ROSE User Manual:
A Tool for Building
Source-to-Source Translators
Draft User Manual
(version 0.9.3a)

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*).

October 14, 2008

October 14, 2008

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 Collingbough (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), 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)

ME: *Check spelling of student 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 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.

1.2 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:

- Array Class abstraction optimizer,
- Source-to-source instrumenter,
- Loop analyzer,
- Symbolic complexity analyzer,
- 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.

1.3 What Is 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.

ME: *Check spelling of recent work by Bjarne*

1.4 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.

1.5 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.

1.6 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.

1.7 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].

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

1.9 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

[1] All ROSE documentation is still in development

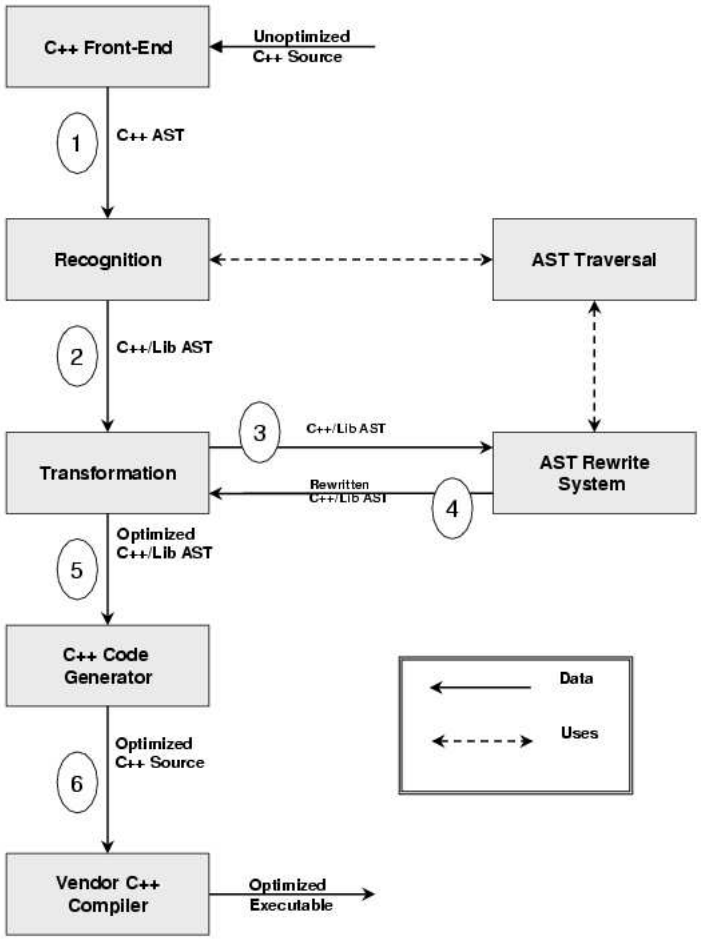

Figure 1.1: Different phases of internal processing within translators built using ROSE infrastructure

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 Processing**
  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-0.9.3a-HTML-Docs.ps.gz
  ROSE HTML documentation that is available without internet access.
- **MANUAL** : ROSE/docs/Rose/ROSE-0.9.3a-UserManual.ps.gz
  This is the ROSE User Manual which explains basic concepts about and capabilities within ROSE.
- **TUTORIAL** : ROSE/docs/Rose/Tutorial/ROSE-0.9.3a-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 a mailing list (casc-rose *at* llnl *dot* gov). The email list is only used to request help and announce internal and external releases. Anyone who would like to be on the email list can send an email to *dquinlan *at* llnl *dot* gov*. Those with LLNL internal access may check who is on the email list at: ROSE Email List (current subscribers, LLNL internal URL link: cmg-r.llnl.gov/casc/computing/mailing_lists/casc-rose.html). At some point the email list will be made more generally open to automated substription.

2.2 ROSE Installation

2.2.1 Requirements and Options

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

Required Hardware/Operating System

ROSE has been developed on Linux/Intel platforms. We have not yet addressed significant portability issues for ROSE. But EDG has addressed portability issues for their C++ frontend and it is available on nearly all platforms (see `www.EDG.com` for details). ROSE is currently developed on Linux/Intel platforms and works with all modern versions of the GNU compilers (3.4.x, and later). ROSE also works on both 32-bit and 64-bit architectures, as well as with the Intel C and C++ compiler. Future work will focus on portability to other platforms important to users.

Software Requirements

You will require **ONLY** a C++ compiler to compile ROSE; ROSE is written in C++. Present development work is done on Intel/Linux platforms using the GNU g++ 3.4.x, and 4.x; and the Intel compilers.

ROSE users may either obtain a free research license from EDG and hence ROSE with EDG source code, or alternatively, obtain ROSE that contains a binary version of the EDG work. The latter is limited to specific platforms and versions of compilers. See EDG (www.edg.com) for details and limitations on how their software may be used. There is more information in the ROSE Manual (see chapter *Getting Started* section *Getting a Version of the EDG License for Research Use*).

Use of Required Software:
The following software is required in order to build and use ROSE:

- **ROSE** :
  There are three versions of ROSE supported: the *Distribution Version* for users (typical), the *External Development Version* for advanced users and collaborators, and the *Internal Development Version* (intended only for ROSE development team). The development versions are what are found in the ROSE software repositories and have additional software requirements (subversion, JDK, autoconf, automake, Doxygen, LaTeX, etc.).

  - **Distribution Version**
    Provided as a tared and compressed file in the form, ROSE-0.9.3a.tar.gz. It can be obtained from outreach.scidac.gov/projects/rose. This is the most typical way that users will see and work with ROSE. But it is less up-to-date compared to development versions.
  - **External Development Version**
    It is available from the SciDAC Outreach Center's subversion repository: outreach.scidac.gov/projects/rose. We put a subset (excluding the EDG part essentially) of the internal developer version of ROSE into the external repository to enable people to have quick access to the most recent new features in ROSE. The external repository is synchronized with the internal repository once a day in ideal conditions. Several branches also exist to accept contributions from external collaborators.
  - **Internal Development Version**
    Only available directly from the LLNL's internal Subversion (SVN) repository. The details of building this version are located in the Appendix of the Manual.

- **g++** : version >= 3.4.x
  In order to use OpenMP or gFortran g++ >= 4.2.x is required.

- **BOOST** : version >= 1.35.0
  Visit www.boost.org for more details about BOOST and www.boost.org/users/download for download and installation instructions.

- **JAVA** : version >1.5.0_11
  A SUN Java virtual machine (JVM) is needed. A Java compiler (JDK) is also required for development versions.

- **Autoconf** : version >= 2.59. Needed *ONLY* for development versions.
  Autoconf is an extensible package of M4 macros that produce shell scripts to automatically configure software source code packages.

- **Automake** : version >= 1.96. Needed *ONLY* for development versions.
  Automake is a tool for automatically generating 'Makefile.in' files compliant with the GNU Coding Standards.

- **Libtool**: version >=1.5.6. Needed *ONLY* for development versions.

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

- **Source Tree**
  Location of source code directory (there is only one source tree).

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

We *strongly* recommend that the **Source Tree** and the **Compile Tree** be different. This avoids many potential problems with the *make clean* rules. Note that the **Compile Tree** will be the same as the **Source Tree** if the user has *not* explicitly generated a separate directory in which to run *configure* and compile ROSE.

If the **Source Tree** and **Compile Tree** are the same, then there is only one combined **Source/Compile Tree**. Alternatively, numerous different **Compile Trees** can be used from a single **Source Tree**. More than one **Compile Tree** allows ROSE to be generated on different platforms from a single source (either a generated distribution or checked out from SVN). ROSE is developed and tested internally using separate **Compile Trees**.

