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

Draft Tutorial

(version 0.9.5a)

Daniel Quinlan, Markus Schordan, Richard Vuduc, Qing Yi
Thomas Panas, Chunhua Liao, and Jeremiah J. Willcock
Lawrence Livermore National Laboratory
Livermore, CA 94550
925-423-2668 (office)
925-422-6278 (fax)
{dquinlan,panas2,liao6}@llnl.gov
markus@complang.tuwien.ac.at
qingyi@cs.utsa.edu
richie@cc.gatech.edu
jewillco@osl.iu.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)

January 20, 2013

January 20, 2013

Contents

List of Figures

Chapter 1

Introduction

1.1 What is ROSE

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

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

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

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

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

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

1.2 Why you should be interested in ROSE

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

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

1.3 Problems that ROSE can address

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

- OpenMP translator,
- Array class abstraction optimizer,
- Source-to-source instrumenter,
- Loop analyzer,
- Symbolic complexity analyzer,

- Inliner and outliner,
- Code coverage tools,
- and many more...

Example tools that *can* be built include:

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

1.4 Examples in this ROSE Tutorial

This tutorial lays out a set of progressively complex example programs (located in <ROSE_SOURCE>/tutorial/*) that serve as a tutorial for the use of ROSE. Translators built using ROSE can either just analyze (and output results) or compile the input programs just like a compiler (generating object files or executables). Many of the examples in this tutorial just do simple analysis of the input source code, and a few show the full compilation of the input source code. Where the translators generate either object files of executables, the vendor's compiler is used to compile the final ROSE-generated code. Within ROSE, the call to generate source code from the AST and call the vendor's compiler is referred to as the *backend processing*. The specification of the vendor's compiler as a backend is done within the configuration step within ROSE (see options for `configure` in the ROSE User Manual).

Within the example programs below, the user can provide alternative input programs for more complex evaluation of the tutorial examples and ROSE. The end of the chapter, section 1.7, shows the makefiles required to compile the tutorial programs using an installed version of ROSE (compiled using `make install`). This `example_makefile` is run as part of the testing using the `make installcheck` rule.

Chapters are organized in topics including simple ROSE AST visualization, dealing with complex data types, program analysis, program transformation and optimization, correctness checking, binary support, interacting with other tools, and parallelism. We hope readers can easily find the information they want.

Specific chapters in this tutorial include:

FIXME: *We should constantly update this*

- Introduction
  1. Introduction *(this chapter)*
  2. Problems that ROSE can address
  3. Getting Started
     This chapter covers where to find ROSE documentation and how to install ROSE.

4. Example Makefiles demonstrating the command lines to compile and link the example translators in this tutorial are found in <ROSE_Compile_Tree>/tutorial/exampleMakefile.

- Working with the ROSE AST:

  1. Identity Translator
     This example translator reads a C or C++ application, builds the AST internally, generates source code from the AST (unparsing), and calls the backend vendor compiler to compile the generated C or C++ application code. Thus the translator acts like and can be used to replace any compiler since it takes in source code and outputs an object code (or executable). This example also shows that the output generated from and ROSE translator is a close reproduction of the input; preserving all comments, preprocessor control structure, and most formating.
  2. Scopes of Declarations (scopeInformation.C)
     This example shows the scopes represented by different IR nodes in the AST.
  3. AST Graph Generator
     This translator reads a C or C++ application code and builds the AST, internally. The translator does not regenerate code from the AST and so does not call the backend vendor's compiler. This shows how simple it could be to build source code analysis tools; the code calls an internal ROSE function to generate a dot graph of the AST, the makefile has the details of converting the dot graph into a postscript file (also shown).
  4. AST PDF Generator
     This translator reads an C or C++ application code builds the AST internally. The translator does not regenerate code from the AST and so does not call the backend vendor's compiler. This shows how simple it could be to build source code analysis tools, the code calls an internal ROSE function to generate a `pdf` file with bookmarks representing the AST. The pdf file show as output is in this case a previously generated figure of a screen shot obtained by viewing the output pdf file using `acroread`.
  5. Introduction to AST Traversals and Attributes
     This collection of examples show the use of the simple visitor pattern for the traversal of the AST within ROSE. The simple visitor pattern permits operations to be programmed which will be invoked on different nodes within the AST. To handle communication of context information down into the AST and permit communication of analysis information up the AST, we have provided inherited and synthesized attributes (respectively). Note that an AST is most often represented as a tree with extra edges and with shared IR nodes that make the full graph (representing all edges) not a tree. We present two styles of traversal, one over the tree representing the AST (which excludes some types of IR nodes) and one over the full AST with all extra nodes and shared nodes. Extra nodes are nodes such as SgType and SgSymbol IR nodes.

     (a) AST traversals
         These traversals visit each node of the tree embedded within the AST (excluding shared SgType and SgSymbol IR nodes). These traversals visit the IR nodes is

an order dependent upon the structure of the AST (the source code from which the AST is built).

i. Classic Object-Oriented Visitor Patterns
This example, classicObjectOrientedVisitorPatternMemoryPoolTraversal.C, show the use of a classic visitor patterns. At the moment this example uses the AST's memory pools as a basis but it is identical to a future traversal. The ROSE visitor Pattern (below) is generally more useful. The classic visitor pattern traversals are provided for completeness.

ii. Visitor Traversal (visitorTraversal.C)
Conventional visitor patterns without no attributes. This pattern can explicitly access global variables to provide the effect of accumulator attributes (using static data members we later show the handling of accumulator attributes).

iii. Inherited Attributes (inheritedAttributeTraversal.C)
Inherited attributes are used to communicate the context of any location within the AST in terms of other parent AST nodes.

iv. Synthesized Attributes (synthesizedAttributeTraversal.C)
Synthesized attributes are used to pass analysis results from the leaves of the AST to the parents (all the way to the root of the AST if required).

v. Accumulator Attributes (accumulatorAttributeTraversal.C)
Accumulator attributes permit the interaction of data within inherited attributes with data in synthesized attributes. In our example program we will show the use of accumulator attributes implemented as static data members. Accumulator attributes are a fancy name for what is essentially global variables (or equivalently a data structure passed by reference to all the IR nodes in the AST).

vi. Inherited and Synthesized Attributes (inheritedAndSynthesizedAttributeTraversal.C)
The combination of using inherited and synthesized attributes permits more complex analysis and is often required to compute analysis results on the AST within a specific context (e.g. number of loop nests of specific depth).

vii. Persistent Attributes (persistantAttributes.C)
Persistent attributes may be added the AST for access to stored results for later traversals of the AST. The user controls the lifetime of these persistent attributes.

viii. Nested traversals
Complex operations upon the AST can require many subordinate operations. Such subordinate operations can be accommodated using nested traversals. All traversals can operate on any subtree of the AST, and may even be nested arbitrarily. Interestingly, ROSE traversals may also be applied recursively (though care should be take using recursive traversals using accumulator attributes to avoid *over* accumulation).

(b) Memory Pool traversals
These traversals visit all IR nodes (including shared IR nodes such as SgTypes and SgSymbols). By design this traversal can visit ALL IR nodes without the worry

of getting into cycles. These traversals are mostly useful for building specialized tools that operate on the AST.

i. Visit Traversal on Memory Pools
This is a similar traversal as to the Visitor Traversal over the tree in the AST.

ii. Classic Object-Oriented Visitor Pattern on Memory Pools
This is similar to the Classic Object-Oriented Visitor Pattern on the AST.

iii. IR node Type Traversal on Memory Pools
This is a specialized traversal which visits each type of IR node, but one one of each type of IR nodes. This specialized traversal is useful for building tools that call static member functions on each type or IR node. A number of memory based tools for ROSE are built using this traversal.

6. AST Query Library
This example translator shows the use of the AST query library to generate a list of function declarations for any input program (and output the list of function names). It can be trivially modified to return a list of any IR node type (C or C++ language construct).

7. Symbol Table Handling (symbolTableHandling.C)
This example shows how to use the symbol tables held within the AST for each scope.

8. AST File I/O (astFileIO_GenerateBinaryFile.C)
This example demonstrates the file I/O for AST. This is part of ROSE support for whole program analysis.

9. Debugging Tips
There are numerous methods ROSE provides to help debug the development of specialized source-to-source translators. This section shows some of the techniques for getting information from IR nodes and displaying it. Show how to use the PDF generator for AST's. This section may contain several subsections.
   (a) Generating the code representing any IR node
   (b) Displaying the source code position of any IR node

- Complex Types

1. Getting the type parameters in function declaration (functionParameterTypes.C)
This example translator builds a list to record the types used in each function. It shows an example of the sort of type information present within the AST. ROSE specifically maintains all type information.

2. Resolving overloaded functions (resolvingOverloadedFunctions.C – C++ specific)
The AST has all type information pre-evaluated, particularly important for C++ applications where type resolution is required for determining function invocation. This example translator builds a list of functions called within each function, showing that overloaded function are fully resolved within the AST. Thus the user is not required to compute the type resolution required to identify which over loaded functions are called.

3. Getting template parameters to a templated class (templateParameters.C – C++ specific)

All template information is saved within the AST. Templated classes and functions are separately instantiated as specializations, as such they can be transformed separately depending upon their template values. This example code shows the template types used the instantiate a specific templated class.

- Program Analysis

  1. Recognizing loops within applications (loopRecognition.C)
     This example program shows the use of inherited and synthesized attributes form a list of loop nests and report their depth. The inherited attributes are required to record when the traversal is within outer loop and the synthesized attributes are required to pass the list of loop nests back up of the AST.
  2. Generating a CFG (buildCFG.C)
     This example shows the generation of a control flow graph within ROSE. The example is intended to be simple. Many other graphs can be built, we need to show them as well.
  3. Generating a CG (buildCallGraph.C)
     This example shows the generation of a call graph within ROSE.
  4. Generating a CH (classHierarchyGraph.C)
     This example shows the generation of a class hierarchy graph within ROSE.
  5. Building custom graphs of program information
     The mechanisms used internally to build different graphs of program data is also made externally available. This section shows how new graphs of program information can be built or existing graphs customized.
  6. Database Support (dataBaseUsage.C)
     This example shows how to use the optional (see `configure --help`) SQLite database to hold persistent program analysis results across the compilation of multiple files. This mechanism may become less critical as the only mechanism to support global analysis once we can support whole program analysis more generally within ROSE.

- Program Transformations and Optimizations

  1. Generating Unique Names for Declarations (generatingUniqueNamesFromDeclaration.C)
     A recurring issue in the development of many tools and program analysis is the representation of unique strings from language constructs (functions, variable declarations, etc.). This example demonstrated support in ROSE for the generation of unique names. Names are unique across different ROSE tools and compilation of different files.
  2. Command-line processing
     ROSE includes mechanism to simplify the processing of command-line arguments so that translators using ROSE can trivially replace compilers within makefiles. This example shows some of the many command-line handling options within ROSE and the ways in which customized options may be added.
     (a) Recognizing custom command-line options

   (b) Adding options to internal ROSE command-line driven mechanisms

3. Tailoring the code generation format: how to indent the generated source code and others.
4. AST construction: how to build AST pieces from scratch and attach them to the existing AST tree.
   (a) Adding a variable declaration (addingVariableDeclaration.C)
   Here we show how to add a variable declaration to the input application. Perhaps we should show this in two ways to make it clear. This is a particularly simple use of the AST IR nodes to build an AST fragment and add it to the application's AST.
   (b) Adding a function (addingFunctionDeclaration.C)
   This example program shows the addition of a new function to the global scope. This example is a bit more involved than the previous example.
   (c) Simple Instrumentor Translator (simpleInstrumentor.C)
   This example modifies an input application to place new code at the top and bottom of each block. The output is show with the instrumentation in place in the generated code.
   (d) Other examples for creating expressions, structures and so on.
5. Handling source comments, preprocessor directives.
6. Calling the inliner (inlinerExample.C)
   This example shows the use of the inliner mechanism within ROSE. The function to be inlined in specified and the transformation upon the AST is done to inline the function where it is called and clean up the resulting code.
7. Calling the outliner (outlinerExample.C)
   This example shows the use of the outliner mechanism within ROSE. A segment of code is selected and a function is generated to hold the resulting code. Any required variables (including global variables) are passed through the generated function's interface. The outliner is a useful part of the empirical optimization mechanisms being developed within ROSE.
8. Call loop optimizer on set of loops (loopOptimization.C)
   This example program shows the optimization of a loop in C. This section contains several subsections each of which shows different sorts of optimizations. There are a large number of loop optimizations only two are shown here, we need to add more.
   (a) Optimization of Matrix Multiply
   (b) Loop Fusion Optimizations
9. Parameterized code translation: How to use command line options and abstract handles to have the translations you want, the order you want, and the behaviors you want.

Does this tutorial still exist?

10. Program slicing (programSlicingExample.C)
    This example shows the interface to the program slicing mechanism within ROSE. Program slicing has been implemented to two ways within ROSE.

- Correctness Checking

1. Code Coverage Analysis (codeCoverage.C)
   Code coverage is a useful tool by itself, but is particularly useful when combined with automated detection of bugs in programs. This is part of work with IBM, Haifa.
2. Bug seeding: how to purposely inject bugs into source code.

- Binary Support
  1. Instruction semantics
  2. Binary Analysis
  3. Binary construction
  4. DWarf debug support

- Interacting with Other Tools
  1. Abstract handles: uniform ways of specifying language constructs.
  2. ROSE-HPCT interface: How to annotate AST with performance metrics generated by third-party performance tools.
  3. Tau Performance Analysis Instrumentation (tauInstrumenter.C)
     Tau currently uses an automate mechanism that modified the source code text file. This example shows the modification of the AST and the generation of the correctly instrumented files (which can otherwise be a problem when macros are used). This is part of collaborations with the Tau project.
  4. The Haskell interface: interacting with a function programming language.

- Parallelism
  1. Shared-memory parallel traversals
  2. Distributed-memory parallel traversals
  3. Parallel checker
  4. Reduction variable recognition

Other examples included come specifically from external collaborations and are more practically oriented. Each is useful as an example because each solves a specific technical problem. More of these will be included over time.

FIXME: *The following tutorials no longer exist?*

1. Fortran promotion of constants to double precision (typeTransformation.C)
   Fortran constants are by default singe precision, and must be modified to be double precision. This is a common problem in older Fortran applications. This is part of collaborations with LANL to eventually automatically update/modify older Fortran applications.

2. Automated Runtime Library Support (charmSupport.C)
   Getting research runtime libraries into use within large scale applications requires automate mechanism to make minor changes to large amounts of code. This is part of collaborations with the Charm++ team (UIUC).

(a) Shared Threaded Variable Detection Instrumentation (interveneAtVariables.C)
Instrumentation support for variables, required to support detection of threaded bugs in applications.

(b) Automated Modification of Function Parameters (changeFunction.C)
This example program addresses a common problem where an applications function must be modified to include additional information. In this case each function in a threaded library is modified to include additional information to a corresponding wrapper library which instruments the library's use.

Add make installcheck e.am to build example ors using the installed libraries.

1.5 ROSE Documentation and Where To Find It

There are three forms of documentation for ROSE, and also a ROSE web Page and email lists. For more detailed information on getting started, see the ROSE User Manual, chapter *Getting Started* for more details).

1. ROSE User Manual
The User Manual presents how to get started with ROSE and documents features of the ROSE infrastructure. The User Manual is found in ROSE/docs/Rose directory, or at:
ROSE User Manual (pdf version, relative link)

2. ROSE Tutorial
The ROSE Tutorial presents a collection of examples of how to use ROSE (found in the ROSE/tutorial directory). The ROSE Tutorial documentation is found in ROSE/docs/Rose/Tutorial directory. The tutorial documentation is built in the following steps:

   (a) actual source code for each example translator in the ROSE/tutorial directory is included into the tutorial documentation
   (b) each example is compiled
   (c) inputs to the examples are taken from the ROSE/tutorial directory
   (d) output generated from running each example is placed into the tutorial documentation

   Thus the `ROSE/tutorial` directory contains the exact examples in the tutorial and each example may be modified (changing either the example translators or the inputs to the examples). The ROSE Tutorial can also be found in the `ROSE/docs/Rose/Tutorial` directory (the LaTeX document; ps or pdf file)
   : ROSE Tutorial (pdf version, relative link),

3. ROSE HTML Reference: Intermediate Representation (IR) documentation
This web documentation presents the detail interfaces for each IR nodes (documentation generated by Doxygen). The HTML IR documentation is found in ROSE/docs/Rose directory (available as html only):
ROSE HTML Reference (relative link)

4. ROSE Web Page
The ROSE web pages are located at: http://www.rosecompiler.org

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

1.6 Using the Tutorial

First install ROSE (see ROSE User Manual, chapter *Getting Started* for details). Within the ROSE distribution at the top level is the `tutorial` directory. All of the examples in this documentation are represented there with Makefiles and sample input codes to the example translators.

1.7 Required Makefile for Tutorial Examples

FIXME: *The exampleMakefile needs to be modified to include the support for compiling all of the tutorial examples.*

This section shows an example makefile 1.1 required for the compilation of many of the tutorial example programs using the installed libraries (assumed to be generated from `make install`). The build process can be tested by running `make installcheck` from within the ROSE compile tree. This `makefile` can be found in the compile tree *(not the source tree)* for ROSE in the `tutorial` directory.

```
# 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 SVN 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/hudson-rose/.hudson/tempInstall/include

# Location of Boost include directory
BOOST_CPPFLAGS = -pthread -I/export/tmp.hudson-rose/opt/boost_1_40_0-inst/include

# Location of Dwarf include and lib (if ROSE is configured to use Dwarf)
ROSE_DWARF_INCLUDES =
ROSE_DWARF_LIBS_WITH_PATH =

# Location of library directory after "make install"
ROSE_LIB_DIR = /home/hudson-rose/.hudson/tempInstall/lib

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

ROSE_LIBS = $(ROSE_LIB_DIR)/librose.la

# Location of source code
ROSE_SOURCE_DIR = \
   ../../tutorial

executableFiles = identityTranslator ASTGraphGenerator \
               visitorTraversal inheritedAttributeTraversal \
               synthesizedAttributeTraversal \
               inheritedAndSynthesizedAttributeTraversal \
               accumulatorAttributeTraversal persistantAttributes \
               queryLibraryExample nestedTraversal \
               loopRecognition \
               typeInfoFromFunctionParameters \
               resolveOverloadedFunction templateParameter \
               instrumentationExample addVariableDeclaration \
               addFunctionDeclaration loopOptimization \
               buildCFG debuggingIRnodeToString \
               debuggingSourceCodePositionInformation \
               commandlineProcessing \
               loopNestingInfoProcessing

# Default make rule to use
all: $(executableFiles)
	@if [ x$${ROSE_IN_BUILD_TREE:+present} = xpresent ]; then echo "ROSE_IN_BUILD_TREE should not be set" >&2; exit 1; fi

# Example of how to use ROSE (linking to dynamic library, which is must faster
# and smaller than linking to static libraries).  Dynamic linking requires the
# use of the "-L$(ROSE_LIB_DIR) -Wl,-rpath" syntax if the LD_LIBRARY_PATH is not
# modified to use ROSE_LIB_DIR.  We provide two example of this; one using only
# the "-lrose -ledg" libraries, and one using the many separate ROSE libraries.
$(executableFiles):
#	g++ -I$(ROSE_INCLUDE_DIR) -o $@ $(ROSE_SOURCE_DIR)/$@.C -L$(ROSE_LIB_DIR) -Wl,-rpath $(ROSE_LIB_DIR) $(ROSE_LIBS)
#	g++ -I$(ROSE_INCLUDE_DIR) -o $@ $(ROSE_SOURCE_DIR)/$@.C $(LIBS_WITH_RPATH) $(ROSE_LIBS)
#	/bin/sh ../../libtool --mode=link $(CXX) $(CPPFLAGS) $(CXXFLAGS)  $(LDFLAGS) -I$(ROSE_INCLUDE_DIR) $(BOOST_CPPFLAGS) -o $@ $(ROSE_SOU
	/bin/sh ../../libtool --mode=link $(CXX) $(CPPFLAGS) $(CXXFLAGS)  $(LDFLAGS) -I$(ROSE_INCLUDE_DIR) $(BOOST_CPPFLAGS) $(ROSE_DWARF_IN
```

Figure 1.1: Example `Makefile` showing how to use an installed version of ROSE (generated by `make install`).

Part I

Working with the ROSE AST

Getting familiar with the ROSE AST is the basis for any advanced usage of ROSE. This part of tutorial collects examples for AST visualization, traversal, query, and debugging.

Chapter 2

Identity Translator

What To Learn From This Example This example shows a trivial ROSE translator which does not transformation, but effectively wraps the the backend vendor compiler in an extra layer of indirection.

Using the input code in Figure 2.2 we show a translator which builds the AST (calling `frontend()`), generates the source code from the AST, and compiles the generated code using the backend vendor compiler[1]. Figure 2.1 shows the source code for this translator. The AST graph is generated by the call to the `frontend()` using the standard `argc` and `argv` parameters from the C/C++ `main()` function. In this example code, the variable `project` represents the root of the AST[2]. The source code also shows what is an optional call to check the integrity of the AST (calling function `AstTests::runAllTests()`); this function has no side-effects on the AST. The source code generation and compilation to generate the object file or executable are done within the call to `backend()`.

The identity translator (*identityTranslator*) is probably the simplest translator built using ROSE. It is built by default and can be found in
ROSE_BUILD/exampleTranslators/documentedExamples/simpleTranslatorExamples or
ROSE_INSTALL/bin. It is often used to test if ROSE can compile input applications. Typing *identityTranslator –help* will give you more information about how to use the translator.

Figure 2.3 shows the generated code from the processing of the `identityTranslator` build using ROSE and using the input file shown in figure 2.2. This example also shows that the output generated from and ROSE translator is a close reproduction of the input; preserving all comments, preprocessor control structure, and most formating. Note that all macros are expanded in the generated code.

In this trivial case of a program in a single file, the translator compiles the application to build an executable (since `-c` was not specified on the command-line).

[1] Note: that the backend vendor compiler is selected at configuration time.

[2] The AST is technically a *tree* with additional attributes that are represented by edges and additional nodes, so the AST is a tree and the AST *with* attributes is a more general graph containing edges that would make it technically *not* a tree.

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

#include "rose.h"

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

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

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

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

Figure 2.1: Source code for translator to read an input program and generate an object code (with no translation).

```
// Example input file for ROSE tutorial
#include <iostream>

typedef float Real;

// Main function
int main()
   {
     int x = 0;
     bool value = false;

  // for loop
     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 2.2: Example source code used as input to identity translator.

```
// Example input file for ROSE tutorial
#include <iostream>
typedef float Real;
// Main function

int main()
{
  int x = 0;
  bool value = false;
// for loop
  for (int i = 0; i < 4; i++) {
    int x;
  }
  return 0;
}
```

Figure 2.3: Generated code, from ROSE identity translator, sent to the backend (vendor) compiler.

Chapter 3

Simple AST Graph Generator

What To Learn From This Example This example shows how to generate a DOT file to visualize a simplified AST from any input program.

DOT is a graphics file format from the AT&T GraphViz project used to visualize moderate sized graphs. It is one of the visualization tools used within the ROSE project. More inforamtion can be readily found at *www.graphviz.org/*. We have found the *zgrviewer* to be an especially useful program for visualizing graphs generated in the DOT file format (see chapter 5 for more information on zgrviewer).

Each node of the graph in figure 3.3 shows a node of the Intermediate Representation (IR); the graphs demonstrates that the AST is formally a tree. Each edge shows the connection of the IR nodes in memory. The generated graph shows the connection of different IR nodes that form the AST for an input program source code. Binary executables can similarly be vizualized using DOT files. The generation of such graphs is appropriate for small input programs, chapter 6 shows a mechanism using PDF files that is more appropriate to larger programs (e.g. 100K lines of code). More information about generation of specialized AST graphs can be found in 5 and custom graph generation in 28.

Note that a similar utility program named *dotGenerator* already exists within ROSE/exampleTranslators/DOTGenerator. It is also installed to *ROSE_INS/bin*.

The program in figure 3.1 calls an internal ROSE function that traverses the AST and generates an ASCII file in `dot` format. Figure 3.2 shows an input code which is processed to generate a graph of the AST, generating a `dot` file. The `dot` file is then processed using `dot` to generate a postscript file 3.3 (within the `Makefile`). Figure 3.3 (../../..//tutorial/test.ps) can be found in the compile tree (in the tutorial directory) and viewed directly using ghostview or any postscript viewer to see more detail.

Figure 3.3 displays the individual C++ nodes in ROSE's intermediate representation (IR). Each circle represents a single IR node, the name of the C++ construct appears in the center of the circle, with the edge numbers of the traversal on top and the number of child nodes appearing below. Internal processing to build the graph generates unique values for each IR node, a pointer address, which is displays at the bottom of each circle. The IR nodes are connected for form a tree, and abstract syntax tree (AST). Each IR node is a C++ class, see SAGE III reference for

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

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

  // Generate a DOT file to use in visualizing the AST graph.
     generateDOT ( *project );

     return 0;
   }
```

Figure 3.1: Example source code to read an input program and generate an AST graph.

```

// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);
void foo(double)
   {
     int x = 1;
     int y;

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }
```

Figure 3.2: Example source code used as input to generate the AST graph.

details, the edges represent the values of data members in the class (pointers which connect the IR nodes to other IR nodes). The edges are labeled with the names of the data members in the classes representing the IR nodes.

Use this first example e to explain the use of der files (config.h and d the code to build the SgProject object.

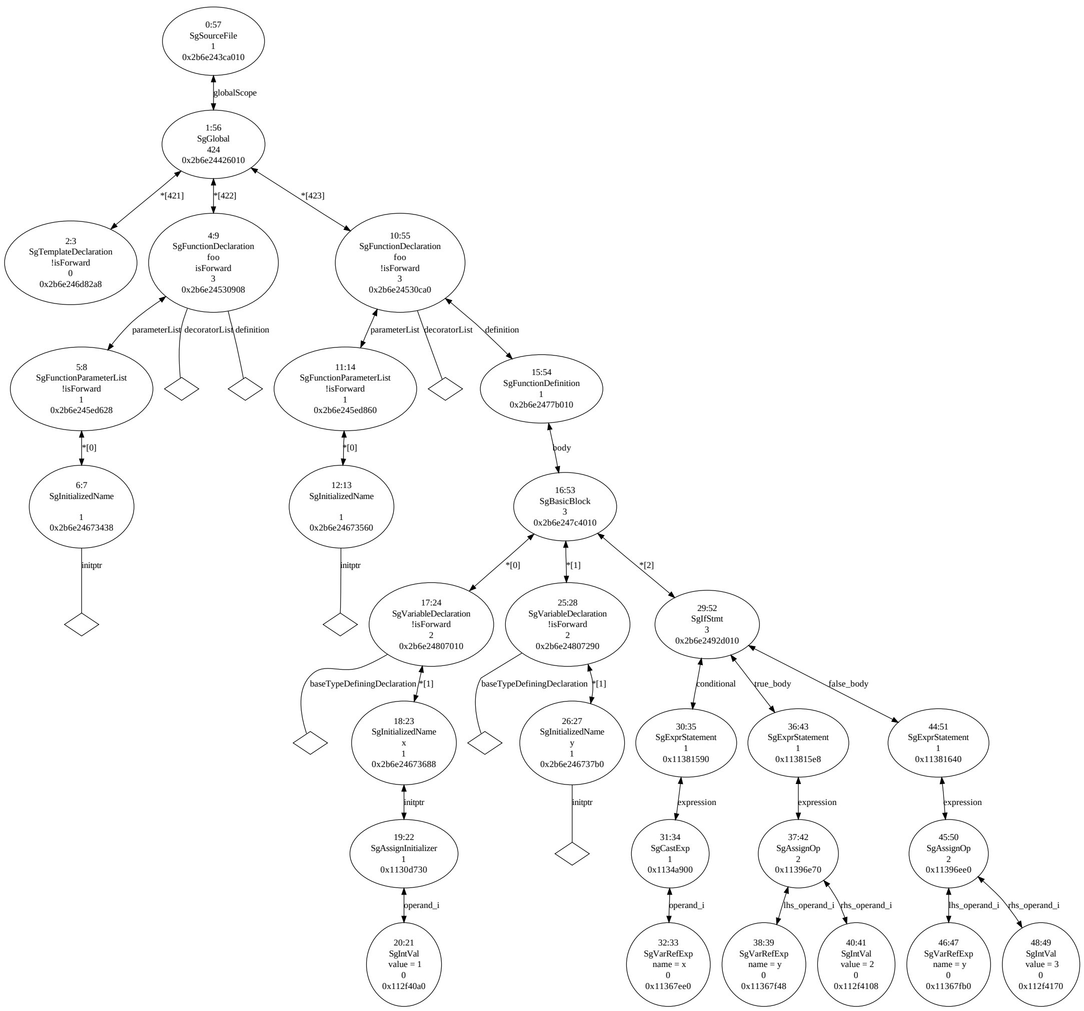


Figure 3.3: AST representing the source code file: inputCode_ASTGraphGenerator.C.

Chapter 4

AST Whole Graph Generator

What To Learn From This Example This example shows how to generate and visualize the AST from any input program. This view of the AST includes all additional IR nodes and edges that form attributes to the AST, as a result this graph is not a tree. These graphs are more complex but show significantly more detail about the AST and its additional edges and attributes. Each node of the graph in figure **??** shows a node of the Intermediate Representation (IR). Each edge shows the connection of the IR nodes in memory. The generated graph shows the connection of different IR nodes to form the AST and its additional attributes (e.g types, modifiers, etc). The generation of such graphs is appropriate for very small input programs, chapter 6 shows a mechanism using PDF files that is more appropriate to larger programs (e.g. 100K lines of code). More information about generation of specialized AST graphs can be found in 5 and custom graph generation in 28.

Again, a utility program, called *dotGeneratorWholeASTGraph* is provided within ROSE to generate detailed dot graph for input code. It is available from *ROSE_BUILD/exampleTranslators/DOTGenerator* or *ROSE_INS/bin*. A set of options is available to further customize what types of AST nodes to be shown or hidden. Please consult the screen output of *dotGeneratorWholeASTGraph -help* for details.

Viewing these dot files is best done using: **zgrviewer** at `http://zvtm.sourceforge.net/zgrviewer.html`. This tool permits zooming in and out and viewing isolated parts of even very large graphs. **Zgrviewer** permits a more natural way of understanding the AST and its additional IR nodes than the pdf file displayed in these pages. The few lines of code used to generate the graphs can be used on any input code to better understand how the AST represents different languages and their constructs.

