish your contributions there using a fork of our repository and send a pull request to us to review and merge your contributions.

The number one most important aspect about any contribution you make is that it should include test codes that demonstrate the feature and test it within our automate test mechanism (the *make check* makefile rules).. Depending upon the feature this can include an additional demonstrative example of how it works, such examples go into the ROSE Tutorial (often as a separate chapter). Most new work starts in the *Experimental* part of the ROSE Tutorial and is moved forward in the document over time.

The purpose of the test codes in our automated tests are:

- Make sure that future great ideas in ROSE don't break your feature.
- Allow us to easily detect maintenance problems as early as possible.
- Help us sleep at night knowing that ROSE is really working.
- Give everyone else using ROSE confidence in future releases.

We take this subject very seriously, since it can be a significant problem. In the future we will likely not accept contributions that are not accompanied by sufficient test codes that demonstrate that they work and will be part of the automated tests (*make check* makefile rule). If you want to add a new feature to ROSE, show us your tests.

21.2 Working with the ROSE Git repositories

This section is most useful for ROSE developers who have access to LLNL's network file system (NFS).

We have (3) Git repositories:

- internal: */nfs/casc/overture/ROSE/git/ROSE.git*
- external: `https://github.com/rose-compiler/rose` – is synced with the internal repository.

- external(being phased out): http://www.rosecompiler.org/rose.git – is synced with the internal repository.

21.2.1 Continuous Integration in ROSE

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.

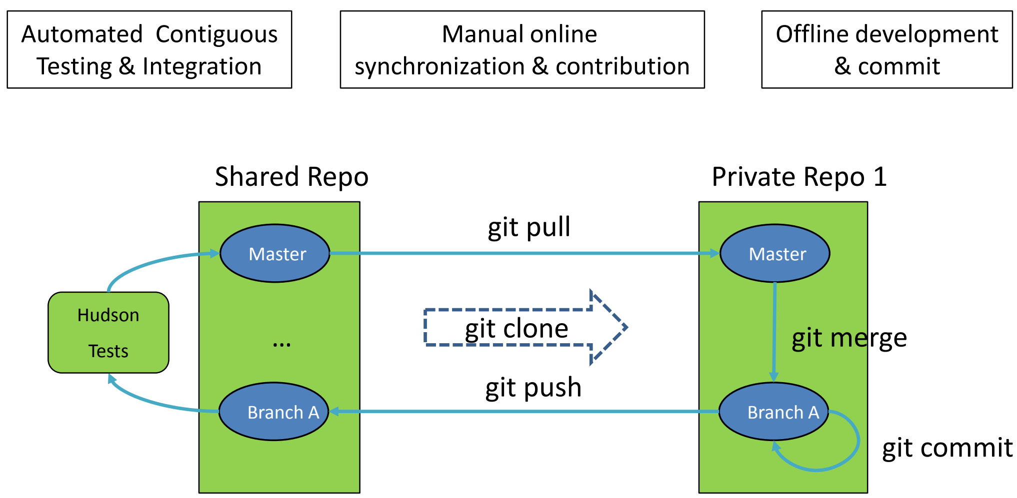

Figure 21.1: Contiguous integration using Git and Hudson

Figure 21.1 shows a high level overview of the development model used by ROSE developers. Taking advantage of the distributed source code repositories based on git, each developer should first clone his/her own repository from a shared repository. 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. When work is finished and locally tested, he can push all accumulated commits within the private repository to his branch within the shared repository. We create a dedicated branch within the shared 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.

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 Hudson is actively monitoring each developer's branch within the shared repository and will initiate comprehensive commit tests upon the branch once new commits are detected. Finally, Hudson will merge the new commits to the master branch of the shared repository if all tests pass. If a single test fails, Hudson 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 shared git repository is mostly stable and can be a good candidate for external release. On top of the master branch of the shared git repository, we further have more comprehensive release tests in Hudson. If all the release tests pass, an external release based on the master branch will be made available outside.

21.2.2 The Internal Git Repository

The internal ROSE Git repository (Fig. 21.1) is hosted under */nfs/casc/overture/ROSE/git/ROSE.git*. External collaborators can access this NFS path through any of these internal LLNL hosts: `tuxblue[1-6|9-13]`. Your LLNL account must be in the casc, overture, and rose POSIX groups(contact 4HELP to be added to these groups).

```
$ git clone ssh://<user>@tuxblue[1-6|9-13]/nfs/casc/overture/ROSE/git/ROSE.git
```

Updates to the "Release Candidate" (*-rc) branches in this internal repository trigger our Continuous Integration framework. Although you'll want to make a local clone of ROSE for development purposes, you must make sure to push your changes to your remote branch for testing and integration purposes.

The `master` branch always contains the latest work of ROSE and can only be updated by our continuous integration framework.

21.2.3 The External Git Repository

External users (who don't have an account with LLNL) are recommended to use ROSE's external SVN repository, which is described in 21.3.

For advanced external users who are comfortable with git. We have an external git repository which is cloned and synchronized with our internal shared repository. To clone the external git repository, simply type:

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

Or

git clone http://www.rosecompiler.org/rose.git

Depending on your network speed, the commandline above may take 3 to 5 minutes, or even longer.

How to work with us on ROSE

Figure 21.2 shows the suggested organization of git repositories to support external collaborations. LLNL provides external accesss to its public git repository; internal LLNL developers work with the internal git repository. Once a feature or bug fix is ready, it is pushed to the external public git repository.

External collaborators should clone their private repository from the LLNL public git repository. Collaborators should work on their private repository and push their features and bug fixes to their public git repository (which should be cloned from their private repository).

LLNL will accept changes by pulling them from the external collaborator's public repository. All changes will be inspected as part of internal LLNL policies. External collaborators can obtain the latest version of the LLNL work by pulling from our external public repository to their private git repository as often as they wish.

21.2.4 Our Git Naming Conventions

Both the internal and the external ROSE repositories are structured according to n continuous integration workflow using Git best practices. The master branch contains the latest work in ROSE that passes the tests

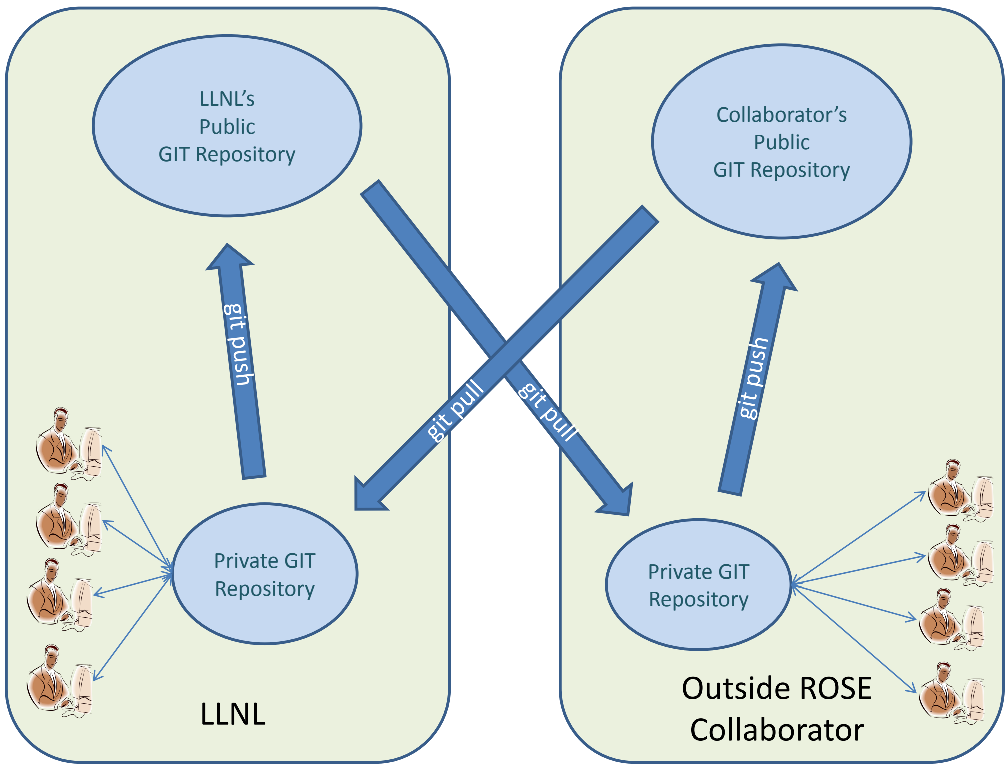

Figure 21.2: How to setup external collaborations with Git

and each developer can create any number of branches that he owns. No developer can modify a branch owned by another developer, but a group branch can be created that can be modified by groups of users.

Branches in the ROSE repository is named according to naming convention so that it is easy to see who owns a branch and if the developer intends this branch to be tested, released or not tested at all. This is done with a prefix and postfix scheme.

The prefix of a branch name maps directly to your LLNL official user name (OUN). For example, a developer 'John Brady' has the OUN 'brady1'. If he is working on a bugfix for bug number 1234 he should create a branch 'brady1-bug1234' where he works on this.

If work on a branch has reached a stage where a developer deems it ready for integration, he should push his changes to a new branch with a '-rc' postfix. RC stands for 'Release Candidate' as this work will be integrated with master if it passes all the tests.

21.2.5 Making a Copy of the Internal repository

Before starting to work a developer must create a local clone of the central git repository. Please configure this copy with your name and email before starting work.

- Configuration:
  - Set your name: *git config –global user.name 'John Doe'*
  - Set your email: *git config –global user.email johndoe@example.com*
  - Tell git-branch and git-checkout to setup new branches so that git-pull(1) will appropriately merge from that remote branch: *git config branch.autosetupmerge true*
- Get content in the repository:
  - Clone a private repository from the shared repository: *git clone file:///nfs/casc/overture/ROSE/git/ROSE.git localRoseName* . A directory named localRoseName will show up after this command. Type *cd localRoseName* to operate within the private repository from now on.
  - Retrieve EDG source files: we use a separated git repository for EDG to protect its proprietary source files. The EDG git repository is linked to the ROSE git repository as a submodule. You have to explicitly get the submodule if you want to modify the EDG source files. Just type: *git submodule init; git submodule update*
  - list all branches in the central git repository: *git branch -r*
  - list all branches in the local copy of the git repository: *git branch -l*
  - list all local and central branches: *git branch -a*
  - show the log: *git log*
- Using a branch:
  - creating a local branch within a local repo, based on whatever current branch's content: *git branch branch-name*
  - creating a local branch to track a central branch: *git branch branch-name –track origin/branch-name*
  - creating a branch in the central git: *git push origin origin:refs/heads/branch-name* . This command is not allowed for average developer unless the branch name has a prefix matching the developer's LLNL OUN. Please ask ROSE administrator to create a remote branch if needed.
  - start working on (switch to) a branch: *git checkout branch-name*

  - deleting a local branch: *git branch -d branch-name*
  - deleting a remote branch: *git push origin :branch-name* . Again, we have access control so developers can only delete their own branches, not others'.

- Adding / Deleting files: always be careful not to add/delete wrong files!
  - add <file> to the project *git add <file>*
  - add all files under directory <dir> to the project, including subdirectories *git add <dir>*
  - remove <file> from the project *git rm <file>*

- Committing : intermediate changes can be committed locally. You don't have to have active connection to the shared repository to do your daily work at all.
  - Always check the current status of your local repo by typing: *git status* . This allows you to see what files/directories have been modified, added, or out of control of git.
  - Commit changes to the local repository: *git commit -a*

- Pulling and Merging the shared master branch: It is recommended to synchronize the master branches often and merge new commits from others to your local branch.
  - Switch to and synchronize a local master branch: *git checkout master; git pull origin master*
  - Don't forget to update the link to the EDG submodule: *git submodule update*
  - Switch to your local branch and merge with local master: *git checkout your-local-branch-name; git merge master*
  - Alternatively, you can stay in your local branch and directly pull the remote master and update the EDG submodule: *git pull origin master; git submodule update*

- Pushing: A set of local commits can be pushed to the central repository when ready. It is highly recommended to pull and merge the shared master branch before doing this step.
  - Push all commits of the current local branch to a branch of the central repository:
    git push origin HEAD:branch-name Please always try to synchronize with the remote master branch (using git pull and merge as mentioned above) before pushing commits. **Note**: It is highly recommended not to push too often than needed since each git push will trigger a large set of Hudson tests which tax our own workstations and many other test machines.

- Modifying the EDG submodule. In rare cases, you may have to modify the EDG files within the submodule and update the link between ROSE and the EDG module. Please be **EXTREMELY cautious** when you have to do this and always ask for help from senior LLNL developers for your first attempt. Here are some brief instructions about how to do this right.
  - First, make sure you have tried to pull the content from the EDG submodule. You can run *git submodule init; git submodule update* again from the top level of your private ROSE git repository to be safe.
  - By default, the EDG submodule files checked out via *git submodule init; git submodule update* is not on any branch (You can verify this by typing *git status* within *src/frontend/CxxFrontend/EDG*. You can create a local branch based on the current content before you modify the EDG related files. Just type *git checkout -b local-branch-name* under *src/frontend/CxxFrontend/EDG* so you can create a local branch off the current content and switch to the branch.
  - We have access control over the EDG git repo the same way we have for the ROE git repo. You have to have your own remote branch in the shared EDG git repo so you can push any contributions to your own branch. The initial content of your remote branch MUST be the same as the current EDG version linked

to ROSE. This should be the default EDG version checked out and also the local branch you just created from the previous step. To create a remote branch for yourself based the latest linked EDG version, type *git push origin HEAD:refs/heads/your_LLNL_OUN-main-rc* within *src/frontend/CxxFrontend/EDG*.

  - You can then modify EDG files and commit changes to your private repository of the submodule. You should also be able to push your commits to your own branch within the remote EDG repository. Again, always be sure do all commits and push within *src/frontend/CxxFrontend/EDG* for EDG related changes.
  - In the end, change ROSE git repository's link to the changed submodule.

```
$ cd ROSE/src/frontend/CxxFrontend  # Go to submodule's parent repository's path
$ git add EDG  # Do not type 'git add EDG/',
               # which will add all files under EDG/ to the super project!!!
$ git commit -m "Updated submodule EDG."
$ git push   # assuming you are on your local ROSE branch tracking your own remote branch
             # and have set up the push mode by: git config push.default 'tracking'
```

  - Due to the limited support of submodule of Hudson's git plugin, a developer's branch of the EDG repo will not automatically be merged into the master of the EDG repo. So the EDG branch linked to the parent ROSE repo often is often the master of the EDG repo. This should not be a problem since two repositories are linked through HASH numbers.

- Get more info on the repository:
  - show a diff of the changes made since your last commit *git diff*
  - show files added to the staging area, files with changes, and untracked files *git status*
  - show recent commits *git log*
  - show commits between the specified range *git log <ref>..<ref>*
  - show the changeset (diff) of a commit specified by <rev> *git show <rev>*
  - show who authored each line in <file> *git blame <file>*
  - really nice GUI interface to git blame *git gui blame*
  - show only the commits which affected <file> listing the most recent first *git whatchanged <file>*
- Sharing changes:
  - fetch changes from the server, and merge them into the current branch: *git pull origin branch-name*
  - pushing local changes to the central git (from working branch): *git push origin branch-name*
- Reverting changes:
  - reverse commit specified by <rev> and commit the result: *git revert <rev>*
  - re-checkout <file>, overwriting any local changes: *git checkout <file>*
  - re-checkout all files, overwriting any local changes: *git checkout .*
- Fix mistakes / Undo:
  - abandon everything since your last commit: *git reset –hard*
  - undo your most recent *successful* merge *and* any changes that occurred after *git reset –hard ORIG_HEAD*
  - forgot something in your last commit? That's easy to fix. Undo your last commit, but keep the changes in the staging area for editing. *git reset –soft HEAD^*

  - redo previous commit, including changes you've staged in the meantime. Also used to edit commit message of previous commit. *git commit –amend*

- Stashing:
  - save your local modifications to a new stash: *git stash save <optional-name>*
  - restore the changes recorded in the stash on top of the current working tree state *git stash apply*
  - restore the changes from the most recent stash, and remove it from the stack of stashed changes *git stash pop*
  - list all current stashes *git stash list*
  - show the contents of a stash *git stash show <stash-name> -p*
  - delete current stashes *git stash clear*
- Remotes: Again, only administrators can change remote repository's branches.
  - delete a branch in a remote repository *git push <remote> :refs/heads/<branch>*
  - create a branch on a remote repository *git push <remote> <remote>:refs/heads/<remote_branch>*
  - create a branch on a remote repository based on +<remote> *git push <repository> +<remote>:<new_remote>*
  - prune deleted remote-tracking branches from "git branch -r" listing *git remote prune <remote>*

Git cheat sheet: http://cheat.errtheblog.com/s/git

21.3 Working with the ROSE SVN repository (phasing out)

We maintain an external subversion repository for ROSE at SciDAC Outreach Center. It is synchronized with the internal shared git repository using a vendor drop scheme (building a distribution from the git repository and load the content of the distribution to the svn repository). Some tips for using them are gathered in this section.

If you are our external (non-LLNL) users who make contributions to ROSE, we highly recommend you to work on a dedicated branch of the external repository. We can create the branch for you on request. And you need to apply an account of the SciDAC Outreach Center to have write access to your branch.

Here are the steps to have an account with write access to ROSE's branches: Please follow the link on https://outreach.scidac.gov/account/register.php to fill out a registration form (Project name: ROSE, PI: Daniel Quinlan) and fax a signed use policies form as instructed on the registration page. After getting your account, you need to log into the website and go to page https://outreach.scidac.gov/projects/rose/. Click "Request to join" on the top-right screen to request to join the ROSE project and we will grant you the write access to your branch.

Some frequently used commands for ROSE external developers are listed below:

- Install your svn client (>=1.5.1 is recommended) with *libsvn_ra_dav* support (http://www.webdav.org/neon and *–with-ssl*) or set the right *LD_LIBRARY_PATH* for it (*libsvn_ra_dav-1.so*) if you encounter the following problem:
  svn: Unrecognized URL scheme for 'https://outreach.scidac.gov/svn/rose/trunk'
- To check out the main trunk, type:
  svn checkout https://outreach.scidac.gov/svn/rose/trunk rose
- To check out a branch, type:
  svn checkout https://outreach.scidac.gov/svn/rose/branches/branch_name rose

- Merge the new updates of the main trunk into your working branch. Conceptually, svn merge works as two step: diff two revisions and merge the different into a working copy. So you need to know two revision numbers of the main trunk: the first is the latest revision number of the main trunk from which your branch was created (or most recently synchronized); the second is usually the head revision of the main trunk. [1]:
  - find the revision in which your branch was created or the last synchronization point with the trunk: *svn log https://outreach.scidac.gov/svn/rose/branches/branch_name*
  - cd local work copy of your branch, do the merge (overlapped merging seems possible using subversion 1.5.1), assume the last synchronization point(or originating point) is rev 56:
    svn merge –dry-run -r 56:69 https://outreach.scidac.gov/svn/rose/trunk
    svn merge -r 56:69 https://outreach.scidac.gov/svn/rose/trunk
  - Solve conflicts as needed.
  - svn commit: **Note:please record the start and end revision numbers of the main trunk being merged into the commit log to keep track of merging. Please put this information on the first line if this is a commit following a merge of your branch with the main trunk (see Commit Message Format in subsection 21.3.1 for details)**
- You can check the archive of email notifications of the svn commits from https://osp5.lbl.gov/pipermail/rose-commits

21.3.1 Commit Message Format

The automatically generated ChangeLog2 file will provide everyone with detailed information about what changes are made to ROSE over time. To make this information as clear and consistent as possible we have two (slightly different) commit message formats: 1) normal commits of your local contributions to your branch or to the internal SVN trunk; and 2) commits after a merge of the main trunk's changes.

1. Normal svn commit (not those following an svn merge)
   - `svn commit` will start your favorite editor where you should enter a description of your changes. The first line of that description should be a short, one-line summary (*i.e.*, a title with just the first word capitalized), followed by a blank line, and as much detail as necessary. There is generally no need to includes your name, date, names of files, etc. as this information is readily available from the source revision management system. Do not prefix the summary with tags like "Summary:", "Title:" etc. since it's already implied that the first line is the summary.
     Here's an example specific to a commit on the internal SVN or an SVN branch:

```
Adjusted test case for new binary function detection

  This test case assumed that the only functions in a binary executable
  were those that had symbols in the symbol table.  This is no longer
  true since we now determine function boundaries with a wider variety of
  heuristics.
```

2. For the svn commit at any point after your svn merge
   Here's an example specific to a commit message on an SVN branch after a merge:

[1] Subversion 1.5 is said to support svn merge with the head of a main trunk without explicitly specifying the beginning and end revision numbers. But this new feature is not mature enough to be used in our work as our tests showed. We will try to use the new feature later on when it becomes dependable.

```
svn merge -r 402:428 https://outreach.scidac.gov/svn/rose/trunk
```

3. If you mix an svn merge and some local contributions in one svn commit (we don't suggest mixing them) Here's an example specific to the commit on an SVN branch (*note first line*):

```
svn merge -r 402:428
  Adjusted test case for new binary function detection

  This test case assumed that the only functions in a binary executable
  were those that had symbols in the symbol table.  This is no longer
  true since we now determine function boundaries with a wider variety of
  heuristics.
```

21.3.2 Check In Process

The following information applies to both the internal SVN reposiotry and the branches that we provide to external collaborators. There are a number of details that we need to make sure that your developent work can be used to update ROSE.

For internal SVN users: **Please get permission from the ROSE Development Team before you make your first check-in!**

For all SVN users: If you have access to the SVN repository (at LLNL) and are building the development version of ROSE (available only from SVN, not what we package as a ROSE distribution; e.g. not from a file name such as ROSE-version-number.tar.gz) then there are a number of steps to the checkin process:

1. Make sure you are working with the latest update (run `svn update` in the top level directory.
2. Run `make` && `make docs` && `make check` && `make dist` && `make distcheck` && `make install` && `make installcheck`, depending on how aggressively you want your changes to be tested.
   - Not all tests must be run, but we will know who you are (via `svn blame` if the nightly test fail :-)).
   - All changes must at least compile, so that you don't hold back other developers who update often.
3. The commit will fail if someone else has committed while you were running your pre-commit tests. If this happens you will generally need to restart the check-in process from the top.
4. Please follow the commit message format (see Commit Message Format in subsection 21.3.1 for details).

If you do not have access to the SVN repository at LLNL, and you wish to contribute work to the ROSE project, please make a patch. Using the external SVN access via LBL use `svn diff` to build a patch. Consider options: *–diff-cmd arg*. DQ(7/28/2008): This section still needs to be completed!

21.4 Resync-ing with a full version of ROSE

As part of work with external collaborators, where they have access to the EDG source code, we sometime have to update their version of the parts of ROSE that are not released publicly (e.g. EDG and the EDG/ROSE translation work which uses EDG). A typical reason why this is required is that the external collaborator has made a change to the ROSE IR that is incompatible with the binary distribution of the EDG and EDG/ROSE translation code, and so they need a most recent version of the full distribution of ROSE so that they can build EDG and the EDG/ROSE translation fresh and run the automated tests.

We wish to outline this process:

1. Let us know that you are trying to follow these directions.
2. Ask for a tarball of the full source code of ROSE from our internal SVN repository. We will provide you a tarball of ROSE that matches a specific revision number that was externally released on the web (thus we know that it has passed all of our tests to be released). This will also define a mapping between internal and external SVN revision numbers, which is also in the commit log message on the web site. For example, it shows the lat log entry of the main trunk on the page of (https://outreach.scidac.gov/plugins/scmsvn/viewcvs.php/?root=rose):

   ```
   File      Rev.   Age       Author   Last log entry
   trunk/    243   6 hours    liaoch   Load rose-0.9.4a-4275 into trunk.
   ```

   In this case the internal SVN revision number is *4275* and it mapped to the external SVN revision number *243*. We will make a tarball of ROSE using revision number *4275*. The command to do this, on our side is:

   ```
   scripts/make_svn_tarball 4275
   ```

   with typical output:

   ```
   Built tarball ROSE-svn-Feb03-2009-r4275.tar.gz from SVN revision r4275
   ```

   This builds the file: `ROSE-svn-Feb03-2009-r4275.tar.gz` which we then send to you. This is a full source code release of ROSE which includes the protected EDG source code, we will know if you have a license for this. *You should not distribute this to anyone who does not have an EDG license.*
3. Then you update your branch with the trunk at the external revision number (in our example this would be revision *243*). See the instructions in `Working with The ROSE SVN repositories` of this guide about how to merge the new updates of the main trunk into your branch. Make sure it pass make check.
4. Then build a patch to represent your branch's changes from the external trunk revision. A typical command to generate a patch looks like the following:

   ```
   diff -NaU5 -rbB -x \*.orig -x \*.o -x \*.swp -x \*.bak -x \*.pdf \
     -x \*.html -x \*.rej -x \*~ -x Makefile.in -x \*.gz \
     -x autom4te.cache -x .svn -x aclocal.m4 -x config.guess \
     -x configure -x config.sub external_trunk your_updated_branch > my.patch
   ```

   You may need to check the generated patch and add or remove the items in the exclusion list to regenerate a desired patch as needed. The final patch should only contains your contributions.
5. Apply that patch to the tarball of the internal ROSE's trunk that we sent you (representing the full source code for EDG and everything) and you how have a way to test your work and recompile the EDG work for either a new machine or with the IR changes that you have added.

   ```
   cd internal_ROSE_trunk
   # test run only
   patch -p1 --dry-run <../my.patch
   # if everything looks normal, do the actual patching
   patch -p1 < ../my.patch
   ```

6. After you have passed all tests, then build a patch between the patched internal ROSE trunk and it's original form.

```
diff -NaU5 -rbB -x \*.orig -x \*.o -x \*.swp -x \*.bak -x \*.pdf \
  -x \*.html -x \*.rej -x \*~ -x Makefile.in -x \*.gz \
  -x autom4te.cache -x .svn -x aclocal.m4 -x config.guess \
  -x configure -x config.sub internal_ROSE_trunk_orig
  internal_ROSE_trunk_patched > my.patch2
```

   Again, please tweak the exclusion list above to generate a clean and complete patch. This patch contains your contributions tested against with the full internal source tree. Please record the revision number of your branch associated with this patch. The number will be treated as a synchronization point between your branch and the main trunk.

7. Let us know when you are done and we can get your patch applicable to our internal SVN repository. At this point we can review and apply the patch to the internal ROSE and the next external release of ROSE (usually nightly) will reflect your changes.

21.5 How to recover from a file-system disaster at LLNL

Disasters can happen (cron scripts can go very very badly). If you loose files on the CASC cluster at LLNL you can get the backup from the night before. It just takes a while.

To restore from backups at LLNL: use the command:
`restore`

1. `add <directory name>`
   This will build the list of files to be recovered.
2. recover
   This will start the process to restore the files from tape.

This process can take a long time if you have a lot of files to recover.

21.6 Generating Documentation

There is a standard GNU `make docs` rule for building all documentation.