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

- **libxml2-devel** :
  Several optional features of ROSE need to handle XML files, such as roseHPCT and BinaryContextLookup.
- **Doxygen** :
  Most ROSE documentation is generated using LaTex and Doxygen, thus Doxygen is required for ROSE developers that want to regenerate the ROSE documentation. This is not required for ROSE users, since all documentation is included in the ROSE distribution. Visit *www.doxygen.org* for details and to download software. There are no ROSE-specific configure options to use Doxygen; it must only be available within your path.
- **LaTeX** :
  LaTeX is used for a significant portion of the ROSE documentation. LaTex is included on most Unix systems. There are no ROSE specific configure options to use LaTeX; it must only be available within your path.
- **DOT (GraphViz)** :
  ROSE uses DOT for generating graphs of ASTs, Control Flow, etc. DOT is also used internally by Doxygen. Visit *www.graphviz.org* for details and to download software. An example showing the use of the DOT to build graphs is in the ROSE Tutorial. There are no ROSE-specific configure options to use dot; it must only be available within your path.
- **SQLite** :
  ROSE users can store persistent data across separate compilation of files by storing information in an SQLite database. This is used by several features in ROSE (call-graph generation, for example) and may be used directly by the user for storage of user-defined analysis data. Such database support is one way to handle global analysis (the other way is to build the whole application AST). Visit *www.sqlite.com* for details and to download software. An example showing the use of the ROSE database mechanism is in the ROSE Tutorial. Use of SQLite requires special ROSE configuration options (so that the SQLite library can be added to the link line at compile time). See ROSE configuration options for more details (`configure --help`).
- **mpicc** :
  mpicc is a compiler for MPI development. If ROSE is configures with MPI enabled, one can utilize features in ROSE that allow for distributed parallel AST traversals.

2.2.2 Building BOOST

The following is a quick guide on how to install BOOST. For more details please refer to www.boost.org:

1. Download BOOST.
   Download BOOST at www.boost.org/users/download.

2. Untar BOOST.
   Type `tar -zxf BOOST-[VersionNumber].tar.gz` to untar the BOOST distribution.

3. Create a separate compile tree.
   Type `mkdir compileTree` to build a location for the object files and documentation (use any name you like for this directory, e.g. BOOST_BUILD).

4. Create a separate install tree.
   Type `mkdir installTree` to create a location for the install filesto reside (e.g. BOOST_INSTALL).

5. Change directory to the new compile tree directory.
   Type `cd compileTree;` . This changes the current directory to the newly created directory.

6. Run the `configure` script.
   Type `{AbsoluteOrRelativePathToSourceTree}/configure --prefix=[installTree]` to run the BOOST `configure` script. The path to the configure script may be either relative or absolute. The prefix option specifies the installation directory (e.g. BOOST_INSTALL).

7. Run `make`.
   Type `make` to build all the source files.

8. Run `make install`.
   Type `make install` to copy all build files into the install directory. BOOST is now available in your installTree (e.g. BOOST_INSTALL) to be used by ROSE.

Note that the installation of Boost will frequently output warnings (e.g. **(Unicode/ICU support for boost.regex?..not found).**, these can be ignored.

2.2.3 Building ROSE From a Distribution (ROSE-0.9.3a.tar.gz)

The process for building ROSE from a ROSE *Distribution Version* is the same as for most standard software distributions (e.g those using autoconf tools):

1. Untar ROSE.
   Type `tar -zxf ROSE-0.9.3a.tar.gz` to untar the ROSE distribution.

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

3. Change directory to the new compile tree directory.
   Type `cd compileTree;` . This changes the current directory to the newly created directory.

4. Add JAVA environment variables.
   For example:
   ```
   export JAVA_HOME=/usr/apps/java/jdk1.5.0_11
   export LD_LIBRARY_PATH=$JAVA_HOME/jre/lib/i386/server:$LD_LIBRARY_PATH
   ```

5. Add the Boost library path into your LD_LIBRARY_PATH.
   For example: `LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/youraccount/opt/boost_1_35_0/lib`

6. Run the `configure` script.
   Type {`AbsoluteOrRelativePath`}`/configure --prefix=‘pwd‘ --with-boost=[BOOST_installTree]` to run the ROSE `configure` script. The path to the configure script may be either relative or absolute. The prefix option on the configure command line is only required if you run `make install` (suggested), because the default location for installation is `/usr/local` and most users don't have permission to write to that directory. This is common to all projects that use autoconf. ROSE follows the GNU Makefile Standards as a result of using autoconf and automake tools for its build system. As of ROSE-0.8.9a, the default setting for the install directory (prefix) is the build tree. For more on ROSE configure options, see section 2.2.4.

7. Run `make`.
   Type `make` to build all the source files. See details of running `make` in parallel in section 2.2.5.

8. To test ROSE (optional).
   Type `make check` to test the ROSE library against a collection of test codes. See details of running make in parallel 2.2.5.

9. To install ROSE, type `make install`.
   Installation is optional, but suggested. Users can simplify their use of ROSE by using it from an installed version of ROSE. This permits compilation using a single include directory and the specification of only two libraries. See details of installing ROSE in section 2.2.6.

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

2.2.4 ROSE Configure Options

A few example configure options are:

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

- Minimal configuration (prefered)
  `../ROSE/configure --prefix=‘pwd‘ --with-boost=[BOOST_installTree]`
  Configure in the current directory so that installation will also happen in the current directory (a `install` subdirectory will be built).

- Turning on compiler debugging options (prefered)
  `../ROSE/configure --with-CXX_DEBUG=-g --with-C_DEBUG=-g --with-CXX_WARNINGS=-Wall --prefix=‘pwd‘--with-boost=[BOOST_installTree]`
  Configure as above, but with debugging and warnings turned on (`-Wall` is specific to the gnu compilers).

- Adding Fortran support
  `../ROSE/configure --prefix=‘pwd‘ --with-boost=[BOOST_installTree] --with-java`
  The Open Fortran Parser will also be enabled, allowing ROSE to process Fortran code. The programs `java`, `javac`, and `jar` must be either in your PATH or in `$JAVA_HOME/bin`.

- Adding SQLite support
  `../ROSE/configure --with-sqlite3=/home/dquinlan/SQLite/sqliteCompileTree --prefix=‘pwd‘ --with-boost=[BOOST_installTree]`
  Configure as above, but permit use of SQLite database for storage of analysis results between compilation of separate files (one type of support in ROSE for global analysis).

- Adding parallel distributed memory analysis support (using MPI)
  `../ROSE/configure --prefix=‘pwd‘ --with-mpi --with-gcc-omp --with-boost=[BOOST_installTree]`
  Configure as above, but with MPI and OpenMP support for ROSE to run AST traversals in parallel (distributed and shared memory).

- Adding IDA Pro support
  `../ROSE/configure --prefix=‘pwd‘ --with-binarysql --with-boost=[BOOST_installTree]`
  The binarysql flag allows ROSE to read a binary file previously stored as a sql file (e.g. fetched from IDA Pro).

- Adding support for SWIG (Python connection)
  `../ROSE/configure --prefix=‘pwd‘ --with-javaport=yes SWIG=swig --with-boost=[BOOST_installTree] --with-java`
  This allows ROSE to be build with javaport, a support that connects ROSE to Java via SWIG. The Eclipse plug-in to ROSE is based on this work.

- Additional Examples
  More detailed documentation on configure options can be found by typing `configure --help`, or see figure 2.2.4 for complete listing.