The program in figure 4.1 calls an internal ROSE function that traverses the AST and generates an ASCII file in `dot` format. Figure **??** shows an tiny input code which is processed to generate a graph of the AST with its attributes, generating a `dot` file. The `dot` file is then processed using `dot` to generate a pdf file 4.3 (within the `Makefile`). Note that a similar utility program already exists within ROSE/exampleTranslators (and includes a utility to output an alternative PDF representation (suitable for larger ASTs) as well). Figure **??** (../../..//tutorial/test.ps) can be found in the compile tree (in the tutorial directory) and viewed directly using any pdf or dot viewer to see more detail (**zgrviewer** working with the dot file directly is strongly advised).

Note that AST's can get very large, and that the additional IR nodes required to represent

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

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

  // Build the DOT file to visualize the AST with attributes (types, symbols, etc.).
  // To protect against building graphs that are too large an option is
  // provided to bound the number of IR nodes for which a graph will be
  // generated.  The layout of larger graphs is prohibitively expensive.
     const int MAX_NUMBER_OF_IR_NODES = 2000;
     generateAstGraph(project,MAX_NUMBER_OF_IR_NODES);

     return 0;
   }
```

Figure 4.1: Example source code to read an input program and generate a *whole* AST graph.

the types, modifiers, etc, can generate visually complex graphs. ROSE contains the mechanisms to traverse these graphs and do analysis on them. In one case the number of IR nodes exceeded 27 million, an analysis was done through a traversal of the graph in 10 seconds on a desktop x86 machine (the memory requirements were 6 Gig). ROSE organizes the IR in ways that permit analysis of programs that can represent rather large ASTs.

```
// Trivial function used to generate graph of AST
// with all types and additional edges shown.
// Graphs of this sort are large, and can be
// viewed using "zgrviewer" for dot files.
int foo()
   {
     return 0;
   }
```

Figure 4.2: Example tiny source code used as input to generate the small AST graph with attributes.

Figure 4.3 displays the individual C++ nodes in ROSE's intermediate representation (IR). Colors and shapes are used to represent different types or IR nodes. Although only visible using **zgrviewer** the name of the C++ construct appears in the center of each node in the graph, with the names of the data members in each IR node as edge labels. Unique pointer values are includes and printed next to the IR node name. These graphs are the single best way to develop an intuitive understanding how language constructs are organized in the AST. In these graphs, the color yellow is used for types (SgType IR nodes), the color green is used for expressions (SgExpression IR nodes), and statements are a number of different colors and shapes to make them more recognizable.

Figure 4.5 shows a graph similar to the previous graph but larger and more complex because it is from a larger code. Larger graphs of this sort are still very useful in understanding how more significant language constructs are organized and reference each other in the AST. Tools such as **zgrviewer** are essential to reviewing and understanding these graphs. Although such graphs

```
// Larger function used to generate graph of AST
// with all types and additional edges shown.
// Graphs of this sort are large, and can be
// viewed using "zgrviewer" for dot files.
int foo ( int x );

int globalVar = 42;

void foobar_A()
   {
     int a = 4;
     int b = a + 2;
     int c = b * globalVar;
     int x;
     x = foo (c);
     int y = x + 2;
     int z = globalVar * y;
   }



void foobar_B()
   {
     int p;
     int i = 4;
     int k = globalVar * (i+2);
     p = foo (k);
     int r = (p+2) * globalVar;
   }
```

Figure 4.4: Example source code used as input to generate a larger AST graph with attributes.

can be visualized, in practice this is only useful for debugging small codes in the construction of custom analysis and transformation tools. The graphs for real million line applications would never be visualized. Using ROSE one can build automated tools to operate on the AST for large scale applications where visualization would not be possible.

Chapter 5

Advanced AST Graph Generation

FIXME: *This chapter brings more confusions. I suggest to remove it until it is done right . -Leo*

What To Learn From This Example This example shows a maximally complete representation of the AST (often in more detail that is useful).

Where chapter 3 presented a ROSE-based translator which presented the AST as a tree, this chapter presents the more general representation of the graph in which the AST is embedded. The AST may be thought of as a subset of a more general graph or equivalently as an AST (a tree in a formal sense) with annotations (extra edges and information), sometimes referred to as a '*decorated* AST.

We present tools for seeing all the IR nodes in the graph containing the AST, including all types (SgType nodes), symbols (SgSymbol nodes), compiler generated IR nodes, and supporting IR nodes. In general it is a specific filtering of this larger graph which is more useful to communicating how the AST is designed and internally connected. We use these graphs for internal debugging (typically on small problems where the graphs are reasonable in size). The graphs presented using these graph mechanism present all back-edges, and demonstrate what IR nodes are shared internally (typically SgType IR nodes).

First a few names, we will call the AST those nodes in the IR that are specified by a traversal using the ROSE traversal (SgSimpleTraversal, etc.). We will call the graph of all IR nodes the *Graph of all IR nodes.* the AST is embedded in the *Graph of all IR nodes.* The AST *is* a tree, while the *graph of all IR nodes* typically not a tree (in a Graph Theory sense) since it typically contains cycles.

We cover the visualization of both the AST and the *Graph of all IR nodes.*

- AST graph
  These techniques define ways of visualizing the AST and filtering IR nodes from being represented.
  - Simple AST graphs
  - Colored AST graphs
  - Filtering the graph
    The AST graph may be generated for any subtree of the AST (not possible for the graphs of all IR nodes). Additionally runtime options permit null pointers to be ignored. .

FIXME: *Is this true?*

- *Graph of all IR nodes*
  These techniques define the ways of visualizing the whole graph of IR nodes and is based on the memory pool traversal as a means to access all IR nodes. Even disconnected portions of the AST will be presented.
  - Simple graphs
  - Colored graphs
  - Filtering the graph

Removed this example newer mechanism for the whole AST graphs needs to be presented.

Chapter 6

AST PDF Generator

What To Learn From This Example This example demonstrates a mechanism for generating a visualization of the AST using pdf files. A pdf file is generated and can be viewed using `acroread`. The format is suitable for much larger input programs than the example shown in previous chapters using dot format 4. This mechanism can support the visualization of input files around 100K lines of code.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

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

  // Generate a PDF file for interactive exploration of the AST.
     generatePDF ( *project );

     return 0;
   }
```

Figure 6.1: Example source code to read an input program and generate a PDF file to represent the AST.

The program in figure 6.1 calls an internal ROSE function that traverses the AST and generates an ASCI file in `dot` format. Figure 3.2 shows an input code which is processed to generate a graph of the AST, generating a `pdf` file. The `pdf` file is then processed using `acroread` to generate a GUI for viewing the AST.

A standalone utility tool, called *pdfGenerator* is provided within ROSE. It is available from *ROSE_BUILD/exampleTranslators/PDFGenerator* or *ROSE_INS/bin*. Users can use it to generate AST in a pdf format from an input code.

Figure 6.3 displays on the left hand side the individual C++ nodes in ROSE's intermediate representation (IR). The page on the right hand side shows that IR nodes member data. Pointers in boxes can be clicked on to navigate the AST (or nodes in the tree hierarchy can be clicked on jump to any location in the AST. This representation shows only the IR nodes that are traversed

```

// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);
void foo(double)
   {
     int x = 1;
     int y;

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }
```

Figure 6.2: Example source code used as input to generate the PDF file of the AST.

by the standard traversal (no SgSymbol or SgType IR nodes are presented in this view of the AST).

The output of this translator is shown in figure 6.3. The left hand side of the screen is a tree with click-able nodes to expand/collapse the subtrees. The right hand side of the screen is a description of the data at a particular node in the AST (the node where the user has clicked the left mouse button). This relatively simple view of the AST is useful for debugging transformation and finding information in the AST required by specific sorts of analysis. It is also useful for developing an intuitive feel for what information is in the AST, how it is organized, and where it is stored.

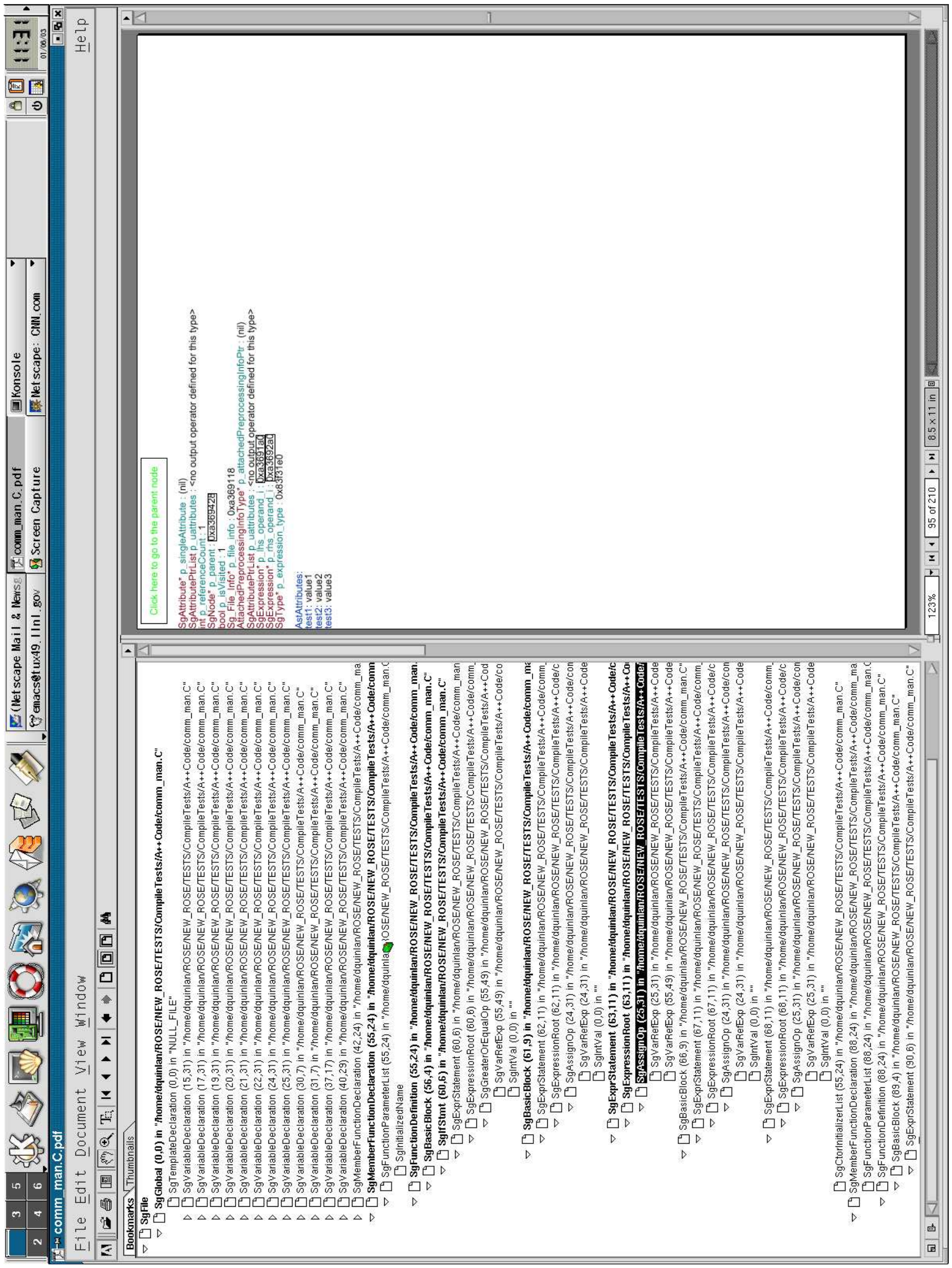


Figure 6.3: Example output from translator which outputs PDF representation of AST. The generated PDF file makes use of the bookmark mechanism to expand and collapse parts of the AST.

Chapter 7

Introduction to AST Traversals

FIXME: *Add* **What to learn from this example** *paragraph to each example.*

FIXME: *Add* **What is different from previous example** *paragraph to each example.*

FIXME: *Add a table and/or graph at the end of this chapter to summarize the traversals.*

An essential operation in the analysis and construction of ASTs is the definition of traversals upon the AST to gather information and modify targeted internal representation (IR) nodes. ROSE includes different sorts of traversals to address the different requirements of numerous program analysis and transformation operations. This section demonstrates the different types of traversals that are possible using ROSE.

ROSE translators most commonly introduce transformations and analysis through a traversal over the AST. Alternatives could be to generate a simpler IR that is more suitable to a specific transformation and either convert modification to that transformation specific IR into changes to the AST or generate source code from the transformation specific IR directly. These approaches are more complex than introducing changes to the AST directly, but may be better for specific transformations.

Traversals represent an essential operation on the AST and there are a number of different types of traversals. The suggested traversals for users are explained in Section 7.2. Section 7.3 introduces specialized traversals (that traverse the AST in different orders and traverse types and symbols), typically not appropriate for most translators (but perhaps appropriate for specialized tools, often internal tools within ROSE).

See the ROSE User Manual for a more complete introduction to the different types of traversals. The purpose of this tutorial is to present examples, but we focus less on the background and philosophy here than in the ROSE User Manual.

This chapter presents a number of ways of traversing the AST of any input source code. These *traversals* permit operations on the AST, which may either read or modify the AST in place. Modifications to the AST will be reflected in the source code generated when the AST is *unparsed*; the code generation phase of the source-to-source process defined by ROSE. Note that for all examples, the input code described in section 7.1 is used to generate all outputs shown with each translator.

7.1 Input For Example Traversals

The code shown in figure 7.1 shows the input code that will be used to demonstrate the traversals in this chapter. It may be modified by the user to experiment with the use of the traversals on

```

// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;
          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);

void foo(double)
   {
     int x = 1;
     int y;

     for (int i=0; i < 4; i++)
        {
          int x;
        }

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }

int main()
   {
     foo(42);
     foo(3.14159265);

     templateClass<char> instantiatedClass;
     instantiatedClass.foo(7);
     instantiatedClass.foo(7.0);

     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 7.1: Example source code used as input to program in traversals shown in this chapter.

alternative input codes.

7.2 Traversals of the AST Structure

This collection of traversals operates on the AST in an order which matches the structure of the AST and the associated source code. These types of traversals are the most common traversals for users to use. A subsequent section of this chapter demonstrated more specialized traversals over all IR nodes (more than just those IR nodes in the AST representing the structure of the source code) that are suitable for some tools, mostly tools built internally within ROSE.

Because the traversals in this section traverse the structure of the source code (see the AST graph presented in the first tutorial example) they are more appropriate for most transformations of the source code. We suggest that the user focus on these traversals which represent the interface we promote for analysis and transformation of the AST, instead of the memory pools traversals which are suitable mostly for highly specialized internal tools. The simple traversals of both kinds have the same interface so the user may easily switch between them with out significant difficulty.

7.2.1 Classic Object-Oriented Visitor Pattern for the AST

We show this example first, but it is rarely used in practice, and more useful traversals follow. It is however most closely similar to traversals that are available in other compiler infrastructures, and so a concept with which many people will be familar. In this case because this implementation is based on the memory pool infrstructure it will visit all node and not in any order based on the AST. The ASTSimpleProcessing traversal in section 7.2.2 is closer to a common visitor pattern that visits the IR nodes in the order in which they appear in the AST.

Figure 7.2 shows the source code for a translator using the classic object-oriented visitor pattern to traverse the AST. *This visitor pattern is only implemented for the memory pool based traversal.* Thus it works on the whole of the attributed AST and does not work on a restricted subset of the AST (e.g. a subtree of the unattributed AST). Figure 7.3 shows the output from this traversal using the example input source from figure 7.1.

7.2.2 Simple Traversal (no attributes)

Figure 7.4 shows the source code for a translator which traverses the AST. The traversal object is from the type `visitorTraversal` derived from `AstSimpleProcessing`. The `visit()` function is required to be defined because it is defined as a pure virutal funciton in the `AstSimpleProcessing` base class. The member function `traverseInputFiles()` of `AstSimpleProcessing` is called to traverse the AST and call the `visit()` function on each IR node. Note that the function `traverse()` (not used) would visit each IR nodes while `traverseInputFiles()` will only visit those IR nodes that originate in the input source code (thus skipping all header files).

For each node where the `visit()` function is called a `SgNode` pointer is to the node is passed into the `visit` function. Note that using this simple traversal the only context information available to the visit function is what is stored in its member variables (though access to other nodes is possible along any edges in the attributed AST graph). The only option is to traverse the AST in either pre-order or postorder. The `atTraversalEnd()` function may be defined by the user to do final processing after all nodes have been visited (or to perform preparations before the nodes are visited, in the case of the corresponding `atTraversalStart()` function). Figure 7.5 shows the output from this traversal using the example input source from figure 7.1.

7.2.3 Simple Pre- and Postorder Traversal

Figure 7.6 shows the source code for a translator that traverses the AST without attributes (like the one in the previous subsection), but visiting each node twice, once in preorder (before its children) and once in postorder (after all children). Figure 7.7 shows the output from this traversal using the example input source from figure 7.1.

```

#include "rose.h"

// Classic Visitor Pattern in ROSE (implemented using the traversal over
// the elements stored in the memory pools so it has no cycles and visits
// ALL IR nodes (including all Sg_File_Info, SgSymbols, SgTypes, and the
// static builtin SgTypes).
class ClassicVisitor : public ROSE_VisitorPattern
   {
     public:
       // Override virtual function defined in base class
          void visit(SgGlobal* globalScope)
             {
               printf ("Found the SgGlobal IR node \n");
             }

          void visit(SgFunctionDeclaration* functionDeclaration)
             {
               printf ("Found a SgFunctionDeclaration IR node \n");
             }
          void visit(SgTypeInt* intType)
             {
               printf ("Found a SgTypeInt IR node \n");
             }

          void visit(SgTypeDouble* doubleType)
             {
               printf ("Found a SgTypeDouble IR node \n");
             }
   };


int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Classic visitor pattern over the memory pool of IR nodes
     ClassicVisitor visitor_A;
     traverseMemoryPoolVisitorPattern(visitor_A);

     return backend(project);
   }
```

Figure 7.2: Example source showing simple visitor pattern.

7.2.4 Inherited Attributes

Figure 7.8 shows the use of inherited attributes associated with each IR node. Within this traversal the attributes are managed by the traversal and exist on the stack. Thus the lifetime of the attributes is only as long as the processing of the IR node and its subtree. Attributes such as this are used to communicate context information **down** the AST and called *Inherited attributes*.

In the example the class `Inherited Attribute` is used to represent inherited attribute. Each instance of the class represents an attribute value. When the AST is traversed we obtain as output the loop nest depth at each point in the AST. The output uses the example input source from figure 7.1.

Note that inherited attributes are passed by-value down the AST. In very rare cases you might want to pass a pointer to dynamically allocated memory as an inherited attribute. In this case you can define the virtual member function `void destroyInheritedValue(SgNode *n, InheritedAttribute inheritedValue)` which is called after the last use of the inherited attribute computed at this node, i. e. after all children have been visited. You can use this function to free the memory allocated for this inherited attribute.

```
Found a SgTypeInt IR node
Found a SgTypeDouble IR node
Found the SgGlobal IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
```

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

class visitorTraversal : public AstSimpleProcessing
   {
     public:
          visitorTraversal();
          virtual void visit(SgNode* n);
          virtual void atTraversalEnd();
   };

visitorTraversal::visitorTraversal()
   {
   }

void visitorTraversal::visit(SgNode* n)
   {
     if (isSgForStatement(n) != NULL)
        {
          printf ("Found a for loop ... \n");
        }
   }

void visitorTraversal::atTraversalEnd()
   {
     printf ("Traversal ends here. \n");
   }

int
main ( int argc, char* argv[] )
   {
     if (SgProject::get_verbose() > 0)
          printf ("In visitorTraversal.C: main() \n");

     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal function (member function of AstSimpleProcessing)
  // starting at the project node of the AST, using a preorder traversal.
     exampleTraversal.traverseInputFiles(project,preorder);

     return 0;
   }
```

Figure 7.4: Example source showing simple visitor pattern.

```
Found a for loop ...
Found a for loop ...
Traversal ends here.
```

Figure 7.5: Output of input file to the visitor traversal.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

class PreAndPostOrderTraversal : public AstPrePostProcessing
   {
     public:
          virtual void preOrderVisit(SgNode* n);
          virtual void postOrderVisit(SgNode* n);
   };

void PreAndPostOrderTraversal::preOrderVisit(SgNode* n)
   {
     if (isSgForStatement(n) != NULL)
        {
          printf ("Entering for loop ... \n");
        }
   }

void PreAndPostOrderTraversal::postOrderVisit(SgNode* n)
   {
     if (isSgForStatement(n) != NULL)
        {
          printf ("Leaving for loop ... \n");
        }
   }

int
main ( int argc, char* argv[] )
   {
     if (SgProject::get_verbose() > 0)
          printf ("In prePostTraversal.C: main() \n");

     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object
     PreAndPostOrderTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project);

     return 0;
   }
```

Figure 7.6: Example source showing simple pre- and postorder pattern.

```
Entering for loop ...
Leaving for loop ...
Entering for loop ...
Leaving for loop ...
```

Figure 7.7: Output of input file to the pre- and postorder traversal.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

// Build an inherited attribute for the tree traversal to test the rewrite mechanism
class InheritedAttribute
   {
     public:
      // Depth in AST
         int depth;
         int maxLinesOfOutput;

       // Specific constructors are required
          InheritedAttribute (int x) : depth(x), maxLinesOfOutput(20) {};
          InheritedAttribute ( const InheritedAttribute & X ) : depth(X.depth), maxLinesOfOutput(20){};
   };

class visitorTraversal : public AstTopDownProcessing<InheritedAttribute>
   {
     public:
       // virtual function must be defined
          virtual InheritedAttribute evaluateInheritedAttribute(SgNode* n, InheritedAttribute inheritedAttribute);
   };

InheritedAttribute
visitorTraversal::evaluateInheritedAttribute(SgNode* n, InheritedAttribute inheritedAttribute)
   {
     static int linesOfOutput = 0;
     if (linesOfOutput++ < inheritedAttribute.maxLinesOfOutput)
          printf ("Depth in AST at %s = %d \n",n->sage_class_name(),inheritedAttribute.depth);

     return InheritedAttribute(inheritedAttribute.depth+1);
   }

int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // DQ (1/18/2006): Part of debugging
     SgFile & localFile = project->get_file(0);
     localFile.get_file_info()->display("localFile information");

  // Build the inherited attribute
     InheritedAttribute inheritedAttribute(0);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project,inheritedAttribute);

  // Or the traversal over all AST IR nodes can be called!
     exampleTraversal.traverse(project,inheritedAttribute);

     return 0;
   }
```

Figure 7.8: Example source code showing use of inherited attributes (passing context information **down** the AST.

```
Inside of Sg_File_Info::display(localFile information)
     isTransformation                      = false
     isCompilerGenerated                   = false
     isOutputInCodeGeneration              = false
     isShared                              = false
     isFrontendSpecific                    = false
     isSourcePositionUnavailableInFrontend = false
     isCommentOrDirective                  = false
     isToken                               = false
     filename = /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inp
     line     = 1  column = 1
Depth in AST at SgSourceFile = 0
Depth in AST at SgGlobal = 1
Depth in AST at SgTemplateDeclaration = 2
Depth in AST at SgFunctionDeclaration = 2
Depth in AST at SgFunctionParameterList = 3
Depth in AST at SgInitializedName = 4
Depth in AST at SgFunctionDeclaration = 2
Depth in AST at SgFunctionParameterList = 3
Depth in AST at SgInitializedName = 4
Depth in AST at SgFunctionDefinition = 3
Depth in AST at SgBasicBlock = 4
Depth in AST at SgVariableDeclaration = 5
Depth in AST at SgInitializedName = 6
Depth in AST at SgAssignInitializer = 7
Depth in AST at SgIntVal = 8
Depth in AST at SgVariableDeclaration = 5
Depth in AST at SgInitializedName = 6
Depth in AST at SgForStatement = 5
Depth in AST at SgForInitStatement = 6
Depth in AST at SgVariableDeclaration = 7
```

Figure 7.9: Output of input file to the inherited attribute traversal.

7.2.5 Synthesized Attributes

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

#include <algorithm>
#include <functional>
#include <numeric>

typedef bool SynthesizedAttribute;

class visitorTraversal : public AstBottomUpProcessing<SynthesizedAttribute>
   {
     public:
       // virtual function must be defined
          virtual SynthesizedAttribute evaluateSynthesizedAttribute (
                       SgNode* n, SynthesizedAttributesList childAttributes );
   };

SynthesizedAttribute
visitorTraversal::evaluateSynthesizedAttribute ( SgNode* n, SynthesizedAttributesList childAttributes )
   {
  // Fold up the list of child attributes using logical or, i.e. the local
  // result will be true iff one of the child attributes is true.
     SynthesizedAttribute localResult =
         std::accumulate(childAttributes.begin(), childAttributes.end(),
                         false, std::logical_or<bool>());

     if (isSgForStatement(n) != NULL)
        {
          printf ("Found a for loop ... \n");
          localResult = true;
        }

     return localResult;
   }

int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     SynthesizedAttribute result = exampleTraversal.traverse(project);

     if (result == true)
        {
          printf ("The program contains at least one loop!\n");
        }

     return 0;
   }
```

Figure 7.10: Example source code showing use of synthesized attributed (passing analysis information **up** the AST).

Figure 7.10 shows the use of attributes to pass information **up** the AST. The lifetime of the

```
Found a for loop ...
Found a for loop ...
The program contains at least one loop!
```

Figure 7.11: Output of input file to the synthesized attribute traversal.

attributes are similar as for inherited attributes. Attributes such as these are called synthesized attributes.

This code shows the code for a translator which does an analysis of an input source code to determine the presence of loops. It returns true if a loop exists in the input code and false otherwise. The list of synthesized attributes representing the information passed up the AST from a node's children is of type `SynthesizedAttributesList`, which is a type that behaves very similarly to `vector<SynthesizedAttribute>` (it supports iterators, can be indexed, and can be used with STL algorithms).

The example determines the existence of loops for a given program.

7.2.6 Accumulator Attributes

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

// Build an accumulator attribute, fancy name for what is essentially a global variable :-).
class AccumulatorAttribute
   {
     public:
          int forLoopCounter;

       // Specific constructors are optional
          AccumulatorAttribute () { forLoopCounter = 0; }
          AccumulatorAttribute ( const AccumulatorAttribute & X ) {}
          AccumulatorAttribute & operator= ( const AccumulatorAttribute & X ) { return *this; }
   };

class visitorTraversal : public AstSimpleProcessing
   {
     public:
          static AccumulatorAttribute accumulatorAttribute;
          virtual void visit(SgNode* n);
   };

// declaration required for static data member
AccumulatorAttribute visitorTraversal::accumulatorAttribute;

void visitorTraversal::visit(SgNode* n)
   {
     if (isSgForStatement(n) != NULL)
        {
          printf ("Found a for loop ... \n");
          accumulatorAttribute.forLoopCounter++;
        }
   }

int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
  // can be specified to be preorder or postorder).
     exampleTraversal.traverseInputFiles(project,preorder);

     printf ("Number of for loops in input application = %d \n",exampleTraversal.accumulatorAttribute.forLoopCounter);

     return 0;
   }
```

Figure 7.12: Example source code showing use of accumulator attributes (typically to count things in the AST).

Figure 7.12 shows the use of a different sort of attribute. This attribute has a lifetime equal to the lifetime of the traversal object (much longer than the traversal of any subset of IR nodes). The same attribute is accessible from each IR node. Such attributes are called *accumulator*

```
Found a for loop ...
Found a for loop ...
Number of for loops in input application = 2
```

Figure 7.13: Output of input file to the accumulator attribute traversal.

attributes and are semantically equivalent to a global variable. Accumulator attributes act as global variables which can easily be used to count application specific properties within the AST.

Note that due to the limitation that the computation of inherited attributes cannot be made dependent on the values of synthesized attributes, counting operations cannot be implemented by combining these attributes as is usually done in attribute grammars. However, the use of accumulator attributes serves well for this purpose. Therefore all counting-like operations should be implemented using accumulator attributes (= member variables of traversal or processing classes).

Although not shown in this tutorial explicitly, accumulator attributes may be easily mixed with inherited and/or synthesized attributes.

In this example we count the number of for-loops in an input program.

7.2.7 Inherited and Synthesized Attributes

Figure 7.14 shows the combined use of inherited and synthesized attributes. The example source code shows the mixed use of such attributes to list the functions containing loop. Inherited attributes are used to communicate that the traversal is in a function, which the synthesized attributes are used to pass back the existence of loops deeper within the subtrees associated with each function.

List of functions containing loops.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.

#include "rose.h"

#include <algorithm>
#include <functional>
#include <numeric>

typedef bool InheritedAttribute;
typedef bool SynthesizedAttribute;

class Traversal : public SgTopDownBottomUpProcessing<InheritedAttribute,SynthesizedAttribute>
   {
     public:
       // Functions required
          InheritedAttribute evaluateInheritedAttribute (
             SgNode* astNode,
             InheritedAttribute inheritedAttribute );

          SynthesizedAttribute evaluateSynthesizedAttribute (
             SgNode* astNode,
             InheritedAttribute inheritedAttribute,
             SubTreeSynthesizedAttributes synthesizedAttributeList );
   };

InheritedAttribute
Traversal::evaluateInheritedAttribute (
     SgNode* astNode,
     InheritedAttribute inheritedAttribute )
   {
     if (isSgFunctionDefinition(astNode))
        {
       // The inherited attribute is true iff we are inside a function.
          return true;
        }
     return inheritedAttribute;
   }

SynthesizedAttribute
Traversal::evaluateSynthesizedAttribute (
     SgNode* astNode,
     InheritedAttribute inheritedAttribute,
     SynthesizedAttributesList childAttributes )
   {
     if (inheritedAttribute == false)
        {
       // The inherited attribute is false, i.e. we are not inside any
       // function, so there can be no loops here.
          return false;
        }
     else
        {
       // Fold up the list of child attributes using logical or, i.e. the local
       // result will be true iff one of the child attributes is true.
          SynthesizedAttribute localResult =
          std::accumulate(childAttributes.begin(), childAttributes.end(),
                          false, std::logical_or<bool>());
          if (isSgFunctionDefinition(astNode) && localResult == true)
             {
               printf ("Found a function containing a for loop ...\n");
             }
          if (isSgForStatement(astNode))
             {
               localResult = true;
             }
          return localResult;
        }
   }

int
main ( int argc, char* argv[] )
   {
  // Build the abstract syntax tree
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the inherited attribute
     InheritedAttribute inheritedAttribute = false;

  // Define the traversal
     Traversal myTraversal;

  // Call the traversal starting at the project (root) node of the AST
     myTraversal.traverseInputFiles(project,inheritedAttribute);

  // This program only does analysis, so it need not call the backend to generate code.
     return 0;
   }
```

Figure 7.14: Example source code showing use of both inherited and synthesized attributes working together (part 1).