Note to developers: To build the documentation (`make docs`*) you will need LaTeX, Doxygen and DOT to be installed (check the list of dependences in the* `ROSE/ChangeLog`*). If you want to build the reference manual of Latex documentation generated by Doxygen (not suggested) you may have to tailor your version of LaTeX to permit larger internal buffer sizes. All the other LaTeX documentation, such as the User Manual but not the Reference Manual may be built without problems using the default configuration for LaTeX.*

21.7 Adding New SAGE III IR Nodes (Developers Only)

We don't expect users to add nodes to the SAGE III Intermediate Representation (IR), however, we need to document the process to support developers who might be extending ROSE. It is hoped that if you proceed to add IR nodes that you understand just what this means (you're not extending any supported language (e.g.

C++); you are only extending the internal representation. Check with us so that we can help you and understand what you're doing.

The SAGE III IR is now completely generated using the ROSETTA IR generator tool which we developed to support our work within ROSE. The process of adding new IR nodes using ROSETTA is fairly simple: one adds IR node definitions using a BNF syntax and provides additional headers and implementations for customized member data and functions when necessary.

There are lots of examples within the construction of the IR itself. So you are encouraged to look at the examples. The general steps are:

FIXME: *Need to co* *Fortr*

1. Add a new node's name into *src/ROSETTA/astNodeList*
2. Define the node in ROSETTA's source files under *src/ROSETTA/src*
   For example, an expression node has the following line in *src/ROSETTA/src/expression.C*:

   ```
   NEW_TERMINAL_MACRO (VarArgOp,"VarArgOp","VA_OP");
   ```

   This is a macro (currently) which builds an object named *VarArgOp* (a variable in ROSETTA) to be named *SgVarArgOp* in SAGE III, and to be referenced using an enum that will be called *V_SgVarArgOp*. The secondary generated enum name *VA_OP* is historical and will be removed in a future release of ROSE.
3. In the same ROSETTA source file, specify the node's SAGE class hierarchy.
   This is done through the specification of what looks a bit like a BNF production rule to define the abstract grammar.

   ```
   NEW_NONTERMINAL_MACRO (Expression,
        UnaryOp        | BinaryOp              | ExprListExp   | VarRefExp       | ClassNameRefExp |
        FunctionRefExp | MemberFunctionRefExp  | ValueExp      | FunctionCallExp | SizeOfOp        |
        TypeIdOp       | ConditionalExp        | NewExp        | DeleteExp       | ThisExp         |
        RefExp         | Initializer           | VarArgStartOp | VarArgOp        | VarArgEndOp     |
        VarArgCopyOp   | VarArgStartOneOperandOp ,"Expression","ExpressionTag");
   ```

   In this case, we added the VarArgOp IR node as an expression node in the abstract grammar for C++.
4. Add the new node's members (fields): both data and function members are allowed.
   ROSETTA permits the addition of data fields to the class definitions for the new IR node. Many generic access functions will be automatically generated if desired.

   ```
       VarArgOp.setDataPrototype  ( "$GRAMMAR_PREFIX_Expression*","operand_expr","= NULL",
   CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, DEF_TRAVERSAL, NO_DELETE);
   ```

   The new data fields are added to the new IR node. Using the first example above, the new data member is of type `SgExpression*`, with name `operand_expr`, and initialized using the source code string `= NULL`. Additional properties that this IR node will have include:

   - Its construction will take a parameter of this type and use it to initialize this member field.
   - Access functions to *get* and *set* the member function will be automatically generated.
   - The automatically generated AST traversal will traverse this node (i.e. it will visit its children in the AST).
   - Have the automatically generated destructor not call delete on this field (the traversal will to that).

   In the case of the VarArgOp, an additional data member was added.

   ```
       VarArgOp.setDataPrototype ( "$GRAMMAR_PREFIX_Type*", "expression_type", "= NULL",
   CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, NO_TRAVERSAL || DEF2TYPE_TRAVERSAL);
   ```

5. Most IR nodes are simpler, but SgExpression IR nodes have explicit precedence.
   All expression nodes have a precedence in the evaluation, but the precedence must be specified. This precedence must match that of the C++ frontend. So we are not changing anything about the way that C++ evaluates expressions here! It is just that SAGE must have a defined value for the precedence. ROSETTA permits variables to be defined and edited to tailor the automatically generated source code for the IR.

```
VarArgOp.editSubstitute ( "PRECEDENCE_VALUE", "16" );
```

6. Associate customized source code.
   Automatically generated source code sometimes cannot meet all requirements, so ROSETTA allows user to define any custom code that needs to be associated with the IR node in some specified files. If customized code is needed, you have to specify the source file containing the code. For example, we specify the file containing customized source code for *VarArgOp* in *src/ROSETTA/src/expression.C*:

```
     VarArgOp.setFunctionPrototype ( "HEADER_VARARG_OPERATOR", "../Grammar/Expression.code" );
     VarArgOp.setDataPrototype  ( "SgExpression*", "operand_expr"   , "= NULL",
 CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, DEF_TRAVERSAL, NO_DELETE);
     VarArgOp.setDataPrototype ( "SgType*", "expression_type", "= NULL",
 CONSTRUCTOR_PARAMETER, BUILD_ACCESS_FUNCTIONS, NO_TRAVERSAL || DEF2TYPE_TRAVERSAL, NO_DELETE);
   // ...
   VarArgOp.setFunctionSource ( "SOURCE_EMPTY_POST_CONSTRUCTION_INITIALIZATION",
                                "Grammar/Expression.code" );
```

   Pairs of special markers (such as *SOURCE_VARARG_OPERATOR* and *SOURCE_VARARG_END_OPERATOR*) are used for marking the header and implementation parts of the customized code. For example, the marked header and implementation code portions for *VarArgOp* in *src/ROSETTA/Grammar/Expression.code* are:

```
HEADER_VARARG_OPERATOR_START
   virtual unsigned int cfgIndexForEnd() const;
   virtual std::vector<VirtualCFG::CFGEdge> cfgOutEdges(unsigned int index);
   virtual std::vector<VirtualCFG::CFGEdge> cfgInEdges(unsigned int index);
HEADER_VARARG_OPERATOR_END

// ....
SOURCE_VARARG_OPERATOR_START

  SgType*
  $CLASSNAME::get_type() const
   {
     SgType* returnType = p_expression_type;
     ROSE_ASSERT(returnType != NULL);
     return returnType;
   }

  unsigned int $CLASSNAME::cfgIndexForEnd() const {
    return 1;
  }
  //....

SOURCE_VARARG_OPERATOR_END
```

   The C++ source code is extracted from between the named markers (text labels) in the named file and inserted into the generated source code. Using this technique, very small amounts of specialized code can be tailored for each IR node, while still providing an automated means of generating all the rest. Different locations in the generated code can be modified with external code. Here we add the source code for a function.

7. Adding the set_type and get_type member functions.
   It is not clear that this is required, but all expressions must define a function that can be used to describe its type (of the expression). It is unfortunate, but it is generally in compiling the generated source code that details like this are discovered. (ROSETTA has room for improvement!)

   ```
   VarArgOp.setFunctionSource ( "SOURCE_SET_TYPE_DEFAULT_TYPE_EXPRESSION",
                                "Grammar/Expression.code" );
   VarArgOp.setFunctionSource ( "SOURCE_DEFAULT_GET_TYPE",
                                "Grammar/Expression.code" );
   ```

8. Creating the new IR node at some point, such as somewhere within frontend, midend, or even backend if desired. This step is decided by the purpose of the newly added IR types. If the new types of IR come from their counterparts in EDG, then modifications to the EDG/SAGE connection code are needed. If you are trying to represent some pragma information, you don't need to touch EDG and its connection. You just parse the pragma string in the original AST and create your own nodes to get a new version of AST. Then it should be done.

   Here are the instructions if the new IR node is connected to EDG IR node. Modify the EDG/SAGE connection code to have the new IR node built in the translation from EDG to SAGE III. This step often requires a bit of expertise in working with the EDG/SAGE connection code. In general, it requires no great depth of knowledge of EDG.
   Two source files are usually involved: a) *src/frontend/CxxFrontend/EDG_SAGE_Connection/sage_gen_be.C* which converts IL tree to SAGE III AST and is derived from EDG's C++/C-generating back end *cp_gen_be.c*; b) *sage_il_to_str.C* contains helper functions forming SAGE III AST from various EDG IL entries. It is derived from EDG's *il_to_str.c*. For the *SgVarArgOp* example, the following EDG-SAGE connection code is needed in *sage_gen_be.C*:

```
a_SgExpression_ptr
sage_gen_expr ( an_expr_node_ptr expr,
                a_boolean need_parens,
    ...
              )
{
  // ...
  case eok_va_arg:
  {
   sageType = sage_gen_type(expr->type);
   sageLhs = sage_gen_expr_with_parens(operand_1,NULL);
   if (isSgAddressOfOp(sageLhs) != NULL)
     sageLhs = isSgAddressOfOp(sageLhs)->get_operand();
   else
     sageLhs = new SgPointerDerefExp(sageLhs,NULL);
  //....
   result = new SgVarArgOp(sageLhs, sageType);
   goto done_with_operation;
                       }
  }
  //.....

}
```

9. Modify the unparser to have whatever code you want generated in the final code generation step of the ROSE source-to-source translator. The source files of the unparser are located at *src/backend/unparser*. For *SgVarArgOp*, it is unparsed by the following function in *src/backend/unparser/CxxCodeGeneration/unparseCxx_expressions.C*:

```
void
Unparse_ExprStmt::unparseVarArgOp(SgExpression* expr, SgUnparse_Info& info)
   {
     SgVarArgOp* varArg = isSgVarArgOp(expr);
     SgExpression* operand = varArg->get_operand_expr();
     SgType* type = varArg->get_type();
     curprint ( "va_arg(");
     unparseExpression(operand,info);
     curprint ( ",");
     unp->u_type->unparseType(type,info);
     curprint ( ")");
   }
```

21.8 Separation of EDG Source Code from ROSE Distribution

The EDG research license restricts the distribution of their source code. Working with EDG is still possible within an open source project such as ROSE because EDG permits binaries of their work to be freely distributed (protecting their source code). As ROSE matured, we designed the autoconf/automake distribution mechanism to build distributions that exclude the EDG source code and alternatively distribute a Linux-based binary version of their code.

All releases of ROSE, starting with 0.8.4a, are done without the EDG source code by default. An optional configure command line option is implemented to allow the construction of a distribution of ROSE which includes the EDG source code (see `configure --help` for the `--with-edg_source_code` option).

The default options for configure will build a distribution that contains no EDG source code (no source files or header files). This is not a problem for ROSE because it can still exist as an almost entirely open source project using only the ROSE source and the EDG binary version of the library.

Within this default configuration, ROSE can be freely distributed on the Web (eventually). Importantly, this simplifies how we work with many different research groups and avoid the requirement for a special research license from EDG for the use of their C and C++ front-end. Our goal has been to simplify the use of ROSE.

Only the following command to configure with EDG source code is accepted:

```
configure --with-edg_source_code=true
```

This particularly restrictive syntax is used to prevent it from ever being used by accident. Note that the following will not work. They are equivalent to not having specified the option at all:

```
configure --with-edg_source_code
configure --with-edg_source_code=false
configure --with-edg_source_code=True
configure --with-edg_source_code=TRUE
configure --with-edg_source_code=xyz
configure
```

To see how any configuration is set up, type `make testEdgSourceRule` in the `ROSE/src/frontend/CxxFrontend/EDG_3.3/src` directory.

To build a distribution without EDG source code:

1. Configure to use the EDG source code and build normally,
2. Then rerun configure to not use the EDG source code, and
3. Run `make dist`.

21.9 How to Deprecate ROSE Features

There comes a time when even the best ideas don't last into a new version of the source code. This section covers how to deprecated specific functionality so that it can be removed in later releases (typically after a couple of releases, or before our first external release). When using GNU compilers these mechanisms will trigger the use of GNU attribute mechanism to permit use of such functions in applications to be easily flagged (as warnings output when using the GNU options `-Wall`).

Both functions and data members can be deprecated, but the process if different for each case:

- Deprecated functions and member functions.
  Use the macro `ROSE_DEPRECATED_FUNCTION` after the function declaration (and before the closing `;`). As in:

  ```
  void old_great_idea_function() ROSE_DEPRECATED_FUNCTION;
  ```

- Deprecated data members.
  Use the macro `ROSE_DEPRECATED_VARIABLE` to specify that a data members or variables is to be deprecated. This is difficult to do because data members of the IR are all automatically generated and thus can't be edited in this way. Where a data member of the IR is to be deprecated, it should be specified explicitly in the documentation for that specific class (in the `ROSE/docs/testDoxygen` directory, which is the staging area for all IR documentation, definitely *not* in the `ROSE/src/frontend/SageIII/docs` directory, which is frequently overwritten). See details on how to document ROSE (Doxygen-Related Pages).

  ```
  void old_great idea_data_member ROSE_DEPRECATED_VARIABLE;
  ```

21.10 Code Style Rules for ROSE

I don't want to constrain anyone from being expressive, but we have to maintain your code after you leave, so there are a few rules:

1. Document your code. Explain every function and use variable names that clearly indicate the purpose of the variable. Explain what the tests are in your code (and where they are located).
2. Write test codes to test your code (these are assembled in the `ROSE/tests` directory (or subdirectories of `ROSE/tests/nonsmoke/functional/roseTests`).
3. Use assertions liberally, use boolean values arguments to `ROSE_ASSERT(<expression>)`. Use of `ROSE_ASSERT(true/false)` for error branches is preferred.
4. Put your code into source files (*.C) and as little as possible into header files.
5. If you use templates, put the code into a *.C file and include that *.C file at the bottom of your header file.
6. If you use a *for loop* and break out of the loop (using `break;` at some point in the iteration, then consider a *while loop* instead.
7. Don't forget a default statement within switch statements.
8. Please don't open namespaces in source files, i.e. use the fully qualified function name in the function definition to make the scope of the function as explicitly clear as possible.
9. Think about your variable names. I too often see `Node`, `node`, and `n` in the same function. Make your code *obvious* so that I can understand it when I'm tired or stupid (or both).
10. Write good code so that we don't have to debug it after you leave.
11. Indent your code blocks.

My rules for style are as follows. Adhere to them if you like, or don't, if you're appalled by them.

1. Indent your code blocks (I use five spaces, but some consider this excessive).
2. Put spaces between operators for clarity.

21.11 Things That May Happen to Your Code

No one likes to have their code touched, and we would like to avoid having to do so. We would like to have your contribution to ROSE always work and never have to be touched. We don't wish to pass critical judgment on style since we want to allow many people to contribute to ROSE. However, if we have to debug your code, be prepared that we may do a number of things to it that might offend you:

1. We will add documentation where we think it is appropriate.
2. We will add assertion tests (using `ROSE_ASSERT()` macros) wherever we think it is appropriate.
3. We will reformat your code if we have to understand it and the formatting is a problem. This may offend many people, but it will be a matter of project survival, so all apologies in advance. If you fix anything later, your free to reformat your code as you like. We try to change as little as possible of the code that is contributed.

21.12 ROSE Email Lists

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

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

21.13 How To Build a ROSE Distribution with EDG Binaries

The construction of a binary distribution is done as part of making ROSE available externally on the web to users who do not have an EDG license. We make only the EDG part of ROSE available as a binary (library) and the rest is left as source code (just as in an all source distribution).

This step is automated in our daily regression test script in `rose/script/roseFreshTest` and is turned on by a flag `ENABLE_BUILD_BINARY_EDG`. A simplified excerpt of the script is shown below:

```
if [ 0$ENABLE_BUILD_BINARY_EDG -ne 0 ]; then
  cd ${ROSE_TOP}/build
  make binary_edg_tarball || exit 1
  ${ROSE_TOP}/sourcetree/scripts/copy_binary_edg_tarball_to_source_tree_svn
  ${ROSE_TOP}/sourcetree/scripts/copy_binary_edg_tarball_to_source_tree ${ROSE_TOP}/sourcetree
make $MAKEFLAGS source_with_binary_edg_dist DOT_SVNREV=-${svnversion}
echo "############ make source_with_binary_edg_dist done"
fi
```

As shown in the script above, the steps to build a binary distribution of ROSE are:

1. Configure and build ROSE normally, using configure (use all options that you require in the binary distribution).
2. Run `make binary_edg_tarball`. This will make a binary tar ball from EDG and EDG-SAGE connection source files.

3. Copy the binary to the svn repository: `copy_binary_edg_tarball_to_source_tree_svn`
4. Copy the binary to your local copy: `copy_binary_edg_tarball_to_source_tree`
5. Make the ROSE distribution with EDG binaries: `make source_with_binary_edg_dist`

To make sure the binaries are up to date for different platforms, we generate a hash number from the source files within the EDG and EDG-SAGE connection source tree and treat the number as a signature for the binary package. Please see the makefile target `rose_binary_compatibility_signature` in `rose/Makefile.am` for details. Our regression test script will check for the availability and consistency of the binaries for all supported platforms before making an external ROSE release with EDG binaries.

21.14 Avoiding Nightly Backups of Unrequired ROSE Files at LLNL

If your at LLNL and participating in the nightly builds and regression testing of ROSE, then it is kind to the admin staff to avoid having your testing directory *often many gigabytes of files* backed up nightly.

There is a file `.nsr` that you can put into any directory that you don't need to have backed up. The syntax of the text in the file is: `skip: .`

Additional examples are:

```
# The directives in this file are for the legato backup system
# Here we specify not to backup any of the following file types:
+skip: *.ppm *.o *.show*
```

More information can be found at:
www.ipnom.com/Legato-NetWorker-Commands/nsr.5.html or
https://computation-int.llnl.gov/casc/computing/tutorials/toolsmith/backups.htm

Example used in ROSE:
+skip: *.C *.h *.f *.F *.o *.a *.la *.lo *.so *.so.* Makefile rose_test* *.dot

Note: There does not appear to be a way of avoiding the backup on executables.

Thanks for saving a number of people a lot of work.

21.15 Setting Up Nightly Regression Tests

Directions for using a bash script (rose/scripts/roseFreshTest) to set up periodic regression tests:

1. Get an account on the machine you are going to run the tests on.
2. Get a scratch directory (normally /export/0/tmp.¡your username¿) on that machine.
3. Copy (using svn cp) a stub script (scripts/roseFreshTestStub-*) to one with your name.
4. Edit your new stub script as appropriate:
   (a) Set the versions of the different tools you want to use (compiler, ...).
   (b) Change ROSE_TOP to be in your scratch directory.
   (c) Set ROSE_SVNROOT to be the URL of the trunk or branch you want to test.
   (d) Set MAILADDRS to the people you want to be sent messages about the progress and results of your test.

(e) MAKEFLAGS should be set for most peoples' needs, but the -j setting might need to be modified if you have a slower or faster computer.

(f) If you would like the copy of ROSE that you test to be checked out using "svn checkout" (rather than the default of "svn export"), add a line "SVNOP=checkout" to the stub file.

(g) The default mode of roseFreshTest is to use the most current version of ROSE on your branch as the one to test. If you would like to test a previous version, you can set SVNVERSIONOPTION to the revision specification to use (one of the arguments to -r in "svn help checkout").

5. Check your stub script in so that it will be backed up, and so that other people can copy from it or update it to match (infrequent) changes in the underlying scripts.

6. Run "crontab -e" on the machine you will be testing on:

   (a) Make sure there is a line with "MAILTO=¡your email¿".

   (b) Add new lines for each test you would like to run:

      i. If other people are using the machine you are running tests on, be sure to coordinate the time your scripts are going to run with them.

      ii. See "man crontab" for the format of the time and date specification.

      iii. The command to use is (all one line):

```
  cd <your ROSE source tree>/scripts && \
  ./roseFreshTest ./roseFreshTestStub-<your stub name>.sh \
  <extra configure options>
Where <extra configure options> are things like
--enable-edg\_union\_struct\_debugging, --with-C\_DEBUG=...,
--with-java, etc.
```

7. Your tests should then run on the times and dates specified.

8. If you would ever like to run a test immediately, copy and paste the correct line in "crontab -e" and set the time to the next minute (note that the minute comes first, and the hour is in 24-hour format); ensure the date specification includes today's date. Be sure to quit your editor – just suspending it prevents your changes from taking effect.

21.15.1 When We Test and Release ROSE

We have the following timeline (Pacific Time Zone) for testing and releasing ROSE.

Daily tests and updates to the rose website and the SciDAC subversion repository.

1. 1:00 am: Start the regression test on a 32-bit workstation. The test will update the website and the SciDAC repository also if the test passes.
2. 3:40 am: Finish the 32-bit regression test and the updates to the external website and subversion repository.
3. 12:00 pm: Start another the regression test on a 32-bit workstation.
4. 14:40 pm: Finish the 32-bit regression test and updates to the external website and subversion repository.
5. 4:00 am: Run nightly NMI tests (Compile Farm Tests)

We also have a weekly release of a ROSE file package on the SciDAC project page. The script starts every Monday morning 5:00 am and should finish around 7:00am.

21.15.2 Enabling Testing Using External Benchmarks

In addition to testing ROSE using the embedded test cases via *make check*, roseFreshTest also supports automatically testing ROSE on external benchmarks based on the installed copy of ROSE generated by *make install*. Currently, it supports using ROSE's identityTranslator as a compiler to compile a growing subset of the SPEC CPU 2006 benchmark suite.

To enable this feature, do the following:

1. Install the SPEC CPU 2006 benchmark suite to a desired path (e.g. */home/liao6/opt/spec_cpu2006/*) as instructed in its user manual.

2. Prepare a configuration file (e.g. *rose.cfg*) for compiler name, compilation options, and other benchmark options based on the sample config file in *spec_installed_path/config*. A set of relevant options in *rose.cfg* are:

```
 #We want to the test to abort on errors and report immediately
 ignore_errors = no

 # we want have ascii and table-based output (Screen) for results
 output_format = asc, Screen

 #The result is not intended for official reports to the SPEC organization
 reportable    = 0

 # compilers to compile benchmarks
 CC           = identityTranslator
 CXX          = identityTranslator
 FC           = identityTranslator

# compilation options: turn off ROSE 's EDG frontend warnings
# since we are not interesting in fixing the benchmarks
COPTIMIZE     = -O2 --edg:no_warnings
CXXOPTIMIZE  = -O2 --edg:no_warnings
FOPTIMIZE    = -O2 --edg:no_warnings
```

3. Finally, add the following lines in your stub script:

```
# using external benchmarks during regression testing sessions
ENABLE_EXTERNAL_TEST=1

# installation path of spec cpu and the config file for using rose
SPEC_CPU2006_INS=/home/liao6/opt/spec_cpu2006
SPEC_CPU2006_CONFIG=rose.cfg
```

That is it. Now your daily regression test has incorporated the SPEC benchmark.

The subset of the SPEC benchmark and the command line to run them is defined in *rose/script/testOnExternalBenchmarks.sh*. We will continue to enhance the quality of ROSE and add more external benchmarks as time passes by.

21.16 Updating The External Website and Repository

(For the LLNL internal developers only) We have several special top level makefile targets to update the rosecompiler.org and SciDAC Outreach subversion repository. They are controlled by the regression test scripts automatically. Here are some instructions if you really want to do it manually:

21.16.1 rosecompiler.org

Here are the commands to update the rosecompiler.org website:

```
# 1. enter your build tree for ROSE. You should have ran make docs already
 cd build/docs/Rose

# 2. change the scidac.outreach account to yours in the Makefile. e.g
  # in build/docs/Rose/Makefile
  copyWebPages: logo
       cd ROSE_WebPages?; rsync -avz *   yourAccount@web-dev.nersc.gov:/www/host/rosecompiler

# 3. do the uploading, input your password when prompted.
    make copyWebPages
```

21.16.2 The External Repository

To build the binary file of the EDG frontend and the corresponding *EDG_SAGE_CONNECTION* code for the current platform:

- *make binary_edg_tarball*
- add the binary into the internal SVN repository, remove any stale binaries for other platforms as well.
  make copy_binary_edg_tarball_to_source_tree_svn
- make a source release package with EDG binaries.
  make source_with_binary_edg_dist DOT_SVNREV=-svnversion

Finally, a dedicated script will import the release package into the external ROSE svn repository hosted at the SciDAC Outreach Center. You must have an active account with https://outreach.scidac.gov to do this!

rose/scripts/importRoseDistributionToSVN ROSE_TOP_TEST_DIR

It conducts a set of sanity checks and postprocessing before the actual importing. e.g. No EDG copyrighted files in the package, remove .svn and other undesired directories or files, make sure all EDG binaries for supported platforms are available.

21.17 Generating ChangeLog2

You can generate a GNU-style ChangeLog from ROSE's subversion commit logs using a program named *svn2cl*. Download it from `http://ch.tudelft.nl/~arthur/svn2cl/` and install it as directed in its documentation.

The command line we use to generate ChangeLog2 is:

```
svn log --xml --verbose \
| xsltproc --stringparam include-rev yes \
--stringparam ignore-message-starting "Automatic updates" \
/home/liao6/opt/svn2cl-0.11/svn2cl.xsl - > ChangeLog2
```

The command above will include revision numbers into the change log and filter out the automatically generated commits updating EDG binary files.

21.18 Compiling ROSE using ROSE Translators

It is possible to use a ROSE-based translator to compile the ROSE source tree. The motivation could be using Compass checkers to check for bugs or violations in ROSE's source files. However, there are some pending bugs preventing the process from being fully successful.

Here are some instructions:

- rename or copy your translator (such as identityTranslator) executable file to a file named *roseTranslator*, which is the only name that can be recognized by ROSE's configure script to set up necessary flags and system headers etc.
- define *CXX=roseTranslator* to specify the compiler(translator) to compile ROSE during configuration.
- define *-DNDEBUG* as a workaround for a bug related to assert statements.
- define *CXXLD=g++* as a workaround for rose translator's limitations as a linker.

In summary, you have to set the necessary search path and shared library path for your translator and rename it to textitroseTranslator. Then a configuration line likes like the following:

```
../sourcetree/configure --with-boost=/home/liao6/opt/boost_1_35_0 \
--with-CXX_DEBUG="-g -DNDEBUG" --with-C_DEBUG="-g -DNDEBUG" \
--prefix=/home/liao6/daily-test-rose/compass/install \
CXX=roseTranslator CXXLD=g++
```

21.19 Enabling PHP Support

1. Fetch and install PHP (tested with 5.2.6) from `http://www.php.net/downloads.php`. PHC requires a few specific configure flags in order to be able to use PHP properly. Fill in your choice of PHP install location where appropriate in place of `/usr/local/php`.