Output of `configure --help` is detailed in Figures 2.2.4 (Part 1) and 2.2.4 (Part 2):

2.2.5 Running *GNU Make* in Parallel

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

2.2.6 Installing ROSE

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

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

See the example makefile in
`ROSE/exampleTranslators/documentedExamples/simpleTranslatorExamples/exampleMakefile`
in Section 3.3 for exact details of building a translator on your machine (setup by configure and tested by

`make installcheck`). Note that the tutorial example codes are also tested by `make installcheck` and the `example_makefile` there can also serve as an example.

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

2.2.7 Testing ROSE

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

2.2.8 Getting Help

You may use the following mailing list to ask for help from the ROSE development team: *casc-rose *dot* llnl *dot* gov*.

2.3 Building Translators Using ROSE

At this point you should have installed ROSE. For examples of ROSE translators see the ROSE-0.9.3a-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).

Application Codes

ROSE will be released after tests are complete on approximately 10 separate one-million-line application codes:

1. KULL
   This is an important application at LLNL.
2. ALE3D
   This is an important application at LLNL.
3. ARES
   This is an important application at LLNL.
4. CHOMBO
   This is an Adaptive Mesh Refinement (AMR) library at Lawrence Berkeley National Laboratory.
5. DiffPack
   This is a numerical library originally developed at University of Oslo, Norway. The developers have been substantial collaborators to the ROSE project.
6. ROSE
   The compilation of compiler project (ROSE) with itself is a milestone for any compiler project. ROSE can be used to compile the ROSE source code and has provided a good test of the internal compiler robustness.
7. Overture
   This is an internal DOE library that supports Overset Grid applications. It is well in excess of one million lines of code. It includes the A++/P++ library and other libraries upon which it depends.
8. CHROMA
   This is an Molecular Dynamics application developed at University of Illinois at Urbana-Champaign (UIUC). This is not really a one million line code, I think, but Overture more than makes up the difference.

The first six are mostly done, in the sense that there are about 10 bugs that have been isolated which appear to be the only remaining problems. I am working on these bugs, but some are non-trivial (read *hard*).

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 Version of the EDG License for Research 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, but since ROSE does not yet routinely package a separate binary, we provide the EDG source code as part of the distribution of ROSE. So at present we only give out ROSE to people who 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@edg.com
  - John Spicer jhs@edg.com
  - Daveed Vandevoorde daveed@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 0.9.3a to adapt to many kinds of systems.

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

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

Defaults for the options are specified in brackets.

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

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

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

For better control, use the options below.

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

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

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

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

Optional Features:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
--enable-dot2gml_translator   Configure option to have DOT to GML translator built (bison version specific tool).
  --disable-dependency-tracking  speeds up one-time build
  --enable-dependency-tracking   do not reject slow dependency extractors
  --with-ROSE_LONG_MAKE_CHECK_RULE=yes  specify longer internal testsing by "make check" rule)
  --disable-xmltest       Do not try to compile and run a test LIBXML program
  --disable-binary-analysis-tests
                          Disable tests of ROSE binary analysis code
  --enable-edg_union_struct_debugging   Specify if EDG Union/Struct debugging support is to be used
$with_Mesa_help_string
  --enable-doxygen-developer-docs ..... Enable display of internal project detail with Doxygen.
  --enable-doxygen-generate-fast-docs ..... Enable faster generation of Doxygen documents using
                                        tag file mechanism to connect Sage III documentation to
                                        Rose documentation (documentation not as presentable).
  --enable-static[=PKGS]
                          build static libraries [default=no]
  --enable-ltdl-install   install libltdl
  --enable-shared[=PKGS]
```

Figure 2.1: Example output from configure –help in `ROSE` directory (Part 1).

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

```
                          build shared libraries [default=yes]
  --enable-fast-install[=PKGS]
                          optimize for fast installation [default=yes]
  --disable-libtool-lock  avoid locking (might break parallel builds)
  --enable-purify-api ..................... Enable purify API in code.
  --enable-purify-linker ..................... Augment the linker with purify.
  --enable-purify-windows ..... turn on use of PURIFY windows option
  --enable-insure ..................... Augment the linker with insure.
--enable-dq-developer-tests   Development option for Dan Quinlan (disregard).
  --enable-purify use memory managment that purify can understand
  --disable-checking don't do consistency checking in parser
  --enable-stand-alone compile standalone edgcpfe
  --enable-cp-backend generate c++ code as output
  --enable-sage-backend generate sage++ tree
  --enable-rosehpct       enable build of the ROSE-HPCT module
  --enable-assembly-semantics
                          Enable semantics-based analysis of assembly code

Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-boost[=DIR]
                          use boost (default is yes) - it is possible to
                          specify the root directory for boost (optional)
  --with-boost-libdir=LIB_DIR
                          Force given directory for boost libraries. Note that
                          this will overwrite library path detection, so use
                          this parameter only if default library detection
                          fails and you know exactly where your boost
                          libraries are located.
  --with-boost-thread[=special-lib]
                          use the Thread library from boost - it is possible
                          to specify a certain library for the linker e.g.
                          --with-boost-thread=boost_thread-gcc-mt
  --with-boost-date-time[=special-lib]
                          use the Date_Time library from boost - it is
                          possible to specify a certain library for the linker
                          e.g.
                          --with-boost-date-time=boost_date_time-gcc-mt-d-1_33_1
  --with-boost-regex[=special-lib]
                          use the Regex library from boost - it is possible to
                          specify a certain library for the linker e.g.
                          --with-boost-regex=boost_regex-gcc-mt-d-1_33_1
  --with-boost-program-options[=special-lib]
                          use the program options library from boost - it is
                          possible to specify a certain library for the linker
                          e.g.
                          --with-boost-program-options=boost_program_options-gcc-mt-1_33_1
  --with-boost-serialization[=special-lib]
                          use the Serialization library from boost - it is
                          possible to specify a certain library for the linker
                          e.g.
                          --with-boost-serialization=boost_serialization-gcc-mt-d-1_33_1
  --with-boost-system[=special-lib]
                          use the System library from boost - it is possible
                          to specify a certain library for the linker e.g.
                          --with-boost-system=boost_system-gcc-mt
  --with-boost-filesystem[=special-lib]
                          use the Filesystem library from boost - it is
                          possible to specify a certain library for the linker
                          e.g. --with-boost-filesystem=boost_filesystem-gcc-mt
  --with-boost-wave[=special-lib]
                          use the Wave library from boost - it is possible to
                          specify a certain library for the linker e.g.
                          --with-boost-wave=boost_wave-gcc-mt-d-1_33_1
  --with-sqlite3=[ARG]
                          use SQLite 3 library [default=yes], optionally
                          specify the prefix for sqlite3 library
  --with-mysql=[ARG]
                          use MySQL client library [default=yes],
                          optionally specify path to mysql_config
  --with-alternate_backend_Cxx_compiler=<compiler name>
                              Specify an alternative C++ back-end compiler
  --with-alternate_backend_C_compiler=<compiler name>
                              Specify an alternative C back-end compiler
  --with-alternate_backend_fortran_compiler=<compiler name>
                              Specify an alternative fortran back-end compiler
  --with-xml-prefix=PFX   Prefix where libxml is installed (optional)
  --with-xml-exec-prefix=PFX Exec prefix where libxml is installed (optional)
  --with-valgrind ... Run uninitialized field tests that use Valgrind
```

Figure 2.2: Example output from configure –help in `ROSE` directory (Part 2).

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[] )
   {
  // Build the AST used by ROSE
     SgProject* sageProject = frontend(argc,argv);

  // Run internal consistancy tests on AST
     AstTests::runAllTests(sageProject);

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

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

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 exampleT

```
# 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 = /home/liao6/daily-test-rose/20081014_120001/install/include