```
Found a function containing a for loop ...
Found a function containing a for loop ...
```

Figure 7.15: Output of input file to the inherited and synthesized attribute traversal.

7.2.8 Persistent Attributes

Figure 7.16 shows the use of another form of attribute. This attribute has a lifetime which is controlled explicitly by the user; it lives on the heap typically. These attributes are explicitly attached to the IR nodes and are not managed directly by the traversal. There attributes are called *persistent* attributes and are not required to be associated with any traversal. Persistent attributes are useful for storing information across multiple traversals (or permanently within the AST) for later traversal passes.

Persistent attributes may be used at any time and combined with other traversals (similar to accumulator attributes). Traversals may combine any or all of the types of attributes within in ROSE as needed to store, gather, or propagate information within the AST for complex program analysis.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

class persistantAttribute : public AstAttribute
   {
     public:
          int value;
          persistantAttribute (int v) : value(v) {}
   };

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

void visitorTraversalSetAttribute::visit(SgNode* n)
   {
     if (isSgForStatement(n) != NULL)
        {
          printf ("Found a for loop (set the attribute) ... \n");

       // Build an attribute (on the heap)
          AstAttribute* newAttribute = new persistantAttribute(5);
          ROSE_ASSERT(newAttribute != NULL);

       // Add it to the AST (so it can be found later in another pass over the AST)
          n->addNewAttribute("MyNewAttribute",newAttribute);
        }
   }

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

void visitorTraversalReadAttribute::visit(SgNode* n)
   {
     if (isSgForStatement(n) != NULL)
        {
          printf ("Found a for loop (read the attribute) ... \n");

       // Add it to the AST (so it can be found later in another pass over the AST)
       // AstAttribute* existingAttribute = n->attribute["MyNewAttribute"];
       // DQ (1/2/2006): Added support for new attribute interface.
       // printf ("visitorTraversalReadAttribute::visit(): using new attribute interface \n");
       // AstAttribute* existingAttribute = n->attribute()["MyNewAttribute"];
          AstAttribute* existingAttribute = n->getAttribute("MyNewAttribute");
          ROSE_ASSERT(existingAttribute != NULL);

          printf ("Existing attribute at %p value = %d \n",n,dynamic_cast<persistantAttribute*>(existingAttrib
        }
   }

int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object to set persistant AST attributes
     visitorTraversalSetAttribute exampleTraversalSettingAttribute;

  // Call the traversal starting at the project node of the AST
     exampleTraversalSettingAttribute.traverseInputFiles(project,preorder);

  // Build the traversal object to read any existing AST attributes
     visitorTraversalReadAttribute exampleTraversalReadingAtribute;

  // Call the traversal starting at the project node of the AST
     exampleTraversalReadingAtribute.traverseInputFiles(project,preorder);

     return 0;
   }
```

Figure 7.16: Example source code showing use of persistent attributes used to pass information across multiple passes over the AST.

```
Found a for loop (set the attribute) ...
Found a for loop (set the attribute) ...
Found a for loop (read the attribute) ...
Existing attribute at 0x2b8b77d18010 value = 5
Found a for loop (read the attribute) ...
Existing attribute at 0x2b8b77d18130 value = 5
```

Figure 7.17: Output of input file to the persistent attribute traversal showing the passing of information from one AST traversal to a second AST traversal.

7.2.9 Nested Traversals

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

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

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

void visitorTraversal::visit(SgNode* n)
   {
     if (isSgFunctionDeclaration(n) != NULL)
        {
          printf ("Found a function declaration ... \n");

       // Build the nested traversal object
          nestedVisitorTraversal exampleTraversal;

       // Call the traversal starting at the project node of the AST (traverse in postorder just to be differe
       // Note that we call the traverse function instead of traverseInputFiles, because we are not starting a
       // the AST root.
          exampleTraversal.traverse(n,postorder);
        }
   }

void nestedVisitorTraversal::visit(SgNode* n)
   {
     if (isSgFunctionDefinition(n) != NULL)
        {
          printf ("Found a function definition within the function declaration ... \n");
        }
   }

int
main ( int argc, char* argv[] )
   {
     if (SgProject::get_verbose() > 0)
          printf ("In visitorTraversal.C: main() \n");

     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project,preorder);

     return 0;
   }
```

Figure 7.18: Example source code showing use nested traversals.

Figure 7.18 shows the use of multiple traversals in composition. Figure 7.19 shows the output

```
Found a function declaration ...
Found a function declaration ...
Found a function definition within the function declaration ...
Found a function declaration ...
Found a function declaration ...
Found a function declaration ...
Found a function declaration ...
Found a function definition within the function declaration ...
```

Figure 7.19: Output of input file to the nested traversal example.

of the nested traversal.

7.2.10 Combining all Attributes and Using Primitive Types

```
int main() {
  int x=1;
  for(int i=1; i<10;i++)
    for(int j=i;j<10;j++)
      for(int k=i;k<10;k++)
        for(int l=i;l<10;l++)
          for(int m=i;m<10;m++)
            x++;

  int i=5,j=7;
  while(i>0) {
    while(j>0) {
      x++;
      j--;
      i--;
    }
  }

  i=10;
  do {
    x++;
    i--;
  } while (i>0);

  return x;
}
```

Figure 7.20: Input code with nested loops for nesting info processing

The previous examples have shown cases where attributes were classes, alternatively attributes can be any primitive type (int, bool, etc.). This example demonstrates how to use AstTopDownBottomUpProcessing to compute inherited and synthesized attributes, generate pdf and dot output, how to accumulate information, and how to attach attributes to AST nodes in the same pass.

The attributes are used to compute the nesting level and the nesting depth of for/while/do-while loops: The nesting level is computed using an inherited attribute. It holds that $nesting - level(innerloop) = nesting - level(outerloop) + 1$ starting with 1 at the outer most loop. The nesting depth is computed using a synthesized attribute. It holds that $nesting-depth(innerloop) = nesting - level(outerloop) - 1$ starting with 1 at the inner most loop.

To compute the values we use a primitive type (unsigned int). This example also shows how to use defaultSynthesizedAttribute to initialize a synthesized attribute of primitive type. The values of the attributes are attached to the AST using AstAttribute and the AST node attribute mechanism available at every AST node (which can be accessed with `node->attribute`). (see loopNestingInfoProcessing.C)

For the entire program the maximum nesting level (= max nesting depth) is computed as accumulated value using member variable `_maxNestingLevel` of class LoopNestingInfoProcessing. We also demonstrate how to customize an AstAttribute such that the value of the attribute is printed in a pdf output. (by overriding toString, see LoopNestingInfo class)

In the generated pdf file (for some C++ input file) the values of the attributes can be viewed for each node (see printLoopInfo implementation). Further more we also generate a dot file, to visualize the tree using the graph visualization tool dot. The generated file can be converted to postscript (using dot) and viewed with gv.

7.2.11 Combined Traversals

Performing a large number of program analyses as separate traversals of the AST can be somewhat inefficient as there is some overhead associated with visiting every node several times. ROSE therefore provides a mechanism for combining traversal objects of the same base type and evaluating them in a single traversal of the AST. This is entirely transparent to the individual traversal object, so existing code can be reused with the combination mechanism, and analyzers can be developed and tested in isolation and combined when needed.

The one requirement that is placed on traversals to be combined is that they be independent of each other; in particular, this means that they should not modify the AST or any shared global data. Any output produced by the analyzers will be interleaved.

Figure 7.24 shows the source code for a translator that combines three different analyzers into one traversal, each one counting the occurrences of a different type of AST node (as determined by a VariantT value). First three traversals are run after each other, as usual; then three traversal objects are passed (by pointer) to an object of type `AstCombinedSimpleProcessing` using its `addTraversal` method. One then invokes one of the usual traverse methods on this combined object with the same effect as if it had been called for each of the traversal objects individually.

Any operation on the list of analyzers is possible using the `get_traversalPtrListRef` method of the combined processing class that returns a reference to its internal list of analyzers (an object of type `vector<AstSimpleProcessing *>`). Any changes made through this reference will be reflected in further traversals.

In addition to `AstCombinedSimpleProcessing`, there is also a combined class for each of the other types of traversals discussed above: `AstCombinedTopDownProcessing`, `AstCombinedBottomUpProcessing`, etc. Where traversals using attributes are combined, all of the combined traversals must have the same attribute types (i. e. the same template parameters). Attributes are passed to and returned from the combined traversal as a vector.

```
// Author: Markus Schordan, Vienna University of Technology, 2004.
// $Id: loopNestingInfoProcessing.C,v 1.1 2006/04/24 00:22:00 dquinlan Exp $

// #include <string>
// #include <iostream>

#include "rose.h"

using namespace std;

typedef unsigned int NestingLevel;
typedef unsigned int NestingDepth;
typedef NestingLevel InhNestingLevel;
typedef NestingDepth SynNestingDepth;

/*! This class is used to attach information to AST nodes.
    Method 'toString' is overridden and
    called when a pdf file is generated. This allows to display
    the value of an AST node attribute (annotation) in a pdf file.
*/
class NestingLevelAnnotation : public AstAttribute {
public:
  NestingLevelAnnotation(NestingLevel n,NestingDepth d)
    : _nestingLevel(n),_nestingDepth(d) {}
  NestingLevel getNestingLevel() { return _nestingLevel; }
  NestingDepth getNestingDepth() { return _nestingDepth; }
  string toString() {
    ostringstream ss; ss<<_nestingLevel<<","<<_nestingDepth;
    return ss.str();
  }
private:
  NestingLevel _nestingLevel;
  NestingDepth _nestingDepth;
};

/* !
The loop nesting level and nesting depth for each while/dowhile/for
loop nest is computed. It is attached to the AST as annotation and
can be accessed as node->attribute["loopNestingInfo"]
after the processing has been performed.
The maximum nesting level of the whole AST is computed as
"accumulated" value in a member variable and can be accessed with
getMaxNestingLevel().
*/
class LoopLevelProcessing : public AstTopDownBottomUpProcessing<InhNestingLevel,SynNestingDepth> {
public:
  LoopLevelProcessing():_maxNestingLevel(0) {}

  /*! Performs a traversal of the AST and computes loop-nesting information by using
      inherited and synthesized attributes. The results are attached to the AST as
      annotation.
  */
  void attachLoopNestingAnnotaton(SgProject* node) { traverseInputFiles(node,0); }

  /*! Returns the maximum nesting level of the entire AST (of the input file).
      Requires attachLoopNestingAnnotation (to be called before)
  */
  NestingLevel getMaxNestingLevel() { return _maxNestingLevel; }

protected:
  //! computes the nesting level
  InhNestingLevel evaluateInheritedAttribute(SgNode*,InhNestingLevel);
  //! computes the nesting depth
  SynNestingDepth evaluateSynthesizedAttribute(SgNode*,InhNestingLevel,SynthesizedAttributesList);
  //! provides the default value 0 for the nesting depth
  SynNestingDepth defaultSynthesizedAttribute(InhNestingLevel inh);
private:
  NestingLevel _maxNestingLevel;
};


NestingLevel
LoopLevelProcessing::evaluateInheritedAttribute(SgNode* node, NestingLevel loopNestingLevel) {

  //! compute maximum nesting level of entire program in accumulator (member variable)
  if(loopNestingLevel>_maxNestingLevel)
    _maxNestingLevel=loopNestingLevel;

  switch(node->variantT()) {
  case V_SgGotoStatement:
```

Figure 7.21: Example source code showing use of inherited, synthesized, accumulator, and per-

```
      cout << "WARNING: Goto statement found. We do not consider goto loops.\n";
 // DQ (11/17/2005): Added return statment to avoid g++ warning: control reaches end of non-void function
      return loopNestingLevel;
      break;
   case V_SgDoWhileStmt:
   case V_SgForStatement:
   case V_SgWhileStmt:
      return loopNestingLevel+1;
   default:
      return loopNestingLevel;
   }
}

SynNestingDepth
LoopLevelProcessing::defaultSynthesizedAttribute(InhNestingLevel inh) {
   /*! we do not need the inherited attribute here
       as default value for synthesized attribute we set 0, representing nesting depth 0.
   */
   return 0;
}

SynNestingDepth
LoopLevelProcessing::evaluateSynthesizedAttribute(SgNode* node,InhNestingLevel nestingLevel,SynthesizedAttributesList l)
    {
      if (nestingLevel>_maxNestingLevel)
           _maxNestingLevel=nestingLevel;

   // compute maximum nesting depth of synthesized attributes
      SynNestingDepth nestingDepth=0;
      for(SynthesizedAttributesList::iterator i=l.begin();i!=l.end();i++)
         {
           if(*i>nestingDepth) nestingDepth=*i;
         }

      switch(node->variantT())
         {
           case V_SgDoWhileStmt:
           case V_SgForStatement:
           case V_SgWhileStmt:
              {
                nestingDepth++;
                cout << "Nesting level:" << nestingLevel << ", nesting depth:" << nestingDepth << endl;
                break;
              }

           default:
              {
             // DQ (11/17/2005): Nothing to do here, but explicit default in switch avoids lots of warnings.
              }
         }

   // add loop nesting level as annotation to AST
      NestingLevelAnnotation* nla = new NestingLevelAnnotation(nestingLevel,nestingDepth);
      ROSE_ASSERT(nla != NULL);

   // DQ (1/2/2006): Added support for new attribute interface.
   // printf ("LoopLevelProcessing::evaluateSynthesizedAttribute(): using new attribute interface \n");
#if 0
      if (node->get_attribute() == NULL)
         {
           AstAttributeMechanism* attributePtr = new AstAttributeMechanism();
           ROSE_ASSERT(attributePtr != NULL);
           node->set_attribute(attributePtr);
         }
#endif
// node->attribute.add("loopNestingInfo",nla);
// node->attribute().add("loopNestingInfo",nla);
    node->addNewAttribute("loopNestingInfo",nla);

   //! return the maximum nesting depth as synthesized attribute
   return nestingDepth;
}

int main ( int argc, char** argv) {

    // command line parameters are passed to EDG
    // non-EDG parameters are passed (through) to ROSE (and the vendor compiler)
    SgProject* root=frontend(argc,argv);
    LoopLevelProcessing t;
```

Figure 7.22: Example source code showing use of inherited, synthesized, accumulator, and per-

```

Output:
Nesting level:5, nesting depth:1
Nesting level:4, nesting depth:2
Nesting level:3, nesting depth:3
Nesting level:2, nesting depth:4
Nesting level:1, nesting depth:5
Nesting level:2, nesting depth:1
Nesting level:1, nesting depth:2
Nesting level:1, nesting depth:1
Max loop nesting level: 5
```

Figure 7.23: Output code showing the result of using inherited, synthesized, and accumulator attributes.

```
#include <rose.h>

class NodeTypeCounter: public AstSimpleProcessing {
public:
    NodeTypeCounter(enum VariantT variant, std::string typeName)
        : myVariant(variant), typeName(typeName), count(0) {
    }

protected:
    virtual void visit(SgNode *node) {
        if (node->variantT() == myVariant) {
            std::cout << "Found " << typeName << std::endl;
            count++;
        }
    }

    virtual void atTraversalEnd() {
        std::cout << typeName << " total: " << count << std::endl;
    }

private:
    enum VariantT myVariant;
    std::string typeName;
    unsigned int count;
};

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

    std::cout << "sequential execution of traversals" << std::endl;
    NodeTypeCounter forStatementCounter(V_SgForStatement, "for loop");
    NodeTypeCounter intValueCounter(V_SgIntVal, "int constant");
    NodeTypeCounter varDeclCounter(V_SgVariableDeclaration, "variable declaration");
    // three calls to traverse, executed sequentially
    forStatementCounter.traverseInputFiles(project, preorder);
    intValueCounter.traverseInputFiles(project, preorder);
    varDeclCounter.traverseInputFiles(project, preorder);
    std::cout << std::endl;

    std::cout << "combined execution of traversals" << std::endl;
    AstCombinedSimpleProcessing combinedTraversal;
    combinedTraversal.addTraversal(new NodeTypeCounter(V_SgForStatement, "for loop"));
    combinedTraversal.addTraversal(new NodeTypeCounter(V_SgIntVal, "int constant"));
    combinedTraversal.addTraversal(new NodeTypeCounter(V_SgVariableDeclaration, "variable declaration"));
    // one call to traverse, execution is interleaved
    combinedTraversal.traverseInputFiles(project, preorder);
}
```

Figure 7.24: Example source showing the combination of traversals.

```
sequential execution of traversals
Found for loop
Found for loop
for loop total: 2
Found int constant
Found int constant
Found int constant
Found int constant
Found int constant
Found int constant
Found int constant
Found int constant
Found int constant
Found int constant
int constant total: 10
Found variable declaration
Found variable declaration
Found variable declaration
Found variable declaration
Found variable declaration
Found variable declaration
Found variable declaration
Found variable declaration
variable declaration total: 8

combined execution of traversals
Found variable declaration
Found int constant
Found variable declaration
Found for loop
Found variable declaration
Found int constant
Found int constant
Found variable declaration
Found int constant
Found int constant
Found variable declaration
Found int constant
Found variable declaration
Found int constant
Found for loop
Found variable declaration
Found int constant
Found int constant
Found variable declaration
Found int constant
for loop total: 2
int constant total: 10
variable declaration total: 8
```

Figure 7.25: Output of input file to the combined traversals. Note that the order of outputs changes as execution of several analyzers is interleaved.

7.2.12 Short-Circuiting Traversals

The traversal short-circuit mechanism is a simple way to cut short the traversal of a large AST once specific information has been obtained. It is purely an optimization mechanism, and a bit of a hack, but common within the C++ Boost community. Since the technique works we present it as a way of permitting users to avoid the full traversal of an AST that they might deam to be redundant of inappropriate. We don't expect that this mechanism will be particularly useful to most users and we don't recommend it. It may even at some point not be supported. However, we present it because it is a common technique used in the C++ Boost community and it happens to work (at one point it didn't work and so we have no idea what we fixed that permitted it to work now). We have regarded this technique as a rather ugly hack. It is presented in case you really need it. It is, we think, better than the direct use of lower level mechanisms that are used to support the AST traversal.

```

// Input for translator to show exception-based exiting from a translator.

namespace A
{
  int __go__;
  struct B
  {
    static int __stop__;
  };
};

void foo (void)
{
  extern void bar (int);
  bar (A::__go__);
  bar (A::B::__stop__);
}
```

Figure 7.26: Input code with used to demonstrate the traversal short-circuit mechanism.

Figure 7.27 shows the example code demonstrating a traversal setup to support the short-circuit mechanism (a conventional mechanism used often within the C++ Boost community). The input code shown in figure 7.26 is compiled using the example translator, the output is shown in figure 7.28.

The output shown in figure 7.28 demonstrates the initiation of a traversal over the AST and that traversal being short-circuited after a specific point in the evaluation. The result is that there is no further traversal of the AST after that point where it is short-circuited.

```
// Example of an AST traversal that uses the Boost idiom of throwing
// an exception to exit the traversal early.

#include <rose.h>
#include <string>
#include <iostream>

using namespace std;

// Exception to indicate an early exit from a traversal at some node.
class StopEarly
{
public:
  StopEarly (const SgNode* n) : exit_node_ (n) {}
  StopEarly (const StopEarly& e) : exit_node_ (e.exit_node_) {}

  // Prints information about the exit node.
  void print (ostream& o) const
  {
    if (exit_node_) {
      o << '\t' << (const void *)exit_node_ << ":" << exit_node_->class_name () << endl;
      const SgLocatedNode* loc_n = isSgLocatedNode (exit_node_);
      if (loc_n) {
        const Sg_File_Info* info = loc_n->get_startOfConstruct ();
        ROSE_ASSERT (info);
        o << "\tAt " << info->get_filename () << ":" << info->get_line () << endl;
      }
    }
  }

private:
  const SgNode* exit_node_; // Node at early exit from traversal
};

// Preorder traversal to find the first SgVarRefExp of a particular name.
class VarRefFinderTraversal : public AstSimpleProcessing
{
public:
  // Initiate traversal to find 'target' in 'proj'.
  void find (SgProject* proj, const string& target)
  {
    target_ = target;
    traverseInputFiles (proj, preorder);
  }

  void visit (SgNode* node)
  {
    const SgVarRefExp* ref = isSgVarRefExp (node);
    if (ref) {
      const SgVariableSymbol* sym = ref->get_symbol ();
      ROSE_ASSERT (sym);
      cout << "Visiting SgVarRef '" << sym->get_name ().str () << "'" << endl;
      if (sym->get_name ().str () == target_) // Early exit at first match.
        throw StopEarly (ref);
    }
  }

private:
  string target_; // Symbol reference name to find.
};

int main (int argc, char* argv[])
{
  SgProject* proj = frontend (argc, argv);
  VarRefFinderTraversal finder;

  // Look for a reference to "__stop__".
  try {
    finder.find (proj, "__stop__");
    cout << "*** Reference to a symbol '__stop__' not found. ***" << endl;
  } catch (StopEarly& stop) {
    cout << "*** Reference to a symbol '__stop__' found. ***" << endl;
    stop.print (cout);
  }

  // Look for a reference to "__go__".
  try {
    finder.find (proj, "__go__");
    cout << "*** Reference to a symbol '__go__' not found. ***" << endl;
  } catch (StopEarly& go) {
    cout << "*** Reference to a symbol '__go__' found. ***" << endl;
    go.print (cout);
  }
```

```
Visiting SgVarRef '__go__'
Visiting SgVarRef '__stop__'
*** Reference to a symbol '__stop__' found. ***
        0x201317f8:SgVarRefExp
        At /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_traversal
Visiting SgVarRef '__go__'
*** Reference to a symbol '__go__' found. ***
        0x20131790:SgVarRefExp
        At /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_traversal
```

Figure 7.28: Output code showing the result of short-circuiting the traversal.

7.3 Memory Pool Traversals

Allocation of IR nodes in ROSE is made more efficient through the use of specialized allocators implemented at member function new operators for each class of the IR in Sage III. Such specialized memory allocators avoid significant fragmentation of memory, provide more efficient packing of memory, improve performance of allocation of memory and IR node access, and additionally provide a secondary mechanism to accessing all the IR nodes. Each IR node has a memory pool which is an STL vector of blocks (a fixed or variable sized array of contiguously stored IR nodes).

The three types of traversals are:

1. ROSE Memory Pool Visit Traversal
   This traversal is similar to the one provided by the SimpleProcessing Class (using the visit() function and no inherited or synthesized attributes).

2. Classic Object-Oriented Visitor Pattern for Memory Pool
   This is a classic object-oriented visitor pattern.

3. IR node type traversal, visits one type of IR node for all IR types in the AST. This is useful for building specialized tools.

7.3.1 ROSE Memory Pool Visit Traversal

Figure 7.29 shows the source code for a translator which traverses the memory pool containing the AST. At each node the `visit()` function is called using only the input information represented by the current node. Note that using this simple traversal no context information is available to the visit function. All the IR nodes for a given memory pool are iterated over at one time. The order of the traversal of the different memory pools is random but fixed. Thus the order of the traversal of the IR nodes is in no way connected to the structure of the AST (unlike the previous non-memory pool traversals that were very much tied to the structure of the AST and which matched the structure of the original input source code being compiled).

```

#include "rose.h"

// ROSE Visit Traversal (similar interface as Markus's visit traversal)
// in ROSE (implemented using the traversal over
// the elements stored in the memory pools so it has no cycles and visits
// ALL IR nodes (including all Sg_File_Info, SgSymbols, SgTypes, and the
// static builtin SgTypes).
class RoseVisitor : public ROSE_VisitTraversal
   {
     public:
          int counter;
          void visit ( SgNode* node);

          RoseVisitor() : counter(0) {}
   };


void RoseVisitor::visit ( SgNode* node)
   {
  // printf ("roseVisitor::visit: counter %4d node = %s \n",counter,node->class_name().c_str());
     counter++;
   }

int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // ROSE visit traversal
     RoseVisitor visitor;
     visitor.traverseMemoryPool();

     printf ("Number of IR nodes in AST = %d \n",visitor.counter);

     return backend(project);
   }
```

Figure 7.29: Example source showing simple visit traversal over the memory pools.

```
Number of IR nodes in AST = 7705
```

Figure 7.30: Output of input file to the visitor traversal over the memory pool.

7.3.2 Classic Object-Oriented Visitor Pattern for Memory Pool

Figure 7.31 shows the source code for a translator which traverses the memory pools containing the AST. At each node the `visit()` function is called using only the input information represented by the current node. Note that using this simple traversal no context information is available to the visit function. The traversal order is the same as in the 7.29.

```

#include "rose.h"

// Classic Visitor Pattern in ROSE (implemented using the traversal over
// the elements stored in the memory pools so it has no cycles and visits
// ALL IR nodes (including all Sg_File_Info, SgSymbols, SgTypes, and the
// static builtin SgTypes).
class ClassicVisitor : public ROSE_VisitorPattern
   {
     public:
       // Override virtual function defined in base class
          void visit(SgGlobal* globalScope)
             {
               printf ("Found the SgGlobal IR node \n");
             }

          void visit(SgFunctionDeclaration* functionDeclaration)
             {
               printf ("Found a SgFunctionDeclaration IR node \n");
             }
          void visit(SgTypeInt* intType)
             {
               printf ("Found a SgTypeInt IR node \n");
             }

          void visit(SgTypeDouble* doubleType)
             {
               printf ("Found a SgTypeDouble IR node \n");
             }
   };


int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Classic visitor pattern over the memory pool of IR nodes
     ClassicVisitor visitor_A;
     traverseMemoryPoolVisitorPattern(visitor_A);

     return backend(project);
   }
```

Figure 7.31: Example source showing simple visitor pattern.

```
Found a SgTypeInt IR node
Found a SgTypeDouble IR node
Found the SgGlobal IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
Found a SgFunctionDeclaration IR node
```

7.3.3 ROSE IR Type Traversal (uses Memory Pools)

Figure 7.33 shows the source code for a translator which traverses only one type of IR node using the memory pool containing the AST. This traversal is useful for building specialized tools (often tools which only call static functions on each type of IR node).

This example shows the use of an alternative traversal which traverses a representative of each type or IR node just one, but only if it exists in the AST (memory pools). This sort of traversal is useful for building tools that need only operate on static member functions of the IR nodes or need only sample one of each type or IR node present in the AST. this specific example also appears in: *ROSE/src/midend/astDiagnostics/AstStatistics.C*.

The user's use of the traversal is the same as for other ROSE AST traversals except that the ROSE_VisitTraversal::traverseRepresentativeIRnodes() member function is called instead of ROSE_VisitTraversal::traverseMemoryPool().

This mechanism can be used to generate more complete reports of the memory consumption of the AST, which is reported on if *-rose:verbose 2* is used. Figure 7.35 shows a partial 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), sorted by memory consumption. The AST contains approximately 280K IR nodes. Note that the Sg_File_Info IR nodes is most frequent and consumes 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. *Note: more complete information about the memory use of the AST in in the ROSE User Manual appendix.*

```
// This example code shows the traversal of IR types not available using the other traversal mechanism.
#include "rose.h"
using namespace std;

// CPP Macro to implement case for each IR node (we could alternatively use a visitor pattern and a function template, ma
#define IR_NODE_VISIT_CASE(X) \
          case V_##X: \
             { \
               X* castNode = is##X(node); \
               int numberOfNodes   = castNode->numberOfNodes(); \
               int memoryFootprint = castNode->memoryUsage(); \
               printf ("count = %7d, memory use = %7d bytes, node name = %s \n",numberOfNodes,memoryFootprint,castNode->c
               break; \
             }

class RoseIRnodeVisitor : public ROSE_VisitTraversal {
     public:
          int counter;
          void visit ( SgNode* node);
          RoseIRnodeVisitor() : counter(0) {}
   };

void RoseIRnodeVisitor::visit ( SgNode* node)
   {
  // Using a classic visitor pattern should avoid all this casting,
  // but each function must be created separately (so it is wash if
  // we want to do all IR nodes, as we do here).
     switch(node->variantT())
        {
          IR_NODE_VISIT_CASE(Sg_File_Info)
          IR_NODE_VISIT_CASE(SgPartialFunctionType)
          IR_NODE_VISIT_CASE(SgFunctionType)
          IR_NODE_VISIT_CASE(SgPointerType)
          IR_NODE_VISIT_CASE(SgFunctionDeclaration)
          IR_NODE_VISIT_CASE(SgFunctionSymbol)
          IR_NODE_VISIT_CASE(SgSymbolTable)
          IR_NODE_VISIT_CASE(SgInitializedName)
          IR_NODE_VISIT_CASE(SgStorageModifier)
          IR_NODE_VISIT_CASE(SgForStatement)
          IR_NODE_VISIT_CASE(SgForInitStatement)
          IR_NODE_VISIT_CASE(SgCtorInitializerList)
          IR_NODE_VISIT_CASE(SgIfStmt)
          IR_NODE_VISIT_CASE(SgExprStatement)
          IR_NODE_VISIT_CASE(SgTemplateDeclaration)
          IR_NODE_VISIT_CASE(SgTemplateInstantiationDecl)
          IR_NODE_VISIT_CASE(SgTemplateInstantiationDefn)
          IR_NODE_VISIT_CASE(SgTemplateInstantiationMemberFunctionDecl)
          IR_NODE_VISIT_CASE(SgClassSymbol)
          IR_NODE_VISIT_CASE(SgTemplateSymbol)
          IR_NODE_VISIT_CASE(SgMemberFunctionSymbol)

          default:
             {
#if 0
               printf ("Case not handled: %s \n",node->class_name().c_str());
#endif
             }
        }
   }


int
main ( int argc, char* argv[] )
   {
  // ROSE visit traversal
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // ROSE visit traversal
     RoseIRnodeVisitor visitor;
     visitor.traverseRepresentativeIRnodes();
     printf ("Number of types of IR nodes (after building AST) = %d \n",visitor.counter);

#if 1
  // IR nodes statistics
     if (project->get_verbose() > 1)
          cout << AstNodeStatistics::IRnodeUsageStatistics();
#endif

     int errorCode = 0;
     errorCode = backend(project);

     return errorCode;
```

```
count =      24, memory use =    3840 bytes, node name = SgSymbolTable
count =     750, memory use =   30000 bytes, node name = SgStorageModifier
count =    3424, memory use =  246528 bytes, node name = Sg_File_Info
No representative for SgPartialFunctionType found in memory pools
count =     399, memory use =   47880 bytes, node name = SgFunctionType
count =      12, memory use =    1152 bytes, node name = SgPointerType
count =       2, memory use =     576 bytes, node name = SgForStatement
count =       2, memory use =     208 bytes, node name = SgForInitStatement
count =       4, memory use =    2272 bytes, node name = SgCtorInitializerList
count =       1, memory use =     296 bytes, node name = SgIfStmt
count =       9, memory use =     792 bytes, node name = SgExprStatement
count =       6, memory use =    3984 bytes, node name = SgTemplateDeclaration
count =       5, memory use =    3800 bytes, node name = SgTemplateInstantiationDecl
count =       1, memory use =     296 bytes, node name = SgTemplateInstantiationDefn
count =       3, memory use =    3048 bytes, node name = SgTemplateInstantiationMemberFunctionDecl
count =     424, memory use =  390080 bytes, node name = SgFunctionDeclaration
count =       1, memory use =      48 bytes, node name = SgClassSymbol
count =       3, memory use =     144 bytes, node name = SgTemplateSymbol
count =       2, memory use =      96 bytes, node name = SgMemberFunctionSymbol
count =     399, memory use =   19152 bytes, node name = SgFunctionSymbol
count =     750, memory use =  222000 bytes, node name = SgInitializedName
Number of types of IR nodes (after building AST) = 0
```

Figure 7.34: Output of input file to the IR Type traversal over the memory pool.