   ```
   ./configure  --enable-debug --enable-embed --prefix=/usr/local/php
   make && make install
   ```

2. Fetch and install PHC (tested with svn version r1487). Currently only the development release works with ROSE.

   ```
   svn checkout http://phc.googlecode.com/svn/trunk/ phc-read-only
   cd phc-read-only
   touch src/generated/*
   ./configure --prefix=/usr/local/php --with-php=/usr/local/php
   make && make install
   ```

3. Finally, due to an incongruence in the class hierarchies of PHC and ROSE the following changes have to be made to the installed /usr/local/php/include/phc/AST_fold.h. Hopefully this can be resolved soon so that ROSE works with an unmodified upstream PHC.

```
--- src/generated/AST_fold.h    2008-07-30 10:35:32.000000000 -0700
+++ src/generated/AST_fold.h.rose       2008-08-13 15:30:37.000000000 -0700
@@ -1037,7 +1037,7 @@
                        case Nop::ID:
                                return fold_nop(dynamic_cast<Nop*>(in));
                        case Foreign::ID:
-                               return fold_foreign(dynamic_cast<Foreign*>(in));
+                               return 0;
                }
                assert(0);
        }
@@ -1271,7 +1271,7 @@
                        case Nop::ID:
                                return fold_nop(dynamic_cast<Nop*>(in));
                        case Foreign::ID:
-                               return fold_foreign(dynamic_cast<Foreign*>(in));
+                               return 0;
                        case Switch_case::ID:
                                return fold_switch_case(dynamic_cast<Switch_case*>(in));
                        case Catch::ID:
```

4. Once both packages have been installed ROSE must be configured with the additional --with-php=/usr/local/php option.

21.20 Binary Analysis

ME: *Move this binary ocumentation into the anual Binary Analysis chapter.*

The documentation for the binary analysis can be found in the ROSE manual at 14. There are also examples in the ROSE Tutorial. However, there are a collection of details that we need to document about the design; so for now these details can go here. The design behind the support for binary analysis in ROSE has caused a number of design meetings to discuss details. This section is specific to the support in ROSE for binary analysis and the development of the support in ROSE for the binary analysis.

21.20.1 Design of the Binary AST

This subsection is specific to the design of the binary executable file format and specifically the representation of the binary file format in the Binary AST as a tree (in the graph sense) instead of as a directed graph, so that ti can be traversed using the mechanisms available in ROSE.

- Symbols
  Their are multiple references to symbols (as shown in the Whole Graph view of the AST with the binary format). We have selected the SgAsmELFSymbolTable and the SgAsmCoffSymbolTable instead of the SgAsmGenericSymbolTable because it points to the most derived type. An alternative reasoning is that

in stripped binariiies that require DLL support the required symbols in the SgAsmELFSymbolTable and the SgAsmCoffSymbolTable are left in place to support the DLL mechanism where as all entries in the SgAsmGenericSymbolTable are removed (get more details from Robb).

FIXME: *We s reference for the det symbols are left binaries and what required to suppo linking and where they*

- Checking the symbols in the executable using `nm`
  ROSE permits a programmable interface to the binary executable file format, but unix utility functions provide text output of such details. For example, use `nm -D .libs/librose.so | c++filt | less` to generate a list of all the symbols in an executable (text output). In this case `c++filt` resolved the original names from the mangled names for executables built from C++ applications. The C++ symbols appear at the bottom of the listing.

21.20.2 Output from AC_CANONICAL_BUILD Autoconf macro

The ROSE *configure.in* calls the **AC_CANONICAL_BUILD** Autoconf macro as a way to determine some details about the target machine. The results of these for the machines commonly used for development are:

```
Linux (tux270, 64 bit):
   build_cpu    = x86_64
   build_vendor = redhat
   build_os     = linux-gnu

OSX (ninjai: 64bit Mac Desktop):
   build_cpu    = i386
   build_vendor = apple
   build_os     = darwin9.6.0

Cygwin (tux245: 32 bit Windows XP running Cygwin):
   build_cpu    = i686
   build_vendor = pc
   build_os     = cygwin
```

21.21 Testing on the NMI Build and Test Farm

The NMI Build and Test Farm allows us to compile tests on ROSE on a variety of different Operating Systems. For more information on the compile farm see http://nmi.cs.wisc.edu/. These tests can be run against an arbitrary tarball of ROSE source (with EDG binary), or against the HEAD revision of the public svn repository. The purpose of this section is to show how different build and test configurations can be implemented. For a detailed introduction on how to submit jobs to the build system visit http://nmi.cs.wisc.edu/node/31. A refernce manual can be found at http://nmi.cs.wisc.edu/node/65. However, in order to add new tests to ROSE, the information given in this chapter will suffice.

In order to run a test, it has to be submitted on one of the submission hosts provided by the University of Wisconsin. The submission scripts provided with ROSE are developed for Metronome 2.6.0 (This is the framework responsible for parsing and submitting the scripts to the build machines). At the time of this writing, there are three submission hosts. They are:

- `nmi-s001.cs.wisc.edu`
- `nmi-s003.cs.wisc.edu`

- `nmi-s005.cs.wisc.edu`

For every test a build and test run is started.

21.21.1 Adding a test

To add a test which can be run on the Compile Farm you need to add an options file to `<rose_dir>/scripts/nmiBuildAndTestFarm/build_configs/<platform>/`. If using `nmi-submit` (see section 21.21.2) with `--no-skip-update` (the default), the options file does not need to be checked in. However, once your file works, you should check it in to the SVN repository so that it makes it out to the public repository, and then to the NMI cron job.

These option files are simple bash scripts that set variables that determine the configuration of the run on the platform, which is implied by the directory the options file is placed in. The name of the option file itself is not interpreted in any special way.

Overview of the options:

- `TITLE` - The title of the test
- `DESCRIPTION` - Short text to describe the test
- `PREREQS` - Define what software this run needs. The prereqs available can be seen by navigating to: http://nmi-s005.cs.wisc.edu/nmi/index.php?page=pool/platform and clicking on the platforms you want to use.
- `CONFIGURE_OPTIONS` - Define which options you want to pass to `configure`. You will very likely need, at a minimum, to refer to the correct boost directory.
- `JAVA_HOME` - If `java` is included in the prereqs, `JAVA_HOME` should be specified. This will be passed to the environment of the running test.
- `ACLOCAL_INCLUDES` - Some prereqs may have `m4` macros in nonstandard locations. This can be passed to the build script (and subsequently to `aclocal`) via `ACLOCAL_INCLUDES`. The value is passed verbatim, and so should be space separated entries of the form "`-I <dir>`". A common requirement is to include the path to the libxml-2.2.7.3 `m4` macros with a value of "-I /prereq/libxml2-2.7.3/share/aclocal/".

Example options file:

```
TITLE="testing default on all linux platforms"
DESCRIPTION="minimal configuration options, gcc 4.2.4, without java"
PREREQS="gcc-4.2.4, boost-1.35.0"
CONFIGURE_OPTIONS="--with-boost=/prereq/boost-1.35.0 --with-CXX_WARNINGS=-Wall --without-java"
```

21.21.2 Manually submitting tests

Tests can be manually submitted with the script `nmi-submit`. This is a ruby program in the `scripts/nmiBuildAndTestFarm` directory. See `nmi-submit --help` for more information. At the time of this writing, this would output the following:

```
Usage: nmi-submit [options] [TARBALL] CONFIG [CONFIG...]
Submit TARBALL to platforms specified by each CONFIG, which must be
files in the subtree ROSE/scripts/nmiBuildAndTestFarm.
```

```
        --no-tarball                  Submit the current HEAD of the public subversion
                                      repository instead of a tarball.

        --[no-]skip-update            With --no-skip-update, nmi-submit will copy files (e.g.
                                      submit.sh, glue.pl, &c) to the submit host to ensure
                                      that they are up-to-date.  This step can be skipped to
                                      speed up nmi-submit.  Defaults to --no-skip-update.

        --[no-]fork                   If --fork is specified, all subprocesses are forked.
                                      This allows nmi-submit to be more responsive to INT
                                      signals, but then requres that ssh and scp can be run
                                      passwordless (e.g. because ssh-agent has an appropriate
                                      identity loaded).  Defaults to --fork.

        --submit-host = HOST          Specify the submission host to use.  Defaults to
                                      heller@nmi-s005.cs.wisc.edu.

        --user-dir = REMOTE_DIR       Use REMOTE_DIR on the submission host as a working area
                                      to stage files and run the submission from.  Defaults to
                                      'id -un'-rose-nmi.
                                      WARNING: nmi-submit will write to REMOTE_DIR
                                      indiscriminately.  Don't keep your family photos there.

    -h, --help                        Show this message
```

`nmi-submit` can submit a tarball (built with `make binary_tarball` in the compile tree) and a list of options files, or with the `--no-tarball` option, submit a list of options files to be tested against the current version of the public repository.

`nmi-submit` should be run locally. It is recommended to run it from your source tree, specifically in the `scripts/nmiBuildAndTestFarm` directory, although this is not required.

Once you have submitted a test, you should be given a `RunID`, and your test should appear on the search results page at http://nmi-web.cs.wisc.edu/nmi/index.php?page=results%2Foverview&opt_project=rose+compiler.

21.21.3 Cron automated tests

Jobs can be added to the cronjobs file in the directory <rose_dir>/scripts/nmi. These cronjobs will be loaded every night into the crontab file on the submission host (this is done by the first entry, which executes `update.sh`). In order to add your own build tests, simply add a line there (see `man 5 crontab` for more information). Be sure to test your submission before adding it to the cronjobs.

NOTE: If you want your cronjobs to be permanent, add this to your local svn copy, and not the checkout on the submit machines. Also be sure to add the options file that specify the test to the svn repository.

Example entry:

```
# run the minimal_default test every day at midnight
0 0 * * * cd \${CWD}; ./submit.sh build_configs/x86_64_deb_4.0/minimal_default
```

21.21.4 Viewing the Results of Recent Tests

One way to see the results of recent tests is to navigate to http://nmi-web.cs.wisc.edu/nmi/index.php?page=results%2Foverview&opt_project=rose+compiler. A command-line friendlier tool exists, namely `nmi-summary`, which will summarize tasks and give results for the individual tasks `configure`, `make`, `check`.

`nmi-summary` depends on `ruby`, `rubygems` and the `hpricot` gem. See `nmi-summary --help` for more information, which, at the time of this writing, produces the following:

```
usage:      nmi-summary [DAY]
   or:      nmi-summary [DAY] RUNID_RANGE

In the first form, gives a summary for tasks run on DAY.

The second form is the same, except only tasks whose runids fall within
RUNID_RANGE are included.

In either form, if DAY is omitted, it defaults to the most recent day in
which a task was run.

    [DAY]           Should be spcified in the format YYYY/MM/DD.

    [RUNID_RANGE]   Should be specified in the format LOWER..UPPER.
                    Both LOWER and UPPER are inclusive.  Either may be
                    omitted.  LOWER defaults to 0 and UPPER defaults to
                    a large number (essentially infinity).

EXAMPLES
--------

    The following will show results for 11 September 2009 with RunIDs
    181300 or higher:

        nmi-summary 2009/09/11 181300..

    The following will show results for the most recent day with
    submissions, whose RunIDs fall within the range 181300 and 181400,
    inclusive:

        nmi-summary 181300..181400

    The following will show all results for 11 September 2009:

        nmi-summary 2009/09/11
```

```
DEPENDENCIES
------------
    nmi-summary depends on rubygems and the hpricot gem.

        yum install rubygems && gem install hpricot

NOTES
-----
    nmi-summary scrapes data from the NMI website, making N+2 requests.
    It is not speedy, and its users must exercise patience.
```

21.21.5 `cleanup.sh`

The process of submitting tests produces some temporary files. One is a generated environment file that is quite small. However, submitting individual tarballs leaves a copy of the tarball on the submit host, which is necessary so that the run host can access it.

So as not to unduly burden the NMI submit host hard drives, we have a script, `cleanup.sh`, which is included in the crontab and cleans up any such temporary files older than 72 hours. Although not necessary (thanks to cron), it is safe to run the script manually.

21.21.6 Troubleshooting with `nmi-postmortem`

If a run fails, it can be helpful to examine the environment that it ran on. There is a small program, `nmi-postmortem`, that aims to automate some of the tedium of doing this. On the results page for a run, you can find the `RunID` (see Figure 21.3). Alternatively, you can find the `RunID` from `nmi-summary` (see section 21.21.4).

`nmi-postmortem` is intended to be run on the submit host. If it is not in the `PATH` of the account you are using there, then `scp` the file to the submit host and place it somewhere in your `PATH`. If you are using the shared account, then `nmi-postmortem` should already be in your `PATH`.

With this, you can invoke the following on the submit host:

```
nmi-postmortem <runid>
```

This will do the following:

- Determine the machine the test ran on (the `run host`).
- Ensure that it is possible to `ssh` to the run host. This may require you entering the account's password a couple of times, but is otherwise automated. This amounts to copying the public key from the submit host to the run host's `$HOME/.ssh/authorized_keys` file.
- Copy `results.tar.gz` to the run machine and extract it there in a directory called `run`. `WARNING`: Any previous run directory on that machine will be removed first.
- `ssh` you onto the run machine, `cd` to the run directory and source the environment file for the run. At this point you should be able to investigate in an environment very close to the one the actual run failed on.

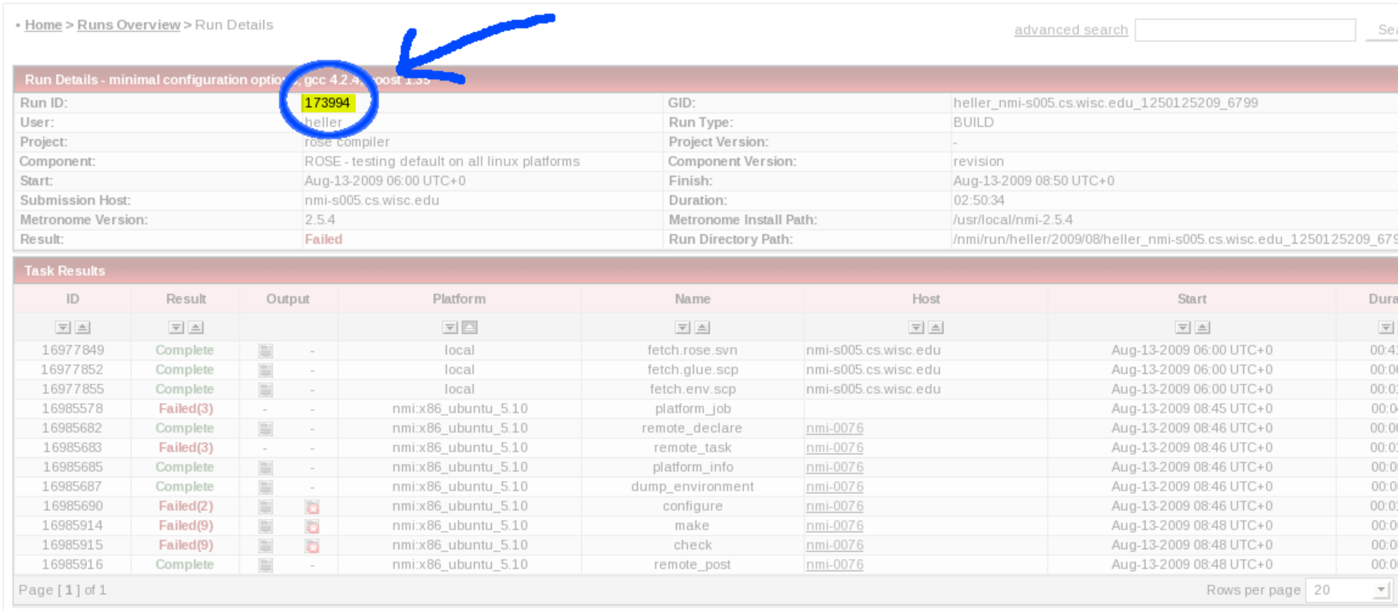

Figure 21.3: Example screenshot of a results page, `runid` highlighted.

NOTE: `nmi-postmortem` invokes `ssh` a lot and assumes that there exists a file `$HOME/.ssh/id_rsa.pub` on the submit host and that this key has no passphrase. If this is not the case for the account you are using, simply invoke `ssh-keygen` and be sure not to specify a passphrase.

21.21.7 Default Timeouts

See the manual.

21.21.8 Where to get help

1. **Mailing Lists** - It is recommended to subscribe to the mailing lists listed at http://nmi.cs.wisc.edu/node/521. As of this writing, these include `uw-nmi-announce and nmi-users`.
2. **Support** - Support's email is nmi-support@cs.wisc.edu.
3. **The Manual** - http://nmi.cs.wisc.edu/node/31.

21.22 ROSE API Refactoring

This is the outline of the API, add API functions to the next section.

This a draft design for a new High Level ROSE API where high level function interfaces will be located that call mechanisms for analysis, transformation, and expected user level support for ROSE tools. This support is presently spread around in ROSE and this API would centralize it and make ROSE more clear to users. There are four levels:

1. ROSE Frontend
   Generation of Abstract Syntax Tree (AST) from source code or binary executable. The AST holds structural representations of the input software.
2. ROSE Midend
   Analysis and transformation support for ROSE-based tools.
   (a) ROSE Analysis API
       This would include intra-procedural analysis, inter-procedural analysis, and whole program analysis (which over comes the issues of separate compilation). This analysis can handle either source code analysis, binary analysis, or both. Program analysis on source code includes:
       i. Program analysis on source code includes:
          A. Call Graph Analysis
          B. Class Hierarchy Analysis
          C. Control Flow Analysis
          D. Def-Use Analysis
          E. Dominance Analysis
          F. Dominator Trees And Dominance Frontiers Analysis (old)
          G. Connection of Open Analysis (old)
          H. Pointer Analysis
          I. Procedural Slicing (old; not used)
          J. Side-Effect Analysis
          K. Value Propagation Analysis
          L. Static Interprocedural Slicing (replaces Procedural Slicing)
          M. Liveness Analysis
          N. Dependence Analysis
          O. AST Interpreter (Interpretation of Concrete Semantics using AST)
       ii. Program analysis on binaries includes:
          A. Call graph Analysis
          B. Control Flow Analysis
          C. Constant Propogation
          D. Data Flow Analysis
          E. InstructionSemantics
          F. Library Identification (FLIRT)
          G. Dwarf Debug Format
          H. Analysis of the Binary File Format
   (b) ROSE Transformation API
       Modifications of the AST can be organized as:
       i. Instrumentation
       ii. Optimization These include a range of optimizations relevant for general performance optimization of scientific applications.

A. Inlining
B. Loop optimizations:fusion, fisson, unrolling, blocking, loop interchange, array copy, etc.
C. Constant Folding
D. Finite Differencing
E. Partial Redundancy Elimination

iii. General Transformations These include outlining,

A. Outlining
B. ImplicitCodeGeneration
This work makes C++ implicit semantics explicit for C style analysis.
C. FunctionCallNormalization
This is a library of function call normizations to support binary analysis.
D. AST Copy support
This support permits arbitrary subtrees (or the whole AST) to be copied with control over deep or shallow copying via a single function.
E. AST Merge support
This work permits the merging of separate AST's and the sharing of their identically names language declarations to support whole program analysis. Duplicate parts of the merged AST are deleted.
F. Static Binary Rewriting
A restricted set of transformations are possible on a binary executable, this section details this work.

(c) AST Traversals
ROSE provides a number of different techniques to define traversals of the AST and associated graphs formed from the AST.

3. ROSE Backend
Code generation from the AST (unparsing) and optionally calling the backend compiler. ROSE includes a number of features specific to the code generation phase:

(a) Code generation from arbitrary subtrees of the AST
Users can generate code from subsets of the AST as part of support for custom code generation.
(b) Generation of abritrary test with generated code
This section contains the support for the output of arbitrary text as part of the generation of code (useful for generating code for specialized GPU tools, etc.).
(c) Code generation Format control
Some control is possible over the formatting of generated code within ROSE.

4. ROSE Util
Utility functions useful in ROSE-based tools.

(a) AST Visualization
AST support for visualization includes representations as PDF, DOT, and a more colorful representation of the whole graph that includes AST plus type attributes (not typically as part of an AST). This work includes support for dot2gml translation (in roseIndependentSupport/dot2gml). this is where interfaces to possible OGDF (Open Graph Drawing Framework) could be put.

(b) AST Query
The AST Query mechanism is a simple approach to getting list of IR nodes. It is typically used within analysis or transformations.

(c) AST Consistancy Tests
The consistancy tests validate that the AST is correctly formed. Note that this is not a test that the code that will be gnerated is leagal code.

(d) Performance monitoring
This section provides support using in ROSE to measuring both space and time complexity for ROSE based tools.

(e) AST Postprocessing
The AST postprocessing is a step used to fix the AST after some types of modification by the user and to make it a correctly formed AST. Not all modifications to the AST can be corrected using this step.

(f) AST File I/O Support
This section contains the support for writing and reading the AST to and from files (binary file I/O is used and the design is for performance (both space and time)).

(g) Language specific name support
This section contains the support for generating unique names for language constructs and handling mangled and unmangled names for use in ROSE based tools.

(h) Support for comments and CPP directives
This section contains the support for reading and writing comments and CPP directives within the AST.

(i) GUI Support
This section contains the support for building GUI based tools using ROSE.

(j) Binary Analysis connection to IDA PRO
This section contains the support for using IDA Pro with ROSE for Binary Analysis.

(k) Database Support
This section contains the support for building tools that use SQLite Database.

(l) Graphs and Graph Analysis
This section contains the support for building custom graphs to represent static and dynamic ananlysis and graph analysis algorithms to suport of analysis of these graphs.

(m) Performance Metric Annotation
This section contains the support for dynamically derived information to be written into the AST (performance inforamtion to support analysis and optimization tools).

(n) Abstract Handles
This section contains the support for building abstract handles into source code. This work is used in the autotuning and also other tools that pass references to source code as part of an interface.

(o) Macro Rewrapper
This is currently in the ROSE/projects directory and should perhaps be a part of the ROSE API.

(p) Command-line processing support
This is the command line handling used internally by ROSE and made available so that users can process the command line for their specific ROSE based tools.

(q) Common string support
These function support common operations on strings used within ROSE and useful within ROSE-based tools.

(r) Common file and path support
This is a collection of function useful for handling directory structures within ROSE-based tools.

(s) Miscellaneous Support
Output of useage information, ROSE version number support, etc.

21.23 ROSE API (PUT YOUR LISTS OF FUNCTIONS HERE)

This is a group effort to define a better set of high level documents for ROSE that will explain what is in ROSE at a high level and what users need to know about the ROSE API and a moderate level of detail. Only functionality expected to be useful to the development of ROSE based tools by external users are presented. Lower level details are available in other documentation or via the doxygen generated documentation.

Some helpful notes, other ideas, issues, etc.:

1. Class-based
   It seems that interfaces are more useful if we place them in a class rather than a namespace.

2. Virtual vs. not-virtual
   Except where performance is an issue, it's useful to have mostly virtual methods.

3. Namespace
   The ROSE API should be in a "rose" namespace which all of our source files each import.

4. Naming style
   Agree on style. Most of ROSE uses SomeClass for classes and someMethod for methods and functions.

5. Macros
   Header file macros should all be the same form, such as ROSE_WHATEVER_H. Feature macros should use a common form (perhaps ROSE_HAS_WHATEVER or ROSE_USES_WHATEVER). Function like macros should be replaced with inline functions. Value macros should be replaced with static const data members.

6. Who manages pointed-to memory
   Classes with pointers should have a clear statement of who manages the pointed-to memory: the class or the caller. Also, if the caller manages memory then when can the caller delete that memory (must it wait until the object that uses it is deleted or did the object make a copy)?

7. Include files
   Must the end user include all of rose (rose.h) in order to use a particular class? Our compiling would be *so*much*faster* if each file included only what it actually used. (We could still have a "rose.h"-like file that includes everything for the lazy end user.)

   FIXME: *DQ has a a single rose.h sin bugs due to different*

8. Some provision for header/library consistency
   For instance, certain functions in the HDF5 API (like H5_init() that initializes the library) pass a version number from a header file that gets compared with a version number compiled into the library. If they don't match then there will probably be runtime issues and HDF5 can report this before the user gets a core dump. ROSE could do something similar.

9. Namespace aliases can be used to provide alternative shorter namespace names for users, so we can focus on having names that are as clear as possible.

Outline where to put list of functions/functionality for each category of the API.

21.23.1 Story Of ROSE (JK)

This is a section that Jeff Keasler has specific ideas about how to write and for which he will provide an outline to start the process.

21.23.2 User API (All)

Proposed location of new ROSE API: *ROSE/src/API*

The ROSE User Application Program Interface (API) is the subset of ROSE that is typically required by users to write ROSE based applications for the general processing of software (source code or executable). Specialized projects may require deeper levels of the ROSE software than present in this API and many project may not use but a small part of this API. This documentation is to present ROSE at a high level while covering enough detail to make it clear what different parts of ROSE are available.

ROSE will soon be represented by a number of namespaces. The API will be represented by four namespaces (the Intermediate Representation is expected to be in its own namespace. It is not clear if there should be a single top level namespace or if there should be namespace aliases that would permit alternative shorter names (an option).

Frontend (Yi)

Interm proposed namespace name: *ROSE_Frontend*

Liao will be proposing names for us to agree upon separately.

Ve might discover that d and backend are too e to deserve their own namespaces.

The frontend of ROSE takes the source code or binary executable and generates an Abstract Syntax Tree (AST), which for the basis of further work. The AST forms a structural representation of the source code or binary executable.

1. **SgProject* frontend (int argc, char** argv);**
   Generates an AST represented by the root node (*SgProject*) from the commandline in the form defined by `main(int argc, char** argv)`.
2. **SgProject* frontend (const std::vector<std::string> & argv);**
   Generates an AST represented by the root node (*SgProject*) from an alternative representation of the command line more useful when custom command line editing is required by the translator.
3. **SgProject* frontendShell (int argc, char** argv);**
   Generates an AST represented by the root node (*SgProject*) from the common command line form, but for files that might be conditionally compiled later.
4. **SgProject* frontendShell (const std::vector<std::string> & argv);**
   Generates an AST represented by the root node (*SgProject*) from the alternative command line form, but for files that might be conditionally compiled later.

These might be too low for the proposed API.

Notes from Robb

1. Parsing functions
   These methods parse a particular entity from a binary file and fill in an existing IR node that was recently constructed. See parse() methods in src/frontend/BinaryFormats/*.C
2. Disassembly
   Disassembling a buffer into a std::map of instructions. ROSE normally calls this automatically, does a little analysis to organize instructions into basic blocks and basic blocks into functions, and links everything into the AST. However, its also useful to call the disassembler explicitly. Disassemblers can be specialized by derivation. There's a number of functions and full doxygen documentation (the actual functions that disassemble a _single_ x86, ARM, or PowerPC instruction are only lightly documented).
   See doxygen for Disassembler class.
   See src/frontend/Disassembler/Disassembler.h

Midend (All)

Interm proposed namespace name: ***ROSE_Midend***
The midend of ROSE is typically where the user interacts with the AST or uses features in ROSE to generate alternative graphs to represent specific types of program analysis. The midend includes both analysis and transformation capabilities and is used by the users to build custom analyzes and transformations.

Analysis (TP) Interm proposed namespace name: ***ROSE_Analysis***
Analysis within ROSE by definition does not modify the AST structure. It might add attributes to IR nodes, but it does not change the structure of the AST. In may cases it may generate specific data structures and separate analysis may traverse these data structures; thus both are covered separately.

Construct (TP & DQ) This covers the construction of various data structures that are part of specific forms of analysis or are used in subsequent forms of analysis. We separate out the API that is specific for source code and binary executable.