# Location of Boost include directory
BOOST_CPPFLAGS = -pthread -I/home/liao6/opt/boost_1_35_0/include

# Location of library directory after "make install"
ROSE_LIB_DIR = /home/liao6/daily-test-rose/20081014_120001/install/lib

ROSE_LIBS = $(ROSE_LIB_DIR)/librose.la

ROSE_SOURCE_DIR = ../../../../sourcetree/exampleTranslators/documentedExamples/simpleT

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

# 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:
        ../../../libtool --mode=compile g++ -I$(ROSE_INCLUDE_DIR) $(BOOST_CPPFLAGS) -c

exampleTranslator: exampleTranslator.lo
        ../../../libtool --mode=link g++ -o exampleTranslator exampleTranslator.lo $(L

# 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

```
ROSE (pre-release alpha version: 0.9.3a)

This ROSE translator provides a means for operating on C, C++, and Fortran
source code, as well as on x86 and ARM object code.

Usage: rose [OPTION]... FILENAME...

If a long option shows a mandatory argument, it is mandatory for the equivalent
short option as well, and similarly for optional arguments.

Main operation mode:
     -rose:(o|output) FILENAME
                            file containing final unparsed C++ code
                            (relative or absolute paths are supported)

Operation modifiers:
     -rose:output_warnings   compile with warnings mode on
     -rose:C_only, -rose:C   follow C89 standard, disable C++
     -rose:C99_only, -rose:C99
                            follow C99 standard, disable C++
     -rose:Cxx_only, -rose:Cxx
                            follow C++ 89 standard
     -rose:UPC_only, -rose:UPC
                            follow Unified Parallel C 1.2 specification
     -rose:upc_threads n    Enable UPC static threads compilation with n threads
                            n>=1: static threads; dynamic(default) otherwise
     -rose:Fortran, -rose:F, -rose:f
                            compile Fortran code, determining version of
                            Fortran from file suffix)
     -rose:Fortran2003, -rose:F2003, -rose:f2003
                            compile Fortran 2003 code
     -rose:Fortran95, -rose:F95, -rose:f95
                            compile Fortran 95 code
     -rose:Fortran90, -rose:F90, -rose:f90
                            compile Fortran 90 code
     -rose:Fortran77, -rose:F77, -rose:f77
                            compile Fortran 77 code
     -rose:Fortran66, -rose:F66, -rose:f66
                            compile Fortran 66 code
     -rose:FortranIV, -rose:FIV, -rose:fIV
                            compile Fortran IV code
     -rose:FortranII, -rose:FII, -rose:fII
                            compile Fortran II code (not implemented yet)
     -rose:FortranI, -rose:FI, -rose:fI
                            compile Fortran I code (not implemented yet)
     -rose:strict           strict enforcement of ANSI/ISO standards
     -rose:astMerge         merge ASTs from different files
     -rose:astMergeCommandFile FILE
                            filename where compiler command lines are stored
                              for later processing (using AST merge mechanism)
     -rose:compilationPerformanceFile FILE
                            filename where compiler performance for internal
                              phases (in CSV form) is placed for later
```

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`

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

IE: *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 effects 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 automating 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):

[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).

```
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. 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
//std::list<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 std::list<SgNode*>'s and returns a
//std::list<SgNode*>s **//

  queryNodeList( NodeQuerySynthesizedAttributeType,
     TypeOfQueryTypeOneParameter elementReturnType)

  queryNodeList( std::list<SgNode*>, roseFunctionPointerOneParameter )

  queryNodeList( std::list<SgNode*>, SgNode*, roseFunctionPointerTwoParameters)

  queryNodeList( std::list<SgNode*>, SgNode*, TypeOfQueryTypeTwoParameters)

  queryNodeList( std::list<SgNode*>, VariantT)

  queryNodeList( std::list<SgNode*>, VariantVector*)

//** Functions that visit only the nodes in the memory pool that is
//specified in a VariantVector and returns a std::list<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 std::list¡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 std::list¡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
};
}
```

Chapter 7

AST Processing

7.1 Introduction

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.

FIXME: *This cha*
cover both
Object-Oriented Visi
Traversal (which h
build, 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 unit
classic Object-Orien
Pattern Traversal n
memory pools as a p
(only the code need b

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

```
#include "rose.h"

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

Figure 7.1: Headerfile *MyVisitor.h*.

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.

```
#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.

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).

```
#include "rose.h"

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*.

```
#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*.

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.

```
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 defaultSynthesizedAttribute.
- 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_loop_body};
```