```
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

Truncated results presented ...
```

Figure 7.35: Example of output using -rose:verbose 2 (memory use report for AST).

Chapter 8

Graph Processing Tutorial

8.1 Traversal Tutorial

ROSE can collect and analyze paths in both source and binary CFGs. At moment it doesn't attempt to save paths because if you save them directly the space necessary is extremely large, as paths grow 2^n with successive if statements and even faster when for loops are involved. Currently a path can only cannot complete the same loop twice. However it is possible for a graph such that 1 -¿ 2 , 2-¿3, 3-¿1, 3-¿5, has paths, 1,2,3,1,2,3,5 and 1,2,3,5 because the loop 1,2,3,1 is not repeated.

The tutorial example works as such:

```
#include <iostream>
#include <fstream>
//#include <rose.h>
#include <string>
#include <err.h>
#include "SgGraphTemplate.h"
#include "graphProcessing.h"

#include "staticCFG.h"
#include "interproceduralCFG.h"
/* Testing the graph traversal mechanism now implementing in AstProcessing.h (inside src/midend/astProcessing/
#include <sys/time.h>
#include <sys/resource.h>
using namespace std;
using namespace boost;

/* You need to use myGraph type here because the conversion of StaticCFG::InterproceduralCFG or StaticCFG::CFG
in a boost form. The SgGraphTemplate.h file handles this conversion and myGraph is specific to that file */
typedef myGraph CFGforT;

/*
Your basic visitor traversal subclassed from SgGraphTraversal on the CFGforT template as defined
above
*/
class visitorTraversal : public SgGraphTraversal<CFGforT>
   {
     public:
          int paths;
        /* This is the function run by the algorithm on every path, VertexID is a type implemented in SgGraphT
          void analyzePath(vector<VertexID>& pth);
   };

/* defining the analyzePath function. This simply counts paths as should be obvious. Again, VertexID is define
void visitorTraversal::analyzePath(vector<VertexID>& pth) {
        paths++;
}

int main(int argc, char *argv[]) {
  /* First you need to produce the project file*/
  SgProject* proj = frontend(argc,argv);
  ROSE_ASSERT (proj != NULL);
 /* Getting the Function Declaration and Definition for producing the graph */
  SgFunctionDeclaration* mainDefDecl = SageInterface::findMain(proj);
  SgFunctionDefinition* mainDef = mainDefDecl->get_definition();
/* Instantiating the visitorTraversal */
    visitorTraversal* vis = new visitorTraversal();
/* This creates the StaticCFG::InterproceduralCFG object to be converted to a boost graph */
     StaticCFG::InterproceduralCFG cfg(mainDef);
     stringstream ss;
     SgIncidenceDirectedGraph* g = new SgIncidenceDirectedGraph();
     /* We got the necessary internal SgIncidenceDirectedGraph from the cfg */
     g = cfg.getGraph();
     myGraph* mg = new myGraph();
/* Converting the cfg to a boost graph */
     mg = instantiateGraph(g, cfg, mainDef);
/* Set internal variables */
     vis->paths = 0;
     /* invoking the traversal, the first argument is the graph, the second is true if you
     do not want bounds, false if you do, the third and fourth arguments are starting and stopping
     vertices respectively, if you are not bounding simply insert 0. Finally the last argument is
     currently deprecated */
     vis->constructPathAnalyzer(mg, true, 0, 0, true);
     std::cout << "finished" << std::endl;
     std::cout << " paths: " << vis->paths << std::endl;
     delete vis;
}
```

Figure 8.1: Source CFG Traversal Example

```cpp
#include <iostream>
#include <fstream>
#include <rose.h>
//#include "interproceduralCFG.h"
#include <string>
#include <err.h>

/* These are necessary for any binary Traversal */

#include "graphProcessing.h"
#include "BinaryControlFlow.h"
#include "BinaryLoader.h"
/* Testing the graph traversal mechanism now implementing in graphProcessing.h (inside src/midend/astProcessing/)*/

using namespace std;
using namespace boost;

/* These should just be copied verbatim */

typedef boost::graph_traits<BinaryAnalysis::ControlFlow::Graph>::vertex_descriptor Vertex;
/**< Graph vertex type. */
typedef boost::graph_traits<BinaryAnalysis::ControlFlow::Graph>::edge_descriptor
Edge;      /**< Graph edge type. */



/* We first make a visitorTraversal, subclassed from SgGraphTraversal templated on the BinaryAnalysis:ControlFlow::Graph
which is implemented as a boost graph */


class visitorTraversal : public SgGraphTraversal<BinaryAnalysis::ControlFlow::Graph>
   {

     public:
         long int pths;
         long int tltnodes;
          /* This needs to be in any visitorTraversal, it is the function that will be run on every path by the graph
          path analysis algorithm, notice the Vertex type is from the above typedefs */
          virtual void analyzePath( vector<Vertex>& pth);


   };


/* This is a very simple incarnation, it just counts paths */
void visitorTraversal::analyzePath(vector<Vertex>& pth) {
      pths++;
}


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

     /* Parse the binary file */
     SgProject *project = frontend(argc, argv);
     std::vector<SgAsmInterpretation*> interps = SageInterface::querySubTree<SgAsmInterpretation>(project);
     if (interps.empty()) {
         fprintf(stderr, "no binary interpretations found\n");
         exit(1);
     }

     /* Calculate plain old CFG. */
         BinaryAnalysis::ControlFlow cfg_analyzer;
         BinaryAnalysis::ControlFlow::Graph* cfg = new BinaryAnalysis::ControlFlow::Graph;

         cfg_analyzer.build_cfg_from_ast(interps.back(), *cfg);
         std::ofstream mf;
         mf.open("analysis.dot");
         /* Declaring the visitorTraversal */
         visitorTraversal* vis = new visitorTraversal;
         /* Setting internal variables*/
         vis->tltnodes = 0;
         vis->pths = 0;

         /* visitorTraversal has 5 arguments, the first is the ambient CFG, the second identifies whether or not
         you are bounding the graph, that is, whether you want all your paths to start at one specific node and end at
        another specific node, the fourth and fifth would be start and end if the graph were bounded. Since they aren't
        you can simply input 0, for the moment the final argument is deprecated, though it's purpose was to tell the progra
         that your analysis function was thread safe, that is that openMP could run it without having a critical command.
        Currently a critical is always used */

         vis->constructPathAnalyzer(cfg, true, 0, 0, false);
         std::cout << "pths: " << vis->pths << std::endl;
         std::cout << "tltnodes: " << vis->tltnodes << std::endl;
```

Chapter 9

Scopes of Declarations

The scope of an IR node may be either stored explicitly in the IR node or obtained through computation through its parent information in the AST. Figure X shows an example where the variable definition for a variable is the scope of namespace X. The declaration for variable a is in the namespace X. In a more common way, the function foo is a member function of B with a declaration appearing in class B, but with a function definition in global scope.

```
namespace X{
   extern int a;
}
int X::a = 0;
class B
  {
    void foo();
  };
void B::foo() {}
```

In C++, using name qualification the scope of a declaration can be independent of it structural location in the AST. The `get_parent()` member function (available on most IR nodes) communicates the structural information of the original source code (also represented in the AST). The scope information must at times be stored explicitly when it can not be interpreted structurally.

The example in this chapter show how to find the scope of each C++ construct . Note that SgExpression IR nodes can take their scope from that of the statement where they are found. SgStatement and SgInitializedName IR nodes are the interesting IR nodes from the point of scope.

The SgInitializedName and all SgStatement IR nodes have a member function `get_scope()` which returns the scope of the associated IR nodes. The example code in this chapter traverses the AST and reports the scope of any SgInitializedName and all SgStatement IR nodes. It is intended to provide a simple intuition about what the scope can be expected to be in an application. The example code is also useful as a simple means of exploring the scopes of any other input application.

9.1 Input For Examples Showing Scope Information

Figure 9.1 shows the input example code form this tutorial example.

```

int xyz;

void foo (int x)
   {
     int y;
     for (int i=0; i < 10; i++)
        {
          int z;
          z = 42;
        }
   }
```

Figure 9.1: Example source code used as input to program in codes used in this chapter.

9.2 Generating the code representing any IR node

The following code traverses each IR node and for a SgInitializedName of SgStatement outputs the scope information. The input code is shown in figure 9.1; the output of this code is shown in figure 9.3.

```
// This example shows the scope of each statement and name (variable names, base class names, etc.).

#include "rose.h"

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

void visitorTraversal::visit(SgNode* n)
   {
  // There are three types ir IR nodes that can be queried for scope:
  //   - SgStatement, and
  //   - SgInitializedName
     SgStatement* statement = isSgStatement(n);
     if (statement != NULL)
        {
          SgScopeStatement* scope = statement->get_scope();
          ROSE_ASSERT(scope != NULL);
          printf ("SgStatement       = %12p = %30s has scope = %12p = %s (total number = %d) \n",
               statement,statement->class_name().c_str(),
               scope,scope->class_name().c_str(),(int)scope->numberOfNodes());
        }

     SgInitializedName* initializedName = isSgInitializedName(n);
     if (initializedName != NULL)
        {
          SgScopeStatement* scope = initializedName->get_scope();
          ROSE_ASSERT(scope != NULL);
          printf ("SgInitializedName = %12p = %30s has scope = %12p = %s (total number = %d)\n",
               initializedName,initializedName->get_name().str(),
               scope,scope->class_name().c_str(),(int)scope->numberOfNodes());
        }
   }

int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project,preorder);

     printf ("Number of scopes (SgScopeStatement) = %d \n",(int)SgScopeStatement::numberOfNodes());
     printf ("Number of scopes (SgBasicBlock)     = %d \n",(int)SgBasicBlock::numberOfNodes());

#if 0
     printf ("\n\n");
     printf ("Now output all the symbols in each symbol table \n");
     SageInterface::outputLocalSymbolTables(project);
     printf ("\n\n");
#endif

     return 0;
   }
```

Figure 9.2: Example source code showing how to get scope information for each IR node.

```
SgStatement       = 0x2adc1d169010 =                      SgGlobal has scope = 0x2adc1d169010 = SgGlobal (tot
SgStatement       = 0x2adc1d41b010 =           SgVariableDeclaration has scope = 0x2adc1d169010 = SgGlobal (tot
SgInitializedName = 0x2adc1d3b6438 =
xyz has scope = 0x2adc1d169010 = SgGlobal (total number = 0)
SgStatement       = 0x2adc1d273908 =           SgFunctionDeclaration has scope = 0x2adc1d169010 = SgGlobal (tot
SgStatement       = 0x2adc1d330628 =         SgFunctionParameterList has scope = 0x2adc1d169010 = SgGlobal (tot
SgInitializedName = 0x2adc1d3b6560 =
x has scope = 0x2adc1d541010 = SgFunctionDefinition (total number = 0)
SgStatement       = 0x2adc1d541010 =            SgFunctionDefinition has scope = 0x2adc1d169010 = SgGlobal (tot
SgStatement       = 0x2adc1d58a010 =                    SgBasicBlock has scope = 0x2adc1d541010 = SgFunctionDe
SgStatement       = 0x2adc1d41b290 =           SgVariableDeclaration has scope = 0x2adc1d58a010 = SgBasicBlock
SgInitializedName = 0x2adc1d3b6688 =
y has scope = 0x2adc1d58a010 = SgBasicBlock (total number = 0)
SgStatement       = 0x2adc1d5cd010 =                  SgForStatement has scope = 0x2adc1d58a010 = SgBasicBlock
SgStatement       =    0x8dd1390 =            SgForInitStatement has scope = 0x2adc1d5cd010 = SgForStatement
SgStatement       = 0x2adc1d41b510 =           SgVariableDeclaration has scope = 0x2adc1d5cd010 = SgForStatemen
SgInitializedName = 0x2adc1d3b67b0 =
i has scope = 0x2adc1d5cd010 = SgForStatement (total number = 0)
SgStatement       =    0x8e52860 =              SgExprStatement has scope = 0x2adc1d5cd010 = SgForStatement
SgStatement       = 0x2adc1d58a120 =                    SgBasicBlock has scope = 0x2adc1d5cd010 = SgForStatemen
SgStatement       = 0x2adc1d41b790 =           SgVariableDeclaration has scope = 0x2adc1d58a120 = SgBasicBlock
SgInitializedName = 0x2adc1d3b68d8 =
z has scope = 0x2adc1d58a120 = SgBasicBlock (total number = 0)
SgStatement       =    0x8e528b8 =              SgExprStatement has scope = 0x2adc1d58a120 = SgBasicBlock (t
Number of scopes (SgScopeStatement) = 0
Number of scopes (SgBasicBlock)     = 2
```

Figure 9.3: Output of input code using scopeInformation.C

Chapter 10

AST Query

This chapter presents a mechanism for simple queries on the AST. Such queries are typically a single line of code, instead of the class that must be declared and defined when using the traversal mechanism. While the traversal mechanism is more sophisticated and more powerful, the AST Query mechanism is particularly simple to use.

10.1 Simple Queries on the AST

This section demonstrates a simple query on the AST.

The program in figure 10.1 calls an internal ROSE Query Library. Queries of the AST using the query library are particularly simple and often are useful as nested queries within more complex analysis. More information of the ROSE AST Query Library is available within ROSE User Manual.

Using the input program in figure 10.2 the translator processes the code and generates the output in figure 10.3.

FIXME: *Put an example of composition of AST queries into the example input code.*

10.2 Nested Query

This section demonstrates a nested AST query, showing how to use composition in the construction of more elaborate queries from simple ones.

The number of traversals of the AST can be reduced by using nested queries. Nested queries permits queries on the result from a NodeQuery. Another advantage is that nested (combined) queries can be formed to query for information without writing new query, the nested query is a new query.

The program in figure 10.4 calls an internal ROSE Query Library. Two different queries are performed to find all access functions within the AST. The first query is nested, the returned list from a query is used in a traversal, and the second query queries the AST for the same nodes.

Using the input program in figure 10.5 the translator processes the code and generates the output in figure 10.6.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

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

  // Build a list of functions within the AST
     Rose_STL_Container<SgNode*> functionDeclarationList = NodeQuery::querySubTree (project,V_SgFunctionDecla

     int counter = 0;
     for (Rose_STL_Container<SgNode*>::iterator i = functionDeclarationList.begin(); i != functionDeclarationL
        {
       // Build a pointer to the current type so that we can call the get_name() member function.
          SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(*i);
          ROSE_ASSERT(functionDeclaration != NULL);

       // DQ (3/5/2006): Only output the non-compiler generated IR nodes
          if ( (*i)->get_file_info()->isCompilerGenerated() == false)
             {
            // output the function number and the name of the function
               printf ("Function #%2d name is %s at line %d \n",
                    counter++,functionDeclaration->get_name().str(),
                    functionDeclaration->get_file_info()->get_line());
             }
            else
             {
            // Output something about the compiler-generated builtin functions
               printf ("Compiler-generated (builtin) function #%2d name is %s \n",
                    counter++,functionDeclaration->get_name().str());
             }
        }

  // Note: Show composition of AST queries

     return 0;
   }
```

Figure 10.1: Example source code for translator to read an input program and generate a list of functions in the AST (queryLibraryExample.C).

```

// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);
void foo(double)
   {
     int x = 1;
     int y;

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }

int main()
   {
     foo(42);
     foo(3.14159265);

     templateClass<char> instantiatedClass;
     instantiatedClass.foo(7);
     instantiatedClass.foo(7.0);

     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 10.2: Example source code used as input to program in figure 10.1 (queryLibraryExample.C).

```
Compiler-generated (builtin) function # 0 name is __builtin_copysign
Compiler-generated (builtin) function # 1 name is __builtin_copysignf
Compiler-generated (builtin) function # 2 name is __builtin_copysignl
Compiler-generated (builtin) function # 3 name is __builtin_acosf
Compiler-generated (builtin) function # 4 name is __builtin_acosl
Compiler-generated (builtin) function # 5 name is __builtin_asinf
Compiler-generated (builtin) function # 6 name is __builtin_asinl
Compiler-generated (builtin) function # 7 name is __builtin_atanf
Compiler-generated (builtin) function # 8 name is __builtin_atanl
Compiler-generated (builtin) function # 9 name is __builtin_atan2f
Compiler-generated (builtin) function #10 name is __builtin_atan2l
Compiler-generated (builtin) function #11 name is __builtin_ceilf
Compiler-generated (builtin) function #12 name is __builtin_ceill
Compiler-generated (builtin) function #13 name is __builtin_coshf
Compiler-generated (builtin) function #14 name is __builtin_coshl
Compiler-generated (builtin) function #15 name is __builtin_floorf
Compiler-generated (builtin) function #16 name is __builtin_floorl
Compiler-generated (builtin) function #17 name is __builtin_fmodf
Compiler-generated (builtin) function #18 name is __builtin_fmodl
Compiler-generated (builtin) function #19 name is __builtin_frexpf
Compiler-generated (builtin) function #20 name is __builtin_frexpl
Compiler-generated (builtin) function #21 name is __builtin_ldexpf
Compiler-generated (builtin) function #22 name is __builtin_ldexpl
Compiler-generated (builtin) function #23 name is __builtin_log10f
Compiler-generated (builtin) function #24 name is __builtin_log10l
Compiler-generated (builtin) function #25 name is __builtin_modff
Compiler-generated (builtin) function #26 name is __builtin_modfl
Compiler-generated (builtin) function #27 name is __builtin_powf
Compiler-generated (builtin) function #28 name is __builtin_powl
Compiler-generated (builtin) function #29 name is __builtin_sinhf
Compiler-generated (builtin) function #30 name is __builtin_sinhl
Compiler-generated (builtin) function #31 name is __builtin_tanf
Compiler-generated (builtin) function #32 name is __builtin_tanl
Compiler-generated (builtin) function #33 name is __builtin_tanhf
Compiler-generated (builtin) function #34 name is __builtin_tanhl
Compiler-generated (builtin) function #35 name is __builtin_powil
Compiler-generated (builtin) function #36 name is __builtin_powi
Compiler-generated (builtin) function #37 name is __builtin_powif
Compiler-generated (builtin) function #38 name is __builtin_strchr
Compiler-generated (builtin) function #39 name is __builtin_strrchr
Compiler-generated (builtin) function #40 name is __builtin_strpbrk
Compiler-generated (builtin) function #41 name is __builtin_strstr
Compiler-generated (builtin) function #42 name is __builtin_nansf
Compiler-generated (builtin) function #43 name is __builtin_nans
Compiler-generated (builtin) function #44 name is __builtin_nansl
Compiler-generated (builtin) function #45 name is __builtin_fabs
Compiler-generated (builtin) function #46 name is __builtin_fabsf
Compiler-generated (builtin) function #47 name is __builtin_fabsl
Compiler-generated (builtin) function #48 name is __builtin_cosf
Compiler-generated (builtin) function #49 name is __builtin_cosl
Compiler-generated (builtin) function #50 name is __builtin_sinf
Compiler-generated (builtin) function #51 name is __builtin_sinl
Compiler-generated (builtin) function #52 name is __builtin_sqrtf
Compiler-generated (builtin) function #53 name is __builtin_sqrtl
Compiler-generated (builtin) function #54 name is __builtin_fpclassify
Compiler-generated (builtin) function #55 name is __builtin_return_address
Compiler-generated (builtin) function #56 name is __builtin_frame_address
Compiler-generated (builtin) function #57 name is __builtin_expect
Compiler-generated (builtin) function #58 name is __builtin_prefetch
Compiler-generated (builtin) function #59 name is __builtin_huge_val
Compiler-generated (builtin) function #60 name is __builtin_huge_valf
Compiler-generated (builtin) function #61 name is __builtin_huge_vall
Compiler-generated (builtin) function #62 name is __builtin_inf
Compiler-generated (builtin) function #63 name is __builtin_inff
Compiler-generated (builtin) function #64 name is __builtin_infl
Compiler-generated (builtin) function #65 name is __builtin_nan
Compiler-generated (builtin) function #66 name is __builtin_nanf
Compiler-generated (builtin) function #67 name is __builtin_nanl
Compiler-generated (builtin) function #68 name is __builtin_nans
Compiler-generated (builtin) function #69 name is __builtin_nansf
Compiler-generated (builtin) function #70 name is __builtin_nansl
Compiler-generated (builtin) function #71 name is __builtin_clz
Compiler-generated (builtin) function #72 name is __builtin_ctz
Compiler-generated (builtin) function #73 name is __builtin_popcount
Compiler-generated (builtin) function #74 name is __builtin_parity
Compiler-generated (builtin) function #75 name is __builtin_ffsl
Compiler-generated (builtin) function #76 name is __builtin_clzl
Compiler-generated (builtin) function #77 name is __builtin_ctzl
Compiler-generated (builtin) function #78 name is __builtin_popcountl
Compiler-generated (builtin) function #79 name is __builtin_parityl
Compiler-generated (builtin) function #80 name is __builtin_ffsll
Compiler-generated (builtin) function #81 name is __builtin_clzll
Compiler-generated (builtin) function #82 name is __builtin_ctzll
```

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;


 // Function querySolverAccessFunctions()
 // find access functions (function name starts with "get_" or "set_")
NodeQuerySynthesizedAttributeType
querySolverAccessFunctions (SgNode * astNode)
   {
     ROSE_ASSERT (astNode != 0);
     NodeQuerySynthesizedAttributeType returnNodeList;

     SgFunctionDeclaration* funcDecl = isSgFunctionDeclaration(astNode);

     if (funcDecl != NULL)
        {
          string functionName = funcDecl->get_name().str();
          if ( (functionName.length() >= 4) && ((functionName.substr(0,4) == "get_") || (functionName.substr(0,4) == "set
               returnNodeList.push_back (astNode);
        }

     return returnNodeList;
   }

// Function printFunctionDeclarationList will print all function names in the list
void printFunctionDeclarationList(Rose_STL_Container<SgNode*> functionDeclarationList)
   {
     int counter = 0;
     for (Rose_STL_Container<SgNode*>::iterator i = functionDeclarationList.begin(); i != functionDeclarationList.end();
        {
       // Build a pointer to the current type so that we can call the get_name() member function.
          SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(*i);
          ROSE_ASSERT(functionDeclaration != NULL);

       // output the function number and the name of the function
          printf ("function name #%d is %s at line %d \n",
               counter++,functionDeclaration->get_name().str(),
               functionDeclaration->get_file_info()->get_line());
        }
   }

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

  // Build a list of functions within the AST and find all access functions
  // (function name starts with "get_" or "set_")

  // Build list using a query of the whole AST
     Rose_STL_Container<SgNode*> functionDeclarationList = NodeQuery::querySubTree (project,V_SgFunctionDeclaration);

  // Build list using nested Queries (operating on return result of previous query)
     Rose_STL_Container<SgNode*> accessFunctionsList;
     accessFunctionsList = NodeQuery::queryNodeList (functionDeclarationList,&querySolverAccessFunctions);
     printFunctionDeclarationList(accessFunctionsList);

  // Alternative form of same query building the list using a query of the whole AST
     accessFunctionsList = NodeQuery::querySubTree (project,&querySolverAccessFunctions);
     printFunctionDeclarationList(accessFunctionsList);

  // Another way to query for collections of IR nodes
     VariantVector vv1 = V_SgClassDefinition;
     std::cout << "Number of class definitions in the memory pool is: " << NodeQuery::queryMemoryPool(vv1).size() << std:

  // Another way to query for collections of multiple IR nodes.
  // VariantVector(V_SgType) is internally expanded to all IR nodes derived from SgType.
     VariantVector vv2 = VariantVector(V_SgClassDefinition) + VariantVector(V_SgType);
     std::cout << "Number of class definitions AND types in the memory pool is: " << NodeQuery::queryMemoryPool(vv2).size

  // Note: Show composition of AST queries

     return 0;
   }
```

Figure 10.4: Example source code for translator to read an input program and generate a list of access functions in the AST (nestedQueryExample.C).

```

// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);
void foo(double)
   {
     int x = 1;
     int y;

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }

int main()
   {
     foo(42);
     foo(3.14159265);

     templateClass<char> instantiatedClass;
     instantiatedClass.foo(7);
     instantiatedClass.foo(7.0);

     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 10.5: Example source code used as input to program in figure 10.4 (nestedQueryExample.C).

```
function name #0 is get_foo at line 0
function name #1 is set_foo at line 0
function name #2 is get_foo at line 28
function name #3 is set_foo at line 29
function name #0 is get_foo at line 0
function name #1 is set_foo at line 0
function name #2 is get_foo at line 28
function name #3 is set_foo at line 29
Number of class definitions in the memory pool is: 1
Number of class definitions AND types in the memory pool is: 436
```

Figure 10.6: Output of input file to the AST query processor (nestedQueryExample.C).

Chapter 11

AST File I/O

Figure 11.1 shows an example of how to use the AST File I/O mechanism. This chapter presents an example translator to write out an AST to a file and then read it back in.

11.1 Source Code for File I/O

Figure 11.1 shows an example translator which reads an input application, forms the AST, writes out the AST to a file, then deletes the AST and reads the AST from the previously written file.

The input code is shown in figure 11.2, the output of this code is shown in figure 11.3.

11.2 Input to Demonstrate File I/O

Figure 11.2 shows the example input used for demonstration of the AST file I/O. In this case we are reusing the example used in the inlining example.

11.3 Output from File I/O

Figure 11.3 shows the output from the example file I/O tutorial example.

11.4 Final Code After Passing Through File I/O

Figure 11.4 shows the same file as the input demonstrating that the file I/O didn't change the resulting generated code. *Much more sophisticated tests are applied internally to verify the correctness of the AST after AST file I/O.*

```
// Example demonstrating function inlining (maximal inlining, up to preset number of inlinings).

#include "rose.h"

using namespace std;

// This is a function in Qing's AST interface
void FixSgProject(SgProject& proj);

int main (int argc, char* argv[])
   {
  // Build the project object (AST) which we will fill up with multiple files and use as a
  // handle for all processing of the AST(s) associated with one or more source files.
     SgProject* project = new SgProject(argc,argv);

  // DQ (7/20/2004): Added internal consistancy tests on AST
     AstTests::runAllTests(project);

     bool modifiedAST = true;
     int  count   = 0;

  // Inline one call at a time until all have been inlined.  Loops on recursive code.
     do {
          modifiedAST = false;

       // Build a list of functions within the AST
          Rose_STL_Container<SgNode*> functionCallList = NodeQuery::querySubTree (project,V_SgFunctionCallExp)

       // Loop over all function calls
       // for (list<SgNode*>::iterator i = functionCallList.begin(); i != functionCallList.end(); i++)
          Rose_STL_Container<SgNode*>::iterator i = functionCallList.begin();
          while (modifiedAST == false && i != functionCallList.end())
             {
               SgFunctionCallExp* functionCall = isSgFunctionCallExp(*i);
               ROSE_ASSERT(functionCall != NULL);

            // Not all function calls can be inlined in C++, so report if successful.
               bool sucessfullyInlined = doInline(functionCall);

               if (sucessfullyInlined == true)
                  {
                 // As soon as the AST is modified recompute the list of function
                 // calls (and restart the iterations over the modified list)
                    modifiedAST = true;
                  }
                 else
                  {
                    modifiedAST = false;
                  }

            // Increment the list iterator
               i++;
             }

       // Quite when we have ceased to do any inline transformations
       // and only do a predefined number of inline transformations
          count++;
        }
     while(modifiedAST == true && count < 10);

  // Call function to postprocess the AST and fixup symbol tables
     FixSgProject(*project);

  // Rename each variable declaration
     renameVariables(project);

  // Fold up blocks
     flattenBlocks(project);

  // Clean up inliner-generated code
     cleanupInlinedCode(project);

  // Change members to public
     changeAllMembersToPublic(project);

  // DQ (3/11/2006): This fails so the inlining, or the AST Interface
  // support, needs more work even though it generated good code.
  // AstTests::runAllTests(project);

     return backend(project);
   }
```

```
// This test code is a combination of pass1 and pass7, selected somewhat randomly
// from Jeremiah's test code of his inlining transformation from summer 2004.

int x = 0;

// Function it increment "x"
void incrementX()
   {
     x++;
   }

int foo()
   {
     int a = 0;
     while (a < 5)
        {
          ++a;
        }

     return a + 3;
   }

int main(int, char**)
   {
  // Two trival function calls to inline
     incrementX();
     incrementX();

  // Somthing more interesting to inline
     for (; foo() < 7;)
        {
          x++;
        }

     return x;
   }
```

Figure 11.2: Example source code used as input to demonstrate the AST file I/O support.

Figure 11.3: Output of input code after inlining transformations.