1. Source (TP)

   (a) Call Graph Analysis

```
<CallGraph.h>
  void buildCallGraph();
  template<typename Predicate> void buildCallGraph(Predicate pred);
  SgIncidenceDirectedGraph *getGraph();

  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, SgFunctionDeclaration* fun
  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, Properties* functionProper
  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, std::string name );
  SgGraphNode* findNode ( Rose_STL_Container<SgGraphNode*> & nodeList, std::string name, int );
  SgGraphEdge* findEdge (SgIncidenceDirectedGraph* graph, SgGraphNode* from, SgGraphNode* to);
  SgGraphNode* findNode(SgGraph* graph, std::string nid);

  sqlite3x::sqlite3_connection* open_db(std::string gDB  );
  void createSchema ( sqlite3x::sqlite3_connection& gDB, std::string dbName );
  //Will load all graphs represented in the database into one graph
  SgIncidenceDirectedGraph* loadCallGraphFromDB (sqlite3x::sqlite3_connection& gDB);
  void writeSubgraphToDB ( sqlite3x::sqlite3_connection& gDB,SgIncidenceDirectedGraph* callGraph
  void solveFunctionPointers (  sqlite3x::sqlite3_connection& gDB);
  void solveVirtualFunctions (  sqlite3x::sqlite3_connection& gDB, std::string dbHierarchy );
```

   (b) Class Hierarchy Analysis

```
<ClassHierarchyGraph.h>
  void setAST( SgNode *proj );
  SgIncidenceDirectedGraph* getClassHierarchyGraph();
  SgGraphNode* findNode(SgNode*);

  SgClassDefinitionPtrList getSubclasses( SgClassDefinition * );
  SgClassDefinitionPtrList getDirectSubclasses( SgClassDefinition * );
  SgClassDefinitionPtrList getAncestorClasses( SgClassDefinition * );
```

(c) Control Flow Analysis

```
<virtualCFG.h>

  //! A node in the control flow graph.  Each CFG node corresponds to an AST
  //! node, but there can be several CFG nodes for a given AST node.
  class CFGNode {
    public:
    CFGNode(): node(0), index(0) {}
    explicit CFGNode(SgNode* node, unsigned int index = 0): node(node), index(index)
    //! Pretty string for Dot node labels, etc.
    std::string toString() const;
    //! String for debugging graphs
    std::string toStringForDebugging() const;
    //! ID to use for Dot, etc.
    std::string id() const;
    //! The underlying AST node
    SgNode* getNode() const {return node;}
    //! An identifying index within the AST node given by getNode()
    unsigned int getIndex() const {return index;}
    //! Outgoing control flow edges from this node
    std::vector<CFGEdge> outEdges() const;
    //! Incoming control flow edges to this node
    std::vector<CFGEdge> inEdges() const;
    //! Test whether this node satisfies a (fairly arbitrary) standard for
    //! "interestingness".  There are many administrative nodes in the raw CFG
    //! (nodes that do not correspond to operations in the program), and this
    //! function filters them out.
    bool isInteresting() const;
    //! Equality operator
    bool operator==(const CFGNode& o) const {return node == o.node && index == o.index;
    //! Disequality operator
    bool operator!=(const CFGNode& o) const {return !(*this == o);}
    //! Less-than operator
    bool operator<(const CFGNode& o) const {return node < o.node || (node == o.node &&
  }; // end class CFGNode

  //! A control flow edge connecting two CFG nodes, with an edge condition to
  //! indicate edge types
  class CFGEdge {
    public:
    //! Constructor
    CFGEdge(CFGNode src, CFGNode tgt): src(src), tgt(tgt) {}
    //! Pretty string for Dot node labels, etc.
    std::string toString() const;
    //! String for debugging graphs
    std::string toStringForDebugging() const;
    //! ID to use for Dot, etc.
    std::string id() const;
    //! The source (beginning) CFG node
```

```
    CFGNode source() const {return src;}
    //! The target (ending) CFG node
    CFGNode target() const {return tgt;}
    //! The control flow condition that enables this edge
    EdgeConditionKind condition() const;
    //! The label of the case represented by an eckCaseLabel edge
    SgExpression* caseLabel() const;
    //! The expression of the computed goto represented by the eckArithmeticIf* conditions
    unsigned int computedGotoCaseIndex() const;
    //! The test or case key that is tested as a condition of this control flow edge
    SgExpression* conditionBasedOn() const;
    //! Variables going out of scope across this edge (not extensively tested)
    std::vector<SgInitializedName*> scopesBeingExited() const;
    //! Variables coming into scope across this edge (not extensively tested)
    std::vector<SgInitializedName*> scopesBeingEntered() const;
    //! Compare equality of edges
    bool operator==(const CFGEdge& o) const {return src == o.src && tgt == o.tgt;}
    //! Compare disequality of edges
    bool operator!=(const CFGEdge& o) const {return src != o.src || tgt != o.tgt;}
  }; // end CFGEdge

    class InterestingNode {
    public:
    InterestingNode(CFGNode n): n(n) {}
    std::string toString() const {return n.toString();}
    std::string toStringForDebugging() const {return n.toStringForDebugging();}
    std::string id() const {return n.id();}
    SgNode* getNode() const {return n.getNode();}
    unsigned int getIndex() const {return n.getIndex();}
    std::vector<InterestingEdge> outEdges() const;
    std::vector<InterestingEdge> inEdges() const;
    bool isInteresting() const {return true;}
    bool operator==(const InterestingNode& o) const {return n == o.n;}
    bool operator!=(const InterestingNode& o) const {return !(*this == o);}
    bool operator<(const InterestingNode& o) const {return n < o.n;}
  };

  class InterestingEdge {
    public:
    InterestingEdge(CFGPath p): p(p) {}
    std::string toString() const {return p.toString();}
    std::string toStringForDebugging() const {return p.toStringForDebugging();}
    std::string id() const {return p.id();}
    InterestingNode source() const {return InterestingNode(p.source());}
    InterestingNode target() const {return InterestingNode(p.target());}
    EdgeConditionKind condition() const {return p.condition();}
    SgExpression* caseLabel() const {return p.caseLabel();}
    SgExpression* conditionBasedOn() const {return p.conditionBasedOn();}
    std::vector<SgInitializedName*> scopesBeingExited() const {return p.scopesBeingExited();}
    std::vector<SgInitializedName*> scopesBeingEntered() const {return p.scopesBeingEntered();}
```

```
    bool operator==(const InterestingEdge& o) const {return p == o.p;}
    bool operator!=(const InterestingEdge& o) const {return p != o.p;}
  };

  inline InterestingNode makeInterestingCfg(SgNode* start);
  //! Returns CFG node for just before start
  inline CFGNode makeCfg(SgNode* start);
  //! The first CFG node for a construct (before the construct starts to
  //! execute)
  inline CFGNode cfgBeginningOfConstruct(SgNode* c) ;
  //! The last CFG node for a construct (after the entire construct has finished
  //! executing). This node may not actually be reached if, for example, a goto
  //! causes a loop to be exited in the middle
  inline CFGNode cfgEndOfConstruct(SgNode* c);
```

(d) Def-Use Analysis

```
<DefUseAnalysis.h>
  // def-use-public-functions -----------
  int run();
  int run(bool debug);
  multitype getDefMultiMapFor(SgNode* node);
  multitype  getUseMultiMapFor(SgNode* node);
  std::vector < SgNode* > getDefFor(SgNode* node, SgInitializedName* initName);
  std::vector < SgNode* > getUseFor(SgNode* node, SgInitializedName* initName);
  bool isNodeGlobalVariable(SgInitializedName* node);
  std::vector <SgInitializedName*> getGlobalVariables();
  // the following one is used for parallel traversal
  int start_traversal_of_one_function(SgFunctionDefinition* proc);

<DefUseAnalysis_perFunction.h>
  FilteredCFGNode < IsDFAFilter > run(SgFunctionDefinition* function, bool& abortme);
```

(e) Dominance Analysis (CI 2007)

```
<DominatorTree.h>
  // ! Constructor for the DominatorForwardBackwardWrapperClass
  DominatorForwardBackwardWrapperClass(Direction dir):treeDirection(dir)
  // ! returns whether this is a dominator tree (PRE) or a
  // post-dominator tree (POST)
  Direction getDirection()

  // CI (01/23/2007): Implemented the DT for the VirtualCFG interface with
  // the Lingauer-Tarjan algorithm
  //! TemplatedDominatorTree constructs a dominator/postdominator tree for a cfg. For th
  //!constructor for the DT. Head is the start point for the DT construction. DT works
  //Direction determines Pre/Post-Dominator construction
  TemplatedDominatorTree(SgNode * head, Direction d =
                  DominatorForwardBackwardWrapperClass <
                  CFGFilterFunction >::PRE_DOMINATOR);
  //! writes the DT in DOT-notation to the file given in filename
```

```
  void writeDot(char *filename);
  // ! returns the number of nodes in the tree
  int getSize()
  //! returns the set of nodes directly dominated by nodeID
  std::set<int> getDirectDominatedSet(int nodeID)
  // ! for a given nodeID, return the id of its immediate dominator
  int getImDomID(int i)
  //! get the ImDomID for given SgNode, returns negative for non-cfg-node
  int getImDomID(VirtualCFG::FilteredCFGNode < CFGFilterFunction > node)
  //! calculates if a dominates b, i.e. a is on the path from b to the root
  bool dominates(int a,int b)
  //! returns true if node a dominates node b, see dominates(int a,int b)
  bool dominates(VirtualCFG::FilteredCFGNode < CFGFilterFunction > a,VirtualCFG::FilteredCFGNode <
  // ! for an CFG Node, return the corresponding id
  int getID(VirtualCFG::FilteredCFGNode < CFGFilterFunction > node)

  /* ! \class DominanceFrontier
     This class constructs the dominance (or post-dominance) frontiers for
        all nodes in a ControlFlowGraph. A dominance (post-dominance) frontier
        for node X is simply the set of nodes such that a given node Y from the
        set is not dominated (post-dominated) by X, but there is an immediate
        predecessor of Y that is dominated (post-dominated) by X.
        The type of frontier we construct is determined by the DominatorTree
        that DominanceFrontier is initialized with.
 */
  template < typename CFGFilterFunction > class TemplatedDominanceFrontier:public DominatorForwar
  //! retunrs a set of ID's with the nodes dominance-frontier
  std::set<int> getFrontier(int node)
  //! construct the domnancefrontier
  TemplatedDominanceFrontier(TemplatedDominatorTree < CFGFilterFunction > dt):DominatorForwardBac
  //! debug method to print forntiers
  void printFrontiers()
```

(f) Dominator Trees And Dominance Frontiers Analysis (might be old)

```
<ControlFlowGraph.h>
  //! The constructor for ControlFlowGraph. Builds a CFG rooted at head
  ControlFlowGraph(SgNode * head);
  //! from a given CFGImpl node, create one (or more) ControlNodes
  void createNode(CFGNodeImpl * node);
  //! return the number of nodes in the CFG
  int getSize() {return _numNodes;}
  //! given a node id (and which numbering scheme to use), return the appropriate control node
  ControlNode * getNode(int id, ID_dir dir) {return (dir == FORWARD)?_forIndex[id]:_backIndex[id]
  //! dump the contents of the original CFGImpl to a dot file
  void outputCFGImpl();

<SimpleDirectedGraphNode.h>
  //! get the nodes which are pointed to by the current node
  std::set<SimpleDirectedGraphNode *> getSuccessors() {return _succs;}
  //! get the nodes which point to the current node
```

```
  std::set<SimpleDirectedGraphNode *> getPredecessors() {return _preds;}
  //! add an edge from the current node to n
  void addSuccessor(SimpleDirectedGraphNode * n) {_succs.insert(n);}
  //! add an edge from n to the current node
  void addPredecessor(SimpleDirectedGraphNode * n) {_preds.insert(n);}
  //! test whether n is a successor of the current node
  bool hasSuccessor(SimpleDirectedGraphNode * n) {return _succs.count(n) != 0;}
  //! test whether n is a predecessor of the current node
  bool hasPredecessor(SimpleDirectedGraphNode * n) {return _preds.count(n) != 0;}
  //! return the number of outgoing edges
  int numSuccessors() {return _succs.size();}
  //! return the number of incoming edges
  int numPredecessors() {return _preds.size();}
  //! virtual function to support displaying node information
  virtual void writeOut(std::ostream & os)

<DominatorTree.h>
  DominatorTree(SgNode * head, Direction d = PRE);
  //! get the CFG the dominator tree is built from
  ControlFlowGraph * getCFG() {return _cfg;}
  //! returns whether this is a dominator tree (PRE) or a post-dominator tree (POST)
  Direction getDirection() {return _dir;}
  //! returns the corresponding direction for the numbering of the CFG.
  ControlFlowGraph::ID_dir getCFGDirection() {return _iddir;}
  //! returns the number of nodes in the tree
  int getSize() {return _size;}
  //! for a given node, return the id of its immediate dominator
  int getDom(ControlNode * node) {return doms[node->getID(_iddir)];}
  //! for a given node id, return the id of its immediate dominator
  int getDom(int id) {return doms[id];}
  void printCFG();
  void printDominators();

<DominanceFrontier.h>
  DominanceFrontier(DominatorTree * dt) : _dt(dt),
                                          _size(_dt->getSize()),
                                          _domFrontier(new std::set<int>[_size])
  /*! get the dominance frontier for a given node (these need to be
  referenced against the CFG to determine the actual nodes in the
  frontier
  */
  std::set<int> getFrontier(int id) {return _domFrontier[id];}
  void printFrontier();
```

(g) Connection of Open Analysis (might be old)
This might be something for Colorado State to comment upon.

```
<CallGraph/CallGraph.h>
  Node *Entry() { return entry; }; // FIXME
  Node *Exit() { return exit; };
  IRInterface &GetIRInterface() { return ir; }
```

```

  class Node : public DGraph::Node {
  public:
    unsigned int getID () { return label; }
    bool IsDefined() { return (def != 0); }
    bool IsUsed() { return (uses.size() != 0); }
    ProcHandle GetDef() { return def; }
    void dump(std::ostream& os);
    void shortdump(CallGraph* cgraph, std::ostream& os);
    void longdump(CallGraph* cgraph, std::ostream& os);
    friend class CallGraph::NodeUsesIterator;
    void add_def(ProcHandle h) { def = h; }
    void add_use(ExprHandle h) { uses.push_back(h); }
  }

  class Edge : public DGraph::Edge {
  public:
    EdgeType getType() { return type; }
    void dump (std::ostream& os);
  };


<SSA/DomeTree.h>
  DomTree (DGraph& g);
  Node* domtree_node (DGraph::Node* n) { return dom_tree_node[n]; }
  void compute_dominance_frontiers ();
  void dump (ostream&);

<SSA/Phi.h>
  Phi (const SymHandle& var_name, CFG* _cfg) { sym = var_name; cfg = _cfg; }
  void dump (ostream&);
  void add_arg (CFG::Node* c_n, LeafHandle a_n) { args[c_n] = a_n; }
  LeafHandle arg (CFG::Node* n) { return args[n]; }
  int num_args () { return args.size(); }

<SSA/SSA.h>
  class Def {
  public:
    virtual void dump (ostream&) = 0;
    virtual std::list<Use*>* uses_list () = 0;
  };
  class Use {
  public:
    virtual void dump (ostream&) = 0;
    virtual Def* def () = 0;
  };
  class LeafDef : public Def {
  public:
    LeafDef (LeafHandle l) : Def() { leaf = l; }
    void dump (ostream&);
```

```
    std::list<Use*>* uses_list () { return &uses; }
  };
  class PhiDef : public Def {
  public:
    PhiDef (Phi* p) : Def() { phi = p; }
    void dump (ostream&);
    std::list<Use*>* uses_list () { return &uses; }
  };
  class LeafUse : public Use {
  public:
    LeafUse (LeafHandle l) : Use() { leaf = l; }
    void dump (ostream&);
    Def* def () { return definition; }
  };
  class PhiUse : public Use {
  public:
    PhiUse (Phi* p) : Use() { phi = p; }
    void dump (ostream&);
    Def* def () { return definition; }
  };

<CFG/CFG.h>
  Node *Entry() { return entry; };
  Node *Exit() { return exit; };
  IRInterface &GetIRInterface() { return ir; }
  Node* splitBlock(Node*, StmtHandle /* CFG::NodeStatementsIterator */);
  void connect (Node* src, Node* dst, EdgeType type)
  void connect (Node* src, Node* dst, EdgeType type, ExprHandle expr)
  void connect (Node*, NodeLabelList&);
  void connect (NodeLabelList&, Node*);
  void disconnect (Edge* e) { remove(e); }
  CFG::Node* node_from_label (StmtLabel);

<CFG/RIFG.h> // Representation Independent Flowgraph Interface
        virtual unsigned int HighWaterMarkNodeId()=0;  // largest node id in the graph
        virtual int IsValid(RIFGNodeId n)=0;      // is the node id still valid, or has
        virtual int GetFanin(TarjanIntervals *, RIFGNodeId)=0;
        virtual RIFGNodeId GetRootNode()=0;
        virtual RIFGNodeId GetFirstNode()=0;
        virtual RIFGNodeId GetLastNode()=0;
        virtual RIFGNodeId GetNextNode(RIFGNodeId n)=0;
        virtual RIFGNodeId GetPrevNode(RIFGNodeId n)=0;
        virtual RIFGNodeId GetEdgeSrc(RIFGEdgeId e)=0;
        virtual RIFGNodeId GetEdgeSink(RIFGEdgeId e)=0;
        virtual RIFGNodeId *GetTopologicalMap(TarjanIntervals *)=0;
        virtual RIFGNode *GetRIFGNode(RIFGNodeId n)=0;
```

```
        virtual RIFGEdge *GetRIFGEdge(RIFGEdgeId e)=0;
        virtual RIFGEdgeIterator *GetEdgeIterator(RIFG &fg, RIFGNodeId n, EdgeDirection ed)=0;
        virtual RIFGNodeIterator *GetNodeIterator(RIFG &fg, ForwardBackward fb)=0;


<CFG/OARIFG.h>
  unsigned int HighWaterMarkNodeId();  // largest node id in the graph
  int IsValid(RIFGNodeId n);      // is the node id still valid, or has it been freed
  int GetFanin(TarjanIntervals *, RIFGNodeId);
  RIFGNodeId GetRootNode();
  RIFGNodeId GetFirstNode();
  RIFGNodeId GetLastNode();
  RIFGNodeId GetNextNode(RIFGNodeId n);
  RIFGNodeId GetPrevNode(RIFGNodeId n);
  RIFGNodeId GetEdgeSrc(RIFGEdgeId e);
  RIFGNodeId GetEdgeSink(RIFGEdgeId e);
  RIFGNodeId *GetTopologicalMap(TarjanIntervals *);
  RIFGNode *GetRIFGNode(RIFGNodeId n);
  RIFGEdge *GetRIFGEdge(RIFGEdgeId e);
  RIFGEdgeIterator *GetEdgeIterator(RIFG &fg, RIFGNodeId n, RIFG::EdgeDirection ed);
  RIFGNodeIterator *GetNodeIterator(RIFG &fg, RIFG::ForwardBackward fb);
```

(h) Pointer Analysis

```
<SteensgaardPtrAnal.h>
  void output(std::ostream& out) { Impl::output(out); }

<steensgaard.h>
   ECR * union_with(ECR *that)
   ECR* get_ecr()  { return find_group(); }
   ECR* get_type()
   void set_type(ECR *that)
   std::list<ECR*>& get_pending() {return find_group()->pending;}
   Lambda* get_lambda() { return lambda; }
   void set_lambda(Lambda* l) { lambda = l; }


<PtrAnal.h>
  void operator()( AstInterface& fa,  const AstNodePtr& program);
  bool may_alias(AstInterface& fa, const AstNodePtr& r1, const AstNodePtr& r2);
  VarRef translate_exp(const AstNodePtr& exp) const;
  StmtRef translate_stmt(const AstNodePtr& stmt) const;

  virtual bool may_alias(const std::string& x, const std::string& y) = 0;
  virtual Stmt x_eq_y(const std::string& x, const std::string& y) = 0;
  virtual Stmt x_eq_addr_y(const std::string& x, const std::string& y) = 0;
  virtual Stmt x_eq_deref_y(const std::string& x,
                            const std::string& field,
                            const std::string& y) = 0;
  virtual Stmt x_eq_field_y(const std::string& x,
```

```
                              const std::string& field,
                              const std::string& y) = 0;
  virtual Stmt deref_x_eq_y(const std::string& x,
                              const std::list<std::string>& field,
                              const std::string& y) = 0;
  virtual Stmt field_x_eq_y(const std::string& x,
                              const std::list<std::string>& field,
                              const std::string& y) = 0;
  virtual Stmt allocate_x(const std::string& x) = 0;
  virtual Stmt x_eq_op_y(OpType op, const std::string& x, const std::list<std::string>&
  virtual Stmt funcdef_x(const std::string& x, const std::list<std::string>& params,
                            const std::list<std::string>& ouput) = 0;
  virtual Stmt funccall_x ( const std::string& x, const std::list<std::string>& args,
                              const std::list<std::string>& result)=0;
  virtual Stmt funcexit_x( const std::string& x) = 0;

  virtual void contrl_flow(Stmt stmt1, Stmt stmt2, CFGConfig::EdgeType t) {}
```

(i) Side-Effect Analysis

```
<sideEffect.h>
  //! "Constructor" to return a concrete instance of side effect implementation.
  static SideEffectAnalysis* create();
  //! Perform the side effect analysis on the given project
  virtual int calcSideEffect(SgProject& project) = 0;
  //! Perform the side effect analysis on a file
  virtual int calcSideEffect(SgFile& file) = 0;
  //! Perform the side effect analysis on a node
  virtual int calcSideEffect(SgNode& node) = 0;
  //! Return the list of invoked functions encountered during the analysis.
  virtual list<const char* > getCalledFunctions() = 0;
  //! Return a list of side effects for the given function.
  virtual list<const char* > getGMOD(const char* func) = 0;
  //! Return a list of side effects for the given statement.
  virtual list<const char* > getDMOD(const char* stmt) = 0;
  //! Return the identifier associated with this node and to be passed to getDMOD
  virtual string getNodeIdentifier(SgNode *node) = 0;

//! Utility function to return the fully qualified name of a function given a function
string getQualifiedFunctionName(SgFunctionCallExp *astNode);
//! Utility function to return the fully qualified name of a function given a fucntion
string getQualifiedFunctionName(SgFunctionDeclaration *astNode);
```

(j) Value Propagation Analysis

```
<ValuePropagate.h>
  void build( AstInterface& fa, const AstNodePtr& head,
                  ReachingDefinitionAnalysis& r,
                  AliasAnalysisInterface& alias,
                  FunctionSideEffectInterface* f = 0);
  void build (AstInterface& fa, const AstNodePtr& head,
```

```
                        AliasAnalysisInterface& alias,
                        FunctionSideEffectInterface* f = 0);
   bool known_value( const AstNodePtr& exp,
                       HasValueDescriptor* result = 0, bool *change = 0);
   HasValueMap& get_value_map() { return valmap; }
```

(k) Static Interprocedural Slicing (replaces Procedural Slicing)

(l) Liveness Analysis

(m) Dependence Analysis

```
    //! Perform dependence analysis on a function
    //! Return a dependence graph
    DependenceGraph doDependenceAnalysis (SgFunctionDeclaration* func);

    //! The details of DependenceGraph need to be discussed
    /* The graph may be based on the graph support in ROSE
        Essential information should include:
     A node:
         SgInitializedName* var;    // the accessed variable
         SgExpression*       varRef; // the original variable reference expression
         AccessType          aType;  // read or write access
         std::vector<edge*> edges;  // associated in/out edges for this node
     An edge:
         Node* src;       // source (i) of the dependence
         Node* sink;       // sink (j) of the dependence
         DependenceType dType; // true, anti or output dependence
         SgStatement* carryLoop; // Which level of loop carries this dependence
         DependenceDirection direction; // < access i happens before j in a loop
                                        // = access i and j happen in the same iteration
                                        // > access i happens before j in a loop
         size_t distance;      // dependence distance:
                               // e.g. for () { b[i-1] -> b [i]; }
                               //   distance = i - (i-1) = 1
    */
```

Use (TP & DQ) The use of the data structures built to some forms of analysis (e.g. call graph) can be used to support subsequent forms of analysis that operate on the generated data structures. We separate out the API that is specific for source code and binary executable.