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:

```
class MyAstAttribute : public AstAttribute {
   public:
   MyAstAttribute(int v):value(v) {}
      ...
   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 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`.

7.11 Visualization

7.11.1 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`).

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

Figure 7.7: Example program used as running example

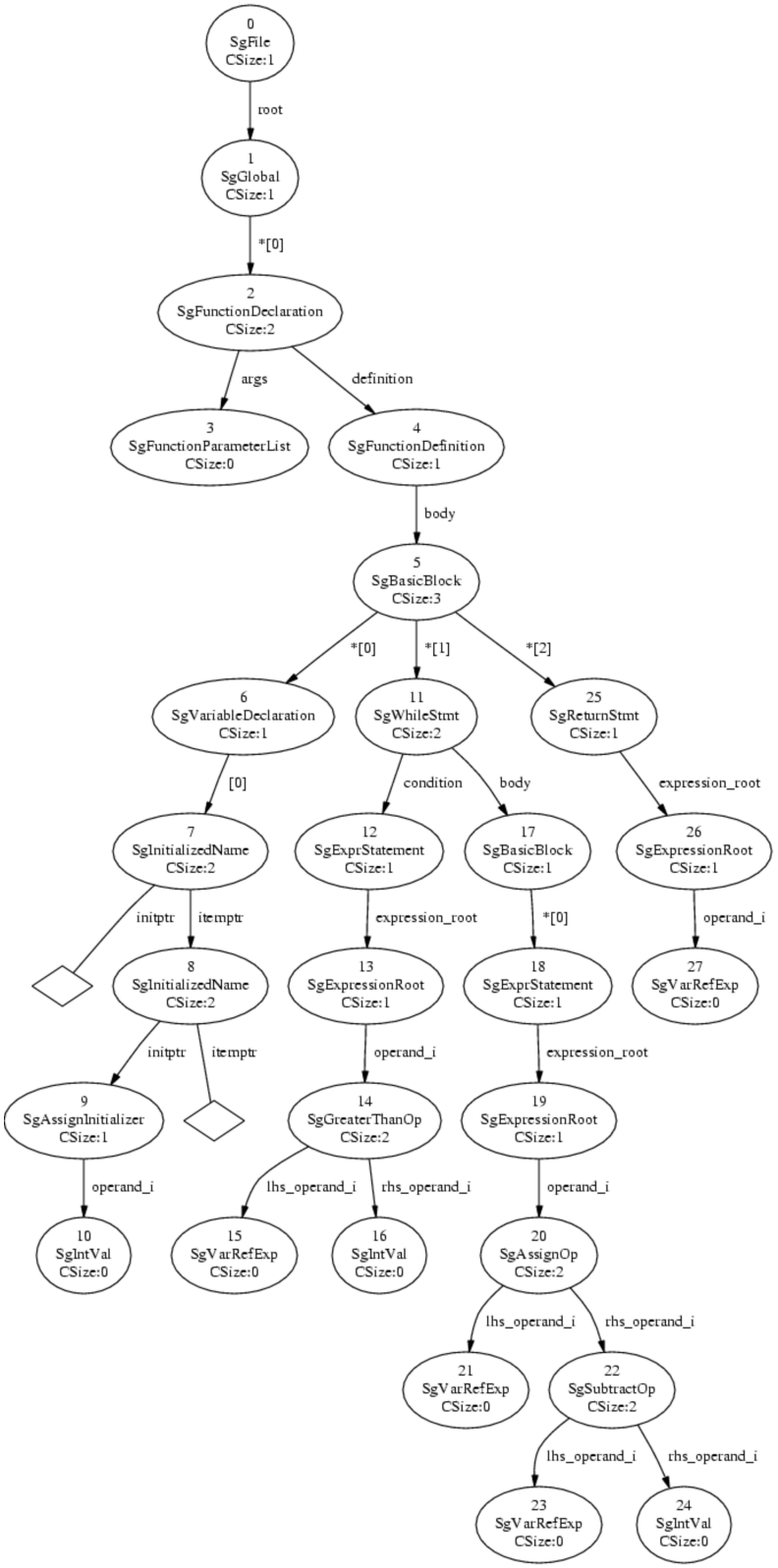

Figure 7.8: Numbers at nodes show the order in which the visit function is called in a preorder traversal

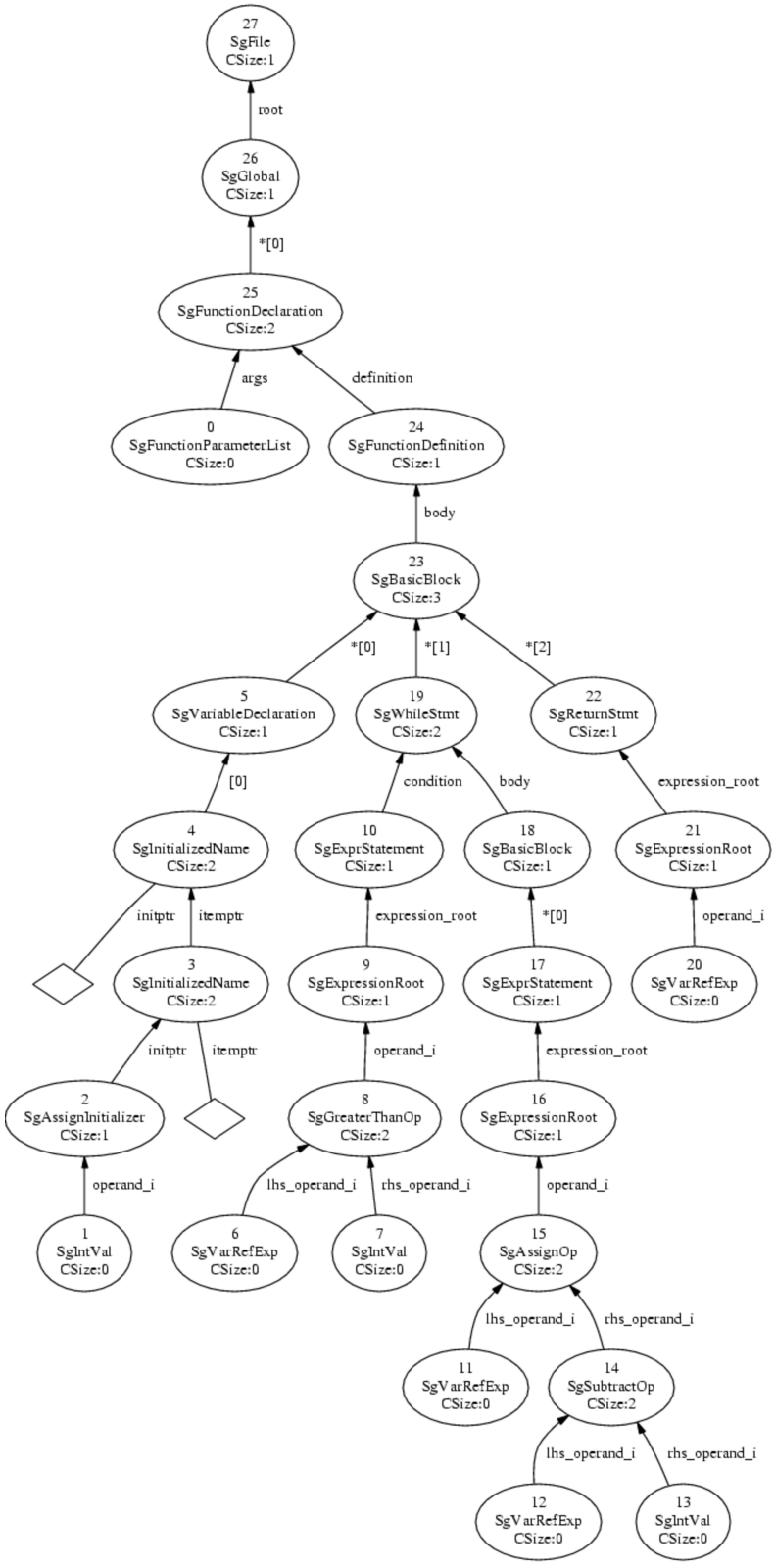

Figure 7.9: Numbers at nodes show the order in which the visit function is called in a postorder traversal

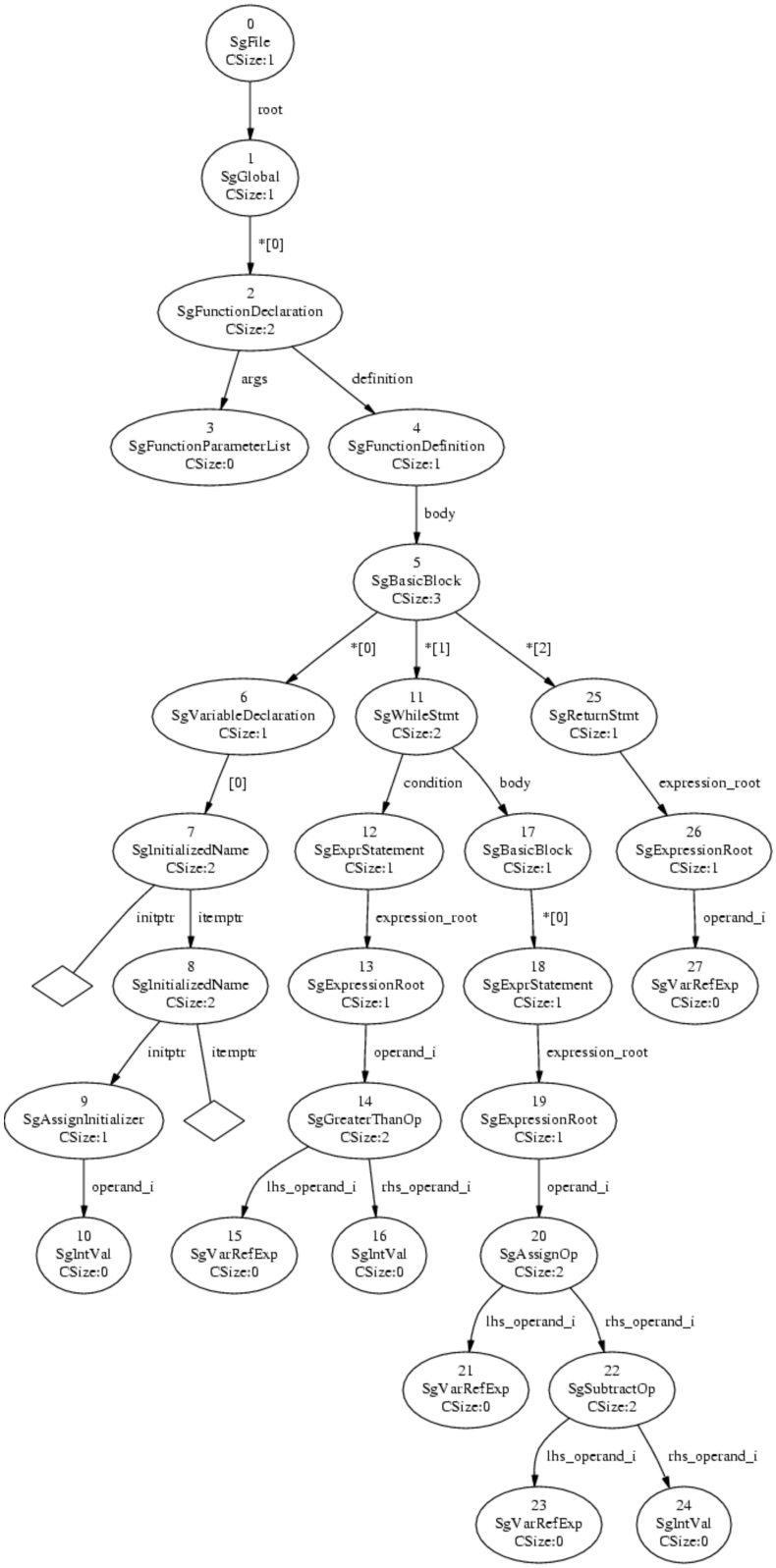

Figure 7.10: Numbers at nodes show the order in which the function evaluateInheritedAttribute is called in a top-down processing

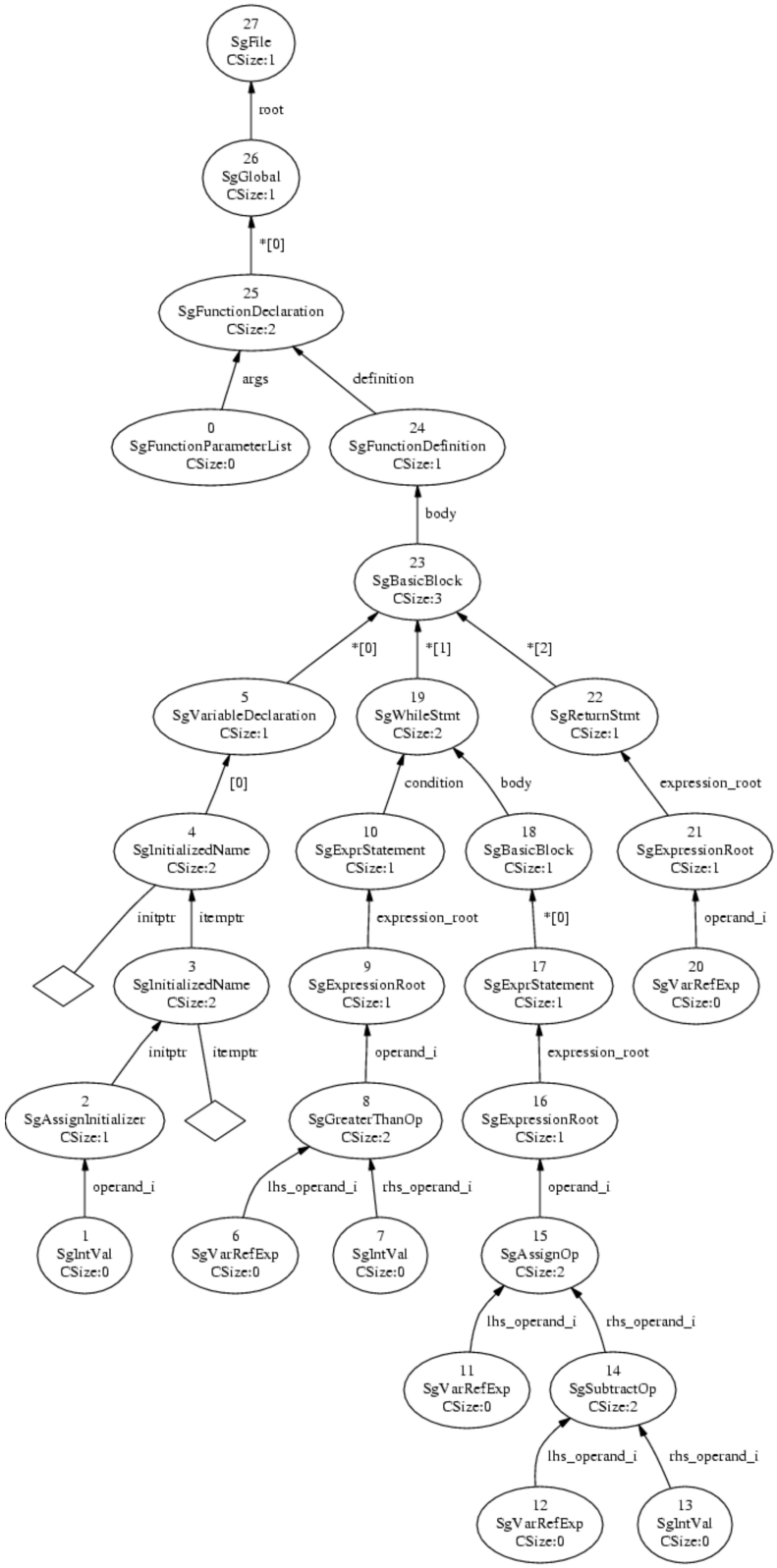

Figure 7.11: Numbers at nodes show the order in which the function evaluateSynthesizedAttribute is called in a bottom up processing

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.

Chapter 8

AST Rewrite Mechanism

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, operations 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

<table>
<tr><td rowspan="1">Relative Positioning
(contains state)</td><td>String-
Based</td><td>High Level Interface
(level 4)</td><td>insert(SgNode*,string,scope,location)
replace(SgNode*,string,scope,location)
remove(SgNode*)</td></tr>
<tr><td rowspan="3">Absolute Positioning
(contains no state)</td><td>String-
Based</td><td>Mid Level Interface
(level 3)</td><td>insert(SgNode*,string,location)
replace(SgNode*,string,location)
remove(SgNode*)</td></tr>
<tr><td rowspan="2">SgNode*</td><td>Low Level Interface
(level 2)</td><td>insert(SgNode*,SgNode*)
replace()
remove(SgNode*)</td></tr>
<tr><td>SAGE III Interface
(level 1)</td><td>insert(SgNode*,SgNode*)
replace(SgNode*,SgNode*)
remove(SgNode*)</td></tr>
</table>

Table 8.1: Different levels of the ROSE Rewrite mechanism.

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 **AstTopDownBottomUpProcessing** 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 "rewriteMidLevelInterfaceTemplatesImpl.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

     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/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.

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.

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 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.

9.1.6 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.

FIXME: *pro*

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 "GlobalDatabaseConnection.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[]) {

        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 "GlobalDatabaseConnection.h"
#include "TableDefinitions.h"
DEFINE_TABLE_PROJECTS();
DEFINE_TABLE_FILES();

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

        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 lin
        string fileName    = "testFile.C";   // this should be retrieved from a SgFile

        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 "GlobalDatabaseConnection.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[]) {

        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, EdgeTypeEmpty>( projectId , GTYPE_TESTGRAPH,
        graph->loadFromDatabase( );

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

Simple Callgaph Example