```
// This test code is a combination of pass1 and pass7, selected somewhat randomly
// from Jeremiah's test code of his inlining transformation from summer 2004.
int x = 0;
// Function it increment "x"

void incrementX()
{
  x++;
}

int foo()
{
  int a__0 = 0;
  while(a__0 < 5){
    ++a__0;
  }
  return a__0 + 3;
}

int main(int ,char **)
{
  x++;
  x++;
// Somthing more interesting to inline
  for (; true; ) {
    int a__1 = 0;
    while(a__1 < 5){
      ++a__1;
    }
    int rose_temp__7__0 = a__1 + 3;
    bool rose__temp__2 = (bool )(rose_temp__7__0 < 7);
    if (!rose__temp__2) {
      break;
    }
    else {
    }
    x++;
  }
  return x;
}
```

Figure 11.4: Output of input code after file I/O.

Chapter 12

Debugging Techniques

There are numerous methods ROSE provides to help debug the development of specialized source-to-source translators. This section shows some of the techniques for getting information from IR nodes and displaying it. More information about generation of specialized AST graphs to support debugging can be found in chapter 5 and custom graph generation in section 28.

12.1 Input For Examples Showing Debugging Techniques

Figure 12.1 shows the input code used for the example translators that report useful debugging information in this chapter.

```
// Example program showing matrix multiply
// (for use with loop optimization tutorial example)

#define N 50

int main()
   {
     int i,j, k;
     double a[N][N], b[N][N], c[N][N];

     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];
                  }
             }
        }

     return 0;
   }
```

Figure 12.1: Example source code used as input to program in codes showing debugging techniques shown in this section.

12.2 Generating the code from any IR node

Any IR node may be converted to the string that represents its subtree within the AST. If it is a type, then the string will be the value of the type; if it is a statement, the value will be the source code associated with that statement, including any sub-statements. To support the generation for strings from IR nodes we use the `unparseToString()` member function. This function strips comments and preprocessor control structure. The resulting string is useful for both debugging and when forming larger strings associated with the specification of transformations using the string-based rewrite mechanism. Using ROSE, IR nodes may be converted to strings, and strings converted to AST fragments of IR nodes.

Note that unparsing associated with generating source code for the backend vendor compiler is more than just calling the unparseToString member function, since it introduces comments, preprocessor control structure and formating.

Figure 12.2 shows a translator which generates a string for a number of predefined IR nodes. Figure 12.1 shows the sample input code and figure 12.5 shows the output from the translator when using the example input application.

12.3 Displaying the source code position of any IR node

This example shows how to obtain information about the position of any IR node relative to where it appeared in the original source code. New IR nodes (or subtrees) that are added to the AST as part of a transformation will be marked as part of a transformation and have no position in the source code. Shared IR nodes (as generated by the AST merge mechanism are marked as shared explicitly (other IR nodes that are shared by definition don't have a SgFileInfo object and are thus not marked explicitly as shared.

The example translator to output the source code position is shown in figure 12.4. Using the input code in figure 12.1 the output code is shown in figure 12.5.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

using namespace std;

int
main ( int argc, char* argv[] )
   {
     ios::sync_with_stdio();     // Syncs C++ and C I/O subsystems!

     if (SgProject::get_verbose() > 0)
          printf ("In preprocessor.C: main() \n");

     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // AST diagnostic tests
     AstTests::runAllTests(const_cast<SgProject*>(project));

  // test statistics
     if (project->get_verbose() > 1)
        {
          cout << AstNodeStatistics::traversalStatistics(project);
          cout << AstNodeStatistics::IRnodeUsageStatistics();
        }

     if (project->get_verbose() > 0)
          printf ("Generate the pdf output of the SAGE III AST \n");
     generatePDF ( *project );

     if (project->get_verbose() > 0)
          printf ("Generate the DOT output of the SAGE III AST \n");
     generateDOT ( *project );

     Rose_STL_Container<SgNode*> nodeList;
  // nodeList = NodeQuery::querySubTree (project,V_SgType,NodeQuery::ExtractTypes);
     nodeList = NodeQuery::querySubTree (project,V_SgForStatement);
     printf ("\nnodeList.size() = %zu \n",nodeList.size());

     Rose_STL_Container<SgNode*>::iterator i = nodeList.begin();
     while (i != nodeList.end())
        {
          printf ("Query node = %p = %s = %s \n",*i,(*i)->sage_class_name(),(*i)->unparseToString().c_str());
          i++;
        }

     return 0;
   }
```

Figure 12.2: Example source code showing the output of the string from an IR node. The string represents the code associated with the subtree of the target IR node.

```

nodeList.size() = 3
Query node = 0x2b82c2bc6010 = SgForStatement = for(i = 0;i <= 50 - 1;i += 1) {for(j = 0;j <= 50 - 1;j += 1) {for(k = 0;k <
Query node = 0x2b82c2bc6130 = SgForStatement = for(j = 0;j <= 50 - 1;j += 1) {for(k = 0;k <= 50 - 1;k += 1) {c[i][j] =(c[
Query node = 0x2b82c2bc6250 = SgForStatement = for(k = 0;k <= 50 - 1;k += 1) {c[i][j] =(c[i][j] +(a[i][k] * b[k][j]));}
```

Figure 12.3: Output of input code using debuggingIRnodeToString.C

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

using namespace std;

int
main ( int argc, char* argv[] )
   {
     if (SgProject::get_verbose() > 0)
          printf ("In preprocessor.C: main() \n");

     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

     Rose_STL_Container<SgNode*> nodeList;
     nodeList = NodeQuery::querySubTree (project,V_SgForStatement);
     printf ("\nnodeList.size() = %zu \n",nodeList.size());

     Rose_STL_Container<SgNode*>::iterator i = nodeList.begin();
     while (i != nodeList.end())
        {
          Sg_File_Info & fileInfo = *((*i)->get_file_info());
          printf ("Query node = %p = %s in %s \n ----- at line %d on column %d \n",
               *i,(*i)->sage_class_name(),fileInfo.get_filename(),
                fileInfo.get_line(), fileInfo.get_col());
          i++;
        }

     if (project->get_verbose() > 0)
          printf ("Calling the backend() \n");

     return 0;
   }
```

Figure 12.4: Example source code showing the output of the string from an IR node. The string represents the code associated with the subtree of the target IR node.

```

nodeList.size() = 3
Query node = 0x2ae33bd05010 = SgForStatement in /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-releas
 ----- at line 11 on column 6
Query node = 0x2ae33bd05130 = SgForStatement in /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-releas
 ----- at line 13 on column 11
Query node = 0x2ae33bd05250 = SgForStatement in /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-releas
 ----- at line 15 on column 16
```

hared IR nodes (as generated by the AST merge mechanism are mar

Figure 12.5: Output of input code using debuggingSourceCodePositionInformation.C

Part II

Complex Types

This part elaborates some details for handling complex types in ROSE.

Chapter 13

Type and Declaration Modifiers

Most languages support the general concept of modifiers to types, declarations, etc. The keyword *volatile* for example is a modifier to the type where it is used in a declaration. Searching for the modifiers for types and declarations,however, can be confusing. They are often not where one would expect, and most often because of corner cases in the language that force them to be handled in specific ways.

This example tutorial code is a demonstration of a how to access the *volatile* modifier used in the declaration of types for variables. We demonstrate that the modifier is not present in the SgVariableDeclaration or the SgVariableDefinitoon, but is located in the SgModifierType used to wrap the type returned from the SgInitializedName (the variable in the variable declaration).

13.1 Input For Example Showing use of *Volatile* type modifier

Figure 13.1 shows the example input used for demonstration of test for the *volatile* type modifier.

```
// Input example of use of "volatile" type modifier

volatile int a,*b;

void foo()
   {
     for (volatile int y = 0; y < 10; y++)
        {
        }
   }
```

Figure 13.1: Example source code used as input to program in codes used in this chapter.

13.2 Generating the code representing the seeded bug

Figure 13.2 shows a code that traverses each IR node and for and SgInitializedName IR node checks its type. The input code is shown in figure 13.1, the output of this code is shown in figure 13.3.

```
#include "rose.h"

using namespace std;

class visitorTraversal : public AstSimpleProcessing
   {
     public:
          void visit(SgNode* n);
   };

void visitorTraversal::visit(SgNode* n)
   {
  // The "volatile" madifier is in the type of the SgInitializedName
     SgInitializedName* initializedName = isSgInitializedName(n);
     if (initializedName != NULL)
        {
          printf ("Found a SgInitializedName = %s \n",initializedName->get_name().str());
          SgType* type = initializedName->get_type();

          printf ("   initializedName: type = %p = %s \n",type,type->class_name().c_str());
          SgModifierType* modifierType = isSgModifierType(type);
          if (modifierType != NULL)
             {
               bool isVolatile = modifierType->get_typeModifier().get_constVolatileModifier().isVolatile();
               printf ("   initializedName: SgModifierType: isVolatile = %s \n",(isVolatile == true) ? "true" : "false");
             }

          SgModifierNodes* modifierNodes = type->get_modifiers();
          printf ("   initializedName: modifierNodes = %p \n",modifierNodes);
          if (modifierNodes != NULL)
             {
               SgModifierTypePtrVector modifierList = modifierNodes->get_nodes();
               for (SgModifierTypePtrVector::iterator i = modifierList.begin(); i != modifierList.end(); i++)
                  {
                    printf ("initializedName: modifiers: i = %s \n",(*i)->class_name().c_str());
                  }
             }
        }

  // Note that the "volatile" madifier is not in the SgVariableDeclaration nor the SgVariableDefinition
     SgVariableDeclaration* variableDeclaration = isSgVariableDeclaration(n);
     if (variableDeclaration != NULL)
        {
          bool isVolatile = variableDeclaration->get_declarationModifier().get_typeModifier().get_constVolatileModifier().isVola
          printf ("SgVariableDeclaration: isVolatile = %s \n",(isVolatile == true) ? "true" : "false");
          SgVariableDefinition* variableDefinition = variableDeclaration->get_definition();
       // printf ("variableDefinition = %p \n",variableDefinition);
          if (variableDefinition != NULL)
             {
               bool isVolatile = variableDefinition->get_declarationModifier().get_typeModifier().get_constVolatileModifier().isV
               printf ("SgVariableDefinition: isVolatile = %s \n",(isVolatile == true) ? "true" : "false");
             }
        }
   }

// must have argc and argv here!!
int main(int argc, char * argv[])
   {
     SgProject *project = frontend (argc, argv);

     visitorTraversal myvisitor;
     myvisitor.traverseInputFiles(project,preorder);

     return backend(project);
   }
```

Figure 13.2: Example source code showing how to detect *volatile* modifier.

```
// Input example of use of "volatile" type modifier
volatile int a;
volatile int *b;

void foo()
{
  for (volatile int y = 0; y < 10; y++) {
  }
}
```

Figure 13.3: Output of input code using volatileTypeModifier.C

Chapter 14

Function Parameter Types

The analysis of functions often requires the query of the function types. This tutorial example shows how to obtain the function parameter types for any function. Note that functions also have a type which is based on their signature, a combination of their return type and functions parameter types. Any functions sharing the same return type and function parameter types have the same function type (the function type, a SgFunctionType IR node, will be shared between such functions).

Figure 14.1 shows a translator which reads an application (shown in figure 14.2) and outputs information about the function parameter types for each function, shown in figure 14.3. This information includes the order of the function declaration in the global scope, and name of the function, and the types of each parameter declared in the function declaration.

Note that there are a number of builtin functions defined as part of the GNU g++ and gcc compatibility and these are output as well. These are marked as compiler generated functions within ROSE. The code shows how to differentiate between the two different types. Notice also that instantiated template functions are classified as *compiler generated*.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

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

  // Build a list of functions within the AST
     Rose_STL_Container<SgNode*> functionDeclarationList = NodeQuery::querySubTree (project,V_SgFunctionDeclar

     int functionCounter = 0;
     for (Rose_STL_Container<SgNode*>::iterator i = functionDeclarationList.begin(); i != functionDeclarationL
        {
       // Build a pointer to the current type so that we can call the get_name() member function.
          SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(*i);
          ROSE_ASSERT(functionDeclaration != NULL);

       // DQ (3/5/2006): Only output the non-compiler generated IR nodes
          if ( (*i)->get_file_info()->isCompilerGenerated() == false)
             {
               SgFunctionParameterList* functionParameters = functionDeclaration->get_parameterList();
               ROSE_ASSERT(functionDeclaration != NULL);

            // output the function number and the name of the function
               printf ("Non-compiler generated function name #%3d is %s \n",functionCounter++,functionDeclarat

               SgInitializedNamePtrList & parameterList = functionParameters->get_args();
               int parameterCounter = 0;
               for (SgInitializedNamePtrList::iterator j = parameterList.begin(); j != parameterList.end(); j+
                  {
                    SgType* parameterType = (*j)->get_type();
                    printf ("    parameterType #%2d = %s \n",parameterCounter++,parameterType->unparseToString(
                  }
             }
            else
             {
               printf ("Compiler generated function name #%3d is %s \n",functionCounter++,functionDeclaration-
             }
        }

     return 0;
   }
```

Figure 14.1: Example source code showing how to get type information from function parameters.

```

// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);
void foo(double)
   {
     int x = 1;
     int y;

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }

int main ( int argc, char* argv[] )
   {
     foo(42);
     foo(3.14159265);

     templateClass<char> instantiatedClass;
     instantiatedClass.foo(7);
     instantiatedClass.foo(7.0);

     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 14.2: Example source code used as input to typeInfoFromFunctionParameters.C.

```
Compiler generated function name #  0 is __builtin_copysign
Compiler generated function name #  1 is __builtin_copysignf
Compiler generated function name #  2 is __builtin_copysignl
Compiler generated function name #  3 is __builtin_acosf
Compiler generated function name #  4 is __builtin_acosl
Compiler generated function name #  5 is __builtin_asinf
Compiler generated function name #  6 is __builtin_asinl
Compiler generated function name #  7 is __builtin_atanf
Compiler generated function name #  8 is __builtin_atanl
Compiler generated function name #  9 is __builtin_atan2f
Compiler generated function name # 10 is __builtin_atan2l
Compiler generated function name # 11 is __builtin_ceilf
Compiler generated function name # 12 is __builtin_ceill
Compiler generated function name # 13 is __builtin_coshf
Compiler generated function name # 14 is __builtin_coshl
Compiler generated function name # 15 is __builtin_floorf
Compiler generated function name # 16 is __builtin_floorl
Compiler generated function name # 17 is __builtin_fmodf
Compiler generated function name # 18 is __builtin_fmodl
Compiler generated function name # 19 is __builtin_frexpf
Compiler generated function name # 20 is __builtin_frexpl
Compiler generated function name # 21 is __builtin_ldexpf
Compiler generated function name # 22 is __builtin_ldexpl
Compiler generated function name # 23 is __builtin_log10f
Compiler generated function name # 24 is __builtin_log10l
Compiler generated function name # 25 is __builtin_modff
Compiler generated function name # 26 is __builtin_modfl
Compiler generated function name # 27 is __builtin_powf
Compiler generated function name # 28 is __builtin_powl
Compiler generated function name # 29 is __builtin_sinhf
Compiler generated function name # 30 is __builtin_sinhl
Compiler generated function name # 31 is __builtin_tanf
Compiler generated function name # 32 is __builtin_tanl
Compiler generated function name # 33 is __builtin_tanhf
Compiler generated function name # 34 is __builtin_tanhl
Compiler generated function name # 35 is __builtin_powil
Compiler generated function name # 36 is __builtin_powi
Compiler generated function name # 37 is __builtin_powif
Compiler generated function name # 38 is __builtin_strchr
Compiler generated function name # 39 is __builtin_strrchr
Compiler generated function name # 40 is __builtin_strpbrk
Compiler generated function name # 41 is __builtin_strstr
Compiler generated function name # 42 is __builtin_nansf
Compiler generated function name # 43 is __builtin_nans
Compiler generated function name # 44 is __builtin_nansl
Compiler generated function name # 45 is __builtin_fabs
Compiler generated function name # 46 is __builtin_fabsf
Compiler generated function name # 47 is __builtin_fabsl
Compiler generated function name # 48 is __builtin_cosf
Compiler generated function name # 49 is __builtin_cosl
Compiler generated function name # 50 is __builtin_sinf
Compiler generated function name # 51 is __builtin_sinl
Compiler generated function name # 52 is __builtin_sqrtf
Compiler generated function name # 53 is __builtin_sqrtl
Compiler generated function name # 54 is __builtin_fpclassify
Compiler generated function name # 55 is __builtin_return_address
Compiler generated function name # 56 is __builtin_frame_address
Compiler generated function name # 57 is __builtin_expect
Compiler generated function name # 58 is __builtin_prefetch
Compiler generated function name # 59 is __builtin_huge_val
Compiler generated function name # 60 is __builtin_huge_valf
Compiler generated function name # 61 is __builtin_huge_vall
Compiler generated function name # 62 is __builtin_inf
Compiler generated function name # 63 is __builtin_inff
Compiler generated function name # 64 is __builtin_infl
Compiler generated function name # 65 is __builtin_nan
Compiler generated function name # 66 is __builtin_nanf
Compiler generated function name # 67 is __builtin_nanl
Compiler generated function name # 68 is __builtin_nans
Compiler generated function name # 69 is __builtin_nansf
Compiler generated function name # 70 is __builtin_nansl
Compiler generated function name # 71 is __builtin_clz
Compiler generated function name # 72 is __builtin_ctz
Compiler generated function name # 73 is __builtin_popcount
Compiler generated function name # 74 is __builtin_parity
Compiler generated function name # 75 is __builtin_ffsl
Compiler generated function name # 76 is __builtin_clzl
Compiler generated function name # 77 is __builtin_ctzl
Compiler generated function name # 78 is __builtin_popcountl
Compiler generated function name # 79 is __builtin_parityl
Compiler generated function name # 80 is __builtin_ffsll
Compiler generated function name # 81 is __builtin_clzll
Compiler generated function name # 82 is __builtin_ctzll
```

Chapter 15

Resolving Overloaded Functions

Figure 15.1 shows a translator which reads an application and reposts on the mapping between function calls and function declarations. This is trivial since all overloaded function resolution is done within the frontend and so need not be computed (this is because all type resolution is done in the frontend and stored in the AST explicitly). Other compiler infrastructures often require this to be figured out from the AST, when type resolution is unavailable, and while not too hard for C, this is particularly complex for C++ (due to overloading and type promotion within function arguments).

Figure 15.2 shows the input code used to get the translator. Figure 15.3 shows the resulting output.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

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

  // Build a list of functions within the AST
     Rose_STL_Container<SgNode*> functionCallList = NodeQuery::querySubTree (project,V_SgFunctionCallExp);

     int functionCounter = 0;
     for (Rose_STL_Container<SgNode*>::iterator i = functionCallList.begin(); i != functionCallList.end(); i++
        {
          SgExpression* functionExpression = isSgFunctionCallExp(*i)->get_function();
          ROSE_ASSERT(functionExpression != NULL);

          SgFunctionRefExp* functionRefExp = isSgFunctionRefExp(functionExpression);

          SgFunctionSymbol* functionSymbol = NULL;
          if (functionRefExp != NULL)
             {
            // Case of non-member function
               functionSymbol = functionRefExp->get_symbol();
             }
            else
             {
            // Case of member function (hidden in rhs of binary dot operator expression)
               SgDotExp* dotExp = isSgDotExp(functionExpression);
               ROSE_ASSERT(dotExp != NULL);

               functionExpression = dotExp->get_rhs_operand();
               SgMemberFunctionRefExp* memberFunctionRefExp = isSgMemberFunctionRefExp(functionExpression);
               ROSE_ASSERT(memberFunctionRefExp != NULL);

               functionSymbol = memberFunctionRefExp->get_symbol();
             }

          ROSE_ASSERT(functionSymbol != NULL);

          SgFunctionDeclaration* functionDeclaration = functionSymbol->get_declaration();
          ROSE_ASSERT(functionDeclaration != NULL);

       // Output mapping of function calls to function declarations
          printf ("Location of function call #%d at line %d resolved by overloaded function declared at line %
               functionCounter++,
               isSgFunctionCallExp(*i)->get_file_info()->get_line(),
               functionDeclaration->get_file_info()->get_line());
        }

     return 0;
   }
```

Figure 15.1: Example source code showing mapping of function calls to overloaded function declarations.

```
// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };

// Overloaded functions for testing overloaded function resolution
void foo(int);
void foo(double)
   {
     int x = 1;
     int y;

  // Added to allow non-trivial CFG
     if (x)
        y = 2;
     else
        y = 3;
   }

int main()
   {
     foo(42);
     foo(3.14159265);

     templateClass<char> instantiatedClass;
     instantiatedClass.foo(7);
     instantiatedClass.foo(7.0);

     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 15.2: Example source code used as input to resolveOverloadedFunction.C.

```
Location of function call #0 at line 29 resolved by overloaded function declared at line 14
Location of function call #1 at line 30 resolved by overloaded function declared at line 15
Location of function call #2 at line 33 resolved by overloaded function declared at line 0
Location of function call #3 at line 34 resolved by overloaded function declared at line 0
```

Figure 15.3: Output of input to resolveOverloadedFunction.C.

Chapter 16

Template Parameter Extraction

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

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

  // Build a list of functions within the AST
     Rose_STL_Container<SgNode*> templateInstantiationDeclList =
          NodeQuery::querySubTree (project,V_SgTemplateInstantiationDecl);

     int classTemplateCounter = 0;
     for (Rose_STL_Container<SgNode*>::iterator i = templateInstantiationDeclList.begin();
          i != templateInstantiationDeclList.end(); i++)
        {
          SgTemplateInstantiationDecl* instantiatedTemplateClass = isSgTemplateInstantiationDecl(*i);
          ROSE_ASSERT(instantiatedTemplateClass != NULL);

       // output the function number and the name of the function
          printf ("Class name #%d is %s \n",
               classTemplateCounter++,
               instantiatedTemplateClass->get_templateName().str());

          const SgTemplateArgumentPtrList& templateParameterList = instantiatedTemplateClass->get_templateArguments();
          int parameterCounter = 0;
          for (SgTemplateArgumentPtrList::const_iterator j = templateParameterList.begin();
               j != templateParameterList.end(); j++)
             {
               printf ("     TemplateArgument #%d = %s \n",parameterCounter++,(*j)->unparseToString().c_str());
             }
        }

     return 0;
   }
```

Figure 16.1: Example source code used to extract template parameter information.

Figure 16.1 shows a translator which reads an application and gathers a list of loop nests. At the end of the traversal it reports information about each instantiated template, including the template arguments.

Figure 16.2 shows the input code used to get the translator. Figure 16.3 shows the resulting output.

```
// Templated class declaration used in template parameter example code
template <typename T>
class templateClass
   {
     public:
          int x;

          void foo(int);
          void foo(double);
   };


int main()
   {
     templateClass<char> instantiatedClass;
     instantiatedClass.foo(7);
     instantiatedClass.foo(7.0);

     templateClass<int> instantiatedClassInt;
     templateClass<float> instantiatedClassFloat;
     templateClass<templateClass<char> > instantiatedClassNestedChar;

     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 16.2: Example source code used as input to templateParameter.C.

```
Class name #0 is templateClass
   TemplateArgument #0 = char
Class name #1 is templateClass
   TemplateArgument #0 = int
Class name #2 is templateClass
   TemplateArgument #0 = float
Class name #3 is templateClass
   TemplateArgument #0 = templateClass< char >
```

Figure 16.3: Output of input to templateParameter.C.

Chapter 17

Template Support

This chapter is specific to demonstrating the C++ template support in ROSE. *This section is not an introduction to the general subject of C++ templates.* ROSE provides special handling for C++ templates because template instantiation must be controlled by the compiler.

Templates that require instantiation are instantiated by ROSE and can be seen in the traversal of the AST (and transformed). Any templates that can be instantiated by the backend compiler **and** *are not transformed* are not output within the code generation phase.

FIXME: *Provide a list of when templates are generated internally in the AST and when template instantiations are output.*

17.1 Example Template Code #1

This section presents figure 17.4, a simple C++ source code using a template. It is used as a basis for showing how template instantiations are handled within ROSE.

```
template <typename T>
class X
   {
     public:
          void foo();
   };

X<int> x;

void X<int>::foo()
   {
   }
```

Figure 17.1: Example source code showing use of a C++ template.

17.2 Example Template Code #2

This section presents figure 17.4, a simple C++ source code using a template function. It is used as a basis for showing how template instantiations are handled within ROSE.

```
template < typename T >
class X
   {
     public :
         void foo ( );
   };
class X< int  > x;
```

Figure 17.2: Example source code after processing using identityTranslator (shown in figure 2.1).

```
// template function
template <typename T>
void foo( T t )
   {
   }

// Specialization from user
template<> void foo<int>(int x) {}

int main()
   {
     foo(1);
   }
```

Figure 17.3: Example source code showing use of a C++ template.

```
// template function
template < typename T >
void foo ( T t )
   {
   }
// Specialization from user

template<> void foo < int > (int x)
{
}

int main()
{
  foo< int  > (1);
  return 0;
}
```

Figure 17.4: Example source code after processing using identityTranslator (shown in figure 2.1).

Part III

Program Analyses

This part exemplifies the use of existing ROSE analyses and how to build customized analyses using ROSE.

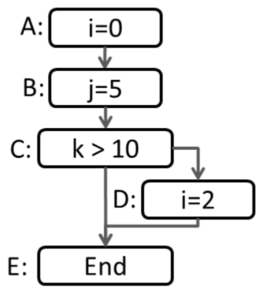
Chapter 18

Generic Dataflow Analysis Framework

This chapter summarizes the basic considerations behind the ROSE dataflow framework as well as how its API can be used to implement inter- and intra-procedural dataflow analyses. It is oriented towards potential users of the framework.

18.1 Basics of DataFlowAnalysis

Dataflow analysis is a technique for determining an applications possible states at various points in a program. It works as a fixed-point iteration over a space of possible facts about each node in the applications Control Flow Graph (CFG). The algorithm starts with no information about each node and iterates by accumulating all the constraints on the application's state at each node until it reaches a fixed point where no additional constraints can be discovered. The designer of a given dataflow analysis must specify an abstract representation of the set of all possible application states that maintains the relevant details of the state (e.g. whether a variable has a constant value or the linear relationships between variable pairs), while igoring the rest. For example, a state abstraction for the constant propagation analysis may have three different values: the special symbol $\perp$ if the variable is uninitialized, a numeric value if this is the only value the variable may hold at a given CFG node or $\top$ if it may have more than one value. More sophisticated abstractions represent application state using polyhedral constraints or predicate logic. Further, the designer must specify a "transfer" function that maps the abstract state before any application operation to the state after it. For example, if before statement $i++$ it is known that $i == n$ then after the statement it is known that $i - 1 == n$. To deal with control flow the designer also specifies a meet function that conjoins the abstract states along multiple control paths. For example, if at the end of the if branch of a conditional it is known that i=5 and at the end of the else branch i<10, the strongest fact that is true immediately after both branches of the conditional is $5 \leq$ i <10.
The set of possible abstract states must form a lattice, which is a partial order where for any pair of elements there exists a unique least upper bound. Intuitively, states that are lower in the partial order represent fewer constraints on the application state and higher states represent



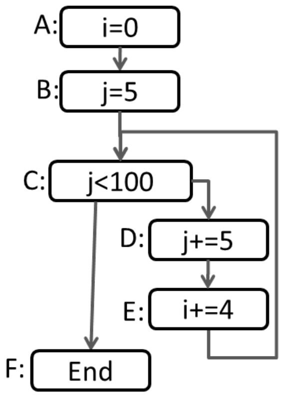
Node	State Below Node
A:	i=0
B:	i=0, j=5
C:	i=0, j=5
D:	i=2, j=5
E:	i=$\top$, j=5

Figure 18.1: Example of a constant propagation analysis.

more constraints. The special state $\perp$ corresponds to the least state in the partial order, where the application has done nothing to constrain its state (e.g. all variables are uninitialized). The meet function must guarantee the uniqueness of the least upper bound and the transfer function must be monotonic (if A$\leq$B then transfer(A)$\leq$transfer(B)). The dataflow fixed-point iteration ensures that the abstract state of every CFG node rises monotonically as it incorporates information about more possible application behaviors. When the analysis reaches a fixed point, the abstract state at each CFG node corresponds to the tightest set of constraints that are can be specified by the abstraction about the application state at that location.

For an intuition about how dataflow analyses work, 18.1 presents an example of a constant propagation analysis. The CFG is on the left and the table on the right shows the fixed-point solution of the abstract state immediately after each node. At each node the abstract application state records for each variable one of the following values: (i) $\perp$, which indicates that the variable is uninitialized, (ii) a specific constant value if the variable may only have this value at node n or (iii) $\top$ which indicates that the variable may have more than one value (i.e., is not representable as a single constant).It shows that immediately after node A it is known that i=0 and similarly after node B, i=0 and j=5. The same is true after node C since it has no side-effects and after the assignment in node D, the state changes to i=2, j=5. When the two conditional branches meet, the abstract state is the union of the states on both branches: the strongest assertions that are true of both states. Since j has the same value and i has two different values, the abstract state after node E is i=$\top$, j=5.

?? presents an example with a more complex abstraction: conjunction of linear relationships between variables. At node B the dataflow analysis computes that i=0 and j=5. When this state is propagated through the loop, the analysis discovers that after the first iteration i=4 and j=5. It then computes the meet of i=0 $\wedge$ j=5 and i=4 $\wedge$ j=5, the facts along both paths. Since this abstraction represents linear relationships, the union finds the tightest linear relationships that are true of both input states. It thus infers that i=0 (mod 4), i=0 (mod 5) (i is divisible by 4 and j by 5) and that 5i=4j-5. When this state is propagated again through the body of the loop, these assertions are discovered to be the invariants of this loop and become the fixed-point solution after node C. If they were not invariants, the algorithm would iterate until invariants were found or it reached the abstract state $\top$ which means that no linear constraints are known. Further, since the conditional j<100 is also linear, j<100 is recorded in the states of the nodes inside the loop and j $\geq$ 100 is recorded at node F after the loop.



Node	State Below Node
A:	i=0
B:	i=0 ∧ j=5
C:	i=0(%4) ∧ j=0(%5) ∧ 5i=4j-5
D:	i=0(%4) ∧ j=0(%5) ∧ 5i=4j-10 ∧ j<100
E:	i=0(%4) ∧ j=0(%5) ∧ 5i=4j-5 ∧ j<100
F:	i=0(%4) ∧ j=0(%5) ∧ 5i=4j-5 ∧ j≥100

Figure 18.2: Example of a dataflow analysis with abstraction of affine constraints.

18.2 ROSE Dataflow Framework

ROSE provides a framework for implementing dataflow analyses. It allows users to specify their dataflow analysis by implement the standard dataflow components: (i) an abstraction of the applications state, (ii) a transfer function that specifies the effects of code on the application state and (iii) a meet operator that combines multiple possible abstract states into one. These are implemented by extending base classes provided by the framework and implementing key virtual methods that correspond to the above functionality. The framework then solves the dataflow equations using the user-provided classes and saves the results at each CFG node. This section describes the functionality provided by the framework and how it can be used to implement analyses. 18.1 summarizes the functionality provided by the framework.

18.2.1 Call and Control-Flow Graphs

The ROSE dataflow analysis framework operates on top of the ROSE Call Graph (CG) and Virtual Control-Flow Graph (VCFG). The CG documents the caller/callee relationships between application functions. The VCFG connects SgNodes in the applications AST to identify the possible execution orders between them. The VCFG is dynamic in that instead of being computing once for the entire application, it computes the outgoing and incoming edges of a given SgNode fresh every time this information is needed. This makes the VCFG very flexible because it automatically responds to changes in the AST with no need for complex adjustments to the graph.

18.2.2 Analyses

ROSE supports both inter-and intra-procedural analyses. Users implement basic, non-dataflow analyses by extending the IntraProceduralAnalysis and InterProceduralAnalysis classes. Intra analyses iterate over the CFG of each function, and inter analyses apply intra analyses to individual

Class	Purpose	Interface	User Responsibilities
Analysis	Implement Simple CFG passes	Classes **IntraProceduralAnalysis InterProceduralAnalysis**	Extend the classes and implement their **runAnalysis** and **transfer** methods
Intra Procedural Dataflow	Implement intraprocedural dataflow iteration	Classes **IntraFWDataflow IntraBWDataflow**	Extend classes and implement **genInitState** and **transfer** methods
Inter Procedural Dataflow	Implement the inter-procedural dataflow	Classes - **Context Insensitive InterProcedural Dataflow**	Execute on a given instance of **IntraDataflow**
Lattice	Generic interface for abstractions of the application's state	Methods **initialize copy meetUpdate operator== str**	Extend the Lattice class and implement interface methods
Nodestate	Stores dataflow information of each CFG node	Methods **setLatticeAbove getLatticeAbove deleteLatticeAbove setFact getFact deleteFacts**	Call methods to access dataflow information
AstInterface	Transforms the CFG	Methods **insertBeforeUsing CommaOp, insertAfterUsing CommaOp, replaceWithPattern**	Call methods

Table 18.1: The Functionality of the Dataflow Interface

functions. To implement an analysis an application developer must derive a class from the IntraProceduralAnalysis and/or InterProceduralAnalysis classes and implement the runAnalysis method. Classes UnstructuredPassInterAnalysis and UnstructuredPassIntraAnalysis Figure 18.2.3 provide examples of simple analyses. UnstructuredPassInterAnalysis takes as an argument a reference to an InterProceduralAnalysis and iterates once through all functions. It applies the runAnalysis method of the intra analysis to each function. UnstructuredPassIntraAnalysis iterates once through all the CFG nodes in the given function, applying its visit method to each node.

These analyses can be used to implement simple passes through the applications CFG and serve as the foundation of the dataflow analysis framework. For example, src/simpleAnalyses/saveDotAnalysis.C and src/simpleAnalyses/printAnalysisStates.C are examples of simple one-pass analyses. saveDotAnalysis prints the applications CFG as a DOT file and printAnalysisStates prints the dataflow