1. Source (TP)

(a) Procedural Slicing (might be the old version; not used)

```
<Slicing.h>
   /*!
     \brief Interface 1:
     Performs a complete slice, that is slices the input file and produces a compilable output fil
   */
   static void completeSlice(SgProject* sgproject);
```

```
  /*!
    \brief
    Interface 2: This function performs the same slicing as sliceOnlyStmts, however in
  */
  static void sliceOnlyStmtWithControl(SgProject* sgproject, set<SgNode*>& stmt);

  /*!
    \brief
    Interface 3:
    This function finds only the statements that directly affect the slicing criterion.
    gives the same statements as the definition use associations gives. The protected f
  */
  static void sliceOnlyStmts(SgProject* sgproject,set<SgNode*>&  stmt_in_slice);
```

(b) Static Interprocedural Slicing (replaces Procedural Slicing)

```
<ControlFlowGraph.h>
  //! The constructor for ControlFlowGraph. Builds a CFG rooted at head
  ControlFlowGraph(SgNode * head);
  //! from a given CFGImpl node, create one (or more) ControlNodes
  void createNode(CFGNodeImpl * node);
  //! return the number of nodes in the CFG
  int getSize() {return _numNodes;}
  //! given a node id (and which numbering scheme to use), return the appropriate contr
  ControlNode * getNode(int id, ID_dir dir) {return (dir == FORWARD)?_forIndex[id]:_bac
  //! dump the contents of the original CFGImpl to a dot file
  void outputCFGImpl();

<CreateSlice.h>
    CreateSlice(std::set < SgNode * >saveNodes):_toSave(saveNodes)
    // bool traverse(SgNode * node) {return traverse(node, false);}
    bool traverse(SgNode * node)

<CreateSliceSet.h>
    CreateSliceSet(SystemDependenceGraph *program,std::list<SgNode*> targets);
    std::set<SgNode*> computeSliceSet();
    std::set<SgNode*> computeSliceSet(SgNode * node);

<DefUseExtension.h>
namespace DUVariableAnalysisExt
{
        SgNode * getNextParentInterstingNode(SgNode* node);
        bool isDef(SgNode * node);
        bool isDef(SgNode * node,bool treadFunctionCallAsDef);
        bool isIDef(SgNode * node);
        bool isIUse(SgNode* node);
        bool test(SgNode* node);
        bool isUse(SgNode * node);
        bool isAssignmentExpr(SgNode*node);
        bool isFunctionParameter(SgNode*node);
        bool isPointerType(SgVarRefExp * ref);
```

```
        bool isComposedType(SgVarRefExp * ref);
        bool isMemberVar(SgVarRefExp * ref);
        bool functionUsesAddressOf(SgVarRefExp * node,SgFunctionCallExp * call);
}

<DependenceGraph.h>
class DependenceGraph {
  void debugCoutNodeList()
  const char *getEdgeName(EdgeType type);
  DependenceNode *createNode(DependenceNode::NodeType type,SgNode * identifyingNode);
  DependenceNode *createNode(SgNode * node);
  void deleteNode(DependenceNode * node);
  DependenceNode *getNode(SgNode * node);
  // (NodeType type, SgNode * node = NULL, std::string depName= "")
  DependenceNode *getNode(DependenceNode::NodeType type,SgNode * identifyingNode);
  DependenceNode * getExistingNode(SgNode * node);
  DependenceNode * getExistingNode(DependenceNode::NodeType type,SgNode * identifyingNode);
  // ! return the InterproceduralInfo object associated with the
  // DependenceGraph
  InterproceduralInfo *getInterprocedural()
  /* ! \brief create an edge of type e between from and to

     Params: - DependenceNode * from: the source of the edge -
     DependenceNode * to: the sink of the edge - EdgeType e: the type of the
     edge

     Side effects: Inserts the Edge (from, to) into the set associated with
     e by _edgetype_map. Inserts e into the set associated with Edge(from,
     to) by _edge_map.

  */
  virtual void establishEdge(DependenceNode * from, DependenceNode * to, EdgeType e=CONTROL);
  virtual void removeEdge(DependenceNode * from, DependenceNode * to, EdgeType e=CONTROL);
  /* ! \brief determine if there is an edge of type e between from and to
     Params: - DependenceNode * from: the source of the edge -
     DependenceNode * to: the sink of the edge - EdgeType e: the type of the
     edge
     Return: true if e is in the set associated with Edge(from, to) by
     _edge_map. */
  bool edgeExists(DependenceNode * from, DependenceNode * to, EdgeType e);
  bool hasOutgingEdge(DependenceNode * src,EdgeType compare);

  /* ! \brief returns all edges between from and to
     Params: - DependenceNode * from: the source of the edge -
     DependenceNode * to: the sink of the edge
     Return: the set of EdgeTypes associated with Edge(from, to) by
     _edge_map.

  */
  std::set < EdgeType > edgeType(DependenceNode * from, DependenceNode * to);
```

```
  // ! writes a dot file representing this dependence graph to filename
  virtual void writeDot(char *filename);
  bool isLibraryFunction(SgFunctionDeclaration * sgFD) const
}

class ControlDependenceGraph:public DependenceGraph {
 public:
  /* ! \brief Contstructor for ControlDependenceGraph
     Params: - SgNode * head: The root of the AST that you want to build the
     CDG for - InterproceduralInfo * ii: the InterproceduralInfo object for
     storing interprocedural information
     Side effects: - initializes _interprocedural
     If ii is NULL, we assume that we are not doing interprocedural
     analysis. Otherwise, we assume that ii is a newly allocated (but not
     yet initialized) object. */
  ControlDependenceGraph(SgFunctionDefinition * head, InterproceduralInfo * ii = NULL);
  void computeInterproceduralInformation(InterproceduralInfo * ii);
  void computeAdditionalFunctioncallDepencencies();

}

class DataDependenceGraph:public DependenceGraph
{
 public:
  /* ! \brief Contstructor for DataDependenceGraph
     Params: - SgNode * head: The root of the AST that you want to build the
     DDG for - InterproceduralInfo * ii: the InterproceduralInfo object for
     storing interprocedural information
     Side effects: - adds data dependence edges to nodes from
     _interprocedural
     If ii is NULL, we assume that we are not doing interprocedural
     analysis. Otherwise, we assume that ii is an InterproceduralInfo object
     that has been initialized by the CDG for the same procedure */
#ifdef NEWDU
  DataDependenceGraph(SgNode * head,EDefUse * du, InterproceduralInfo * ii = NULL);
#else
  DataDependenceGraph(SgNode * head, InterproceduralInfo * ii = NULL);
#endif
  void computeInterproceduralInformation(InterproceduralInfo * ii);
}

class MergedDependenceGraph:public DependenceGraph
{
 public:
  /* ! \brief creates a new dependence node that reflects the argument (not
     a direct copy)
     Params: - DependenceNode * node: The node we want to make a "copy" of
     Return: If we've already "copied" the node, return the existing
     DependenceNode. Otherwise create a new one.
     Side effects: calls createNode appropriately to perform "copies," so
```

```
       _sgnode_map or _depend_map may be updated.
       If the node we are adding is an interprocedural node, we want to copy
       the _interproc pointer, not node itself. If it's an SgNode, we want to
       build the DependenceNode around that, as opposed to node. If it's
       neither, we just copy the argument. */
    DependenceNode * _importNode(DependenceNode * node);
    /* ! \brief creates a backward slice starting from node
       Params: - SgNode * node: the slicing criterion
       Return: returns a set of SgNodes which belong in the slice with slicing
       criterion node.
       This function calls getSlice, and prunes the returned values to find
       just the SgNodes. */
    std::set < SgNode * >slice(SgNode * node);
    /* ! \brief creates a backward slice starting from node
       Params: - DependenceNode * node: the slicing criterion
       Return: returns a set of DependenceNodes which belong in the slice with
       slicing criterion node.
       This is a more general version of slice, which operates on any
       DependenceNode. */
    virtual std::set < DependenceNode * >getSlice(DependenceNode * node) = 0;
}

class FunctionDependenceGraph:public MergedDependenceGraph
{
  public:
    /* ! \brief Constructor for FunctionDependenceGraph, initialized with the
       CDG and DDG for the function.
       Params: - ControlDependenceGraph * cdg: a previously built CDG for the
       function - DataDependenceGraph * ddg: a previously build DDG for the
       function - InterproceduralInfo * ii: If NULL, we aren't doing
       interprocedural. Otherwise, the fully initialized InterproceduralInfo
       object for the function.
    */
    FunctionDependenceGraph(ControlDependenceGraph * cdg, DataDependenceGraph * ddg,
                            InterproceduralInfo * ii = NULL);
    /* ! \brief gets a slice with slicing criterion node
       This simply does a backwards reachability across all edges to produce
       the slice. */
    virtual std::set < DependenceNode * >getSlice(DependenceNode * node);
}

class SystemDependenceGraph:public MergedDependenceGraph
{
  public:
    void addLibraryExtender(SDGLibraryExtender * le)
    SystemDependenceGraph(){debug=false;}
    SgNode *getMainFunction();
    void createSafeConfiguration(SgFunctionDeclaration *fDef);
    bool isKnownLibraryFunction(SgFunctionDeclaration *fDec);
    void createConnectionsForLibaryFunction(SgFunctionDeclaration *fDec);
```

```
  void parseProject(SgProject *project);

  /*! once all functions have been added to the SystemDependenceGraph this function perf
  void performInterproceduralAnalysis();
  void computeSummaryEdges();
  void cleanUp(std::set<SgNode*> preserve);

  /* ! \brief adds a PDG to our SDG

     Params: - FunctionDependenceGraph * pdg: The PDG to add to the SDG

     Side effects: Merges PDG in using _mergeGraph. Maps function PDG
     represents to the PDG itself in _funcs_map. */
  void addFunction(FunctionDependenceGraph * pdg);
  void createFunctionStub(InterproceduralInfo * info);

  void addFunction(ControlDependenceGraph * cdg, DataDependenceGraph * ddg);
  InterproceduralInfo * getInterproceduralInformation(SgFunctionDeclaration * dec)
  void addInterproceduralInformation(InterproceduralInfo * info)
  void doInterproceduralConnections(InterproceduralInfo * ii);


  /* ! \brief links all the functions together

     After the PDGs have been merged into the SDG, each call site is linked
     to the PDG associated with the function that it calls: - The callsite
     node is linked to the entry node with a "call" edge - Each actual-in
     node is linked to the formal-in node with a "call" edge - Each
     formal-out node is linked to the actual-out node with a "return" edge */
  void process();

  /* ! \brief performs a backwards slice with slicing criterion node

     getSlice is defined according to the paper by Horowitz et al. as a two
     phase operation. The first operation does backwards reachability to
     "mark" nodes while not traversing return edges. Thus it ignores functin
     calls. The second phase does backwards reachability from all marked
     nodes while not traversing call edges. Thus it ignores calling
     functions. The final set of reachable nodes is the interprocedural
     slice. */
  virtual std::set < DependenceNode * >getSlice(DependenceNode * node);

  /* ! \brief retrieve the PDGs in the graph

     Returns: a set of FunctionDependenceGraph that comprise the
     SystemDependenceGraph */
  std::set < FunctionDependenceGraph * >getPDGs();
}

<EDefUse.h>
```

```
class EDefUse
{
public:
  EDefUse(SgProject * proj);
      int run(bool debug);
      // get the vector of defining and usage nodes for a specific node and a initializedName
      std::vector < SgNode* > getDefFor(SgNode* node, SgInitializedName* initName);
      std::vector < SgNode* > getUseFor(SgNode* node, SgInitializedName* initName);
      std::vector < std::pair < SgInitializedName* , SgNode* > >
getDefMultiMapFor(SgNode* node);
      // return whether a node is a global node
      bool isNodeGlobalVariable(SgInitializedName* node);
};

<InterproceduralInfo.h>
class InterproceduralInfo
{
        public:
        static SgNode * identifyEntryNode(SgFunctionDeclaration * dec)
        static  SgNode * identifyEntryNode(SgFunctionDefinition * def)
        // ! the callsite - one per SgFunctionCallExp
    void setCallInterestingNode(int id,SgNode * node)
    SgNode * getCallInterestingNode(int id)
        SgNode * getActualReturn(int id)
        SgNode * getActualIn(int id,int varNr)
        int getActualInCount(int id)
        void addActualIn(int id,SgExpression * node)
        void setSliceImportantNode(int id,SgNode * node)
        void setActualReturn(int id,SgNode * node)
        //! returns the node for the function call, which contains the function call
        SgNode * getSliceImportantFunctionCallNode(int i)
        std::set<SgNode *> getExitNodes()
        void addParameterToFunctionCall(SgNode * functionCall,SgExpression * param)
        int callSiteCount()
        SgNode * getFunctionCallExpNode(int i)
        SgNode * getFunctionEntry()
        void setEllipse(SgNode * formal)
        SgNode* getEllipse()
        bool isUndefined()
        int getFormalCount()
        SgNode * getFormal(int nr)
        void setFormalReturn(SgNode *  node)
        SgNode * getFormalReturn()
        // add this DependenceNode to the list of nodes which lead to exiting this function
        void addExitNode(SgNode * node)
        InterproceduralInfo(SgFunctionDeclaration* functionDeclaration)

        /* ! \brief Gets the function declaration that the InterproceduralInfo object is for.
           Returns: The SgFunctionDeclaration node that is associated with this object */
        SgFunctionDeclaration * foo(){return decl;}
```

```
        SgFunctionDefinition * getFunctionDefinition()
    SgFunctionDeclaration * getFunctionDeclaration()
  int addFunctionCall(SgNode * functionCall)

};

<SimpleDirectedGraph.h>
  //! get all the nodes in the graph
  std::set<SimpleDirectedGraphNode *> getNodes() {return _nodes;}
  //! Add a node to the graph
  virtual void addNode(SimpleDirectedGraphNode * node)
  //! Add a link to the graph between "from" and to "to"
  virtual void addLink(SimpleDirectedGraphNode * from, SimpleDirectedGraphNode * to)
  //! Check if a node containing data is in the graph
  bool nodeExists(SimpleDirectedGraphNode * node)
  //! Check if a dependence is in the graph
  bool linkExists(SimpleDirectedGraphNode * from, SimpleDirectedGraphNode * to)
  void printGraph()
  virtual void writeDot(char * filename)
  std::set<SimpleDirectedGraphNode *> getReachable(SimpleDirectedGraphNode * start, Tra
}

<SlicingInfo.h>
    // ! Returns the SgFunctionDeclaration that we are targeting
    SgFunctionDeclaration *getTargetFunction()
    // ! Returns the statements that are part of the slicing criterion
    SgNode *getSlicingCriterion()
    std::list < SgNode * >getSlicingTargets()
```

(c) Liveness Analysis

```
<LivenessAnalysis.h>
 LivenessAnalysis(bool debug, DefUseAnalysis* dfa_p)
  SgFunctionDefinition* getFunction(SgNode* node);
  int getNumberOfNodesVisited();
  // Run liveness analysis on the entire project
  //bool run(bool debug=false);
  // Run liveness analysis for a single function
  FilteredCFGNode < IsDFAFilter > run(SgFunctionDefinition* function, bool& abortme);
  std::vector<SgInitializedName*> getIn(SgNode* sgNode) { return in[sgNode];}
  std::vector<SgInitializedName*> getOut(SgNode* sgNode) { return out[sgNode];}
  int getVisited(SgNode* n) {return visited[n];}
  void setIn(SgNode* sgNode, std::vector<SgInitializedName*> vec) { in[sgNode]= vec;}
  void setOut(SgNode* sgNode, std::vector<SgInitializedName*> vec ) { out[sgNode]=vec;}
  // used by ASTTraversals
  template <class T> T merge_no_dups( T& v1, T& v2);
  void fixupStatementsINOUT(SgFunctionDefinition* funcDecl);
```

(d) Dependence Analysis

(e) AST Interpreter (Interpretation of Concrete Semantics using AST)

```
<interp_core.h>

namespace Interp {
/* Search for a global function in all translation units.
   Can be used to search for the "main" function. */
SgFunctionSymbol *prjFindGlobalFunction(const SgProject *prj, const SgName &fnName);

class Value {
      public:
      /*! These functions return concrete representations where possible.
          They could also be used to implement casting. */
      virtual bool getConcreteValueBool() const;
      virtual char getConcreteValueChar() const;
      virtual double getConcreteValueDouble() const;
      virtual float getConcreteValueFloat() const;
      virtual int getConcreteValueInt() const;
      virtual long double getConcreteValueLongDouble() const;
      virtual long int getConcreteValueLong() const;
      virtual long long int getConcreteValueLongLong() const;
      virtual short getConcreteValueShort() const;
      virtual unsigned char getConcreteValueUnsignedChar() const;
      virtual unsigned int getConcreteValueUnsignedInt() const;
      virtual unsigned long long int getConcreteValueUnsignedLongLong() const;
      virtual unsigned long getConcreteValueUnsignedLong() const;
      virtual unsigned short getConcreteValueUnsignedShort() const;
};

typedef boost::shared_ptr<Value> ValueP;

template <typename PrimType>
class IntegralPrimTypeValue : public GenericPrimTypeValue<PrimType>
   {
      public:
      IntegralPrimTypeValue(Position pos, StackFrameP owner)
              : GenericPrimTypeValue<PrimType>(pos, owner) {}
      IntegralPrimTypeValue(PrimType v, Position pos, StackFrameP owner)
              : GenericPrimTypeValue<PrimType>(v, pos, owner) {}
   }

/* FloatingPointPrimTypeValue - likewise */

typedef IntegralPrimTypeValue<bool> BoolValue;
typedef IntegralPrimTypeValue<char> CharValue;
typedef FloatingPointPrimTypeValue<double> DoubleValue;
typedef FloatingPointPrimTypeValue<float> FloatValue;
typedef IntegralPrimTypeValue<int> IntValue;
typedef FloatingPointPrimTypeValue<long double> LongDoubleValue;
typedef IntegralPrimTypeValue<long int> LongIntValue;
typedef IntegralPrimTypeValue<long long int> LongLongIntValue;
typedef IntegralPrimTypeValue<short> ShortValue;
```

```
typedef IntegralPrimTypeValue<unsigned char> UnsignedCharValue;
typedef IntegralPrimTypeValue<unsigned int> UnsignedIntValue;
typedef IntegralPrimTypeValue<unsigned long long int> UnsignedLongLongIntValue;
typedef IntegralPrimTypeValue<unsigned long> UnsignedLongValue;
typedef IntegralPrimTypeValue<unsigned short> UnsignedShortValue;

class Interpretation {
      /* Create an empty interpretation. */
      Interpretation();
};

class StackFrame {
      public:
      /* Create a stack frame within the given interpretation
         to interpret the given function. */
      StackFrame(Interpretation *currentInterp, SgFunctionSymbol *funSym);

      /* Initialize global variables over the given project.
         To be called before interpretation of the "main" function. */
      void initializeGlobals(SgProject *project);

      /* Interprets this stack frame's function with the given arguments.
         Returns the return value of the function. */
      ValueP interpFunction(const std::vector<ValueP> &args);

};

}
```

2. Binary (DQ)

   (a) InstructionSemantics
   See doxygen documentation for the Rose::BinaryAnalysis::InstructionSemantics2 namespace.

   (b) Library Identification (FLIRT)
   This has not yet been moved from `developersScratchSpace/Dan/libraryIdentification_tests` to a more useful location in ROSE. Likely should be put in the *midend* as part of *analysis*.

   (c) Analysis of the Binary File Format
   This has not yet been moved from `developersScratchSpace/Dan/astEquivalence_tests` to a more useful location in ROSE. Likely should be put in the *midend* as part of *analysis*.

   (d) Dynamic Analysis of Instruction Execution
   There are three interfaces:

      i. Ptrace: Uses the Unix ptrace() system call to trace over the network a binary executing on a remote linux machine.
      ii. Pin: Traces an executable running on the same machine as ROSE.
      iii. Ether: Traces an executable running in Windows XP on a Xen virtual machine on the same hardware as ROSE.

(e) General Dynamic Analysis
Uses Ether/Xen to execute the specimen in Windows XP on a virtual machine and adds new SgAsmGenericSections containing disassembled instructions to the AST as those areas are discovered. Can then unparse the AST to a new executable. *This is work in progress.*

(f) Detecting unreferenced regions
Finding what regions of the binary were never referenced during parsing. The binary I/O utilities keep track of everything that was read during parsing and this information is available through an ExtentMap (see utilities below). The inverse of the ExtentMap will show what hasn't been referenced.
See doxygen for ExtentMap class.

(g) Basic block detection
Organize instructions into basic blocks. The Partitioner class is reponsible for taking a set of instructions and deciding which instructions belong together in a basic block. This analysis is normally called automatically by ROSE as part of its disassembly procedure, but it's also useful to call this explicitly (especially if you also called the Disassembler explicitly, since the disassemlber doesn't actually put things into basic blocks). Fully documented in doxygen. See Partitioner class.
See src/frontend/Disassembler/Partitioner.h

(h) Function boundary detection
Organize basic blocks into functions. The Partitioner class is responsible for taking a set of basic blocks and figuring out how to organize them into functions. It can look at other parts of a binary AST (like symbol tables), is fully configurable, and can be specialized by derivation. See doxygen for Partitioner class.
See src/frontend/Disassembler/Partitioner.h

Transformation Interm proposed namespace name: *ROSE_Transformation*
Transformation by definition modifies the structure of the AST and can be used to define instrumentation, optimizations, and custom translation.

Source (L)

1. Instrumentation

FIXME: *Do u*
section at all? Can be
general t

```

/* Instrument(Add a statement, often a function call) into a function right
   before the return points, handle multiple return statements and return
   expressions with side effects. Return the number of statements inserted.
 */
int instrumentEndOfFunction (SgFunctionDeclaration *func, SgStatement *s);

/* Instrument(Add a statement, often a function call) into a function at
   the very beginning.  */
int instrumentBeginOfFunction (SgFunctionDeclaration *func, SgStatement *s);
```

2. Optimization These include a range of optimizations relevant for general performance optimization of scientific applications.

(a) Inlining

```
/* Inline a function call site*/
bool inlining (SgFunctionCallExp *);
```

```

/* Inline all call sites to a function within a root AST node,
   return the number of call sites being inlined. */
int inlining (SgNode* root, SgFunctionDeclartion *);

/* Inline all call sites to a function with a qualified name,
   return the number of call sites being inlined. */
int inlining (SgNode* root, const std::string &qualified_func_name);

/*Aggressive inline all call sites whenever possible for an AST,
  return the number of call sites being inlined. */
int inlining(SgNode* root);
```

(b) Loop optimizations:fusion, fusion, unrolling, blocking, loop interchange, array copy, etc. Most API functions take SgNode* instead of SgForStatement* to be compatible for both C and Fortran loops.

```

 //! Normalize a loop, return true if successful
 //! The loop can be either C for loop or Fortran DO loop
 bool loopNormalization (SgNode* loop);

/* Check if a for-loop has a canonical form, return loop index,
   bounds, step, body and so on if requested. */
 bool isCanonicalForLoop (SgNode *loop, SgInitializedName **ivar=NULL,
      SgExpression **lb=NULL, SgExpression **ub=NULL, SgExpression **step=NULL,
      SgStatement **body=NULL, bool *hasIncrementalIterationSpace=NULL,
      bool *isInclusiveUpperBound=NULL);

 /* Unroll a target loop with a specified unrolling factor.
    It handles steps larger than 1 and adds a fringe loop
    if the iteration count is not evenly divisible by the unrolling factor. */
 bool loopUnrolling (SgNode *loop, size_t unrolling_factor);

 //! Unroll and jam a loop, with a given unrolling factor
 bool loopUnrollAndJam (SgNode *loop, size_t unrolling_factor);

 /*   Tile the n-level (starting from 1) loop of a perfectly nested loop nest
      using tiling size s. */
 bool loopTiling (SgNode *loopNest, size_t targetLevel, size_t tileSize);

 /* Interchange/permutate a n-level perfectly-nested loop rooted at 'loop'
    using a lexicographical order number within (0,depth!). */
 bool loopInterchange (SgNode*loop, size_t depth, size_t lexicoOrder);

 /*  Normalize loop init stmt by promoting the single variable declaration statem
     outside of the for loop header's init statement, e.g. for (int i=0;) becomes
     int i_x; for (i_x=0;..) and rewrite the loop with the new index variable,
     if necessary. */
 bool normalizeForLoopInitDeclaration (SgForStatement *loop);

 //! Fuse two loops into one loop, return the fused loop
```

```
        SgNode* loopFusion (SgNode* loop1, SgNode* loop2);

        /* Loop fission: break a loop into multiple loops */
        std::vector<std::SgNode*> loopFission (SgNode* src_loop);

// TODO array copy
```

(c) Constant Folding

```
         /* Constant folding an AST subtree rooted at 'root'.
           Avoid folding floating point typed expressions
           by default to ensure accuracy. */
          void constantFolding (SgNode* root, bool foldFloatPoint = false);
```

(d) Finite Differencing

FIXME: *w*

```
        //! Do finite differencing on one expression within one context.
         void doFiniteDifferencingOne(SgExpression* e,
                                     SgBasicBlock* root, RewriteRule* rules);
```

(e) Partial Redundancy Elimination

```
        /* Apply partial redundancy elimination on AST rooted at r*/
        void partialRedundancyElimination(SgNode* r);
```

3. General Transformations
   These include outlining, AST copy, and AST merge support, etc.

(a) Outlining

FIXME: *Dis internal flags an*

```
        //! Accept a set of command line options to set internal behaviors
        void commandLineProcessing(std::vector<std::string> &argvList);

        //! Returns true iff the statement is "outlineable."
        //! Print out reasons for s that can not be outlined if 'verbose' is true
        bool isOutlineable (const SgStatement* s, bool verbose = false);

        //! Preprocess a statement for outlining
        SgBasicBlock* preprocess (SgStatement* s);

         //! Outlines the given basic block into a function named 'name'
         Result outlineBlock (SgBasicBlock* b, const std::string& name);

        //! Outline to a new function with the specified name, calling
        //! preprocessing internally
        Result outline (SgStatement* s, const std::string& func_name);

        //! Stores the main results of an outlining transformation.
        struct Result
        {
          //! The outlined function's declaration and definition.
          SgFunctionDeclaration* decl_;

```

```
    //! A call statement to invoke the outlined function.
    SgStatement* call_;

    //! A SgFile pointer to the newly generated source file containing the
    // outlined function if -rose:outline:new_file is specified (useNewFile==true)
    SgFile* file_;
  }
```

(b) ImplicitCodeGeneration

This work makes C++ implicit semantics explicit for C style analysis.

E: The granularity for
nation targets? flag to
e headers, other files?

```
  //! Make implicit compiler-generated function explicit,
  //  including default constructors, destructors and copy constructors.
  void defaultFunctionGenerator(SgProject *prj);

  //! The same as the above, except that it operates on the file scope
  void defaultFunctionGenerator(SgFile* f);

  //! The same as the above, except that it operates on a target class
  void defaultFunctionGenerator(SgClassDeclaration* c_decl);

  //!Annotates the AST with calls to class destructors whenever objects
  go out of scope.
  void destructorCallAnnotator(SgProject *prj);

  //! Transforms the evaluation of short-circuited expressions to explicitly
  //! evaluate each step. A prerequisite of destructorCallAnnotator.
  void shortCircuitingTransformation(SgProject *prj);
```

(c) FunctionCallNormalization

This is a library of function call normalizations to support binary analysis.

```
  //! Ensure that no statement will have more than one function call
  //! This is to be done by inserting new temporary variables to replace extra cal
   void functionCallNormalization(SgNode* root);
```

(d) AST creation/building

Build AST pieces, transparently taking care of side effects as much as possible. Used to replace the direct call to constructors.

```
  //! Type builders
  SgTypeBool * buildBoolType ();
  SgTypeInt * buildIntType ();
  ...
  //! Expression builders
  SgNullExpression *buildNullExpression ();
  SgIntVal *  buildIntVal (int value=0);
  SgVarRefExp * buildVarRefExp (const std::string &varName,
                                SgScopeStatement *scope=NULL);
  ...
  //! Statement builders
  SgVariableDeclaration * buildVariableDeclaration (const SgName &name,
```

```
                                 SgType *type, SgInitializer *varInit=NULL,
                                          SgScopeStatement *scope=NULL);
      SgWhileStmt * buildWhileStmt (SgStatement *condition, SgStatement *body);
      SgClassDeclaration * buildStructDeclaration (const std::string &name,
                                     SgScopeStatement *scope=NULL);
      ...

      //! Misc builders
      SgFile * buildFile (const std::string &inputFileName,
         const std::string &outputFileName, SgProject *project=NULL);
      SgInitializedName * buildInitializedName (const std::string &name,
                                             SgType *type);
      ...
```

(e) AST insert, deletion, move and replacement
Manipulate AST pieces

```
      PreprocessingInfo * insertHeader (const std::string &filename,
          PreprocessingInfo::RelativePositionType position=PreprocessingInfo::after,
          bool isSystemHeader=false, SgScopeStatement *scope=NULL);
      void insertStatement (SgStatement *targetStmt, SgStatement *newStmt,
           bool insertBefore=true);
      void insertStatementList (SgStatement *targetStmt,
           const std::vector< SgStatement * > &newStmts, bool insertBefore=true);
      void insertStatementBefore (SgStatement *targetStmt, SgStatement *newStmt);
      void insertStatementAfter (SgStatement *targetStmt, SgStatement *newStmt);
      void appendStatement (SgStatement *stmt, SgScopeStatement *scope=NULL);
      void prependStatement (SgStatement *stmt, SgScopeStatement *scope=NULL);
      ...
      void deepDelete (SgNode *root);
      void deleteAST (SgNode *node);
      void removeStatement (SgStatement *stmt);

      ...
      void replaceExpression (SgExpression *oldExp, SgExpression *newExp,
           bool keepOldExp=false);
      void replaceStatement (SgStatement *oldStmt, SgStatement *newStmt,
           bool movePreprocessinInfo=false);

      void moveStatementsBetweenBlocks (SgBasicBlock *sourceBlock,
           SgBasicBlock *targetBlock);
```

(f) AST Copy support
This support permits arbitrary subtrees (or the whole AST) to be copied with control over deep or shallow copying via a single function.

```
        SgNode * deepCopyNode (const SgNode *subtree);
        SgNode * shallowCopyNode (const SgNode *subtree);
        template<typename NodeType>
          NodeType * deepCopy (const NodeType *subtree);
        SgExpression * copyExpression (SgExpression *e);
```

```
        SgStatement * copyStatement (SgStatement *s);
```

(g) AST Merge support
This work permits the merging of separate AST's and the sharing of their identically names language declarations to support whole program analysis. Duplicate parts of the merged AST are deleted.

```
        //! Merge AST tree from multiple files
        void mergeAST( SgProject* project );

        //! Merge AST from mulitple AST binary dump files
        SgProject * mergeAST(std::vector <FILE *> files);
```

Binary (DQ)

1. Static Binary Rewriting
A restricted set of transformations are possible on a binary executable. Existing work supports moving and/or resizing a section. We don't handle all cases since this is incredibly complicated.
See **SgAsmGenericFile::shift_extend()**.