```
#include <iostream>
#include "GlobalDatabaseConnection.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[]) {

        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 lin
        string fileName    = "testFile.C";   // this should be retrieved from a SgFile

        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 =
```

Chapter 10

Loop Transformations

10.1 Introduction

The loop transformation package implements all the algorithms published by Yi and Kennedy [34, 37, 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:

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

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.

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

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 [37]), 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 $N^2 + L^2D$, 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

Binary Analysis: Support for the Analysis of Binary Executables

12.1 Introduction

ROSE supports the disassembly and analysis of binary executables for x86 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.

12.2 The Binary AST

12.2.1 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 or Windows, OS2, or DOS binary.

The binary executable file format can be analyized seperately from the instructions using the command line option: `-rose:read_executable_file_format_only`. this allows graphs generated using the ROSE visualization mechanisms to be easily restricted (in size) to the just the IR nodes specific to the binary executable file format.

FIXME: *We need an*
the binary execut

12.2.2 Instruction Disassembly

ROSE has its own disassembler; 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 roperands and operator expression trees, etc. are stored in the binary AST as seperate IR nodes.

FIXME: *We need an*
the AST for a few i

12.3 Binary Analysis

A number of binary analysis passes are provided, most are a part of the Compass framework for software analysis. See the Compass 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.

12.4 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 reletively 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.

12.5 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 rewritting of the binary permits the development of performance tools that could support the analysis and rewritting of binaries for support of High Performance Computing (HPC). As principal focus in on IBM BGL and Cray XT support (DOE Office of Science supercomputers).

12.5.1 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 aligment 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.

12.5.2 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:

SgAsmFileHeader *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.

FIXME: If the byte order is not specified in the ELF header (e_ident_data_encoding other than 1 or 2) then the parser will make an educated guess and assign a byte order. The unparsed file will differ from the original in this case at the 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:

SgAsmFileHeader *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: Increasing word size probably requires allocating more space for many of the sections. Vice versa for decreasing the word size.

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 SgAsmFileHeader::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_type should probably call set_purpose, but we can't go the other direction because the mapping is N:1.

6. ELF Version

To change the ELF version assign a new value by calling set_version on the object returned by SgAsmGenericHeader::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 (FIXME: Remove these function). 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.

12.5.3 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.

12.5.4 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

DQ: *What text should go here?*

- SgAsmElfSectionTableEntry::set_sh_type
- SgAsmElfSectionTableEntry::set_sh_flags, although the Write and Execute bits are ignored
- SgAsmElfSectionTableEntry::set_sh_info ???
- SgAsmElfSectionTableEntry::set_sh_addralign ???
- SgAsmElfSectionTableEntry::set_sh_entsize ???

12.6 Usage

See the ROSE Tutorial for examples.

Chapter 13

ROSE Tests

13.1 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/roseTests/README for details.

E: *Complete the list of ries that hold tests (in ROSE/tests directory).*

Chapter 14

Testing Within ROSE

```
# 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
```

```
 67   fi
 68 ;; # case C/C++ source files
 69
 70 -o)
 71   if [[ $MODE = 'o' ]]; then
 72     FLAG=1
 73   elif [[ $MODE = 'f' ]]; then
 74     FLAG=2
 75   fi # spike out the object flag
 76
 77   continue
 78 ;; # case -o
 79
 80 *) ;; # default
 81     esac
 82
 83     #esac#fi#############################################################################
 84
 85
 86     if ((FLAG > 0)); then
 87       OFILE=$ARG
 88     fi # name the object file after -o declaration
 89
 90     if ((FLAG == 1)); then
 91       if [[ ${ARG:0:1} != '/' ]]; then
 92         ARG="`pwd`/$ARG"
 93       fi # if argument not specified with absolute path then append PWD
 94
 95       # rename the QMTest output test file, replace space, period, and plus
 96       # with equivalent symbols and change all chars to lower case.
 97       TEST=`echo $ARG | sed -e 's/\//_/g' | sed -e 's/\./_dot_/g' | \
 98     sed -e 's/+/plus/g' | gawk '{print tolower($0)}'`.qmt
 99
100       FLAG=0# reset FLAG
101       continue
102     elif ((FLAG == 2)); then
103       FLAG=0# reset FLAG
104       continue
105     fi # if the -o flag used; create the object name and TEST name from object
106
107     #fi#if#############################################################################
108
109
110     ARGUMENTS="${ARGUMENTS} '$ARG', "
111   fi # if argument is not qm.sh argument
112
113   #fi#done#############################################################################
114
115
116 done # for all command-line arguments to qm.sh
117
118 OBJECT=${TEST%%.*}.o# name the object after the test file name
119 ARGUMENTS="${ARGUMENTS} '-o', '$OBJECT']"
120
121 #done#case#############################################################################
122
123
124 case $TEST_CLASS in
125   strings.SubStringTest) qmtest create -o $TEST -a program="$PROGRAM" -a substring="ERROR SUMMARY: 0 errors from 0 contexts"
-a arguments="$ARGUMENTS" test $TEST_CLASS;;
126
127   *) qmtest create -o "$TEST" -a program="$PROGRAM" -a arguments="$ARGUMENTS" test $TEST_CLASS;;
128 esac # create qmtest test file with test class
129
130 #esac#main#############################################################################
131
132
133 if [[ ${BACKEND:0:4} == "NULL" ]]; then
134   touch $OFILE >& /dev/null # create dummy file and pipe error to NULL
135   exit 0    # always exit 0
136 fi # skip backend compilation
137
138 # Execute backend compilation with original compile-line
139 $BACKEND
140 exit $?
```

14.1 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.

14.2 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.

14.3 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']"`.

14.4 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.

14.4.1 Backend and ROSE arguments

```
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 14.1: Backend and ROSE argument construction block

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.

14.4.2 Relative Path Compile-line Arguments

```
1    case $ARG in
2        -I*) includeFullPath;;

3        [!/]*.c | [!/]*.cpp | [!/]*.C | [!/]*.[cC]* ) ARG="`pwd`/$ARG";;

4        -o) BOOL=1 ; continue;; # spike out -o outputfilename

5        *) ;;
6    esac
```

Figure 14.2: 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.

14.4.3 Naming QMTest Files

```
1      if [[ ${ARG:0:1} != '/' ]]; then
2        ARG=`pwd`/$ARG
3      fi

4      TEST=`echo $ARG | sed -e 's/\//_/g' | sed -e 's/\./_/g' | gawk '{print tolower($0)}'`.qmt

5      OBJECT=${TEST%%.*}.o
```

Figure 14.3: Naming procedure for 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 `'.'`(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.

14.4.4 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 14.4: 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.

14.5 Example

The following example edits a trivial makefile and builds QMTest files with `qm.sh` by editing the makefile.

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:

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.

14.6 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

```
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 14.5: makefile before editing

```
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 14.6: makefile after editing

```
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 14.7: `make` output

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

```
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 14.8: `find . -name "*.qmt"` output

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 15

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*

15.1 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.

15.2 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

15.3 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`)

15.4 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).