```
class UnstructuredPassInterAnalysis : virtual public InterProceduralAnalysis {
  UnstructuredPassInterAnalysis(IntraProceduralAnalysis& intraAnalysis)
  void runAnalysis();
};

class UnstructuredPassIntraAnalysis : virtual public IntraProceduralAnalysis {
   bool runAnalysis(const Function& func, NodeState* state);
   virtual void visit(const Function& func, const DataflowNode& n,
       NodeState& state)=0;
};
```

Figure 18.3: Example of simple analyses

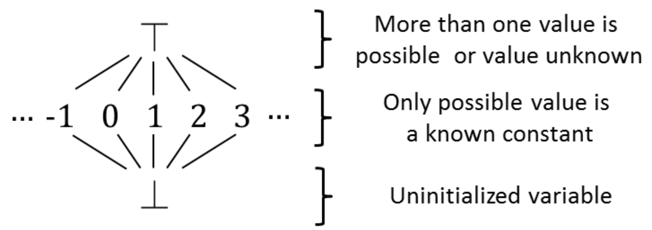


Figure 18.4: Each variable's lattice for constant-propagation analysis

states of all CFG nodes in the application, which is useful for debugging.

18.2.3 Dataflow

To implement a dataflow analysis in ROSE users must first extend the Lattice class to create an abstraction of the applications state that will be used by the analysis. Lattices implement methods such as meet, equality, ordering and operators that allow the Lattice to be moved from one lexical scope to another (e.g. from a caller function to the callee). Users then create an intra-procedural analysis by extending the IntraFWDataflow to create a forward analysis and from IntraBWDataflow to create a backward analysis. Within this class they must implement a function that returns the default abstract state of any given node at the start of the analysis. Further, they implement a transfer function that maps the applications abstract state from before a given CFG node to the state that results from the execution of the nodes expression or statement. Finally, users combine the intra-procedural analysis that they have developed with an inter-procedural analysis of their choice. This analysis will apply the intra-procedural analysis the user has implemented to the applications functions and resolve the effects of function calls on the applications abstract state, utilizing the users own state abstraction.

For a concrete example, consider how the classical constant-propagation analysis is implemented using ROSE. This analysis uses a simple abstraction of application state, where the abstract state of each variable may be a value the lattice shown in 18.4

The code in below shows a class that implements this lattice. This class derives from the FiniteLattice class because the distance between the smallest and largest value in the lattice is finite. Similar functionality is provided for infinite lattices. Its state (lines 4-15) consists of its current level in the lattice as well as its value if the level is valKnown. Since the type of value is long, this abstraction can only represent integral constants. Further, the class has a special uninitialized level that means that the object has not yet been used as part of a dataflow analysis. This class implements methods meetUpdate (lines 42-65) and the equality operator (lines 68-74) to provide the basic semantics of a lattice. meetUpdate computes least upper bound of the constraints in this lattice object and another one, storing the results in this object.If both lattices have the same state, the meet is equal to this state and if they have different states, the meet is $\top$ since the variable represented by the lattice may be set to multiple values on different execution paths. The equality operator determines whether two lattice objects have the same information content. Further, the class implements utility methods that help the dataflow framework manipulate it. The initialize method (lines 24-27) ensures the object is ready to be used. Further, two copy methods (lines 30-38) make it easy to clone lattice objects. Finally, an str method (lines 82-88) simplifies analysis debugging by printing the abstract states at CFG nodes.

.

```
class constPropLat : public FiniteLattice
{
  // The different levels of this object s lattice
  typedef enum {
    uninitialized=0, // This object is uninitialized
    bottom=1, // No constrains on this object s value are known
    valKnown=2, // The value of the variable is known (one assignment seen)
    top=3 // This variable may have more than one value
  } latticeLevels;

  // The level of this object within its lattice
  latticeLevels level;

  // The value of the variable (if level == valKnown)
  long value;

  nodeConstLattice() { level=uninitialized; }

  nodeConstLattice(const nodeConstLattice& that) :
                              value(that.value), level(that.level) {}

  // Initializes this Lattice to its default state,
  // if it is not already initialized
  void initialize() {
    if(level == uninitialized)
      level=bottom;
  }

  // Returns a copy of this lattice
  Lattice* copy() const { return new nodeConstLattice(*this); }

  // Overwrites the state of this Lattice with that of that Lattice
  void copy(Lattice* that) {
    nodeConstLattice* that = dynamic_cast<nodeConstLattice*>(that_arg);

    value = that->value;
    level = that->level;
  }

  // Computes the meet of this and that and saves the result in this
  // returns true if this causes this to change and false otherwise
  bool meetUpdate(Lattice* that) {
    // Record this object s original state to enable change detection
```

```
    unsigned long  oldValue = value;
    latticeLevels oldLevel = level;

    // Cast that into a nodeConstLattice and abort if this is not possible
    nodeConstLattice* that = dynamic_cast<nodeConstLattice*>(that_arg);
    ROSE_ASSERT(that);

    // If that is at a higher lattice level than this, the variable must have
    // multiple possible value on different execution paths
    if(that->level > level) level = top;
    // If both are at the same level
    else if(that->level == level) {
      // If lattices correspond to different values of the variable
      if(level == valKnown && value != that->value)
        level = top; // The union of both these facts is top
    }
    // Otherwise, this lattice doesn t change

    // Return whether this object was modified
    return (oldValID != valID) ||
           (oldLevel != level);
  }

  // Equality Operator
  bool operator==(Lattice* that_arg) {
    // Cast that into a nodeConstLattice and abort if this is not possible
    nodeConstLattice* that = dynamic_cast<nodeConstLattice*>(that_arg);
    ROSE_ASSERT(that);

    return level==that->level && (level!=valKnown || value==that->value);
  }

  // Returns a string representation of this object (this function is
  // required to simplify debugging)
  string str(string indent="") {    }

  // Sets the state of this lattice to the given value. Returns true if this
  // causes the lattice's state to change, false otherwise
  bool set(long value)
  {
    bool modified = this->level != valKnown || this->value != value;
    this-> value = value;
    level = valKnown;
    return modified;
  }
};
```

The second step in implementing constant propagation is to provide a class that implements the dataflow analysis itself. This is done by extending the IntraFWDataflow class, which implements forward intra-procedural analyses and implementing the genInitState and transfer methods, described below.

.

```
class constPropAnalysis : public IntraFWDataflow
{
  constPropAnalysis (): IntraFWDataflow() { }

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

  // The transfer function that is applied to every node in the CFG
  // n - The dataflow node that is being processed
  // state - The NodeState object that describes the state of the node, as
  //         established by earlier analysis passes
  // dfInfo - The Lattices that this transfer function operates on. The
  //          function takes these lattices as input and overwrites them with
  //          the result of the transfer.
  // Returns true if any of the input lattices changed as a result of the
  //    transfer function and false otherwise.
  bool transfer(const Function& func, const DataflowNode& n,
                NodeState& state, const vector<Lattice*>& dfInfo);
}
```

The constPropAnalysis implementation of method genInitState creates a lattice (lines 5-7) that maintains the initial abstract state of the application at CFG node n. This lattice is an instance of the utility class FiniteVarsExprsProductLattice, which creates one copy of constPropLat for each variable that is live at node n. Since it is a product of lattices, this class is also a lattice with well-defined meet and equality operators based on the operators of its constituent lattices. The dataflow framework provides an identical class for infinite lattices as well as a generic ProductLattice class for arbitrary products of lattices. The function then adds (line 4) the lattice to vector initLattices, which is read by the dataflow framework. This function can also specify one or more facts that the framework will maintain at each node. These facts are not subject to dataflow iteration and can be used to maintain information that is useful independently of the current dataflow state.

.

```
1 void constPropAnalysis::genInitState(const Function& func,
2               const DataflowNode& n, const NodeState& state,
3               vector<Lattice*>& initLattices, vector<NodeFact*>& initFacts) {
4   initLattices.push_back(
5           new FiniteVarsExprsProductLattice(true, false,
```

```
6                                  new constPropLat(),
7                                  NULL, func, n, state));
8 }
```

The transfer method maps the abstract state before the CFG node n to the state that results from its execution. It begins by accessing the applications abstract state above node n from the dfInfo argument (lines 6-7) . This is the vector of lattices created by genInitState for node n. It can also be obtained from the state object, which maintains the state of the lattices both below and above each node, as well as the facts at each node. The function then initializes all the constPropLats in the product lattice (10-13) and advances to analyze the effects of the current node on the abstract state.

Lines 16-127 show how the transfer function operates on different types of SgNodes. This code leverages a key feature of how ROSE represents the applications structure. Since ROSE focuses on source-to-source transformations that minimize the changes in the applications source code, all analyses must work on the original AST and cannot perform large normalization passes such as transforming the application into SSA form. Since it is difficult to implement complex analyses on top of the AST, we have developed an "on-demand" normalization that significantly simplifies the analysis development without changing the AST. Working with AST is difficult because AST sub-trees that describe the structure of expressions are complex and difficult to parse (e.g. consider analyzing all the side-effects of a=b=foo(c=d)). As such, our framework treats every SgExpression that does not correspond to an actual memory object as if it produces a temporary object that is read by its parent SgExpression. For example, in the SgExpression a=(b=c*5+d), SgIntVal 5 produces a temporary variable that is consumed by SgMultiplyOp c*5. SgVarRefExp c produces a real application variable, which is also consumed by the SgMultiplyOp. The SgMultiplyOp in turn produces a temporary variable that is consumed by SgAddOp c*5+d, which produces a temporary variable that is consumed by SgAssignOp b=c*5+d, the result of which is consumed by SgAssignOp a=(b=c*5+d). The use of these temporary variables makes it possible for user analyses to focus on just the effects of individual AST nodes without having to analyze sub-trees of the AST. Section 18.2.4 discusses how this on-demand normalization in maintained when updating to the AST.

The effects of integral constants (e.g. SgIntVal or SgLongLongIntVal) are transferred on lines 16-31. On line 20 the analysis calls function SgExpr2Var to convert the SgExpression into a varID, which is an abstract representation of the memory object (either a real or temporary variable) denoted by the SgExpression. On line 21 it queries the FiniteVarsExprsProductLattice prod with this varID to get the constPropLat associated with this memory object. If this variable is live (a non-NULL lattice is returned), on lines 26-28 it sets the lattice object to be at level valKnown and sets the value to be equal to the constant represented by the SgExpression.

The same logic is used for non-integral constants on lines 33-40. However, since our abstraction cannot represent such constants, their lattices are set to $\top$. Lines 44-60 manage assignments and the lattices of the left-hand-side expression and the assignment SgAssignOp itself are set to be equal to the lattice of the right-hand-side expression. The code for variable declaration (lines 63-81) and initialization (lines 84-98) are similar in that the lattice of the right-hand-side is copied to the lattice of the initialized variable. Finally, lines 101-127 focus on arithmetic operations. If the lattices of the left- and right-hand-side expressions are both at levels valKnown,the operation is performed immediately by the analysis on their statically known values and the result is stored in the lattices of the left-hand-side expression and the SgExpression itself. Finally, on line 129 the function returns the modified variable, which keeps track of whether the state of the downstream lattices has changed. Since these lattices are inputs to other SgExpressions, this informs the dataflow framework whether it needs to analyze how these lattices are transferred by those expressions.

.

```
1  bool constPropAnalysis::transfer(const Function& func,
2                          const DataflowNode& n,
3                          NodeState& state, const vector<Lattice*>& dfInfo) {
4    bool modified=false;
5    // Get the lattice object
6    FiniteVarsExprsProductLattice* prodLat =
7              dynamic_cast<FiniteVarsExprsProductLattice*>(*(dfInfo.begin()));
8
9    // Make sure that all the non-constant Lattices are initialized
10   const vector<Lattice*>& lattices = prodLat->getLattices();
11   for(vector<Lattice*>::const_iterator it = lattices.begin();
12       it!=lattices.end(); it++)
13     (dynamic_cast<nodeConstLattice*>(*it))->initialize();
14
15   // Integral Numeric Constants
16   if(isSgLongLongIntVal(n.getNode())  ||
17     // Other types of integral constants
18     ...) {
19     // Memory object and lattice of the expression s result
20     varID res = SgExpr2Var(isSgExpression(n.getNode()));
21     constPropLat* resLat = dynamic_cast<constPropLat*>(
22                                             prodLat->getVarLattice(res));
23
24     // If the result expression is live
25     if(resLat) {
26       if(isSgLongLongIntVal(n.getNode()))
27         modified = resLat->set(isSgLongLongIntVal(n.getNode())->get_value())
28                    || modified;
29       // Same for other types of integral constants
30       ...
31     }
32   // Non-integral Constants
33   } else if(isSgValueExp(n.getNode())) {
34     // Memory object and lattice of the expression s result
35     varID res = SgExpr2Var(isSgExpression(n.getNode()));
36     constPropLat* resLat = dynamic_cast<constPropLat*>(
37                                             prodLat->getVarLattice(res));
38     // If the result expression is live, set it to top since we only work
39     // with integral constants
40     if(resLat) modified = resLat->setTop() || modified;
```

```
  // Plain assignment: lhs = rhs
  } else if(isSgAssignOp(n.getNode())) {
    // Memory objects denoted by the expression s left- and right-hand
    // sides as well as the SgAssignOp itself
    varID lhs = SgExpr2Var(isSgAssignOp(n.getNode())->get_lhs_operand());
    varID rhs = SgExpr2Var(isSgAssignOp(n.getNode())->get_rhs_operand());
    varID res = SgExpr2Var(isSgExpression(n.getNode()));

    // The lattices associated the three memory objects
    constPropLat* resLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(res));
    constPropLat* lhsLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(lhs));
    constPropLat* rhsLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(rhs));

    // If the lhs and/or the SgAssignOp are live, copy lattice from the rhs
    if(lhsLat){ lhsLat->copy(rhsLat); modified = true; }
    if(resLat){ resLat->copy(rhsLat); modified = true; }

  // Variable Declaration
  } else if(isSgInitializedName(n.getNode())) {
    varID var(isSgInitializedName(n.getNode()));
    constPropLat* varLat = dynamic_cast<constPropLat*>(
                                                  prodLat->getVarLattice(var));

    // If this variable is live
    if(varLat) {
      // If there was no initializer, initialize its lattice to Bottom
      if(initName->get_initializer()==NULL)
        modified = varLat->setBot() || modified;
      // Otherwise, copy the lattice of the initializer to the variable
      else {
        varID init = SgExpr2Var(
                        isSgInitializedName(n.getNode())->get_initializer());
        ConstPropLat* initLat = dynamic_cast<ConstPropLat*>(
                                                  prodLat->getVarLattice(init));
        if(initLat) { varLat->copy(initLat); modified = true; }
      }
    }

  // Initializer for a variable
  } else if(isSgAssignInitializer(n.getNode())) {
    // Memory objects of the initialized variable and the
    // initialization expression
```

```
    varID res = SgExpr2Var(isSgAssignInitializer(n.getNode()));
    varID asgn = SgExpr2Var(isSgAssignInitializer(
                                                n.getNode())->get_operand());

    // The lattices associated both memory objects
    constPropLat* resLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(res));
    constPropLat* asgnLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(asgn));

    // If the variable is live, copy lattice from the assignment
    if(resLat){ resLat->copy(asgnLat); modified = true; }

  // += Arithmetic Operation
  } else if(isSgPlusAssignOp(n.getNode())) {
    // Memory objects denoted by the expression s left- and right-hand
    // sides as well as the SgAssignOp itself
    varID lhs = SgExpr2Var(isSgAssignOp(n.getNode())->get_lhs_operand());
    varID rhs = SgExpr2Var(isSgAssignOp(n.getNode())->get_rhs_operand());
    varID res = SgExpr2Var(isSgExpression(n.getNode()));

    // The lattices associated the three memory objects
    constPropLat* resLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(res));
    constPropLat* lhsLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(lhs));
    constPropLat* rhsLat =
            dynamic_cast<constPropLat*>(prodLat->getVarLattice(rhs));

    // If the lhs and/or the SgAssignOp are live and we know both their
    // values of the value of the rhs expression, set their lattice to be the
    // sum of the two.
    if(lhsLat && lhsLat->level==constPropLat::valKnown &&
       rhsLat->level==constPropLat::valKnown)
    { modified = lhsLat->set(lhsLat->value + rhsLat->value) || modified; }
    if(resLat && resLat->level==constPropLat::valKnown &&
       rhsLat->level==constPropLat::valKnown)
    { modified = resLat->set(resLat->value + rhsLat->value) || modified; }
  }
  // Same for other arithmetic operations
  ...

  return modified;
}
```

Once the intra-procedural analysis has been implemented, it can be executed on the application by combining it with an inter-procedural analysis. Currently two such analyses are implemented. ContextInsensitiveInterProceduralDataflow implements a basic context-insensitive analysis that propagates abstract state from callers to callees but does not differentiate between different call sites of the same function. As such, it is sensitive to inter-procedural data flows but can be imprecise because it takes into account control flows that are actually impossible, such as entering a function from one call site but returning to another. The code below provides an example of how this analysis is used to create an inter-procedural constant propagation analysis. The dataflow framework is initialized on line 6 and the applications call graph is built on lines 9-11. The intra-procedural analysis object is created on line 17 and the context-insensitive inter-procedural analysis is created on line 20. The user passes into its constructor references to their intra-procedural analysis and the call graph. Finally, on line 23 the user applies the full inter-procedural analysis to the entire application.

.

```
1  int main( int argc, char * argv[] ) {
2    // Build the AST used by ROSE
3    SgProject* project = frontend(argc,argv);
4
5    // Initialize the ROSE dataflow framework
6    initAnalysis(project);
7
8    // Build the call graph
9    CallGraphBuilder cgb(project);
10   cgb.buildCallGraph();
11   SgIncidenceDirectedGraph* graph = cgb.getGraph();
12
13   // Set the debug level to print the progress of the dataflow analysis
14   analysisDebugLevel = 1;
15
16   // Create the intra-procedural constant propagation analysis
17   constPropAnalysis cp(project);
18
19   // Create the inter-procedural analysis for intra-analysis cp
20   ContextInsensitiveInterProceduralDataflow inter_cp(&cp, graph);
21
22   // Run inter-procedural constant propagation on the entire application
23   inter_cp.runAnalysis();
24 }
```

To simplify debugging the framework also provides the UnstructuredPassInterDataflow analysis, which simply applies the users intra-procedural analysis on each function within the application. While this produces globally incorrect results, it simplifies debugging analyses on individual functions.

18.2.4 Transferring Information Between Analyses

Since in practice users need to implement multiple analyses where one depends on the results of another, the ROSE dataflow framework maintains the results of all analyses at each CFG nodes and makes it easy for analyses to access this data. The lattices and facts of a given CFG node are stored in its associated NodeState object. The data produced by an analysis can be retrieved by using its pointer, as shown in the example below.

This code shows analysis exAnalysis, which takes in its constructor a pointer to the constPropAnalysis described above (lines 4-5). Inside its transfer function this analysis calls the getLatticeBelow method of its argument state (instance of the NodeState class) to get the lattice associated with constPropAnalysis that has index 0 (lines 11-13). It then gets the constPropLat of any variable it cares about and make analysis decisions based on what is statically known about its state.

.

```
1  class exAnalysis {
2    // Class maintains a pointer to the constant propagation analysis to make
3    // it possible to access its results
4    constPropAnalysis& cpAnalysis;
5    exAnalysis(constPropAnalysis* cpAnalysis) : cpAnalysis (cpAnalysis) {}
6
7    bool transfer(const Function& func, const DataflowNode& n,
8                  NodeState& state, const vector<Lattice*>& dfInfo) {
9      // Get the Lattices computed by the constant propagation analysis for the
10     // current CFG node
11     FiniteVarsExprsProductLattice* prodLat =
12                 dynamic_cast<FiniteVarsExprsProductLattice*>(
13                                       state->getLatticeBelow(cpAnalysis, 0));
14
15     // Some application variable of interest
16     varID var = ...;
17
18     // The constPropLat of this variable
19     constPropLat varCPLat = dynamic_cast<constPropLat*>(
20                                                 prodLat->getVarLattice(res));
21
22     // Analyze differently depending on what is known about the
23     // variable s value
24     if(varCPLat)
25       if(varCPLat->level == constPropLat::bottom) {
26         ...
27       } else if(varCPLat->level == constPropLat::valKnown) {
28         ...
29       } else if(varCPLat->level == constPropLat::top) {
30         ...
31       }
32   }
33   ...
```

```
};
```

The code below shows the full functionality of the NodeState class. . Lines 5-16 show the functions to set, get and delete the lattices above and below the associated CFG node. Lines 20-26 provide the same functionality for facts. The str method on line 20 returns a string representation of the lattices and facts associated with the CFG node, which is very useful for debugging. Lines 37-50 show the objects static methods. The getNodeState method on line 37 returns the NodeState object of a given CFG node. Since the ROSE virtual CFG can have multiple CFG nodes for the same AST node, this method requires an additional index argument to identify the node in question. Finally, method copyLattices_aEQa and related methods (lines 39-50) copy lattice information from above a CFG node to below it and vice versa, from one node to another or from one analysis at a given node to another analysis at the same node.

.

```
1  class NodeState
2  {
3    // Sets the lattices above/below this node for the given analysis to the
4    // given lattice vector
5    void setLatticeAbove(const Analysis* analysis, vector<Lattice*>& lattices);
6    void setLatticeBelow(const Analysis* analysis, vector<Lattice*>& lattices);
7
8    // Returns the lattice latticeName generated by the given analysis from
9    // above/below the node
10   Lattice* getLatticeAbove(const Analysis* analysis, int latticeName) const;
11   Lattice* getLatticeBelow(const Analysis* analysis, int latticeName) const;
12
13   // Deletes all lattices above/below this node that are associated with the
14   // given analysis
15   void deleteLatticeAbove(const Analysis* analysis);
16   void deleteLatticeBelow(const Analysis* analysis);
17
18   // Sets the facts at this node for the given analysis to the given
19   // fact vector
20   void setFacts(const Analysis* analysis, const vector<NodeFact*>& newFacts);
21
22   // Returns the given fact, owned by the given analysis
23   NodeFact* getFact(const Analysis* analysis, int factName) const ;
24
25   // Deletes all facts at this node associated with the given analysis
26   void deleteFacts(const Analysis* analysis);
27
28   // Returns a string representation of all the lattices and facts
29   // associated with the CFG node
30   string str(Analysis* analysis, string indent) const;
31
32   // --- Static Methods ---
33   // Returns the NodeState object associated with the given dataflow node
```

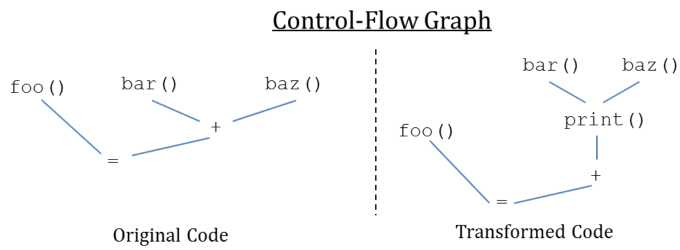


Figure 18.5: Example of Transformation on the CFG

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

  // Copies from's above lattices for the given analysis to to's above
  // lattices for the same analysis
  static void copyLattices_aEQa(Analysis* analysis, NodeState& to, const
                                NodeState& from);

  // Copies from's above lattices for analysisA to to's above lattices for
  // analysisB
  static void copyLattices_aEQa(Analysis* analysisA, NodeState& to,
                                Analysis* analysisB, const NodeState& from);

  // Similar methods for copying in different permutations
  ...
};
```

18.2.5 CFG Transformations

ROSE makes it easy to modify the applications AST as a result of dataflow analyses. The dataflow framework maintains an on-demand normal form, where analyses can focus on the actions of individual SgNodes and ignore how they are arranged within the AST. ROSE maintains this abstraction by providing an API that inserts new SgExpressions into the applications CFG, making all the needed changes in the AST to make sure that the correct control flow is maintained.

To get the intuition of this functionality consider the expression foo()=(bar()+baz()). Suppose the user has decided based on the dataflow state before the SgAddOp + that it they want to add a call to function print immediately before it. From the perspective of the CFG, this is a simple and well-defined operation, as shown at the top of Figure 4. The side-effects of the calls to bar and baz must complete before the call to print and the side-effects of print must complete before the execution of the + operation. The call to foo is not well-ordered relative print or the other

Source Code

```
foo() = bar() + baz()
```

Original Code

```
int tmp1, tmp2
foo() = (tmp1=bar(),
         tmp2=baz(),
         print(),
         tmp1 + tmp2)
```

Transformed Code

Figure 18.6: Example of the Transformation on the Source Code

operations by the structure of the CFG.

Unfortunately, it is difficult to implement these semantics in the context of the AST because (i) there is no way to add a function call to an SgAddOp and (ii) because in C++ the sequence points required by the above semantics (some side-effects much complete before others) are provided by a few specific constructs such as statement boundaries and the comma operator. As such, the transformation requires the complex set of AST changes shown in the Figure 5. We must create temporary variables to hold the results of the calls to bar and baz. We then transform the original SgAddOp into a longer SgCommaOpExp, where we first call bar and baz, saving their results into the temporary variables, then call print and finally perform the addition. The result of the addition is the result of the entire comma expression, so this transformation correctly enforces the semantics of the desired transformation.

ROSE provides the three functions to make it easy to insert expressions into the CFG. Functions insertBeforeUsingCommaOp and insertAfterUsingCommaOp insert SgExpressions before or after existing SgExpressions using a generalization of the transformation described in Figure 18.6.

.

```
// Insert an expression (new_exp) before another expression (anchor_exp) has
// possible side effects, without changing the original semantics. This is
// achieved by using a comma operator: (new_exp, anchor_exp). The comma
// operator is returned.
SgCommaOpExp *insertBeforeUsingCommaOp(SgExpression* new_exp,
                                        SgExpression* anchor_exp);
// Insert an expression (new_exp) after another expression (anchor_exp) has
// possible side effects, without changing the original semantics. This is
// done by using two comma operators:
//     type T1; ... ((T1 = anchor_exp, new_exp),T1) )... ,
// where T1 is a temp variable saving the possible side effect of anchor_exp.
// The top level comma op exp is returned. The reference to T1 in T1 =
// anchor_exp is saved in temp_ref.
SgCommaOpExp *insertAfterUsingCommaOp(SgExpression* new_exp,
                         SgExpression* anchor_exp, SgStatement** temp_decl = NULL,
                         SgVarRefExp** temp_ref = NULL);
```

Function replaceWithPattern (Figure 18.2.5) replaces one SgExpression with another. However, since the original expression may still be valuable, it allows the original expression to be

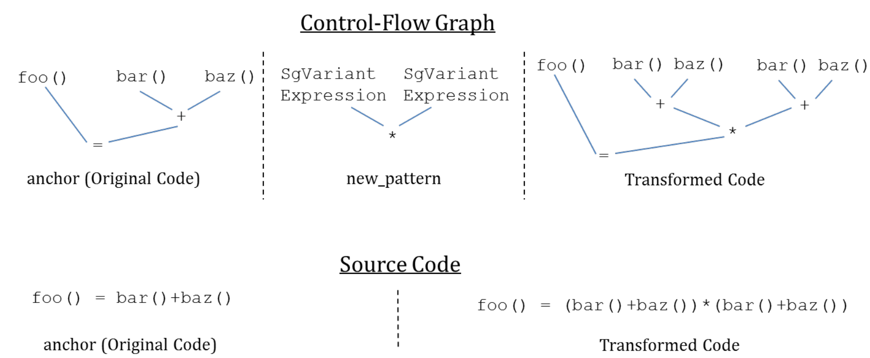


Figure 18.7: Code Replacement Transformation

included at one or more locations inside the new expression that contain nodes of type SgVariantExpression.