Backend (DQ)

Interm proposed namespace name: *ROSE_Backend*
The backend part of ROSE generates the code and produces a final executable (just like any other compiler). Users don't typically work on any aspect of the backend; thus is has a simple API.

1. **int backend (SgProject* project, UnparseFormatHelp *unparseFormatHelp = NULL, UnparseDelegate* unparseDelagate = NULL);**
This function generates source code from the AST and calls the backend compiler. The integer error code from the backend compiler is returned. UnparseFormatHelp permits limited control over the formatting of the generated source code. UnparseDelegate is currently ignored. For binaries, this function generates an assembler listing with section information and a reassembled binary executable.
2. **int backendUsingOriginalInputFile (SgProject* project);**
This is useful as a test code for testing ROSE for use on projects that target Compass or any other analysis only tool using ROSE. Called in tests/nonsmoke/functional/testAnalysis.C for example.
3. Assembler
Generating machine code from SgAsmInstruction nodes. The Assembler class is reponsible for taking SgAsmInstruction nodes and generating machine code, placing the result in a buffer. The assembler can be specialized by derivation.

   Note: currently we only have an x86-64 assembler. The 32-bit needs a bit more work. The x86 assembler is generated automatically from the Intel Instruction Set Reference documentation and is thus substantially smaller than the x86 disassembler.
See doxygen Assembler class.
See src/frontend/Disassembler/Assembler.h
4. Control of Assembler
Various aspects of assembly can be controlled through properties of the Assembler object. For instance, should the assembler use smallest possible data encodings or honor the sizes of the instruction operands in the AST; should instruction prefixes be emitted in the same order and cardinality as the original parse, or

in the order recommended by Intel; etc.
See doxygen Assembler class.
See src/frontend/Disassembler/Assembler.h

The ROSE Install Tree (DQ)

The installation of ROSE is automated using **make install** and generates a GNU standard form of package installation. We should decide what we want this to look like, what it should include and what it should exclude.

FIXME: *This is*
that has not been dis

Utility

Interm proposed namespace name: *ROSE_Support*
These features are important to how applications are developed using ROSE.

AST Utility A collection of the utility support in ROSE is specific to the AST and this are presented together:

1. AST Traversal (Yi)
2. AST Query (A)
3. AST Postprocessing (L)
   After transformations to the AST, it is frequently required to call a standard AST fixup that will fill in missing pieces of the AST and do a few simple tests to validate the AST. It can also support bottom-up AST construction by patching up symbols, scope information etc.

```
        //! Fixup missing pieces of the AST
        void AstPostProcessing(SgNode* node);

        // Do we want to expose individual fixup to users?
        // Some examples are given below

        //! Connect variable reference to the right variable symbols when feasible,
        //! return the number of references being fixed.
        int fixVariableReferences (SgNode *root);

        //! Patch up symbol, scope, and parent information when
        //! a declaration's scope is known.
        void fixVariableDeclaration (SgVariableDeclaration *varDecl, SgScopeStatement *s);
        void fixClassDeclaration (SgClassDeclaration *classDecl, SgScopeStatement *s);
        void fixNamespaceDeclaration (SgNamespaceDeclarationStatement *decl, SgScopeStatement *s);
        void fixLabelStatement (SgLabelStatement *label_stmt, SgScopeStatement *s);
```

4. AST Consistency Tests (L)
   This is the highest level of internal consistency testing available in ROSE. This test is typically run after the frontend processing (by the user) to verify a correct AST before midend processing.

```
       //! Run all known AST consistency tests
        AstTests::runAllTests(SgNode* root);

       //! Do we want to expose individual tests?
```

```
        // Derived class vs. a function call
        // Tests on a tree vs. memory pools

         //! Test on memory pools
         AstTests::testUniqueStatementsInScopes();

         //! Test on a whole/sub tree
         AstTests::testUniqueStatementsInScopes(SgNode* root);

         AstTests::testMangledNames();
         AstTests::testMangledNames(SgNode* root);

         AstTests::testCompilerGeneratedNodes();
         AstTests::testCompilerGeneratedNodes(SgNode* root);

         AstTests::testAstCycles();
         AstTests::testTemplates();
         AstTests::testDefiningAndNondefiningDeclarations();
         AstTests::testSymbolTables();
         AstTests::testMemberFunctions();
         AstTests::testExpressionTypes();

         AstTests::testParentPointersInMemoryPool();
         AstTests::testChildPointersInMemoryPool();

         AstTests::testMappingOfDeclarationsToSymbols();
         AstTests::testExpressionLValue();
        //! Test the declarations to make sure that
        //defining and non-defining appear in the same file
         AstTests::testMultipleFiles();
         AstTests::testTypesInMemoryPool();
```

5. AST Visualization (DQ)
   Visualization of the AST and related graphs generated from program analysis forms an approach to both internal debugging and presentation of specific sorts of results.
   (a) **void generatePDF (const SgProject & project);**
       Generates a PDF file from the AST.
   (b) **void generateDOT (const SgProject & project, std::string filenamePostfix);**
       Generates a DOT file representing the AST (information about types and many IR nodes that are considered attributes to AST nodes are not represented). The resulting graph is of the input source code excluding header files. The result is a tree (formally).
   (c) **void generateDOT_withIncludes (const SgProject & project, std::string filenamePostfix);**
       Generates a DOT file representing the AST (information about types and many IR nodes that are considered attributes to AST nodes are not represented). The resulting graph is of the input source code plus all header files (so it can be very large). The result is a tree (formally).
   (d) **void generateDOTforMultipleFile (const SgProject & project, std::string filenamePostfix);**

Generates a DOT file representing the AST (information about types and many IR nodes that are considered attributes to AST nodes are not represented). The resulting graph is of all of the files specified on the command line. The result is a tree (formally).

(e) **void generateAstGraph (const SgProject* project, int maxSize, std::string filenameSuffix);**
Generates a DOT file representing the AST and includes information about types and many IR nodes that are considered attributes to AST nodes are not represented by the other functions above. The resulting graph is of all of the files specified on the command line. The result is general graph (not a tree) (formally).
By Liao, What about
void generateDOTforWholeAST(**const** SgProject∗ project, std::string filenameSuffix, FilterSetting∗ fs).

6. AST File I/O Support (T)
This support permits the AST to be written to a file and read in from a file. It is useful for assembling the AST from a whole application and many other specialized tools.

```
<AST_FILE_IO.h>
    // sets up the lost of pool sizes that contain valid entries
       static void startUp ( SgProject* root );

    // sets up the lost of pool sizes that contain valid entries
       static unsigned long getSizeOfMemoryPool ( const int position );
       static unsigned long getSizeOfMemoryPoolUpToAst ( AstData* astInPool, const int position );
       static unsigned long getAccumulatedPoolSizeOfNewAst( const int position);
       static unsigned long getAccumulatedPoolSizeOfAst( AstData* astInPool, const int position);
       static unsigned long getPoolSizeOfNewAst( const int sgVariant);
       static unsigned long getTotalNumberOfNodesOfAstInMemoryPool ( );
       static unsigned long getTotalNumberOfNodesOfNewAst ();
       static bool areFreepointersContainingGlobalIndices ( );

    // some methods not used so far ... or not more used
       static unsigned long getGlobalIndexFromSgClassPointer ( SgNode* pointer ) ;
       static SgNode* getSgClassPointerFromGlobalIndex ( unsigned long globalIndex) ;
       static void compressAstInMemoryPool() ;
       static void resetValidAstAfterWriting();
       static void clearAllMemoryPools ( );
       static void deleteStaticData( );
       static void deleteStoredAsts( );
       static void setStaticDataOfAst(AstData* astInPool);
       static int getNumberOfAsts ();
       static void addNewAst (AstData* newAst);
       static void extendMemoryPoolsForRebuildingAST ( );
       static void writeASTToStream ( std::ostream& out );
       static void writeASTToFile ( std::string fileName );
       static std::string writeASTToString ();
       static SgProject* readASTFromStream ( std::istream& in );
       static SgProject* readASTFromFile (std::string fileName );
       static SgProject* readASTFromString ( const std::string& s );
       static void printFileMaps () ;
       static void printListOfPoolSizes () ;
```

```
        static void printListOfPoolSizesOfAst (int index) ;
        static AstData* getAst (int index) ;
        static AstData* getAstWithRoot (SgProject* root) ;

        template <class TYPE>
        static void registerAttribute ( );
        static const std::map <std::string, CONSTRUCTOR>& getRegisteredAttributes ();
```

Other useful utility functions in ROSE

1. Performance monitoring (DQ)
   The *TimingPerformance* class defines a simple mechanism used throughout ROSE to report the execution performance of different parts of the compilation process. As a class variables can be generated on the stack (to record the starting time of an execution phase and the destructor for the class will record the elapsed time of the execution phase.

   (a) **TimingPerformance <variable name> (std::string);**
   This constructor builds and starts a timer, the descructor is automatically called the end of the scope and records the elapsed time. The data is saved internally and output in a final report in either of two forms (using cout or to a file).

   (b) **TimingPerformance::generateReport();**
   A report is generated at the end of the execution when either the last *TimingPerformance* destructor is called of when the report function is called explicitly.

   (c) **TimingPerformance::generateReportToFile(project);**
   Write the CSV formatted file of performance data (accumulated over multiple runs) Execute the function to generate the data into the report fine independent of the level of verbosity specified from the command-line (does no output to cout or cerr). This data can be used by a separate program to graph the different times required to run different parts of ROSE on a wide range of files. It is used mostly for debugging complexity issues inside the compiler or in user developed tools using ROSE.

   (d) **TimingPerformance::set_project (SgProject* project);**
   If set, the report will be generated upon calling the destructor for the *TimingPerformance* object.

2. Language specific name support (DQ)
   This section contains the support for generating unique names for language constructs and handling mangled and unmangled names for use in ROSE based tools.

   (a) **virtual SgName SgStatment::get_qualified_name() const;**
   Qualified names provide a more readable for of newarly unique name for constructs. This function is implemented on all SgStatment objects.

   (b) **virtual SgName SgStatment::get_mangled_name() const;**
   Mangled name support so that unique names can be generated.

   (c) **SgName SgDeclarationStatement::generate_alternative_name_for_unnamed_declaration (SgNode* parent) const;**
   Support for name mangling of unnamed classes embedded in SgVariableDeclaration and SgTypedefDeclaration.

   (d) **SgName SgDeclarationStatement::generate_alternative_name_for_unnamed_declaration_in_scope (SgScopeStatement* scope) const;**
   Support for generation of names for unnamed declarations in scopes.

3. Support for comments and CPP directives (Yi)
4. GUI Support (JK & T)

```
<QRButtons.h>
        QRButtons(std::string caption = "");
        QRButtons(QROSE::Orientation orientation, std::string caption = "");
        void addButtons(int numButtons, Type type);
        unsigned numButtons() const;
        QAbstractButton* operator [](int id) const;
        int getId(QAbstractButton *button);
    Type getType(int id) const;
        void setCaption(const char *caption0, ...);
        void setCaption(int id, const char *fmt_caption, ...);
        void setPicture(int id, const char *filename);
    void setPicture(int id, const char *xpm[]);
        void setBtnChecked(int id, bool check);
        bool isBtnChecked(int id) const;
    bool isBtnCheckable(int id) const;
    void setBtnEnabled(int id, bool enable);
    bool isBtnEnabled(int id) const;


<QREdit.h>
    QREdit(Type type = Line, const char *caption = 0);
    void setText(std::string text);
    std::string getText() const;
    void clear();
    void setReadOnly(bool readOnly);
    bool isReadOnly() const;
    void setFont(QFont font);
    QFont getFont() const;
    void setTextColor(QColor color);
    QColor getTextColor() const;
    void setBgColor(QColor color);
    QColor getBgColor() const;


<QRProgress.h>
    QRProgress(bool useLabel = true, bool autoHide = false);
    void set(int totalSteps);
    void set(int currentStep, int totalSteps);
    void tick(int steps = -1);
    void tick(std::string txt, int steps = -1);
    int value() const;
    int maximum() const;
    void reset();


<QRSelect.h>
    QRSelect(Type type = Combo, const char *caption = 0);
```

```
    Type getType() const;
    void setList(const std::list<std::string> &lst, bool append = false);
    std::list<std::string> getList() const;
    void addItem(std::string text);
    void removeItem(int index);
    void clear();
    unsigned count() const;
    void setText(int index, std::string text);
    std::string getText(int index) const;
    void setPicture(int index, const char *filename);
    void setPicture(int index, const char* xpm[]);
    void setSelected(int index, bool checked = true);
    std::list<int> getSelected() const;
    bool isSelected(int index) const;

<QRSeperator.h>
    QRSeparator();
    QRSeparator(QROSE::Orientation orientation);


<QRTable.h>
    QRTable();
    QRTable(int numCols, ...);
    // colId is the column after which to insert n columns
    // if colId = -1, then it adds columns at the end
    void addCols(int numCols, int colId = -1);
    void removeCol(int colId);
    void setColHeaders(const char *caption0, ...);
    void showColHeader(bool show = true);
    void addRows(int numRows, int rowId = -1);
    void removeRow(int rowId);
    void setRowHeaders(const char *caption0, ...);
    void showRowHeader(bool show = true);

    // the following methods allow you to set the attributes of one or more cells
    // col=c, row=r : sets attribute of cell (c,r)
    // col=All, row=r: sets attribute of row r
    // col=c, row=All, sets attributes of column c
    // col=All, row=All: sets attributes of all cells
    // col=c, row=Header: sets attributes of column header c
    // col=Header, row=r: sets attribute of row header r
    // col=All, row=Header: sets attributes of all column headers
    // col=Header, row=All: sets attribute of all row headers
    // col=Header, row=Header: sets atttributes of all headers
    void setText(std::string text, int col, int row = All);
    void clearText(int col, int row = All);
    void setPicture(const char *icon_filename, int col, int row = All);
    void setPicture(const char *xpm[], int col, int row = All);
    void clearPicture(int col, int row = All);
    void setTextColor(QColor color, int col, int row = All);
```

```
    void setBgColor(QColor color, int col, int row = All);
    void setFont(QFont font, int col, int row = All);
    QFont getFont(int col, int row) const;
    void setType(Type type, int col, int row = All);
    Type getType(int col, int row) const;
    void setChecked(bool checked, int col, int row = All);
    bool isChecked(int col, int row) const;
    bool isCheckable(int col, int row) const;
    void setEnabled(bool enabled, int col, int row = All);
    bool isEnabled(int col, int row) const;
    void setHAlignment(bool left, bool right, int col, int row = All);
    void setVAlignment(bool top, bool bottom, int col, int row = All);
    void activateCell(int col, int row);
    // sets width and height of columns and rows
    void setHDim(int col, int width = -1);
    void setVDim(int row, int height = -1);

<QRToolBar.h>
     QRToolBar(QROSE::Orientation orientation, bool showText = true,
               bool showPic = true, bool picBesidesText = true);
     int addButton(std::string caption, std::string icon_filename = "");
     int addButton(std::string caption, const char *icon_xpm[]);
     int addToggleButton(std::string caption, std::string icon_filename = "");
     int addToggleButton(std::string caption, const char *icon_xpm[]);
     void insertSeparator();
     void setCaption(int id, std::string caption);
     void setPicture(int id, std::string filename);
     void setPicture(int id, const char *xpm[]);
     void setEnabled(int id, bool enable);
     bool isEnabled(int id) const;
     void setChecked(int id, bool checked);
     bool isChecked(int id) const;
     bool isCheckable(int id) const;
     void setHidden(bool enable);
     bool isHidden() const;
     void setHidden(int id, bool enable);
     bool isHidden(int id) const;
     unsigned numButtons() const;
     QAction* getAction(int id) const;
     int getId(QAction *action) const;

<QRTree.h>
      TableModel(QRTree *tree);
      int rowCount(const QModelIndex &parent = QModelIndex()) const;
      QVariant data(const QModelIndex &index, int role) const;
      QVariant headerData(int section, Qt::Orientation orientation,
                          int role = Qt::DisplayRole) const;
```

5. Database Support (A)
6. Graphs and Graph Analysis (T)

```

<HEADER_GRAPH_START>
  //! Simple edge type used to input data to Boost algorithms
      typedef std::pair<int, int> BoostEdgeType;
   // DQ (4/29/2009): Added support for boost edges to be used in boost graph library algori
   // We need this local type so that the member access functions for data members of this t
   // typedef std::vector<SgGraph::BoostEdgeType> SgBoostEdgeList;
      typedef std::vector<BoostEdgeType> SgBoostEdgeList;
   // typedef SgBoostEdgeList* SgBoostEdgeListPtr;
   // DQ (4/29/2009): Added support for boost edges to be used in boost graph library algori
      typedef std::vector<int> SgBoostEdgeWeightList;
   // typedef SgBoostEdgeWeightList* SgBoostEdgeWeightListPtr;
      int hashCode( const char* p, int len) const; // hash a character array
   // void initialize_graph_id();
      void append_properties( int addr, const std::string & prop );
  //! Support for adding SgGraphNode to SgGraph.
      SgGraphNode* addNode( const std::string & name = "", SgNode* sg_node = NULL);
  //! Add support for externally build SgGraphNode objects
      SgGraphNode* addNode( SgGraphNode* node );
  //! Support for adding SgGraphEdge to SgGraph.
      SgGraphEdge* addEdge( SgGraphNode* a, SgGraphNode* b, const std::string & name = "");
  //! Add support for externally build SgGraphNode objects
      SgGraphEdge* addEdge( SgGraphEdge* edge );
      void post_construction_initialization();

  //! Support for Boost Minimum Spanning Tree.
   // std::vector <BoostEdgeDescriptor> generateSpanningTree();
      std::vector <SgGraphEdge*> generateSpanningTree();
   // tps (4/30/2009): Added properties for nodes and edges
   // todo: this will be replaced with AstAttributes once the graph conversion is done.
   std::string getProperty(SgGraph::GraphProperties property, SgGraphNode* node);
   std::string getProperty(SgGraph::GraphProperties property, SgGraphEdge* edge);
   void setProperty(SgGraph::GraphProperties property, SgGraphNode* node, std::string value)
   void setProperty(SgGraph::GraphProperties property, SgGraphEdge* edge, std::string value)

  // tps (4/30/2009): The following are functions on the graph that were used before
  // in the old graph implementation
   //std::set<SgGraphEdge*> getEdge(SgGraphNode* src, SgGraphNode* trg);
      void checkIfGraphNodeExists(const std::string& trg_mnemonic, std::vector<SgGraphNode*>
  //! Check if the node is present in the graph.
      bool exists( SgGraphNode* node );
  //! Check if the edge is present in the graph.
      bool exists( SgGraphEdge* edge );
  //! Builds a set of edges that are associated with a specific node.
      std::set<SgGraphEdge*> computeEdgeSet( SgGraphNode* node );
   //! Integer index version of "std::set<SgGraphEdge*> computeEdgeSet( SgGraphNode* node );
      std::set<int> computeEdgeSet( int node_index );
  //! Build set of node index pairs associated with node index (one of the value of the pair
      std::set< std::pair<int,int> > computeNodeIndexPairSet( int node_index );
  //! Builds a set of node index values associated with a label.
```

```
      std::set<SgGraphNode*> computeNodeSet( const std::string & label );
 //! Builds a set of all nodes in the graph.
      std::set<SgGraphNode*> computeNodeSet();
  //! Integer index version of "std::set<SgGraphNode*> computeNodeSet( const std::string & label );"
      std::set<int> computeNodeIndexSet( const std::string & label );
  // Number of nodes in graph.
      size_t numberOfGraphNodes() const;
  // Number of edges in graph.
      size_t numberOfGraphEdges() const;

      void display_node_index_to_edge_multimap() const;
      void display_node_index_to_node_map() const;
      void display_edge_index_to_edge_map() const;
      void display_node_index_pair_to_edge_multimap() const;
      void display_string_to_node_index_multimap() const;
      void display_string_to_edge_index_multimap() const;

//! Resize the internal hash tables based on the number of nodes (hash_maps and hash_multimaps for e
      void resize_hash_maps( size_t numberOfNodes, size_t numberOfEdges = 10 );
  //! Report the size in bytes of the graph (includes all edges and ndes from all hash_maps and hash
      size_t memory_usage();


<HEADER_GRAPH_POSTDECLARATION_START>
// DQ (4/29/2009): Added support for boost edges to be used in boost graph library algorithms.
// We need this global type so that the member access functions (defined outside the class)
// for data members of this type can be resolved.
typedef SgGraph::SgBoostEdgeList SgBoostEdgeList;
typedef SgBoostEdgeList* SgBoostEdgeListPtr;
// DQ (4/29/2009): Added support for boost edges to be used in boost graph library algorithms.
typedef SgGraph::SgBoostEdgeWeightList SgBoostEdgeWeightList;
typedef SgBoostEdgeWeightList* SgBoostEdgeWeightListPtr;
// Supporting graph type required by Boost Graph Library.
// typedef boost::graph_traits < SgGraph::BoostGraphType >::edge_descriptor BoostEdgeDescriptor;


<HEADER_GRAPHNODE_START>
      void append_properties( int addr, const std::string & prop );
      void post_construction_initialization();

<HEADER_GRAPHEDGE_START>
      void append_properties( int addr, const std::string & prop );
      void post_construction_initialization();

<HEADER_DIRECTED_GRAPH_EDGE_START>
      public:
        // DQ (8/18/2008): This is part of the OLD interface introduced for backward compatability!
           SgDirectedGraphEdge(std::string name, std::string type, int n, SgGraphNode* from, SgGraphN
           SgGraphNode* get_from() { return p_node_A; }
           SgGraphNode* get_to()   { return p_node_B; }
```

```

<HEADER_GRAPH_NODE_LIST_START>
      public:
           typedef rose_hash::hash_multimap<std::string,SgGraphNode*, rose_hash::hash_string
           typedef local_hash_multimap_type::iterator iterator;


<HEADER_GRAPH_EDGE_LIST_START>
      public:
           typedef rose_hash::hash_multimap<SgGraphNode*,SgGraphEdge*, rose_hash::hash_graph
           typedef local_hash_multimap_type::iterator iterator;

<HEADER_UNDIRECTED_GRAPH_EDGE_START>
           SgGraphNode* get_node1() { return p_node_A; }
           SgGraphNode* get_node2() { return p_node_B; }


<HEADER_INCIDENCE_DIRECTED_GRAPH_START>
  // tps (4/30/2009): The following are functions on the graph that were used before
  // in the old graph implementation
           std::set<SgGraphEdge*> getEdge(SgGraphNode* src);
           bool checkIfGraphEdgeExists(SgGraphNode* src);
      void post_construction_initialization();
 //! Support for adding SgGraphEdge to SgGraph.
      SgDirectedGraphEdge* addDirectedEdge( SgGraphNode* a, SgGraphNode* b, const std::strin
 //! Add support for externally build SgGraphNode objects
      SgDirectedGraphEdge* addDirectedEdge( SgDirectedGraphEdge* edge  );
          // tps (4/30/2009): Added to support functionality for DirectedGraphs
      void getSuccessors(SgGraphNode* node, std::vector <SgGraphNode*>& vec );
      void getPredecessors(SgGraphNode* node, std::vector <SgGraphNode*>& vec );
      std::set<SgDirectedGraphEdge*> getDirectedEdge(SgGraphNode* src, SgGraphNode* trg);
      bool checkIfDirectedGraphEdgeExists(SgGraphNode* src, SgGraphNode* trg);
  // DQ (8/18/2009): Added support for construction of sets of edges.
      std::set<SgDirectedGraphEdge*> computeEdgeSetIn( SgGraphNode* node );
      std::set<int> computeEdgeSetIn( int node_index );
      std::set<SgDirectedGraphEdge*> computeEdgeSetOut( SgGraphNode* node );
      std::set<int> computeEdgeSetOut( int node_index );
```

: single or plural form in name convention?

7. Performance Metric Annotation (L) namespace PerformanceMetric

```
//! Loads HPCToolkit XML or GNU gprof text profiling data files given on the command-line
// and create a list of tree representations (called profile trees) for them.
ProfileTreeList_t loadProfilingFiles(std::vector<std::string>& argvList);

//! Attach performance metrics from the profile trees to the AST tree.
void attachMetrics (const ProfileTreeList_t& profiles,
                    SgProject* proj, bool verbose = false);

//! Propagate specified metrics from statement level to parent scopes
// (loop, function, file and project levels)
void propagateMetrics(SgProject * project, const std::vector<string> metricNames);
```

```

  //! Collect metric names from an AST attached with performance metrics
  void collectMetricnames(SgNode* root, std::vector<std::string>& metricNames);

  //! Get a metric from a SgNode based on its name
  MetricAttr* getMetric(SgNode*, std::string metric_name);
```

8. Abstract Handles (L)
   Abstract handles (namespace AbstractHandle) are used to specify arbitrary language constructs within an application.

```
    //! Get SgNode from an abstract handle string
    SgNode* getSgNodeFromAbstracthandle(const std::string& s);

    //! Create an abstract node from a SgNode
    abstract_node * buildAbstractNode(SgNode*);

    //! Create and abstract handle from an abstract node
    abstract_handle* buildAbstractHandle (abstract_node *);

    //! Create an abstract handle from a handle string
    abstract_handle* buildAbstractHandle(std::string handle_string);
```

9. Macro Rewrapper (A)
   This is a feature not yet ready to be a part of the user API in ROSE.
10. Command line processing support (Yi)
    ROSE based tools can add options to their command line or process options of the command line. These function represent command line support for users to detect specialized options or manipulate the command line to add options before processing using ROSE (by the frontend API).
11. Common string support (DQ)
    These function support common operations on strings used within ROSE and useful within ROSE-based tools.
    (a) **int Rose::containsString (const std::string & masterString, const std::string & targetString);**
    Returns result (zero or nonzero) based on containment of *targetString* in *masterString*.

FIXME: *This fun*
not be significant eno

12. Common file and path support (Yi)
    ROSE based tools frequently have to do some simple simple file name handling and this API provides simple access to these functions.