15.5 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
```

15.6 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 16

Developer's Appendix

16.1 Building ROSE from the Source Code Repository Checkout *(for developers only)*

The instructions for building ROSE from SVN are a little more complex. A few GNU software build tools are required (not required for the user *ROSE Distribution, e.g. ROSE-0.9.3a.tar.gz*. Required tools (**** Note current dependencies**):

- autoconf
  Autoconf version 2.53 or higher is required. Newer versions of Autoconf introduce experimental features that could also be problematic. The Autoconf development has not been particularly good at verifying compatibility with previous releases of their work. Some users have reported having to install version 2.53 specifically to use ROSE. *Check the ROSE/ChangeLog for current version numbers to be used with ROSE.*

- automake
  Automake version 1.9 or higher is required. Most software projects appear to be less sensitive to the specific version of automake. *Check the ROSE/ChangeLog for current version numbers to be used with ROSE.*

The `ROSE/ChangeLog` details the changes between versions of ROSE and lists the specific version numbers of all software upon which ROSE depends. Comments of this type appear in the `ChangeLog` as: **FIXME:** *This*

```
********* TESTED with **************
(*)  automake (GNU automake) 1.6.3
(*)  autoconf (GNU Autoconf) 2.57
(*)  GNU Make version 3.79.1
(**) g++ (GCC) 3.3.6
(**) gcc (GCC) 3.3.6
(*)  doxygen 1.3.8
(*)  dot version 1.12 (Sun Aug 15 02:43:07 UTC 2004)
(*)  TeX (Web2C 7.3.1) 3.14159
(*)  Original LaTeX2HTML Version 2002 (1.62)
(*)  sqlite (requires g++ 3.3.2) 3.2.1

(*)  Optional for use of ROSE (by users), but required for internal ROSE development (by ROSE project team)
(**) Required for use of ROSE (and for all internal development)
```

The build process for a *Developer Version* is:

1. Checkout a NEW version from SVN:
   The newest work on ROSE (as of March 2008) is using SVN, instead of CVS. This switch to SVN means the directions for how developers use ROSE have changed. This effects developers of ROSE only (or anyone with access to the newer SVN repository). **Please learn about SVN on another project before using it on ours.**

   To checkout ROSE (assumes access to repository at LLNL), type:
   `svn checkout file:///usr/casc/overture/ROSE/svn/ROSE/trunk/ROSE svn-rose`
   This will checkout a copy of the source code for ROSE from the svn repository. Any directory name can be used for `svn-rose` in the example commandline.

2. Update an *existing* version from SVN:
   Run `svn update` from inside the ROSE directory (at the top level) to update an existing version of ROSE with the new changes in the SVN repository.

3. After being checked out (or updated) from SVN:
   Run the `build` script in the top level ROSE directory to build all configure scripts and `Makefile.in` files (using **automake**). This is the difference between the development environment and the distribution. This script will call the different **autoconf** tools required to setup ROSE and also checkout other work common to multiple projects within CASC.

4. Build a compile directory (for the compile tree):
   Make a separate directory to be the root of the compile tree. There can be many compile trees if you want.

   *Note: Before the next step be sure you are using the correct compiler (***g++** *C++ compiler [see ChangLog file for current version used for development, generally any 3.x version]) and that you are using the correct version of* **autoconf** *and* **automake**.

5. Running configure:
   Type `<pathToSourceTree>/configure --help` to see the different configuration options. `<pathToSourceTree>` is meant to be the absolute or relative path to the source tree where the **SVN** version was checked out. After options have been selected, type `<pathToSourceTree>/configure <selected-options>` to run the configure script. Running the configure script with no options is sufficient (uses default values which are either already set or which the configure script will figure out on your machine). For more on ROSE configure options, see 2.2.4.

6. Running Make after running configure:
   After configuration (after the configure script is finished) run `make` or `gmake`. If you have a development version then you can also make distributions by running `make dist`. If you want to build a new distribution *AND* test it, run `make distcheck` (make or gmake may be used interchangeably). See details of running make in parallel 2.2.5.

7. Testing your new version of ROSE:
   Automated tests are available within the distribution of ROSE. To run these tests, type `make check`. Tests on a modern Intel/Linux machine currently take about 15 minutes to run.

8. Installing ROSE:
   From this point you can generate ROSE the way a user would see it (as if you had started with a *ROSE Distribution*). Type `make install` to install ROSE. See details of installing ROSE 2.2.6.

9. Testing the installed version of ROSE:
   To test the installed version of ROSE type `make installcheck`. To test compilation, this forces one or more of the Example translators to be built using only the header files from the {`install_dir`}`/include` directory for their compilation. To test linking ROSE translators forces the previously compiled example translator to only use the libraries installed in {`install_dir`}`/lib`. This is sufficient to test the installation the way that users are expected to use ROSE (only from an installed version). A sample `makefile` is generated, see 3.3.

16.2 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.

16.3 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.*

16.4 Check In Process

NOTE: Get permission from the ROSE Development Team before you make your first check-in!

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-0.9.3a.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 tests.

   - 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. svn commit -m "¡description of what you did¿".

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!

16.5 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.

Need to cover the new Fortran support.

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:

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. 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 ( ")");
   }
```

16.6 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`.

16.7 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;
```

16.8 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/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.

16.9 Things That May Happen to Your Code After You Leave

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 will 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.

16.10 Maintaining the ROSE Email List (casc-rose@llnl.gov)

There is an open email list for ROSE which can be subscribed to automatically. The list name is: **casc-rose**.

These are the email commands available to users of the list. To use them, a user sends a message to Majordomo with one or more of these commands in the body of the message. Each mailing list has a special "request" address where commands can be sent. For example, to use the casc-rose mailing list (casc-rose@lists.llnl.gov), send commands to casc-rose-request@lists.llnl.gov.

It is also possible to send commands directly to majordomo@lists.llnl.gov. However, be sure to specify which list you want to use. With all the commands below, you can leave out list if you are sending to casc-rose-request@lists.llnl.gov.

- subscribe list address
  Subscribe yourself (or address if specified) to the named list. The list may be configured so that you can only subscribe yourself; ie. you can't specify an address other than your own.

- Unsubscribe list address
  Unsubscribe yourself (or address if specified) from the named list. "unsubscribe *" will remove you (or address) from all lists; This may not work if you have subscribed using multiple addresses. The list may be configured so that you can only unsubscribe yourself; ie. you can't specify an address other than your own.

- which address
  Find out which lists you (or address if specified) are on. Only lists enabled to supply this information will be returned to the requester.

- who list
  Find out who is on the named list. Only lists enabled to supply this information will be returned to the requester.

- info list
  Retrieve the general introductory information for the named list. Only lists enabled to supply this information will be returned to the requester.

- intro list
  Retrieve the introductory message sent to new users. Non-subscribers may not be able to retrieve this.

- lists
  Show the lists served by this Majordomo server (will not show "private" lists).

- help
  Retrieve some help information on the available user commands.

- end
  Stop processing commands (useful if your email program adds a signature).

Here are the URLs for the *casc-rose* email list:
Instructions on how to use a Majordomo mailing list:

```
https://lists.llnl.gov/mj/user-commands.html
```

Web interface for modifying a Majordomo mailing list:

```
https://lists.llnl.gov/majordomo.
```

Details:

1. List name is: *casc-rose* not *casc-rose@llnl.gov*.

2. Must be on site at LLNL.

16.11 How To Build a Binary Distribution

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 licence. 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).

There are a few steps:

1. Configure and build ROSE normally, using configure (use all options that you require in the binary distribution).
2. (optional) Run `make dist`, this will build an *all source distribution* of ROSE.
3. Rerun configure without the `--with-edg_source_code=true` option.
4. Run `make dist`, this will build a binary distribution using the binary libraries build in step one.

16.12 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

Thanks for saving a number of people a lot of work.

16.13 Setting Up Nightly Tests

Directions for using 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.

16.14 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.

16.15 Binary Analysis

The documentation for the binary analysis can be found in the ROSE manual at 12. However, there are a collection of details that we need to document about the design; so for how 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.

16.15.1 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).

ME: *We should get a for the details of what ols are left in stripped and what symbols are ed to support dynamic where they are stored.*

- 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.

Chapter 17

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).

E: 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 18

Glossary

We define terms used in the ROSE manual which might otherwise be unclear.

FIXME: *Define th terms: IR nod Attribute, Synthesize Accumulator Attr*

- **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] Q. Yi and K. Kennedy. Transforming complex loop nests for locality. Technical Report TR02-386, Computer Science Dept., Rice University, Feb. 2002.