.

```
// Replace an anchor node with a specified pattern subtree with optional
// SgVariantExpression. All SgVariantExpression in the pattern will be
// replaced with copies of the anchor node.
SgNode* replaceWithPattern (SgNode * anchor, SgNode* new_pattern);
```

An example of this transformation is shown Figure 18.7, where the original code is the same as in the example above and the new_pattern expression is a single SgMultOp where the arguments are both SgVariantExpressions. The result of the transformation is that the original SgAddOp is replaced with a multiplication the arguments of which are copies of the SgAddOp: (bar()+baz())*(bar()+baz()).

Chapter 19

Recognizing Loops

Figures 19.1 and 19.2 show a translator which reads an application and gathers a list of loop nests. At the end of the traversal it reports information about each loop nest, including the function where it occurred and the depth of the loop nest.

FIXME: *This example program is unfinished. It will output a list of objects representing information about perfectly nested loops.*

Using this translator we can compile the code shown in figure 19.3. The output is shown in figure 19.4.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

class InheritedAttribute
   {
     public:
          int loopNestDepth;

          InheritedAttribute () : loopNestDepth(0) {};
          InheritedAttribute ( const InheritedAttribute & X ) {};
   };

class SynthesizedAttribute
   {
     public:
         SynthesizedAttribute() {};
   };

class Traversal : public SgTopDownBottomUpProcessing<InheritedAttribute,SynthesizedAttribute>
   {
     public:
       // Functions required
          InheritedAttribute evaluateInheritedAttribute (
             SgNode* astNode,
             InheritedAttribute inheritedAttribute );

          SynthesizedAttribute evaluateSynthesizedAttribute (
             SgNode* astNode,
             InheritedAttribute inheritedAttribute,
             SubTreeSynthesizedAttributes synthesizedAttributeList );
   };

InheritedAttribute
Traversal::evaluateInheritedAttribute (
     SgNode* astNode,
     InheritedAttribute inheritedAttribute )
   {
     switch(astNode->variantT())
        {
          case V_SgForStatement:
             {
               printf ("Found a SgForStatement \n");

            // This loop is one deepper than the depth of the parent's inherited attribute
               inheritedAttribute.loopNestDepth++;

               break;
             }

          default:
             {
            // g++ needs a block here
             }
        }

     return inheritedAttribute;
   }

SynthesizedAttribute
Traversal::evaluateSynthesizedAttribute (
     SgNode* astNode,
     InheritedAttribute inheritedAttribute,
```

Figure 19.1: Example source code showing loop recognition (part 1).

```
      SubTreeSynthesizedAttributes synthesizedAttributeList )
   {
     SynthesizedAttribute returnAttribute;

     switch(astNode->variantT())
        {
          case V_SgForStatement:
             {

               break;
             }

          default:
             {
            // g++ needs a block here
             }
        }

     return returnAttribute;
   }



int
main ( int argc, char* argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

  // Build the inherited attribute
     InheritedAttribute inheritedAttribute;

     Traversal myTraversal;

  // Call the traversal starting at the sageProject node of the AST
     myTraversal.traverseInputFiles(project,inheritedAttribute);

     return 0;
   }
```

Figure 19.2: Example source code showing loop recognition (part 2).

```

int main()
   {
     int x[4];
     int y[4][4];

     for (int i=0; i < 4; i++)
        {
          x[i] = 7;
        }

     for (int i=0; i < 4; i++)
        {
          for (int j=0; j < 4; j++)
             {
               y[i][j] = 42;
             }
        }

     return 0;
   }
```

Figure 19.3: Example source code used as input to loop recognition processor.

```
Found a SgForStatement
Found a SgForStatement
Found a SgForStatement
```

Figure 19.4: Output of input to loop recognition processor.

Chapter 20

Virtual CFG

The ROSE virtual control flow graph interface provides a higher level of detail than ROSE's other control flow graph interfaces. It expresses control flow even within expressions, and handles short-circuited logical and conditional operators properly[1]. The interface is referred to as "virtual" because no explicit graph is ever created: only the particular CFG nodes and edges used in a given program ever exist. CFG nodes and edges are value classes (they are copied around by value, reducing the need for explicit memory management).

A CFG node consists of two components: an AST node pointer, and an index of a particular CFG node within that AST node. There can be several CFG nodes corresponding to a given AST node, and thus the AST node pointers cannot be used to index CFG nodes. The particular index values for the different AST node types are explained in Section 20.1.

20.1 CFGNode Index values

To facilitate traversal and represent sufficient details, each eligible ROSE AST node (expression, statement and SgInitializedName) has several corresponding CFGNodes in virtual CFG. These CFGNodes have indices from 0 to n. CFGNode of index value of 0 is used to represent the beginning CFG node for an AST node, while the CFGNode of index n is the end CFGNode for the AST node. The beginning node represents the point in the control flow immediately before the construct starts to execute, and the ending node represents the point immediately after the construct has finished executing. Note that these two nodes do not dominate the other CFG nodes in the construct due to goto statements and labels.

Reimplementation of SgNode::cfgIndexForEnd() returns the index value for n of each eligible SgNode type. See source file *src/frontend/SageIII/virtualCFG/memberFunctions.C* for valid index values for each type of eligible SgNode.

[1] It assumes operands of expressions are computed in left-to-right order, unlike the actual language semantics, however.

20.2 Important functions

The main body of the virtual CFG interface is in virtualCFG.h; the source code is in src/frontend/SageIII/virtualCFG/ and is linked into librose. The filtered CFG interface explained below is in filteredCFG.h, and functions for converting the CFG to a graph in Dot format are in cfgToDot.h.

Two functions provide the basic way of converting from AST nodes to CFG nodes. Each SgNode has two methods, cfgForBeginning() and cfgForEnd(), to generate the corresponding beginning and end CFG nodes, respectively. These functions require that the AST node is either an expression, a statement, or a SgInitializedName.

20.2.1 Node methods

- CFGNode(SgNode* node, unsigned int index): Build a CFG node from the given AST node and index. Valid index values are in Section 20.1.
- toString(): Produce a string representing the information in the node.
- toStringForDebugging(): Similar, but with more internal debugging information.
- id(): A C identifier representing the node.
- getNode(): Get the underlying AST node.
- getIndex(): Get the index (as explained in Section 20.1) for this CFG node within its underlying AST node.
- outEdges(): Return a vector of outgoing CFG edges from this node. This function internally calls cfgOutEdges(unsigned int idx) to generate out edges for each CFGNode of a given index value.
- inEdges(): Return a vector of CFG edges coming into this node (note that the sources and targets of the edges are not reversed, and so each in edge has its target as the current node). This function internally calls cfgInEdges(unsigned int idx) to generate in edges for each CFGNode of a given index value.
- isInteresting (): See Section 20.6.1.
- Nodes are also comparable using the operators ==, !=, and <.

20.2.2 Edge methods

- toString(): Produce a string representing the information in the node.
- toStringForDebugging(): Similar, but with more internal debugging information.
- id(): A C identifier representing the node.
- source(): The starting CFG node for this edge.
- target(): The ending CFG node for this edge.

- condition(): When there are multiple CFG edges from the same starting node, each of them is taken under certain conditions. The condition() method returns the condition, of type EdgeConditionKind. The possible return values are:
  - eckUnconditional: An edge that is always taken.
  - eckTrue: True case of a two-way branch (either an if statement or a loop
  - eckFalse: False case of a two-way branch
  - eckCaseLabel: Case label in a switch statement (key is given by caseLabel())
  - eckDefault: Default label of a switch statement
  - eckDoConditionPassed: Enter Fortran do loop body
  - eckDoConditionFailed: Fortran do loop finished
  - eckForallIndicesInRange: Start testing forall mask
  - eckForallIndicesNotInRange: End of forall loop
  - eckComputedGotoCaseLabel: Case in computed goto – number needs to be computed separately
  - eckArithmeticIfLess: Edge for the arithmetic if expression being less than zero
  - eckArithmeticIfEqual: Edge for the arithmetic if expression being equal to zero
  - eckArithmeticIfGreater: Edge for the arithmetic if expression being greater than zero
- caseLabel(): For an edge with condition eckCaseLabel, an expression representing the key for the case label.
- computedGotoCaseIndex(): The index of this edge's case within a Fortran computed goto (an edge of kind eckComputedGotoCaseLabel).
- conditionBasedOn(): The test expression or switch expression that is tested by this edge.
- scopesBeingExited(), scopesBeingEntered(): Variables leaving and entering scope during this edge. This information has not been extensively verified, and should not be relied upon.
- Edges can also be compared using the operators == and !=. They are not ordered to avoid dependencies on pointer comparison on different computers.

20.3 Drawing a graph of the CFG

Fig. 20.3 shows a translator to dump full (debug) virtual control flow graphs for all functions within input source files. It also dumps a simplified version (interesting) version of virtual control flow graphs. A standalone tool named *virtualCFG* is installed under *ROSE_INSTALL_TREE/bin* for users to generate both debug and interesting dot files of virtual CFGs.

The example input code is given in Fig. 20.3. Debug and interesting virtualCFG of function main() are shown in Fig. 20.3 and Fig. 20.4, respectively. Debug and interesting virtualCFG of function testIf () are shown in Fig. 20.5 and Fig. 20.6, respectively.

```
// Example translator to generate dot files of virtual control flow graphs
#include "rose.h"
#include <string>
using namespace std;

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

  // Process all function definition bodies for virtual control flow graph generation
  Rose_STL_Container<SgNode*> functions = NodeQuery::querySubTree(sageProject, V_SgFunctionDefinition);
  for (Rose_STL_Container<SgNode*>::const_iterator i = functions.begin(); i != functions.end(); ++i)
  {
    SgFunctionDefinition* proc = isSgFunctionDefinition(*i);
    ROSE_ASSERT (proc != NULL);
    string fileName= StringUtility::stripPathFromFileName(proc->get_file_info()->get_filenameString());
    string dotFileName1=fileName+"."+ proc->get_declaration()->get_name() +".debug.dot";
    string dotFileName2=fileName+"."+ proc->get_declaration()->get_name() +".interesting.dot";

    // Dump out the full CFG, including bookkeeping nodes
    VirtualCFG::cfgToDotForDebugging(proc, dotFileName1);

    // Dump out only those nodes which are "interesting" for analyses
    VirtualCFG::interestingCfgToDot (proc, dotFileName2);
  }

  return 0;
}
```

Figure 20.1: Example source code showing visualization of virtual control flow graph.

As we can see in Fig. 20.5, the debug dot graph has all CFGNodes for each eligible SgNode. For example, there are three CFGNodes for SgIfStmt, with indices from 0 to 2. The interesting CFGNode of SgIfStmt has solid line oval while non-essential CFGNodes have dashed line ovals in the graph. The caption of each node has a format of *<SgNode-Enum-value>@line-number:CFGNode-index-value*. It is obvious from the graph that SgIfStmt's interesting CFGNode has an index value of 1. In comparison, Fig. 20.6 only shows interesting CFGNodes with solid line ovals. Again, the captions tells line numbers and CFGNode's index values for each CFGNode.

```
#include <stdio.h>
#include <string.h>
#include <assert.h>

size_t i;
char buffer[10];
int main(int argc, char *argv[])
{
  for (i=0; i < strlen(argv[1]); i++)
  {
    buffer[i] = argv[1][i];
  }
  return 0;
}

int testIf(int i)
{
  int rt;
  if (i%2 ==0)
    rt =0;
  else
    rt =1;

  return rt;
}
```

Figure 20.2: Example source code used as input to build virtual control graphs.

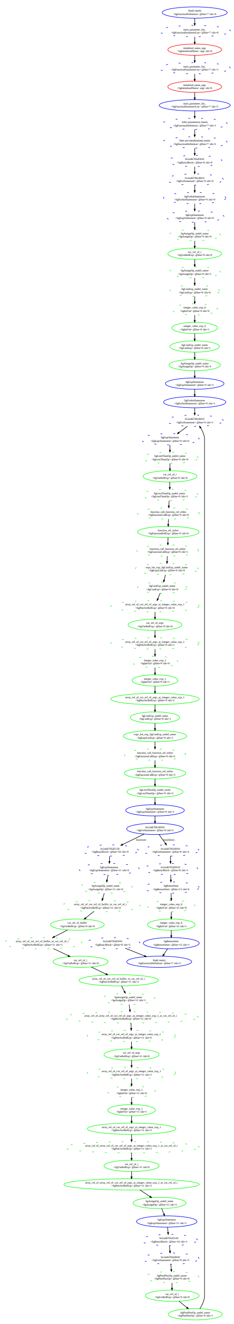


Figure 20.3: The debug virtual control flow graph for function *main()* shows all virtual CFG nodes and edges

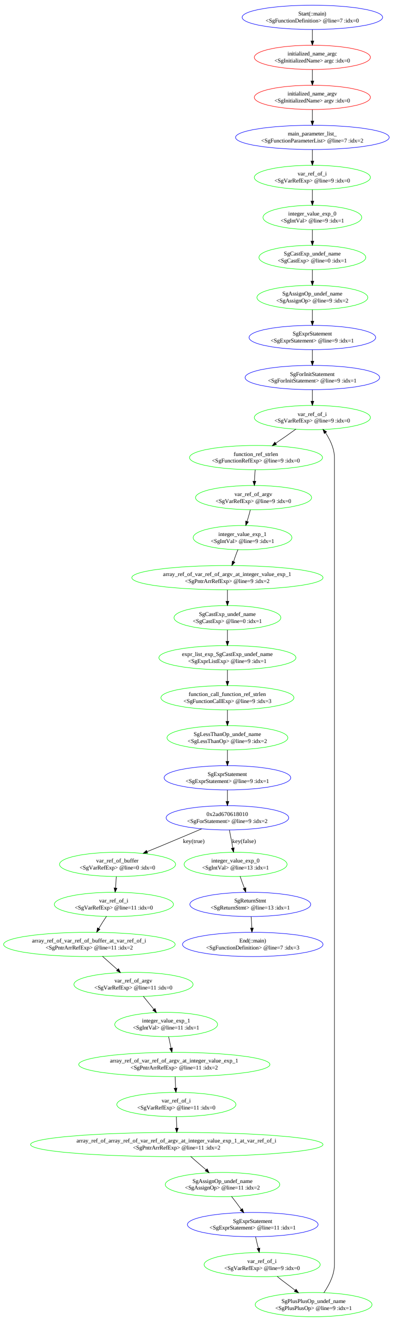


Figure 20.4: The virtual control flow graph for function *main()* shows only interesting virtual CFG nodes and edges. Each CFGNode's caption tells associated source line number and CFGNode index value (@line-num:index-value)

Figure 20.5: The debug virtual control flow graph for function *testIf()* shows all virtual CFG nodes and edges

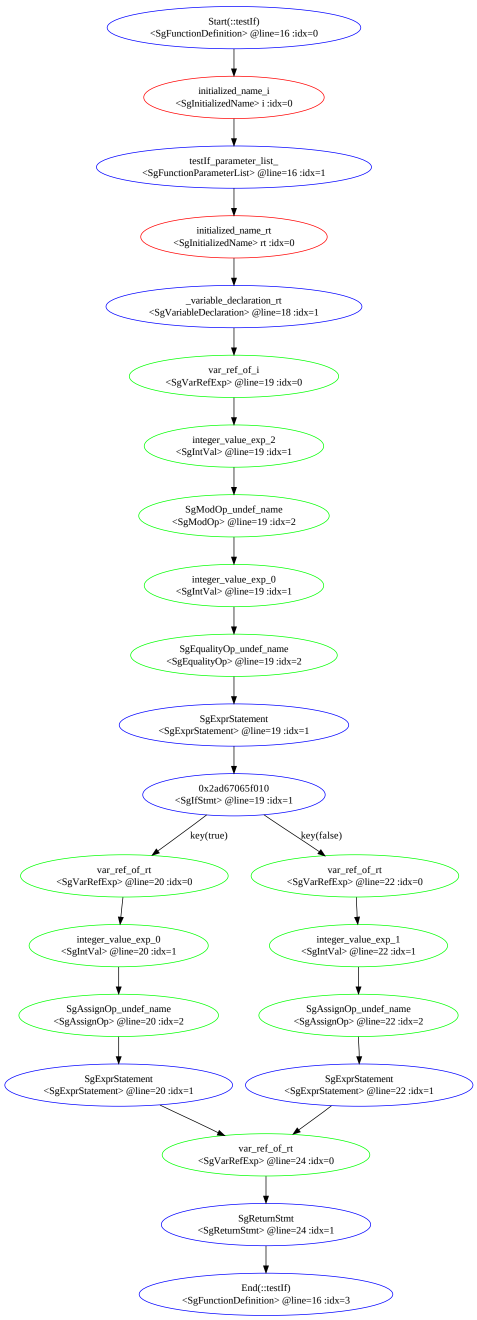


Figure 20.6: The virtual control flow graph for function *testIf()* shows only interesting virtual CFG nodes and edges. Each CFGNode's caption tells associated source line number and CFGNode index value (@line-num:index-value)

20.4 Robustness to AST changes

Control flow graph nodes and edges can be kept (i.e., are not invalidated) in many cases when the underlying AST changes. However, there are some limitations to this capability. Changing the AST node that is pointed to by a given CFG node is not safe. CFG nodes for deleted AST nodes are of course invalid, as are those pointing to AST nodes whose parent pointers become invalid.

20.5 Limitations

Although workable for intraprocedural analysis of C code, the virtual CFG code has several limitations for other languages and uses.

20.5.1 Fortran support

The virtual control flow graph includes support for many Fortran constructs, but that support is fairly limited and not well tested. It is not recommended for production use.

20.5.2 Exception handling

The virtual CFG interface does not support control flow due to exceptions or the setjmp/longjmp constructs. It does, however, support break, continue, goto, and early returns from functions.

20.5.3 Interprocedural control flow analysis

In virtual CFGs, interprocedural control flow analysis is disabled by default. It can be enabled by setting the parameter `virtualInteproceduralControlFlowGraphs` in SgNode::cfgInEdges, SgNode::cfgOutEdges, and their subclasses' definitions. Interprocedural edges are labeled with the `eckInterprocedural` EdgeConditionKind.

In cases where the flow of control cannot be determined statically (calls of virtual functions, function pointers, or functors), the interprocedural control flow graph contains all possible options. In keeping with the 'virtual' nature of ROSE's control flow graphs, the set of options is computed on-the-fly; therefore, changes to the AST will be reflected in subsequent interaction with the control flow graph.

20.6 Node filtering

FIXME

20.6.1 "Interesting" node filter

To simplify the virtual CFG, non-essential CFGNodes, such as the beginning and the end CFGNodes for each AST node, can be filtered out. Each eligible SgNode type has a most important CFGNode out of its all CFGNodes. The interesting CFGNode's index value for each Node type is returned by calling the derived implementation of *virtual bool SgNode::cfgIsIndexInteresting(int idx)*.

20.6.2 Arbitrary filtering

20.7 Static CFG

Since a virtual CFG does not produce any real graph, it is quite inefficient to traverse a virtual CFG frequently. It is necessary to build a static CFG which may improve the performance of some specific operations.

A *SgGraph* object (actually, it's a *SgIncidenceDirectedGraph* object) is created to store the static CFG. Each node in the graph is a *SgGraphNode* object. In a virtual CFG, each node contains two members: node and index. A *SgGraphNode* already holds a pointer to *SgNode*, and we have to add the other property "index" to our *SgGraphNode*. This can be done by adding the corresponding attribute to *SgGraphNode*.

20.7.1 Class methods

- CFG(): The default constructor.
- CFG(SgNode∗ node, bool isFiltered = false): Initialize a static CFG with the start node to build from and a flag indicating if the CFG is a full or filtered one.
- setStart(SgNode∗ node): Set the start node for building a static CFG. Note that the argument has to be an object of any of the following classes: *SgProject*, *SgStatement*, *SgExpression*, and *SgInitializedName*. If a *SgProject* object is passed in, several graphs are built for every function definition.
- isFilteredCFG(): Return the *isFiltered* flag.
- setFiltered(bool flag): Set the *isFiltered* flag.
- buildCFG(): Build a full or filtered CFG according to the *isFiltered* flag.
- buildFullCFG(): Build a full CFG for debugging.
- buildFilteredCFG(): Build a filtered CFG which only contains interesting nodes.
- getOutEdges(SgGraphNode∗ node): Return a vector of outgoing CFG edges (*SgDirectedGraphEdge* objects) from the given node.
- getInEdges(SgGraphNode∗ node): Return a vector of CFG edges coming into the given node.
- cfgForBeginning(SgNode∗ node): Return the CFG node for just before this AST node.
- cfgForEnd(SgNode∗ node): Return the CFG node for just after this AST node.
- getIndex(SgGraphNode∗ node): Return the index of the given CFG node.

- cfgToDot(SgNode∗ node, const std::string& filename): Generate a DOT file for the current CFG. Note that the start node to be drawn can be indicated which is not necessary to be the start node of the CFG.

20.7.2 Drawing a graph of the CFG

Figure 20.7.2 shows a translator to dump full (debug) static control flow graphs for all functions within input source files. It also dumps a simplified version (interesting) version of static control flow graphs.

```
// Example translator to generate dot files of static control flow graphs
#include "rose.h"
#include <string>
using namespace std;

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

  // Process all function definition bodies for static control flow graph generation
  Rose_STL_Container<SgNode*> functions = NodeQuery::querySubTree(sageProject, V_SgFunctionDefinition);
  for (Rose_STL_Container<SgNode*>::const_iterator i = functions.begin(); i != functions.end(); ++i)
  {
    SgFunctionDefinition* proc = isSgFunctionDefinition(*i);
    ROSE_ASSERT (proc != NULL);
    string fileName= StringUtility::stripPathFromFileName(proc->get_file_info()->get_filenameString());
    string dotFileName1=fileName+"."+ proc->get_declaration()->get_name() +".debug.dot";
    string dotFileName2=fileName+"."+ proc->get_declaration()->get_name() +".interesting.dot";

    StaticCFG::CFG cfg(proc);

    // Dump out the full CFG, including bookkeeping nodes
    cfg.buildFullCFG();
    cfg.cfgToDot(proc, dotFileName1);

    // Dump out only those nodes which are "interesting" for analyses
    cfg.buildFilteredCFG();
    cfg.cfgToDot(proc, dotFileName2);
  }

  return 0;
}
```

Figure 20.7: Example source code showing visualization of static control flow graph.

The example input code is given in Fig. 20.3. Debug and interesting static CFG are shown in Fig. 20.5 and Fig. 20.6, respectively.

20.8 Static, Interprocedural CFGs

ROSE supports construction of interprocedural control flow graphs using the InterproceduralCFG class, a subclass of StaticCFG. Like the StaticCFG, the InterproceduralCFG can be constructed from any SgNode that affects control flow. If an InterproceduralCFG is constructed from a given node, it will contain all possible paths of execution from that point. Edges between procedures will be labelled with the 'eckInterprocedural' `EdgeConditionKind`.

In cases where a function call cannot be statically resolved to a function definition, the InterproceduralCFG includes edges from the call node to all possible definitions, which are determined by ROSE's CallGraph.

Chapter 21

Generating Control Flow Graphs

The control flow of a program is broken into *basic blocks* as nodes with control flow forming edges between the basic blocks. Thus the control flow forms a graph which often labeled edges (true and false), and basic blocks representing sequentially executed code. This chapter presents the Control Flow Graph (CFG) and the ROSE application code for generating such graphs for any function in an input code. The CFG forms a fundamental building block for more complex forms of program analysis.

Figure 21.1 shows the code required to generate the control flow graph for each function of an application. Using the input code shown in figure 21.2 the first function's control flow graph is shown in figure 21.3.

Figure 21.3 shows the control flow graph for the function in the input code in figure 21.2.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"
#include <GraphUpdate.h>
#include "CFGImpl.h"
#include "GraphDotOutput.h"
#include "controlFlowGraph.h"
#include "CommandOptions.h"

using namespace std;

// Use the ControlFlowGraph is defined in both PRE
// and the DominatorTreesAndDominanceFrontiers namespaces.
// We want to use the one in the PRE namespace.
using namespace PRE;

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

void visitorTraversal::visit(SgNode* n)
   {
     SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(n);
     if (functionDeclaration != NULL)
        {
          SgFunctionDefinition* functionDefinition = functionDeclaration->get_definition();
          if (functionDefinition != NULL)
             {
               SgBasicBlock* functionBody = functionDefinition->get_body();
               ROSE_ASSERT(functionBody != NULL);

               ControlFlowGraph controlflow;

            // The CFG can only be called on a function definition (at present)
               makeCfg(functionDefinition,controlflow);
               string fileName = functionDeclaration->get_name().str();
               fileName += ".dot";
               ofstream dotfile(fileName.c_str());
               printCfgAsDot(dotfile, controlflow);
             }
        }
   }

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

     CmdOptions::GetInstance()->SetOptions(argc,argv);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project,preorder);

     return 0;
   }
```

Figure 21.1: Example source code showing visualization of control flow graph.

```c
#include <stdio.h>
#include <string.h>
#include <assert.h>

size_t i;
char buffer[10];
int main(int argc, char *argv[])
{
  for (i=0; i < strlen(argv[1]); i++)
  {
    buffer[i] = argv[1][i];
  }
  return 0;
}

int testIf(int i)
{
  int rt;
  if (i%2 ==0)
    rt =0;
  else
    rt =1;

  return rt;
}
```

Figure 21.2: Example source code used as input to build control flow graph.

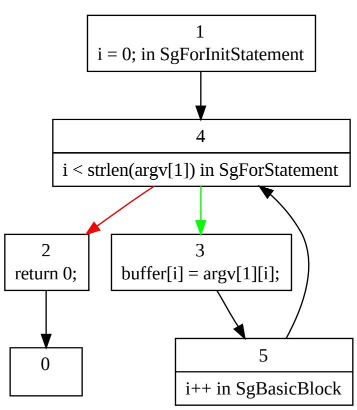


Figure 21.3: Control flow graph for function in input code file: inputCode_1.C.

Chapter 22

Graph Processing Tutorial

22.1 Traversal Tutorial

ROSE can collect and analyze paths in both source and binary CFGs. At moment it doesn't attempt to save paths because if you save them directly the space necessary is extremely large, as paths grow 2^n with successive if statements and even faster when for loops are involved. Currently a path can only cannot complete the same loop twice. However it is possible for a graph such that 1 -¿ 2 , 2-¿3, 3-¿1, 3-¿5, has paths, 1,2,3,1,2,3,5 and 1,2,3,5 because the loop 1,2,3,1 is not repeated.

The tutorial example works as such:

```cpp
#include <iostream>
#include <fstream>
//#include <rose.h>
#include <string>
#include <err.h>
#include "SgGraphTemplate.h"
#include "graphProcessing.h"

#include "staticCFG.h"
#include "interproceduralCFG.h"
/* Testing the graph traversal mechanism now implementing in AstProcessing.h (inside src/midend/astProcessing/
#include <sys/time.h>
#include <sys/resource.h>
using namespace std;
using namespace boost;






/* You need to use myGraph type here because the conversion of StaticCFG::InterproceduralCFG or StaticCFG::CFG
in a boost form. The SgGraphTemplate.h file handles this conversion and myGraph is specific to that file */
typedef myGraph CFGforT;





/*
Your basic visitor traversal subclassed from SgGraphTraversal on the CFGforT template as defined
above
*/
class visitorTraversal : public SgGraphTraversal<CFGforT>
   {
     public:
          int paths;
        /* This is the function run by the algorithm on every path, VertexID is a type implemented in SgGraphT
          void analyzePath(vector<VertexID>& pth);
   };

/* defining the analyzePath function. This simply counts paths as should be obvious. Again, VertexID is define
void visitorTraversal::analyzePath(vector<VertexID>& pth) {
        paths++;
}


int main(int argc, char *argv[]) {
  /* First you need to produce the project file*/
  SgProject* proj = frontend(argc,argv);
  ROSE_ASSERT (proj != NULL);
 /* Getting the Function Declaration and Definition for producing the graph */
  SgFunctionDeclaration* mainDefDecl = SageInterface::findMain(proj);
  SgFunctionDefinition* mainDef = mainDefDecl->get_definition();
/* Instantiating the visitorTraversal */
   visitorTraversal* vis = new visitorTraversal();
/* This creates the StaticCFG::InterproceduralCFG object to be converted to a boost graph */
    StaticCFG::InterproceduralCFG cfg(mainDef);
    stringstream ss;
    SgIncidenceDirectedGraph* g = new SgIncidenceDirectedGraph();
    /* We got the necessary internal SgIncidenceDirectedGraph from the cfg */
    g = cfg.getGraph();
    myGraph* mg = new myGraph();
/* Converting the cfg to a boost graph */
    mg = instantiateGraph(g, cfg, mainDef);
/* Set internal variables */
    vis->paths = 0;
    /* invoking the traversal, the first argument is the graph, the second is true if you
    do not want bounds, false if you do, the third and fourth arguments are starting and stopping
    vertices respectively, if you are not bounding simply insert 0. Finally the last argument is
    currently deprecated */
    vis->constructPathAnalyzer(mg, true, 0, 0, true);
    std::cout << "finished" << std::endl;
    std::cout << " paths: " << vis->paths << std::endl;
    delete vis;
}
```

Figure 22.1: Source CFG Traversal Example

```
#include <iostream>
#include <fstream>
#include <rose.h>
//#include "interproceduralCFG.h"
#include <string>
#include <err.h>

/* These are necessary for any binary Traversal */

#include "graphProcessing.h"
#include "BinaryControlFlow.h"
#include "BinaryLoader.h"
/* Testing the graph traversal mechanism now implementing in graphProcessing.h (inside src/midend/astProcessing/)*/

using namespace std;
using namespace boost;

/* These should just be copied verbatim */

typedef boost::graph_traits<BinaryAnalysis::ControlFlow::Graph>::vertex_descriptor Vertex;
/**< Graph vertex type. */
typedef boost::graph_traits<BinaryAnalysis::ControlFlow::Graph>::edge_descriptor
Edge;      /**< Graph edge type. */



/* We first make a visitorTraversal, subclassed from SgGraphTraversal templated on the BinaryAnalysis:ControlFlow::Graph
which is implemented as a boost graph */


class visitorTraversal : public SgGraphTraversal<BinaryAnalysis::ControlFlow::Graph>
   {

     public:
         long int pths;
         long int tltnodes;
          /* This needs to be in any visitorTraversal, it is the function that will be run on every path by the graph
          path analysis algorithm, notice the Vertex type is from the above typedefs */
          virtual void analyzePath( vector<Vertex>& pth);


   };


/* This is a very simple incarnation, it just counts paths */
void visitorTraversal::analyzePath(vector<Vertex>& pth) {
     pths++;
}


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

     /* Parse the binary file */
     SgProject *project = frontend(argc, argv);
     std::vector<SgAsmInterpretation*> interps = SageInterface::querySubTree<SgAsmInterpretation>(project);
     if (interps.empty()) {
         fprintf(stderr, "no binary interpretations found\n");
         exit(1);
     }

     /* Calculate plain old CFG. */
         BinaryAnalysis::ControlFlow cfg_analyzer;
         BinaryAnalysis::ControlFlow::Graph* cfg = new BinaryAnalysis::ControlFlow::Graph;

         cfg_analyzer.build_cfg_from_ast(interps.back(), *cfg);
         std::ofstream mf;
         mf.open("analysis.dot");
         /* Declaring the visitorTraversal */
         visitorTraversal* vis = new visitorTraversal;
         /* Setting internal variables*/
         vis->tltnodes = 0;
         vis->pths = 0;

         /* visitorTraversal has 5 arguments, the first is the ambient CFG, the second identifies whether or not
         you are bounding the graph, that is, whether you want all your paths to start at one specific node and end at
        another specific node, the fourth and fifth would be start and end if the graph were bounded. Since they aren't
        you can simply input 0, for the moment the final argument is deprecated, though it's purpose was to tell the progra
         that your analysis function was thread safe, that is that openMP could run it without having a critical command.
        Currently a critical is always used */

         vis->constructPathAnalyzer(cfg, true, 0, 0, false);
         std::cout << "pths: " << vis->pths << std::endl;
         std::cout << "tltnodes: " << vis->tltnodes << std::endl;
```

Chapter 23

Dataflow Analysis

The dataflow analysis in Rose is based on the control flow graph (CFG). One type of dataflow analysis is called def-use analysis, which is explained next.