FIXME: *The*
currently are in
namespace which we
to

    (a) **string Rose::getWorkingDirectory ();**
    Returns working directory of ROSE installation, uses call to *getcwd()*.
    (b) **std::string Rose::getSourceDirectory (std::string fileNameWithPath);**
    Return the source directory associated with an installation of ROSE.
    (c) **string Rose::getPathFromFileName (const string fileName);**
    Returns the path associated with the input filename string.
    (d) **std::string Rose::utility_stripPathFromFileName (const std::string & fileNameWithPath);**
    Returns the path associated with the file.

(e) **std::string Rose::getFileNameWithoutPath (SgStatement* statementPointer);**
Returns the name of file associated the specific statement of the AST. The returned string excludes the files path.

(f) **std::string Rose::getFileName (SgLocatedNode* locatedNodePointer);**
Returns the name of file associated the specific statement or expression of the AST. The returned string is the absolute path plus file name.

13. ExtentMap
A class for keeping track of contiguous regions of an address space. Can be used to manage free lists, or keep track what parts of a file have been referenced. Similar to std::map but having different lookup functions and able to combine adjacent regions into single entries in its data structure.
See doxygen ExecMap class.
See src/frontend/BinaryFormats/ExtentMap.C.

14. Debug dumps
The dump() methods scattered throughout src/frontend/BinaryFormats/*.C generate human-readable tables describing all details of the SgAsm* nodes. They all take the same arguments. These are what produce the *.dump files.
See src/frontend/BinaryFormats/*.C

15. String Management
Functions that manage strings that might be stored in various kinds of string tables in a binary file. Modifying strings, sharing storage, repacking tables, reallocating individual strings, avoidance of certain file regions, etc.
See src/frontend/BinaryFormats/GenericString.C

16. Section I/O
A variety of functions for reading the original content of a binary section either by file offset or through a MemoryMap. Also functions for writing back to the file.
Defined for SgAsmGenericFile and SgAsmGenericSection.
See src/frontend/BinaryFormats/GenericFile.C src/frontendBinaryFormats/GenericSection.C

17. MemoryMap
A data structure similar to std::map that maps one address space into another (typically a virtual memory address space into a file address space). This allows ROSE to create a memory address space separate from its own and manage it as a program loader would do.
See doxygen MemoryMap class.
See src/frontend/BinaryFormats/MemoryMap.h

18. Data conversion
Functions for converting data from file format to memory format. Most of these are to handle byte order, but there are some other encodings as well.
See src/frontend/BinaryFormats/ExecGeneric.C

19. Miscellaneous Support (DQ)
Output of usage information, ROSE version number support, etc.

    (a) **void SgFile::usage (int status);**
    This function reports information about ROSE options.

    (b) **std::string version_message ();**
    This function provides a string form of the version message (e.g. output by *–version* option in ROSE).

(c) **std::string version_number ();**
This function provides a string form of the version number of ROSE (in the form major.minor.patch).

21.23.3 IR (PC)

Interm proposed namespace name: *ROSE_IR*
The Intermediate Representation (IR) used in ROSE has data types used within the interfaces of the ROSE API. An understanding of the ROSE API thus requires some familiarity with the hierarchical organization of the IR. However this the IR is perhaps best represented by the Doxygen generated web pages which present both the hierarchy of C++ classes used to represent the IR and the details of the individual classes.

Due to the number of nodes in the IR it is more productive to identify here any elements that should be considered for removal from the existing IR rather than listing elements which should be kept.

For the sake of simplifying the user interface and reducing confusion, the IR has been reviewed to identify nodes which are not being used with a view to a decision as to whether these nodes should remain in the IR. The following nodes have thus far been identified as being unused:

- SgAsmUnaryPlus
- SgAsmUnaryMinus
- SgCommonSymbol
- SgDefaultSymbol
- SgDirectory
- SgDirectoryList
- SgExpressionRoot
- SgFileList
- SgInterfaceSymbol
- SgIntrinsicSymbol
- SgModuleSymbol
- SgOptions
- SgRefExp
- SgSpawnStmt
- SgTypeGlobalVoid
- SgUnknownArrayOrFunctionReference
- SgVariantExpression
- SgVariantStatement

If we choose to split the ROSE header files, it would also be prudent to consider if the IR definition header file (Cxx_Grammar.h) can or should be split. This should be done by identifying groupings of related nodes. Suggested groupings include:

- IRCore.h (SgNode, all other nodes not identified below and common functionality such as variants)
- IRSupport.h (SgSupport and subclasses)
- IRExpr.h (SgExpression and subclasses)

- IRStmt.h (SgStatement and subclasses)
- IRDecl.h (SgDeclarationStatement and subclasses)
- IRType.h (SgType and subclasses)
- IRAsm.h (SgAsmNode and subclasses)

21.24 ROSE Example Projects

These project demonstrate types of tools that have been built using ROSE and are small enough to include within the ROSE distribution (taken from the ROSE/projects directory):

1. Compass
2. Auto Parallelization
3. Assembly To Source AST
4. Auto Tuning
5. Haskell port
6. Java port
7. MPI Code Motion
8. Reverse Computation
9. Qt Designer Plugins
10. Distributed Memory Analysis Compass
11. Name Consistancy Checker
12. Name Similarities
13. Run-time Error Detection (RTED)
14. Mixed Static Dynamic Analysis
15. Taint Check
16. UpcTranslation
17. CERT Secure Code Project
18. Data-Structure Graphing
19. Documentation Generator
20. Bug Seeding
21. (Review the projects directory)

This work should coinside with tutorial examples that should the use of the API and present the simple executables that generated specific forms of analysis (e.g. call graph, outliner, etc.).

Topics that it is less clear where to put (or even if to put) in the ROSE API include:

- Annotation Language Parser (annotationLanguageParser directory; not working)
- Distributed Memory Analysis (distributedMemoryAnalysis directory)
- AST Rewrite Mechanism (supress this from the API for now).
- ompLowering (not for the user interface)
- OpenMP Reduction Variable Recognition

21.25 How to add a commandline switch to ROSE translators

The following instructions are only for ROSE developers to add command line switches to *ALL* translators built using ROSE. the instructions serve mostly as an example and it is helpful to look at the referenced source code in the referenced files.

Note that users may process the commandline directly and support in available in ROSE explicitly for this level of support (and an example is in the ROSE Tutorial).

The following switch is an example of how to add such command line switches to ROSE.

From Support.C in ROSE/src/ROSETTA/src:

```
File.setDataPrototype ( "bool", "F77\_only", "= false",
          NO\_CONSTRUCTOR\_PARAMETER, BUILD\_FLAG\_ACCESS\_FUNCTIONS, NO\_TRAVERSAL, NO\_DELETE);
```

The look for the names (e.g. "F77_only") in "sage_support.cpp" in ROSE/src/frontend/SageIII/sage_support and put in the command line processing support as in:

```
     if ( CommandlineProcessing::isOption(argv,"-rose:","(f77|F77|Fortran77)",true) == true )
        {
          if ( SgProject::get\_verbose() >= 1 )
               printf ("Fortran77 only mode ON \n");
          set\_F77\_only(true);
          set\_Fortran\_only(true);
          if (get\_sourceFileUsesFortran77FileExtension() == false)
             {
               printf ("Warning, Non Fortran77 source file name specificed with explicit -rose:Fortr
               set\_F77\_only(false);
             }
        }
```

There are a few other details, so don't forget the initialize the value in the function "SgFile::initialization()" in Support.code in src/ROSETTA/Grammar:

```
     p\_F77\_only                 = false;
```

If your option takes a parameter, then you have to make this clear by adding it to the function (located in sage_support.cpp):

```
     bool CommandlineProcessing::isOptionTakingSecondParameter( string argument )
```

Also, don't forget to document your option in SgFile::usage() (also in sage_support/cmdline.cpp).

21.26 Managing Hudson

ROSE uses customized versions of Hudson and the Git plugin that display Git branch names in the build list. This section describes the steps necessary to patch and build the latest version of Hudson and the Git plugin from source. Building both requires Apache Maven $\geq$ 2.2.1.

21.26.1 Building the Customized Version

Developers can build an the latest, customized version of Hudson and the Git plugin by running the `build` script in the top level of the `hudson-rose` repository:

```
git clone /usr/casc/overture/ROSE/git/hudson-rose.git
cd hudson-rose
./build
```

The build products (`hudson.war` and `git.hpi`) will appear in the top level of the local repository. The plugin can be installed using the Hudson web interface at:

```
http://hudson-rose.llnl.gov:8080/pluginManager/advanced
```

or by copying `git.hpi` into Hudson's plugins directory, usually:

```
~/.hudson/plugins/
```

21.26.2 Upgrading and Customizing Hudson

These steps are intend to guide a developer through the process of patching a newly-released version of Hudson or the Git Plugin.

1. Changes to Hudson and the Git plugin are tracked in a repository located at `/nfs/casc/overture/ROSE/git/hudson-rose.git`. Clone this repository, which will be referred to as `$HUDSONROSE`. In `$HUDSONROSE`, run

   ```
   git submodule update --init
   ```

   to initialize the appropriate Hudson and Git plugin submodules.

2. To patch the latest version of Hudson, enter `$HUDSONROSE/hudson` and retrieve the latest version of Hudson from the developers' site. With Git, this is accomplished by:

   ```
   git remote add github git://github.com/kohsuke/hudson.git
   git fetch github
   git checkout -b b1.370_custom 1.370
   ```

   This will create a new branch (named `b1.370_custom`) in your local repository that is identical to the tagged release for Hudson version 1.370. Notes:

   - Replace `1.370` with the latest version number.
   - The GitHub URL may change as development progresses. Verify that you are using the appropriate URL.
   - For developers within LLNL, it may be necessary to open ports in order to clone the remote repository. This can be accomplished by submitting an Egress Open Request to `cspservices.llnl.gov`.

3. Next, apply the custom patches to the current branch. Since the number of patches is small (five at present time), it is easier to cherry-pick them from previously customized versions of Hudson. Use your favorite repository browser to locate the required changes. If the changes are represented by commits `$PATCH1` and `$PATCH2`, the following applies them to the latest version of Hudson:

   ```
   git cherry-pick --no-commit $PATCH1
   git cherry-pick --no-commit $PATCH2
   git commit
   ```

When upgrading from Hudson 1.334 to Hudson 1.370 and Git-Plugin 0.7.3 to Git-Plugin 1.0, the following commits were cherry-picked:

- `036444f61316dd8f0ca0d2f191e55045f99762ca`
- `1cb5bf79df2d4fdf2caf1b73dcaf417f1b14329f`
- `3b7a34503610dbd9f6984371d9436f6100648d10`
- `7f9c697fb2b2bc8d71a8ac76a425582c2f60000f`
- `ad2fd58439a85a02242dfda5d0fab7c8f970d17c`

4. Next, patch the Git plugin by changing to the `$HUDSONROSE/hudson-git` directory and following the same procedure above.

5. To build Hudson and the plugin, return to the `$HUDSONROSE` directory. Verify that patches in `$HUDSONROSE/patches` are still required by reading them and comparing them to the latest version of the vanilla distribution. If a patch at location `$PATCHPATH` is no longer required, disable it by appending `-disabled` to the filename:

```
git mv $PATCHPATH $PATCHPATH-disabled
```

Currently, there are three patches in the repository:

`patches/hudson-git/0001-Hacked-the-pom.xml-to-point-it-to-our-hudson-repo.patch`
Adds the ROSE team's Hudson repository to the Project Object Model.

`patches/hudson/0002-Comment-out-some-code-that-fails-to-compile.patch-disabled`
(Disabled) Commented out code that prevented compilation in Hudson 1.334.

`patches/hudson/0001-Specify-local-repository.patch`
Adds the ROSE team's Hudson repository to the Project Object Model.

6. Run the customized build script that applies patches, calls Maven, and copies the resulting `hudson.war` and `git.hpi` to `$HUDSONROSE`:

```
./build
```

The following errors may occur:

Patch did not apply
Significant changes to Hudson or the Git plugin may prevent `git-am` from applying patches automatically. Edit the patches by hand to resolve this issue.

Not a v4.0.0 pom or not this project's pom
For developers within LLNL, the firewall may insert HTTP redirect pages when Maven is pulling in dependencies. Remove the offending `pom` or `xml` files (usually found in `~/.m2/repository`) and rerun `./build`.

When `./build` succeeds, the files `hudson.war` and `git.hpi` will be copied to `$HUDSONROSE`. See the Hudson documentation for directions on deploying these files.

7. To save your work, tag it and push the tag to the central repository:

```
cd $HUDSONROSE/hudson
git tag 1.370_custom
git push origin : 1.370_custom
cd $HUDSONROSE/hudson-git
git tag git-1.0.1_custom
git push origin : git-1.0.1_custom
```

```
cd $HUDSONROSE
git commit
git push origin HEAD
```

Issues Upgrading Hudson Because of changes after Hudson v1.345, the Git plugin no longer compiles against the latest Hudson version. Until the plugin is updated, the developer recommends compiling the plugin against Hudson 1.345. A plugin compiled against Hudson 1.345 will work with the latest version of Hudson, compiled separately. The current `./build` script automatically builds the Git Plugin against a customized version of Hudson 1.345.

21.27 Managing Non-recursive Autotools

The codebase of ROSE spans many directories. This provides the proper level of granularity as a good software development practice. The GNU Autotools has therefore been well-suited for the hierarchical structure of ROSE with its out-of-the-box recursive approach.

However, this default recursive process limits the amount of parallelism during the building and testing of ROSE, especially on our Continuous Integration (CI) server, Hudson. The result is that everyone is slowed down. As we get new machines with more processing cores this will increasingly be a problem (until we saturate the disk or memory bandwidth).

- Recursive Make Considered Harmful by Peter Miller (http://miller.emu.id.au/pmiller/books/rmch/)
- Non-recursive Automake (http://www.flameeyes.eu/autotools-mythbuster/automake/nonrecursive.html)

21.28 A Quick Look at Recursive Automake

A traditional `Automake Makefile.am` would contain the `SUBDIRS` keyword, as in:

```
SUBDIRS = subdir1 subdir2 ...  subdirN
```

This is the mechanism that provides `Automake` with its recursive ability. The generated `Makefile` may resemble the following:

```
---------------------------------------------------------------------
    $(RECURSIVE_TARGETS):
        # Note: this is an ordered-sequential loop
             list='$(SUBDIRS)'; for subdir in $$list; do \
               echo "Making $$target in $$subdir"; \
               if test "$$subdir" = "."; then \
                 dot_seen=yes; \
                 local_target="$$target-am"; \
               else \
                 local_target="$$target"; \
               fi; \
               (cd $$subdir && $(MAKE) $(AM_MAKEFLAGS) $$local_target) \
```

```
          || eval $$failcom; \
        done; \
        if test "$$dot_seen" = "no"; then \
          $(MAKE) $(AM_MAKEFLAGS) "$$target-am" || exit 1; \
        fi; test -z "$$fail"
----------------------------------------------------------------------
```

Don't worry about the gory details of this `Makefile` target, just understand that `Automake` converts the `SUBDIRS` keyword into an ordered-sequential loop. That is, when `make` is executed in this directory, `Make` will recursively invoke `make` on its subdirectories, in the order that they were listed. What this means is that each of the `Make` processes in the different directories are independent of each other. Therefore, the amount of work that `Make` can do in a given directory is limited to the amount of work specified in that directory's `Makefile`. This is a clean and intuitive approach, but it runs into problems when parallel `make` is used. Parallel make can be achieved by executing `make` as follows:

```
$ make -j<# of processes>
```

The problem is that if we execute `make -j16`, for example:

- We are under-utilizing resources if there are less than 16 tasks to be completed
- `Make` will stay in the current directory until everything is complete in it

Our goal then is to show `Make` that there is plenty more work to be done. Well, the only way to do this is to hand `Make` the work. The solution: "flattened" `Makefile`s.

21.29 Manually "Flattening" Recursive Automake Makefiles

For sake of example, let's look at flattening directories within the ROSE/src directory. Ultimately, we are concerned with generating the Libtool library `librose.la`. In our example, we will be flattening a hypothetical ROSE/src/Parent directory.

21.29.1 Relevant files

- ROSE/config/support-rose.m4
- ROSE/src/Makefile.am
- *path/to/directory/for/flattening/Makefile.am*

ROSE/config/support-rose.m4

```
----------------------------------------------------------------------
    AC_CONFIG_FILES([
    ...
    src/Makefile
    src/Parent/Makefile
```

```
    src/Parent/subdir1/Makefile
    src/Parent/subdir2/Makefile
    ...
    )]
---------------------------------------------------------------------
```

`Makefiles` listed in `AC_CONFIG_FILES` are processed by Autoconf, converted into `Makefile.ins` and ultimately converted into GNU Makefiles.

Let's flatten the `src/Parent` directory. In order to do so, we'll want to propagate the information contained in its subdirectory `Makefile.am`'s into its own `Makefile.am`. And because of this propagation upward, we will no longer need to generate the `Makefile`'s in `subdir1` and `subdir2`.

Modified support-rose.m4

```
---------------------------------------------------------------------
    AC_CONFIG_FILES([
    ...
    src/Makefile
    src/Parent/Makefile
    ...
    )]
---------------------------------------------------------------------
```

Be careful to note the disadvantage of this approach: with no `Makefile`s in both `subdir1` and `subdir2`, a developer would not be able to invoke `make` in either of those directories.

The next step is to propagate the subdirectory `Makefile.am` information. To achieve this goal, we will make use of Automake's `include` directive. But first, let's take a look at the `Makefile.am` in the `src/` directory:

ROSE/src/Makefile.am

```
---------------------------------------------------------------------
    ...

    lib_LTLIBRARIES = librose.la

    ...

    librose_la_LIBADD = $(libroseLibraries)

    ...

    libroseLibraries = \
      ...
      $(top_builddir)/src/Parent/libparent.la \
      $(top_builddir)/src/Parent/subdir1/libsubdir1.la \
```

```
      $(top_builddir)/src/Parent/subdir2/libsubdir2.la \
      ...
---------------------------------------------------------------------
```

Here we are creating a `Libtool` library for our `src/` tree, naming it `librose.la`. The individual libraries listed in `libroseLibraries` are `Libtool` convenience libraries that we are creating per-directory. They are deemed "convenience libraries" because the end-goal is not to link a user program against one of them, but rather the convenience libraries serve as intermediate products during Automake's recursive build. After these convenience libraries have been built, `Libtool` will link all of them together into our `librose.la`.

Since we'll be propagating our subdirectories' builds to src/Parent, we will no longer need the intermediate convenience libraries. Rather, all of the compilation products from `src/Parent`, `Parent/subdir1`, and `Parent/subdir2` will be directly linked into `libparent.la` at the same time, so to speak.

```
---------------------------------------------------------------------
    ...

    libroseLibraries = \
      ...
      $(top_builddir)/src/Parent/libparent.la
      ...
---------------------------------------------------------------------
```

After these modifications, nothing will be built in either subdirectory of `Parent`. So our next step is to propagate the subdirectory `Makefiles` into `Parent's Makefile`.

21.29.2 Translating a Makefile.am to a Makefile_variables

Here is a simple, but not lacking, example of what an actual `Makefile.am` looks like.

ROSE/src/Parent/subdir1/Makefile.am

```
---------------------------------------------------------------------
    noinst_LTLIBRARIES = libsubdir1.la

    libsubdir1_la_SOURCES = \
        subdir1_1.C
        subdir1_2.C
        subdir1_3.C

    include_HEADERS = \
        subdir1_1.h \
        subdir1_2.h \
        subdir1_3.h
```

```
EXTRA_DIST = CMakeLists.txt

clean-local:
    echo "Just demonstrating a clean-local rule"
    rm -rf a_test_file
    echo "Done with clean-local"

subdir1_example_target:
    touch a_test_file

---------------------------------------------------------------------
```

In this `Makefile.am`, we are creating a libtool library that links three C source files. We also list headers that should be installed, as well as a `CMake` file that should be included in a distribution of ROSE. Additionally, we have a common `Makefile` target (`clean-local`) that typically does some extra housekeeping. Finally, we have a non-standard user-defined target (`subdir1_example_target`) that creates an empty file named `a_test_file` (by non-standard I mean that it has no formal significance to either the `GNU Autotools` or `GNU Make`).

As its name implies, a `Makefile_variables` file will contain `Makefile variables`. What we need to do is convert `Automake` variables like `*_la_sources` into user variables that can be appended to another variable. We have to be careful to use unique names since there could be name conflicts when you flatten any number of directories. We will also convert our `Makefile` targets into variables that specify a list of commands to execute:

ROSE/src/Parent/subdir1/Makefile.am

```
---------------------------------------------------------------------
    parentSubdir1_noinst_LTLIBRARIES = libsubdir1.la

    parentSubdir1_la_sources = \
        subdir1_1.C
        subdir1_2.C
        subdir1_3.C

    parentSubdir1_includeHeaders = \
        subdir1_1.h \
        subdir1_2.h \
        subdir1_3.h

    parentSubdir1_extraDist = \
        CMakeLists.txt

    parentSubdir1_cleanLocal = \
        echo "Just demonstrating a clean-local rule"; \
        rm -rf a_test_file; \
        echo "Done with clean-local"
```

```
    subdir1_example_target:
        touch a_test_file
---------------------------------------------------------------------
```

As you can see, I've prefixed the variables with `parentSubdir1` with hopes of avoiding naming conflicts in the future. Other than that, the variable names (aside from case) are the same. However, these variables do not have any special meaning to `Automake` – they are just plain-old `Makefile` variables.

Pay special attention to how the `clean-local Makefile` target was converted. It is a variable that lists the commands to be executed for `clean-local`, in a form that resembles how you would execute multiple commands on a single commandline line:

```
$ echo "Two commands at the same time can be separated by a semi-colon"; echo "like so"
```

However, note how I've left our `subdir1_example_target` unchanged. This is because this target is not a recursive one that needs to be propagated up to an identical target in `Parent/Makefile.am` (which doesn't exist and if it did, that would probably be an error because of the naming conflict: one target would override the other). Therefore, choosing to leave `subdir1_example_target` in `subdir1/Makefile.am` becomes a compromise: keeping this target local to `subdir1/Makefile_variables` keeps `subdir_example_target` in its logical place, which is arguably more maintainable. If we were to rather move this target to `Parent/Makefile.am`, then we could avoid naming conflicts which becomes an increasingly large problem when flattening many directories and combining many `Makefile.am`s. Therefore, unique strings should be prefixed to local targets and this change should be updated wherever applicable. I will leave our target as-is since it is relatively unique already, but here is an example of how you may want to alter it:

ROSE/src/Parent/subdir1/Makefile.am

```
---------------------------------------------------------------------
    ...

    parentSubdir1_subdir1_example_target:
        touch a_test_file
---------------------------------------------------------------------
```

(The `Makefile_variables` for `Parent/subdir2` will be similar, therefore it won't be necessary for us to look at it)

Now that we have our `Makefile_variables` in a "transportable" form, we can propagate this information up to `Parent's Makefile.am`. This is how it currently looks:

ROSE/src/Parent/Makefile.am

```
---------------------------------------------------------------------
    SUBDIRS = subdir1 subdir2
```

```
noinst_LTLIBRARIES = libparent.la

libparent_la_SOURCES = parent.C

include_HEADERS = parent.h

EXTRA_DIST = CMakeLists.txt

clean-local:
	echo "Just demonstrating a clean-local rule"
	echo "Done with clean-local"

--------------------------------------------------------------------
```

It should be obvious to you by now how the pieces of the puzzle fit together. Here is how we'll propagate the variables in `subdir1`'s and `subdir2`'s `Makefile_variables`:

ROSE/src/Parent/Makefile.am

```
--------------------------------------------------------------------

noinst_LTLIBRARIES = \
	libparent.la \
	$(parentSubdir1_noinst_LTLIBRARIES) \
	$(parentSubdir2_noinst_LTLIBRARIES)

libparent_la_SOURCES = \
	parent.C \
	$(parentSubdir1_la_sources) \
	$(parentSubdir2_la_sources)

include_HEADERS = \
	parent.h \
	$(parentSubdir1_includeHeaders) \
	$(parentSubdir2_includeHeaders)

EXTRA_DIST = \
	CMakeLists.txt \
	$(parentSubdir1_extraDist) \
	$(parentSubdir2_extraDist)

clean-local:
	echo "Just demonstrating a clean-local rule"
	echo "Done with clean-local"
```

```
        $(parentSubdir1_cleanLocal)
        $(parentSubdir2_cleanLocal)

---------------------------------------------------------------------
```

Not bad, right? However, there are two subleties:

1) If we list files in `Parent/Makefile.am`, Automake will by default search in the current directory, `Parent/`, for the files. Our solution won't work then because `subdir1's` and `subdir2's` files are in their respective directories: `Parent/subdir1` and `Parent/subdir2`. Not a problem you say, let's just cut-and-paste all those files and dump them in the `Parent/` directory. Okay fair enough, but we want to avoid this solution. Imagine if you flattened more directories and moved hundreds of files into one location. This is a maintenance nightmare and is bad practice. We want to maintain the logical hierarchical structure of our codebase.

The simple solution is to prefix subdirectory filenames with a path that is relative to the parent directory. We can take advantage of the variable that `Automake` provides us with: `$(srcdir)`. In the `Parent/` directory, `$(srcdir)` will expand to something like `/home/justintoo/ROSE/src/Parent`, like what you would get if you did:

```
    $ cd "/home/justintoo/ROSE/src/Parent" && pwd
```

Let's add source-path variables to the top of `Parent/Makefile.am`:

```
        parentSubdir1SrcPath=$(srcdir)/subdir1#
        parentSubdir2SrcPath=$(srcdir)/subdir2#
```

A sublety within a sublety: `clean*` targets like `clean-local` will want to clean up BUILD products, which are NOT in the source tree, but are in the build tree. Therefore, `$(srcdir)` will not work. Instead, `Automake` provides us with another handy predefined variable, `$(builddir)`. Don't you love the symmetry? Let's add build-path variables to the top of `Parent/Makefile.am`:

```
        parentSubdir1BuildPath=$(builddir)/subdir1#
        parentSubdir2BuildPath=$(builddir)/subdir2#
```

Note how I've added hashmarks (`#`) to the end of my path variables. This is to ensure that there are no trailing spaces. I do this because some developers may not have their text editors configured to highlight extraneous whitespace. The reason we don't want to have any trailing spaces after our path variables is most apparent in a clean rule:

```
    clean-local:
        echo "Just demonstrating a clean-local rule"
        rm $(parentSubdir1BuildPath)/a_test_file
        echo "Done with clean-local"
```

If there were a trailing space in `parentSubdir1BuildPath`, we would have:

```
clean-local:
    echo "Just demonstrating a clean-local rule"
    rm /home/justintoo/ROSE/src/Parent/subdir1 /a_test_file
    echo "Done with clean-local"
```

This has the adverse effect of removing the whole `subdir1/` directory which is not what we want to do. Just be careful! Although implementing a non-recursive `Autotools` project is seemingly trivial, you must be careful to not overlook the many subleties that exist. Especially in larger, more complex projects, overlooking the subleties of the build system can lead to obscure and unpleasant bugs in the future. Moreover, you want to create a non-recursive build system that is equivalent to its recursive counterpart.

2) The variables that we've added to `Parent/Makefile.am` aren't defined in `Parent/Makefile.am`'s scope. We defined the variables in the subidrectory `Makefile_variables`. This is where the `Makefile include` directive comes in handy. As explained earlier, `include` works by essentially inserting the contents of the included file into the current `Makefile`. Let's add our `include` directives to the top of `Parent/Makefile.am`:

```
include $(srcdir)/subdir1/Makefile_variables
include $(srcdir)/subdir2/Makefile_variables
```

TODO: might want to keep `clean-local` targets as targets, rather than appending them as commands to be executed to the toplevel `clean-local` target. Of course, unique prefixes would have to be given to each subdirectory clean rule (e.g. `parentSubdir1_clean-local`. This way, a user would have the option of cleaning only a specific subdirectory, rather than cleaning everything because then everything would have to be rebuilt when `make` was run again. However, currently all build products are being generated in the toplevel directory. `AM_INIT_AUTOMAKE([subdir-objects])` will have to be utilized in the future.

This is what our final `src/Parent/Makefile.am` looks like:

ROSE/src/Parent/Makefile.am

```
------------------------------------------------------------------------
    include $(srcdir)/subdir1/Makefile_variables
    include $(srcdir)/subdir2/Makefile_variables

    # Source paths
    parentSubdir1Path=$(srcdir)/subdir1#
    parentSubdir2Path=$(srcdir)/subdir2#

    # Build paths
    parentSubdir1BuildPath=$(builddir)/subdir1#
    parentSubdir2BuildPath=$(builddir)/subdir2#

    ############################################################
```

```
    parentSubdir1_noinst_LTLIBRARIES = libsubdir1.la

    parentSubdir1_la_sources = \
        $(parentSubdir1Path)/subdir1_1.C
        $(parentSubdir1Path)/subdir1_2.C
        $(parentSubdir1Path)/subdir1_3.C

    parentSubdir1_includeHeaders = \
        $(parentSubdir1Path)/subdir1_1.h \
        $(parentSubdir1Path)/subdir1_2.h \
        $(parentSubdir1Path)/subdir1_3.h

    parentSubdir1_extraDist = \
        $(parentSubdir1Path)/CMakeLists.txt

    parentSubdir1_cleanLocal = \
        echo "Just demonstrating a clean-local rule"; \
        echo "Done with clean-local"

    subdir1_example_target:
        touch a_test_file
---------------------------------------------------------------------
```

From the `src/Parent` directory, run `$ make` and watch the non-recursive `Automake` magic unfold.

Chapter 22

FAQ

This chapter accumulates frequently ask questions (FAQ) about ROSE. The questions are not created by the authors (such FAQs are not particularly useful).

1. Is ROSE a preprocessor, a translator, or a compiler?
   Technically, no! ROSE is formally a meta-tool, a tool for building tools. *ROSE is an object-oriented framework for building source-to-source translators.* A preprocessor knows nothing of the syntax or semantics of the language being preprocessed, typically it recognizes another embedded language within the input file (or attempts to recognize subsets of source language). In contrast, translators process the input language with precision identical to a compiler. Since ROSE helps build source-to-source translators, we resist calling the translators compilers, since the output is not machine code. This point is not a required part of the definition of a compiler, many language compilers use a particular language as an assembly language level (typically `C`). These are no less a compiler. But since we do source-to-source, we feel uncomfortable with calling the translators compilers (the output language is typically the *same* as the input language). The point is further muddled since it is common in ROSE to have a translator hide the call to the vendor's compiler and thus the translator can be considered to generate machine code. But this gives little credit to the vendor's compiler. So we prefer to refer to our work as a tool (or framework) for building source-to-source translators.

2. What does the output from a ROSE translator look like?
   A great deal of effort has been made to preserve the quality of your original code when regenerated by a translator built using ROSE. ROSE preserves all formatting, comments, and preprocessor control structure. There are examples in the ROSE Tutorial that make this point clear.

3. How do I debug my transformation?
   There are a couple of ways to debug your transformation, but in general the process starts with knowing exactly what you want to accomplish. An example of your transformation on a specific input code is particularly useful. Depending on the type of transformation, there are different mechanisms within ROSE to support the development of a transformation. Available mechanisms include (in decreasing levels of abstractions):

   (a) String-Based Specification.
   A transformation may specify new code to be inserted into the AST by specifying the new code as a source code string. Functions are included to permit `insert()`, `replace()`, `remove()`.

   (b) Calling Predefined Transformations.
   There are a number of predefined optimizing transformations (loop optimizations) that may be called directly within a translator built using ROSE.

   (c) Explicit AST Manipulation.
   The lowest level is to manipulate the AST directly. Numerous functions within SAGE III are provided to support this, but of course it is rather tedious.

4. How do I use the SQLite database?
   ROSE has a connection to SQLite, but you must run configure with the correct command-line options to enable it. Example scripts to configure ROSE to use SQLite are in the `ROSE/scripts` directory. Another detail is that SQLite development generally lags behind ROSE in the use of the newest versions of compilers. So you are likely to be forced to use an older version of your compiler (particularly with GNU g++).

5. What libraries and include paths do I need to build an application using ROSE.
   Run `make installcheck` and observe the command lines used to compile the example applications. These command lines will be what you will want to reproduce in your `Makefile`.

6. Where is the `SgTypedefSeq` used?
   Any type may be hidden behind a chain of *typedefs*. The typedef sequence is the list of typedefs that have been applied to any given type.

7. Why are there defining and non-defining declarations?

```
class X;            // non-defining declaration
X* foo();           // return type of function will refer to non-defining declaration
X* xPointer = NULL; // Again, the type will refer to a pointer-to-a-type that will be the non-defining declaration
class X {};         // defining declaration
```

   The traversal will visit the declarations, so you will, in this case, see the `class X;` class declaration and the `class X ;` class declaration. In general, all references to the class X will use the non-defining declaration, and only the location were X is defined will be a defining declaration. This is discussed in great detail in the chapter on SAGE III of the ROSE User Manual and a bit in the Doxygen Web pages.
   In general, while unparsing, we can't be sure where the definitions associated with declarations are in the AST (without making the code generation significantly more complex).

```
class X;
class X{};
```

   could be unparsed as:

```
class X {};  // should have been "class X;"
class X;     // should have been "class X {};"
```

   The previous example hardly communicates the importance of this concept, but perhaps this one does:

```
class X;
class Y {};
class X { Y y };
```

   would not compile if unparsed as:

```
class X { Y y };
class Y {};
class X
```

Note that we can't just make a declaration as being a defining declarations since they are shared internally (types and symbols can reference them, etc.).

8. Why are comments and CPP directives following the statements being removed and reinserted elsewhere? I have been working on a translator, based on the `ROSE/tutorial/CharmSupport.C` translator. If an include statement is in the top of the input code, then the struct added to the top of the source file will contain the include statements in an obviously bad place:

   ```
   struct AMPI_globals_t
      {
     // A Comment
   #include "stdio.h"
        int a_global_int;
      };
   ```

   I am specifying the end of construct for the SgClassDefinition to be `Sg_File_Info::generateDefaultFileInfoForTransformationNode();` The class declaration is prepended into the global scope. How do I correctly insert the new definition and declaration into the top of a file(either before or after the include statements).

   The answer, for anyone interested, is found in a discussion relative to the ROSE Tutorial example (Global Variable Handling, currently Chapter 30).

   The problem is that comments and preprocessor (`cpp`) directives are attached to the statements. When I wrote the tutorial example showing how to collect the global variables and put them into a data structure, I was not careful to use the low level rewrite mechanism to do the removal of the variable declarations from the global scope and the insertion of the same variable declarations into the scope of the class declaration (the struct that holds the previously global variables). Since the comments and `cpp` directives were attached to the variable declaration, they got moved with the declaration into the new struct that holds them (see the example in the tutorial).

   I should have used the rewrite's mechanism for removing and reinserting the variable declarations since it is careful to disassociate and reassociate comments and cpp directives. In fact, it is quite incredible that I didn't use that slightly higher level interface, because I wrote all that stuff several years ago and it was so much work to get it all correct. I'm a big believer in using the highest level of interfaces possible (which perhaps means I should document them better in the Web pages for the IR instead of just in the ROSE User Manual).
   The AST Rewrite Mechanism functions to use are the

   ```
   LowLevelRewrite::remove ( SgStatement* astNode )
   ```

   and

   ```
   LowLevelRewrite::insert ( SgStatement* targetStatement, SgStatementPtrList newStatementList, bool insertBeforeNode ).
   ```

   These will automatically disassociate any `cpp` directives and comments from the surrounding statements and reattach them so that they don't wander around with the statements being removed, inserted, or replaced.

   I will try to get to fixing up the ROSE Tutorial example so use this interface. Rich and I have been spending a lot of time on the Tutorial lately (after finishing the ROSE User Manual two weeks ago). We are getting all the documentation ready for release on the web. This will likely happen in a few weeks, though all the paperwork and approvals are already in place.

   So as it is, this is a wonderful example of just what a bad idea it is to manipulate the AST at such a low level. It is the reason we have the AST Rewrite Mechanism – provide the highest level of interface required to make manipulation generally more simple.

9. We have read, that the rose compiler is provided under the BSD license. Is every part of the rose compiler under BSD licence and is it free for commercial use?
   ROSE is free for commercial use, our research license with EDG has no restrictions (except that we can only release the binary and not the source code). Obviously the EDG part is not released BSD, only the source code part. If you want to build products using ROSE for C/C++, then you should consider contacting EDG for a license to there work then you could build commercial products and sell them; but you don't have to worry about ROSE. I have no idea what ground your on if you build commercial products for sale based on ROSE and just use the EDG binary that we provide. I expect it would be a complicated install for your customers. In general if you are using EDG, and building commercial projects for sale, then I would encourage you to contact EDG and buy a license from them. This is was a few companies have done, and they have consulted EDG on this point. Our goal is to especially encourage open-source C++ work using ROSE. Clearly we derive robustness in C++ in ROSE from the use of EDG, and we are thankful to there liberal research license.

10. Is there a list of projects compiled with rose?
    I don't release a list of projects and specific research groups using ROSE.

11. We have read, that you plan a windows port. Until which date do you plan to port the project?
    We hope to have a windows port using Cygwin, it worked a while back, but was not tested often, so we have to fix some details for it to work again. So it is not a big deal, but I can't promise when it would happen.

12. ROSE computes different kinds of stuff from the actual AST (and semantic/type info): the docs mention control flow, data flow, slicing(?), and some more. Are these types of things computed accurately? That is, can you fully rely on the computed info? Are they computed for the entire C/C++ language, or a subset? Just to give an example: there are implicit calls to destructors of static objects, e.g. f() A a; will get a Ã call at the end of f()'s scope. Do you take such info into account when computing call/dataflow/control graphs? If so, I wonder how you built this info in: do you first construct some form of IR (intermediate rep.) atop of which you compute the dataflow/call graphs and similar? If so, did you actually add all the 'implicit' semantics of C++ manually to the AST? Hope the question is not too unclear. How do you handle global static objects?

    The information computed is as accurate as possible and alwasy represent the full languge (including full C++, Fortran 2003, etc.). Some languges are newer (e.g. Fortran 2003 and PHP so that will still has to mature). Implicit calls to constructors, destructors, short curcuit evaluation, etc. are not inserted specific analysis in: `src/midend/programTransformation/implicitCodeGeneration` is used. This code introduces implicit calls into the AST as explicit calls which are ignored by the code generation (unparser). . Global static objects are not handles specially, but are structurally represented in the AST (Note that C++ static constructor evaluation orders are compiler implementation dependent).

    C: Check if this code is n our regression tests.

13. Linking: to do general full program analysis, you need linking. How did you implement this? Did you actually build in all the C/C++ linking semantics by hand?

    We support whole program analysis by permitting the AST's from several files to be merged, this saves space in the header file duplication and provides an efficient means of handling large scale applications. This work is currently experemental, and works on a 100K C program seperated over 50+ files, but is less robust for C++ code. It is ongoing research work. A less scalable alternative is to just list multiple source files on the command line, however this is not a meaningful solution for applications contianing hundreds or thousands of files. C++ template details are addressed by having each file instantiate all the templates that it requires and then we record which of these are used by the file. All used instantiated templates are represented as specialized templates in the AST and any transformed instatiated (specialized) templates are output as template specializations, else the backend compiler is used to instantiate the required templates so that we can reduce the code generation required.

14. Filtering: say you have a program like #include ¡iostream¿ f() cout<<”x”; . I assume you don’t save all the stuff in iostream, and included headers, in your fact database - it’ll be huge, then. If not, however, you cannot simply discard all stuff from system headers, since the user code may refer to them, like, you need the def of std::cout in the example above. How do you handle this? There also were some remarks in the docs about something like ’sharing’ of semantically-identical declarations that occur in different parts of the code. Like, if you have n declarations of int f(), you would only store one. Is this done within a translation unit, or across translation units?

    In the file containing a CPP include directive, the the generated file will be essentially identical (i.e. with the CPP include directive). However, a traversal of the AST will include all the items in the include files (and alternative traversal will allow you to only travers the input file and skips all other files (e.g. header files). We don’t have a database, unless you consider the AST as a database (in memory). For the case of `iostream` this will be large, but that is what your program really is, so that is how it has to be represented; such details are important for type analysis and that trickles into every other part of analysis (especially for C++). The **sharing** is part of the support for whole program analysis (global analysis) and it permits redundant parts of the code (e.g. declarations, namespaces, etc.) from being represented more than one when handling many files (acorss translation units; tens, hundreds, or thousands).

15. Preprocessing: you mention that ROSE can refer to code locations as they are before preprocessing, although it inputs preprocessed files. So, where exactly do you get the fine-grained (row,column) info from if you only see the preprocessed files? I assume you use #line directives, but is this really enough (e.g. in the presence of whitespace removal by some preprocessors).

    The frontend of EDG includes CPP and thus it reports source code positions before the CPP translation, thus we get and save this information. For Fortran we have to handle the CPP translation more explicitly and so we only have the source position after translation (but Fortran is always a bit special when it is preprocessed). I am not aware the CPP will remove whitespace, but it is not an issue since we get the information from EDG where it is generated before CPP translation.

16. Code correctness: say someone analyzes some code which isn’t fully correct/complete, e.g. misses some includes, or misses some declarations, or plainly has syntax errors. What do you do in such a case? Skip somehow the erroneous code, or alternatively simply abort?

    We can not currently handl incomplete code, I would argue that any analysis of such code would have huge question marks. The essential reasom for this limitation is that we use EDG for C and C++ and it can’t handle incomplete code in version 3.4. However, the newer 3.11 version of EDG is expected to handle incomplete code and then we will support this, we have no experience with this yet.

17. Dialects: how would you handle different language dialects, e.g. c89,c99,the different flavors of C++, Visual C++, etc? Do you build a ’super’ grammar that unifies all these somehow? Or you have alternative grammars / type checkers?

    We support C89, C99, C++ (98 standard), Fortran 4, Fortran 66, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, PHP and Binary Analysis for x86 and ARM using ELF and PE, NE, LE, and DOS binary formats). We will start work on C++0x when we upgrade to the newest version of EDG. We support C++ compiled using Microsoft Visual Studio, but not all the MS extensions. We support a number of GNU specific C and C++ extensions, but not all. Since we use EDG for the frontend, we don’t have any **super** grammar representation (evn EDG does not have such a construction in the design of there frontend). Such concepts don’t work well for real languges when you need to handle all the corners (which is itself a sad commentary on parser generators and/or modern languages). For C and C++ the typechecking is mostly done by EDG and we save this information and add to it in the ROSE IR.

Chapter 23

Glossary

FIXME: *Define th terms: IR nod Attribute, Synthesize Accumulator Attr*

We define terms used in the ROSE manual which might otherwise be unclear.

- **AST** Abstract Syntax Tree. A very basic understanding of an AST is the entry level into ROSE.
- **Attribute** User defined information (objects) associated with IR nodes. Forms of attributes include: accumulator, inherited, persistent, and synthesized. Both inherited and synthesized attributes are managed automatically on the stack within a traversal. Accumulator attributes are typically something semantically equivalent to a global variable (often a static data member of a class). Persistent attributes are explicitly added to the AST and are managed directly by the user. As a result, they can persist across multiple traversals of the AST. Persistent attributes are also saved in the binary file I/O, but only if the user provides the attribute specific `pack()` and `unpack()` virtual member functions. See the ROSE User Manual for more information, and the ROSE Tutorial for examples.
- **CFG** As used in ROSE, this is the Control Flow Graph, not Context Free Grammar or anything else.
- **EDG** Edison Design Group (the commercial company that produces the C and C++ front-end that is used in ROSE).
- **IR** Intermediate Representation (IR). The IR is the set of classes defined within SAGE III that allow an AST to be built to define any application in C, C++, and Fortran application.
- **Query** (as in AST Query) Operations on the AST that return answers to questions posed about the content or context in the AST.
- **ROSE** A project that covers both research in optimization and a specific infrastructure for handling large scale C, C++, and Fortran applications.
- **Rosetta** A tool (written by the ROSE team) used within ROSE to automate the generation of the SAGE III IR.
- **SAGE++ and SAGE II** An older object-oriented IR upon which the API of SAGE III IR is based.
- **Semantic Information** What abstractions mean (short answer). (This might be better as a description of what kind of semantic information ROSE could take advantage, not a definition.)
- **Telescoping Languages** A research area that defines a process to generate domain-specific languages from a general purpose languages.

- **Transformation** The process of automating the editing (either reconfiguration, addition, or deletion; or some combination) of input application parts to build a new application. In the context of ROSE, all transformations are source-to-source.
- **Translator** An executable program (in our context built using ROSE) that performs source-to-source translation on an existing input application source to generate a second (generated) source code file. The second (generated) source code is then typically provided as input to a vendor provided compiler (which generates object code or an executable program).
- **Traversal** The process of operating on the AST in some order (usually pre-order, post-order, out of order [randomly], depending on the traversal that is used). The ROSE user builds a traversal from base classes that do the traversal and execute a function, or a number of functions, provided by the user.

Bibliography

[1] **Bassetti, F., Davis, K., Quinlan, D.:** "A Comparison of Performance-enhancing Strategies for Parallel Numerical Object-Oriented Frameworks", To be published in Proceedings of the first International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference, Marina del Rey, California, Dec, 1997

[2] **Quinlan, D., Berndt, M.:** "MLB: Multilevel Load Balancing for Structured Grid Applications", Published in Proceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[3] **Brown, D., Chesshire, G., Henshaw, W., and Quinlan, D.:** "OVERTURE: An Object-Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments", Published in Proceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[4] **Balsara, D., Quinlan, D.:** "Parallel Object-Oriented Adaptive Mesh Refinement", Published in Proceedings of the SIAM Parallel Conference, Minneapolis, MN. March, 1997

[5] **Quinlan, D.:** "AMR++: A Design for Parallel Object-Oriented Adaptive Mesh Refinement", Published in Proceedings of the IMA Workshop on Structured Adaptive Mesh Refinement, Minneapolis, MN. March, 1997

[6] **Brislawn, K. D., D. L. Brown, G. S. Chesshire, W. D. Henshaw, K. I. Pao, D. J. Quinlan, W. J. Rider, and J. S. Saltzman:** "An Object-Oriented Approach to Grid Generation and PDE Computations Using Adaptive, Moving, Overlapping Grids", Presented at 1996 Parallel Object-Oriented Methods and Applications Conference, Santa Fe, NM; February 29, 1996, and also at the Fifth International Conference on Numerical Grid Generation in Computational Field Simulations, Mississippi State, April 4, 1996.

[7] **Bradley, Brislawn, Quinlan, Zhang, Nuri:** "Wavelet subband coding of computer simulation output using the A++ array class library," Proceedings of Space Earth Science Data Compression Workshop, Snowbird, UT, March 1995.

[8] **Parsons, R. and Quinlan, D.:** "A++/P++ Array Classes for Architecture Independent Finite Difference Computations," Proceedings of the Second Annual Object-Oriented Numerics Conference, Sunriver, Oregon, April 1994.

[9] **Parsons, R. and Quinlan, D.:** "Run-time Recognition of Task Parallelism within the P++ Parallel Array Class Library," Proceedings of the Conference on Parallel Scalable Libraries, Mississippi State, 1993.

[10] **Angus I. G. and Thompkins W. T.:** Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Dynamics; Fourth Conference on Hypercubes, Concurrent Computers, and Applications, 1989.

[11] **Baden, S. B.; Kohn, S. R.:** Lattice Parallelism: A Parallel Programming Model for Non-Uniform, Structured Scientific Computations; Technical report of University of California, San Diego, Vol. CS92-261, September 1992.

[12] **Balsara, D., Lemke, M., Quinlan, D.:** AMR++, a C++ Object Oriented Class Library for Parallel Adaptive Mesh Refinement Fluid Dynamics Applications, Proceeding of the American Society of Mechanical Engineers, Winter Anual Meeting, Anahiem, CA, Symposium on Adaptive, Multilevel and Hierarchical Computational Stratagies, November 8-13, 1992.

[13] **Berryman, H.; Saltz, J. ; Scroggs, J.:** Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines; Concurrency: Practice and Experience, Vol. 3(3), pg. 159-178, June 1991.

[14] **Chandy, K.M.; Kesselman, C.:** CC++: A Declarative Concurrent Object-Oriented Programming Notation; California Institute of Technology, Report, Pasadena, 1992.

[15] **Chase, C.; Cheeung, A.; Reeves, A.; Smith, M.:** Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures; Proceedings of the 1991 Conference on Parallel Processing, IL.

[16] **Forslund, D.; Wingate, C.; Ford, P.; Junkins, S.; Jackson, J.; Pope, S.:** Experiences in Writing a Distributed Particle Simulation Code in C++; USENIX C++ Conference Proceedings, San Francisco, CA, 1990.

[17] **High Performance Fortran Forum:** Draft High Performance Fortran Language Specification, Version 0.4, Nov. 1992. Available from titan.cs.rice.edu by anonymous ftp.

[18] **Lee, J. K.; Gannon, D.:** Object-Oriented Parallel Programming Experiments and Results; Proceedings of Supercomputing 91 (Albuquerque, Nov.), IEEE Computer Society and ACM SIGARCH, 1991, pg. 273-282.

[19] **Lemke, M.; Quinlan, D.:** Fast Adaptive Composite Grid Methods on Distributed Parallel Architectures; Proceedings of the Fifth Copper Mountain Conference on Multigrid Methods, Copper Mountain, USA-CO, April 1991. Also in Communications in Applied Numerical Methods, Wiley, Vol. 8 No. 9 Sept. 1992.

[20] **Lemke, M.; Quinlan, D.:** P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications; Arbeitspapiere der GMD, No. 611, 20 pages, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany (West), February 1992.

[21] **Lemke, M.; Quinlan, D.:** P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications; accepted for CONPAR/VAPP V, September 1992, Lyon, France; to be published in Lecture Notes in Computer Science, Springer Verlag, September 1992.

[22] **Lemke, M., Quinlan, D., Witsch, K.:** An Object Oriented Approach for Parallel Self Adaptive Mesh Refinement on Block Structured Grids, Preceedings of the 9th GAMM-Seminar Kiel, Notes on Numerical Fluid Mechanics, Vieweg, Germany, 1993.

[23] **McCormick, S., Quinlan, D.:** Asynchronous Multilevel Adaptive Methods for Solving Partial Differential Equations on Multiprocessors: Performance results; Parallel Computing, 12, 1989, pg. 145-156.

[24] **McCormick, S.; Quinlan, D.:** Multilevel Load Balancing, Internal Report, Computational Mathematics Group, University of Colorado, Denver, 1987.

[25] **Peery, J.; Budge, K.; Robinson, A.; Whitney, D.:** Using C++ as a Scientific Programming Language; Report, Sandia National Laboratories, Albuquerque, NM, 1991.

[26] **Schoenberg, R.:** M++, an Array Language Extension to C++; Dyad Software Corp., Renton, WA, 1991.

[27] **Stroustrup, B.:** The C++ Programming Language, 2nd Edition; Addison-Wesley, 1991.

[28] V. Adve, G. Jin, J. Mellor-Crummey, and Q. Yi. High performance fortran compilation techniques for parallelizing scientific codes. In *Proceedings of SC98: High Performance Computing and Networking*, Nov 1998.

[29] V. Adve and J. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, June 1998.

[30] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, San Francisco, October 2001.

[31] J. F. L. Carter and S. F. Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[32] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, Oct. 1991.

[33] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, 1989.

[34] Q. Yi. *Transforming Complex Loop Nests For Locality.* PhD thesis, Rice University, 2002.

[35] Q. Yi, V. Adve, and K. Kennedy. Transforming loops to recursion for multi-level memory hierarchies. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, British Columbia, Canada, June 2000.

[36] Q. Yi and K. Kennedy. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. In *LACSI Symposium*, Santa Fe, NM, Oct 2002.

[37] Chunhua Liao, Daniel J. Quinlan, Richard Vuduc and Thomas Panas. Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization. The 22nd International Workshop on Languages and Compilers for Parallel Computing, Newark, Delaware, USA. October 8-10, 2009.

[38] Chunhua Liao, Daniel J. Quinlan, Jeremiah J. Willcock and Thomas Panas, Extending Automatic Parallelization to Optimize High-Level Abstractions for Multicore, In Proceedings of the 5th international Workshop on OpenMP: Evolving OpenMP in An Age of Extreme Parallelism (Dresden, Germany, June 03 - 05, 2009).

[39] Chunhua Liao, Daniel J. Quinlan , Thomas Panas and Bronis de Supinski A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries, the 6th International Workshop on OpenMP (IWOMP), June 14-16, 2010, Tsukuba, Japan.

[40] Q. Yi and K. Kennedy. Transforming complex loop nests for locality. Technical Report TR02-386, Computer Science Dept., Rice University, Feb. 2002.

[41] Unified Parallel C. http://upc.gwu.edu/

[42] OpenMP Application Program Interface Version 3.0 http://www.openmp.org/mp-documents/spec30.pdf

[43] Berkely UPC - Unified Parallel C. http://upc.lbl.gov/publications/

[44] The Berkeley UPC Runtime Specification V 3.10 http://upc.lbl.gov/docs/system/upcr.txt

[45] Itanium C++ ABI Revision: 1.86 http://www.codesourcery.com/public/cxx-abi/abi.html

[46] Denis Barthou Array Dataflow Analysis in Presence of Non-Affine Constraints. PhD thesis, University of Versailles Saint-Quentin en Yvelins, 1998

[47] M. Belaoucha and D. Barthou and S. Touati Dataflow Analysis for Irregular Programs. FADA Toolkit User's Guide. Oct, 2009

[48] Fuzzy Array Dataflow Analysis library. http://www.prism.uvsq.fr/ bem/fadalib/

[49] Paul Feautrier Dataflow analysis of array and scalar references. Parrallel Processing Letters, 1991.

[50] Sanjay Rajopadhye Dependence Analysis and Parallelizing Transformation

[51] Parma Polyhedral Library http://www.cs.unipr.it/ppl/

[52] PipLib http://www.piplib.org/