23.1 Def-Use Analysis

The definition-usage (def-use) analysis allows to query the definition and usage for each *control flow node* (CFN). Any statement or expression within ROSE is represented as a sequence of CFN's. For instance, the CFG for the following program

```
int main()
{
  int x = 9;
  x = x + 1;
}
```

Figure 23.1: Example input code.

is illustrated in Figure 23.4.

23.1.1 Def-use Example implementation

Fig. 23.2 shows an example program of how the def-use analysis is called. It generates a dot graph showing def/use information within a control flow graph. It also outputs reaching definition information for each variable references of an input code. This program (named as *defuseAnalysis*) is installed under *ROSE_INST/bin* as a standalone tool for users to experiment the def/use analysis of ROSE.

Figure 23.3 shows the screen output of the code(Fig. 23.2) running on the input code(Fig. 23.1). Each variable reference in the input code has at least one reaching definition node. The associated definition statement is also printed out.

```
#include "rose.h"
#include "DefUseAnalysis.h"
#include <string>
#include <iostream>
using namespace std;

int main( int argc, char * argv[] )
{
  vector<string> argvList(argv, argv + argc);
  SgProject* project = frontend(argvList);

  // Call the Def-Use Analysis
  DFAnalysis* defuse = new DefUseAnalysis(project);
  bool debug = false;
  defuse->run(debug);
  // Output def use analysis results into a dot file
  defuse->dfaToDOT();

  // Find all variable references
  NodeQuerySynthesizedAttributeType vars = NodeQuery::querySubTree(project, V_SgVarRefExp);
  NodeQuerySynthesizedAttributeType::const_iterator i = vars.begin();
  for (; i!=vars.end();++i)
  {
    SgVarRefExp * varRef = isSgVarRefExp(*i);
    SgInitializedName* initName = isSgInitializedName(varRef->get_symbol()->get_declaration());
    std::string name = initName->get_qualified_name().str();
    // Find reaching definition of initName at the control flow node varRef
    vector<SgNode* > vec = defuse->getDefFor(varRef, initName);
    ROSE_ASSERT (vec.size() >0 ); // each variable reference must have a definition somewhere

    // Output each reaching definition node and the corresponding statement.
    std::cout<<"---------------------------------------------"<<std::endl;
    std::cout << vec.size() << " definition entry/entries for " << varRef->unparseToString() <<
    " @ line " << varRef->get_file_info()->get_line()<<":"<<varRef->get_file_info()->get_col()
   << std::endl;
    for (size_t j =0; j<vec.size(); j++)
    {
      cout<<vec[j]->class_name()<<" "<<vec[j]<<endl;
      SgStatement * def_stmt = SageInterface::getEnclosingStatement(vec[j]);
      ROSE_ASSERT(def_stmt);
      cout<<def_stmt->unparseToString()<<"  @ line "<<def_stmt->get_file_info()->get_line()
        <<":"<<def_stmt->get_file_info()->get_col() <<endl;
    }
  }
  return 0;
}
```

Figure 23.2: Example source code using def use analysis

```
---------------------------------------------
1 definition entry/entries for x @ line 4:5
SgAssignInitializer 0x2f5c660
int x = 9;  @ line 3:3
---------------------------------------------
1 definition entry/entries for x @ line 4:7
SgAssignInitializer 0x2f5c660
int x = 9;  @ line 3:3
```

Figure 23.3: Output of the program

23.1.2 Accessing the Def-Use Results

For each CFN in the CFG, the definition and usage for variable references can be determined with the public function calls:

```
vector <SgNode*> getDefFor(SgNode*, SgInitializedName*)
vector <SgNode*> getUseFor(SgNode*, SgInitializedName*)
```

where SgNode* represents any control flow node and SgInitializedName any variable (being used or defined at that CFN). The result is a vector of possible CFN's that either define (getDefFor) or use (getUseFor) a specific variable.

Figure 23.4 shows how the variable x is being declared and defined in CFN's between node 1 and 6. Note that the definition is annotated along the edge. For instance at node 6, the edge reads *(6) DEF: x (3) = 5*. This means that variable x was declared at CFN 3 but defined at CFN 5.

The second statement x=x+1 is represented by CFN's from 7 to 12. One can see in the figure that x is being re-defined at CFN 11. However, the definition of x within the same statement happens at CFN 8. Hence, the definition of the right hand side x in the statement is at CFN 5 : *(8) DEF: x (3) = 5*.

Another usage of the def-use analysis is to determine which variables actually are defined at each CFN. The following function allows to query a CFN for all its variables (SgInitializedNames) and the positions those variables are defined (SgNode):

```
std::multimap <SgInitializedName*, SgNode*> getDefMultiMapFor(SgNode*)
std::multimap <SgInitializedName*, SgNode*> getUseMultiMapFor(SgNode*)
```

All public functions are described in *DefuseAnalysis.h*. To use the def-use analysis, one needs to create an object of the class DefUseAnalysis and execute the run function. After that, the described functions above help to evaluate definition and usage for each CFN.

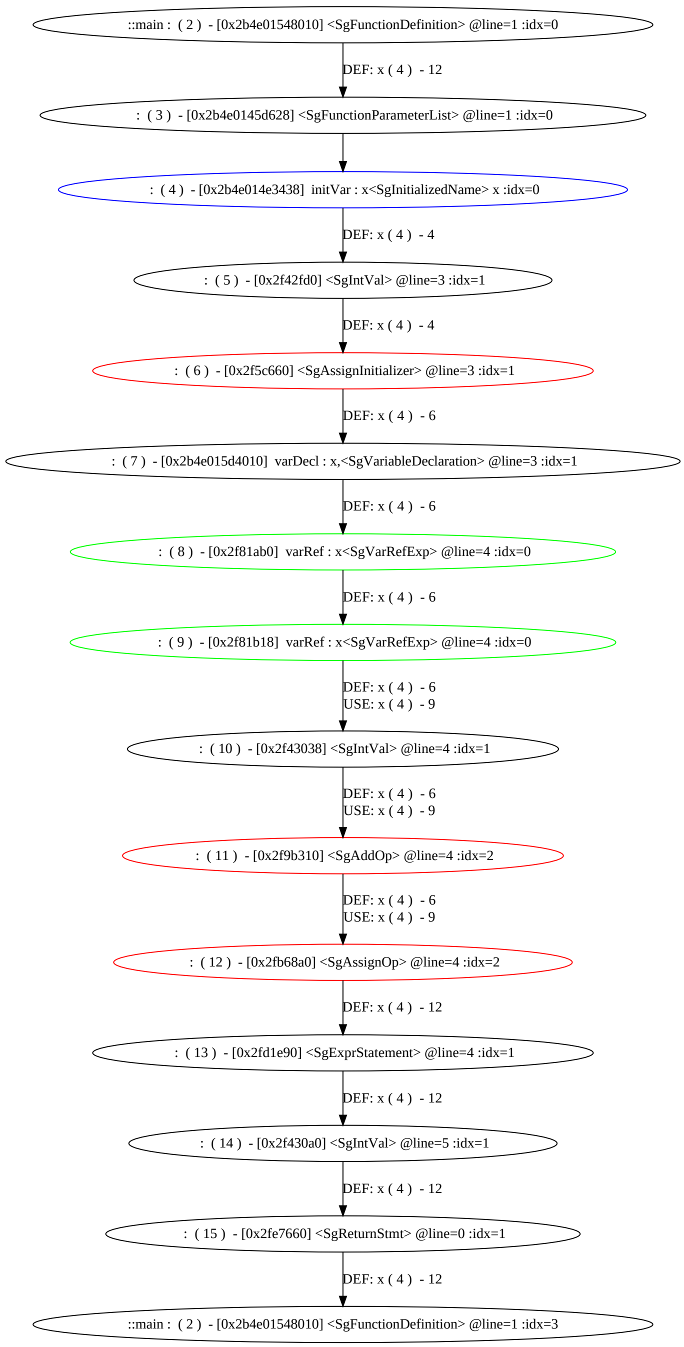


Figure 23.4: Def-Use graph for example program.

23.2 Liveness Analysis

Liveness analysis is a classic data flow analysis performed by compilers to calculate for each program point the variables that may be potentially read before their next write (re-definition). A variable is live at a point in a program's execution path if it holds a value that may be needed in the future.

Fig. 23.5 shows an example program of how the liveness analysis is called in a ROSE-based translator. It generates a dot graph showing def/use information within a control flow graph, alone with live-in and live-out variables. This program (named as *livenessAnalysis*) is installed under *ROSE_INST/bin* as a standalone tool for users to experiment the liveness analysis of ROSE.

Figure 23.7 shows control flow graph with live variable information for the code(Fig. 23.5) running on the input code(Fig. 23.6).

23.2.1 Access live variables

After calling liveness analysis, one can access live-in and live-out variables from a translator based on the virtual control flow graph node. Figure 23.8 shows an example function which retrieves the live-in and live-out variables for a for loop. The code accesses the CFG node (showing in Figure 23.7) of a for statement and retrieve live-in variables of the true edge's target node as the live-in variables of the loop body. Similarly, the live-out variables of the loop are obtained by getting the live-in variables of the node right after the loop (target node of the false edge).

```
#include "rose.h"
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char * argv[] )
{
  vector<string> argvList(argv, argv + argc);
  SgProject* project = frontend(argvList);
  if (project->get_fileList().size() ==0)
    return 0;
  // Prepare the Def-Use Analysis
  DFAnalysis* defuse = new DefUseAnalysis(project);
  bool debug = false;
  defuse->run(debug);
  if (debug)
    defuse->dfaToDOT();

  // Prepare liveness analysis
  LivenessAnalysis* liv = new LivenessAnalysis(debug,(DefUseAnalysis*)defuse);
  ROSE_ASSERT (liv != NULL);

  // Find all function definitions
  Rose_STL_Container<SgNode*> nodeList= NodeQuery::querySubTree(project, V_SgFunctionDefinition);
  std::vector <FilteredCFGNode < IsDFAFilter > > dfaFunctions;
  Rose_STL_Container<SgNode*>::const_iterator i = nodeList.begin();
  bool abortme=false;
  for (; i!=nodeList.end();++i)
  {
    SgFunctionDefinition* func = isSgFunctionDefinition(*i);
    // run liveness analysis
    FilteredCFGNode <IsDFAFilter> rem_source = liv->run(func,abortme);
    if (abortme) {
      cerr<<"Error: Liveness analysis is ABORTING ." << endl;
      ROSE_ASSERT(false);
    }
    if (rem_source.getNode()!=NULL)
      dfaFunctions.push_back(rem_source);
  }

  SgFilePtrList file_list = project->get_fileList();
  std::string firstFileName = StringUtility::stripPathFromFileName(file_list[0]->getFileName());
  std::string fileName = firstFileName+"_liveness.dot" ;
  std::ofstream fs(fileName.c_str());
  dfaToDot(fs, string("var"), dfaFunctions, (DefUseAnalysis*)defuse, liv);
  fs.close();
  return 0;
}
```

Figure 23.5: Example source code using liveness analysis

```
int a[100];

void foo2()
{
  int i;
  int tmp;
  tmp = 10;
  for (i=0;i<100;i++)
  {
    a[i] = tmp;
    tmp =a[i]+i;
  }
   a[0] = 1 ;
}
```

Figure 23.6: Example input code.

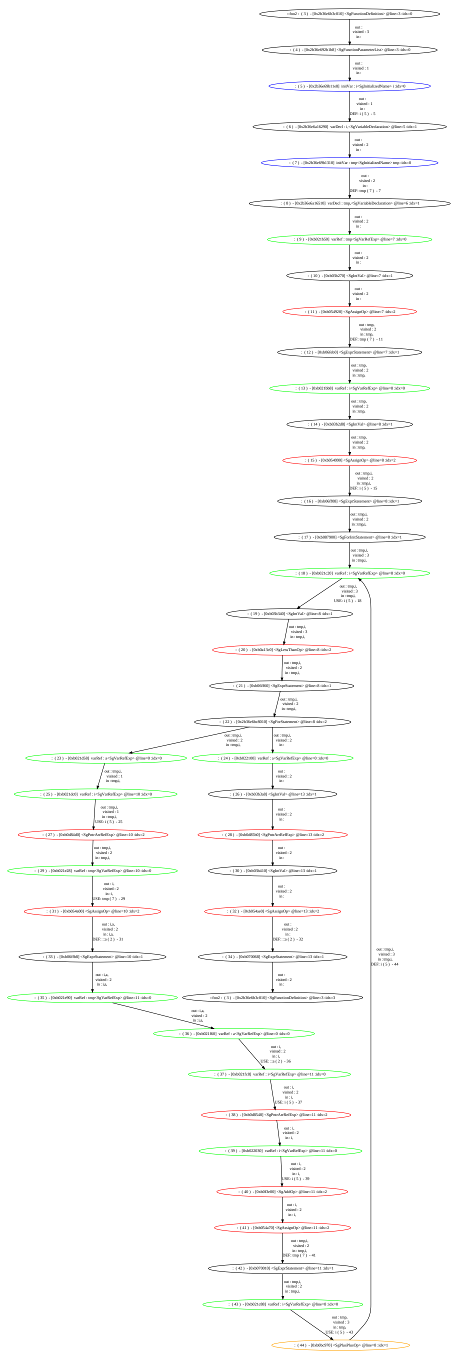


Figure 23.7: Control flow graph annotated with live variables for example program.

```
void getLiveVariables(LivenessAnalysis * liv, SgForStatement* forstmt)
{
  ROSE_ASSERT(liv != NULL);
  ROSE_ASSERT(forstmt!= NULL);
  std::vector<SgInitializedName*> liveIns, liveOuts;

  // For SgForStatement, virtual CFG node which is interesting has an index number of 2,
  // as shown in its dot graph's node caption.
  // "<SgForStatement> @ 8: 2" means this node is for a for statement at source line 8, with an index 2.
  CFGNode cfgnode(forstmt,2);
  FilteredCFGNode<IsDFAFilter> filternode= FilteredCFGNode<IsDFAFilter> (cfgnode);

  // Check edges
  vector<FilteredCFGEdge < IsDFAFilter > > out_edges = filternode.outEdges();
  ROSE_ASSERT(out_edges.size()==2);
  vector<FilteredCFGEdge < IsDFAFilter > >::iterator iter= out_edges.begin();

  for (; iter!=out_edges.end();iter++)
  {
    FilteredCFGEdge < IsDFAFilter > edge= *iter;
    // one true edge going into the loop body
    //x. Live-in (loop) = live-in (first-stmt-in-loop)
    if (edge.condition()==eckTrue)
    {
      SgNode* firstnode= edge.target().getNode();
      liveIns = liv->getIn(firstnode);
    }
    // one false edge going out of loop
    //x. live-out(loop) = live-in (first-stmt-after-loop)
    else if (edge.condition()==eckFalse)
    {
      SgNode* firstnode= edge.target().getNode();
      liveOuts0 = liv->getIn(firstnode);
    }
    else
    {
      cerr<<"Unexpected CFG out edge type for SgForStmt!"<<endl;
      ROSE_ASSERT(false);
    }
  } // end for (edges)
}
```

Figure 23.8: Example code retrieving live variables based on virtual control flow graph

Chapter 24

Generating the Call Graph (CG)

The formal definition of a call graph is:

'A diagram that identifies the modules in a system or computer program and shows which modules call one another.' IEEE

A call graph shows all function call paths of an arbitrary code. These paths are found by following all function calls in a function, where a function in the graph is represented by a node and each possible function call by an edge (arrow). To make a call graph this process is redone for every called function until all edges are followed and there are no ungraphed functions. ROSE has an in-build mechanism for generating call graphs.

ROSE provides support for generating call graphs, as defined in *src/midend/programAnalysis/CallGraphAnalysis/CallGraph.h*. Figure 24 shows the code required to generate the call graph for each function of an application. Using the input code shown in figure 24 the first function's call graph is shown in figure 24.3. A standalone tool named *buildCallGraph* is installed under *ROSE_INSTALL/bin* so users can use it to generate call graphs in dot format.

```
#include "rose.h"
#include <CallGraph.h>
#include <iostream>
using namespace std;
using namespace StringUtility;

// A Function object used as a predicate that determines which functions are
// to be represented in the call graph.
struct keepFunction : public unary_function<bool,SgFunctionDeclaration*>
{
  bool operator()(SgFunctionDeclaration* funcDecl)
  {
    bool returnValue = true;
    ROSE_ASSERT(funcDecl != NULL);
    string filename = funcDecl->get_file_info()->get_filename();
    std::string func_name = funcDecl->get_name().getString();
    string stripped_file_name = stripPathFromFileName(filename);
    //string::size_type loc;

    //Filter out functions from the ROSE preinclude header file
    if(filename.find("rose_edg_required_macros_and_functions")!=string::npos)
      returnValue = false;
    //Filter out compiler generated functions
    else if(funcDecl->get_file_info()->isCompilerGenerated()==true)
      returnValue=false;
    //Filter out compiler generated functions
    else if(funcDecl->get_file_info()->isFrontendSpecific()==true)
      returnValue=false;
    // filter out other built in functions
    //      else if( func_name.find ("__",0)== 0);
    //         returnValue = false;
    // _IO_getc _IO_putc _IO_feof, etc.
    //loc = func_name.find ("_IO_",0);
    //if (loc == 0 ) returnValue = false;

    // skip functions from standard system headers
    // TODO Need more rigid check
    else  if (
              stripped_file_name==string("stdio.h")||
              stripped_file_name==string("libio.h")||
              stripped_file_name==string("math.h")||
              stripped_file_name==string("time.h")||
              stripped_file_name==string("select.h")||
              stripped_file_name==string("mathcalls.h")
          )
      returnValue=false;
    if (returnValue)
      cout<<"Debug:"<<func_name << " from file:"<<stripped_file_name<<" Keep: "<<returnValue<<endl;
    return returnValue;
  }
};
int main( int argc, char * argv[] )
{
  SgProject* project = new SgProject(argc, argv);
  ROSE_ASSERT (project != NULL);

  if (project->get_fileList().size() >=1)
  {
    //Construct a call Graph
    CallGraphBuilder CGBuilder(project);
//     CGBuilder.buildCallGraph(keepFunction());
    CGBuilder.buildCallGraph(builtinFilter());

    // Output to a dot file
    AstDOTGeneration dotgen;
    SgFilePtrList file_list = project->get_fileList();
    std::string firstFileName = StringUtility::stripPathFromFileName(file_list[0]->getFileName());
    dotgen.writeIncidenceGraphToDOTFile( CGBuilder.getGraph(), firstFileName+"_callGraph.dot");
  }

  return 0; // backend(project);
}
```

Figure 24.1: Example source code showing visualization of call graph.

```
// simple (trivial) example code used to demonstrate the call graph generation
class A
{
public:
        int f1() { return 0;}
        int f2() { pf = &A::f1; return (this->*pf)(); }
        int (A::*pf) ();
};

void foo1();
void foo2();
void foo3();
void foo4();

void foo1()
   {
     foo1();
     foo2();
     foo3();
     foo4();
   }

void foo2()
   {
     foo1();
     foo2();
     foo3();
     foo4();
   }

void foo3()
   {
     foo1();
     foo2();
     foo3();
     foo4();
   }

void foo4()
   {
     foo1();
     foo2();
     foo3();
     foo4();
   }

int main()
   {
     foo1();
     foo2();
     foo3();
     foo4();

     return 0;
   }
```

Figure 24.2: Example source code used as input to build call graph.

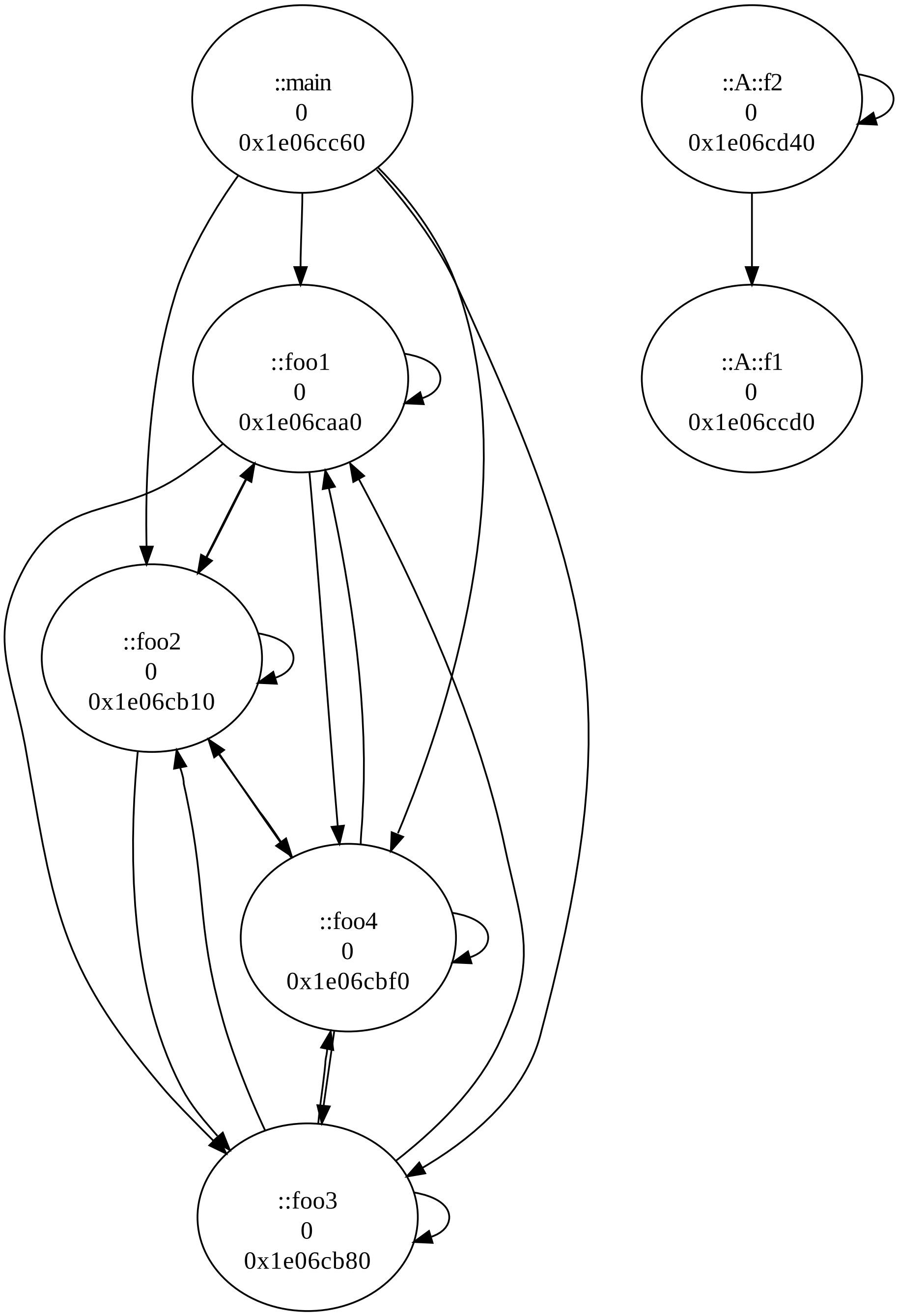


Figure 24.3: Call graph for function in input code file: inputCode_BuildCG.C.

Chapter 25

Dataflow Analysis based Virtual Function Analysis

C++ Virtual function provides polymorphism to the developer but makes it difficult for compilers to do optimizations. Virtual functions are usually resolved at runtime from the vtable. It's very difficult for a compiler to know which functions will be called at compile time. ROSE provides a flow sensitive dataflow analysis based approach to cut down the set of possible function calls. The code for Virtual Function Analysis is located in *src/midend/programAnalysis/VirtualFunctionAnalysis/VirtualFunctionAnalysis.h*. It also provides a mechanism to resolve any function calls. It's a whole program analysis and supposed to be expensive. It memorizes all the resolved function calls for any call site, so that subsequent calls are resolved faster.

Figure 25.1 shows the code required to generate the pruned call graph. Using the input code shown in figure 25 Call Graph Analysis generates call graph shown in figure 25.3. Executing dataflow analysis to resolve virtual function calls resulted in the figure 25.4.

```
#include "sage3basic.h"

#include <string>
#include <iostream>

#include "VirtualFunctionAnalysis.h"
using namespace boost;


using namespace std;
using namespace boost;
// A Function object used as a predicate that determines which functions are
// to be represented in the call graph.
struct keepFunction : public unary_function<bool,SgFunctionDeclaration*>{
  public:
    bool operator()(SgFunctionDeclaration* funcDecl){
      bool returnValue = true;
      ROSE_ASSERT(funcDecl != NULL);
      string filename = funcDecl->get_file_info()->get_filename();

      //Filter out functions from the ROSE preinclude header file
      if(filename.find("rose_edg_required_macros_and_functions")!=string::npos)
        returnValue = false;

      //Filter out compiler generated functions
      if(funcDecl->get_file_info()->isCompilerGenerated()==true)
        returnValue=false;

      return returnValue;
    }
};

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

    SgProject* project = frontend(argc, argv);
    ROSE_ASSERT(project != NULL);

    CallGraphBuilder builder(project);
    builder.buildCallGraph(keepFunction());
    // Generate call graph in dot format
    AstDOTGeneration dotgen;
    dotgen.writeIncidenceGraphToDOTFile(builder.getGraph(), "original_call_graph.dot");

    SgFunctionDeclaration *mainDecl = SageInterface::findMain(project);
    if(mainDecl == NULL) {
            std::cerr<< "Can't execute Virtual Function Analysis without main function\n";
            return 0;
     }

    VirtualFunctionAnalysis *anal = new VirtualFunctionAnalysis(project);
    anal->run();

    anal->pruneCallGraph(builder);

     AstDOTGeneration dotgen2;
     dotgen2.writeIncidenceGraphToDOTFile(builder.getGraph(), "pruned_call_graph.dot");

    delete anal;


    return 0;
}
```

Figure 25.1: Source code to perform virtual function analysis

```
class Animal {
 public:
   virtual void shout() {}
};

class dog : public Animal{
  public:
  virtual void shout () {}
};
class terrier : public dog {
  public:
  virtual void shout () {}
};
class yterrier : public terrier{
  public:
  virtual void shout () {}
};

int main(void) {

Animal **p, **q;
dog *x, d;
terrier *y;

y = new yterrier;
x = &d;
p = (Animal **)&x;
q = p;
*p = y;

x->shout();

return 0;

}
```

Figure 25.2: Example source code used as input for Virtual Function Analysis.

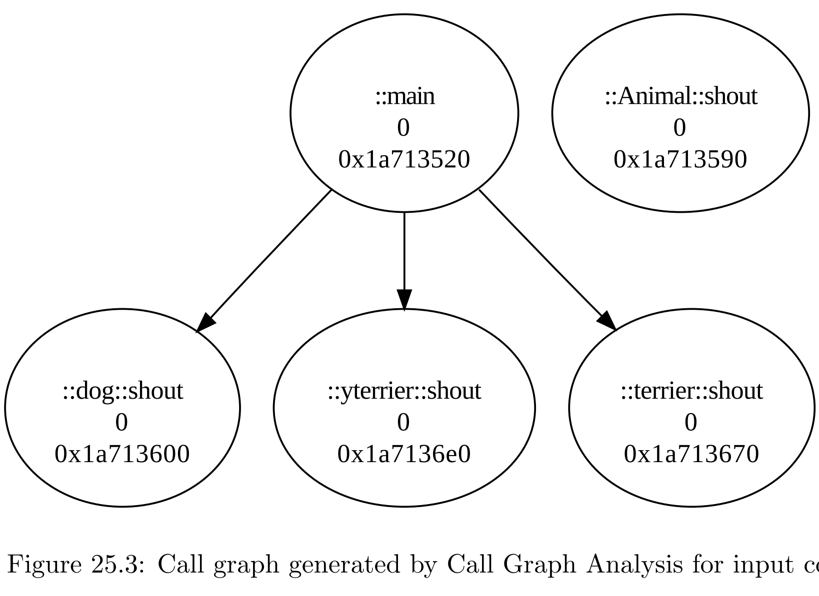


Figure 25.3: Call graph generated by Call Graph Analysis for input code in inputCode_vfa.C.

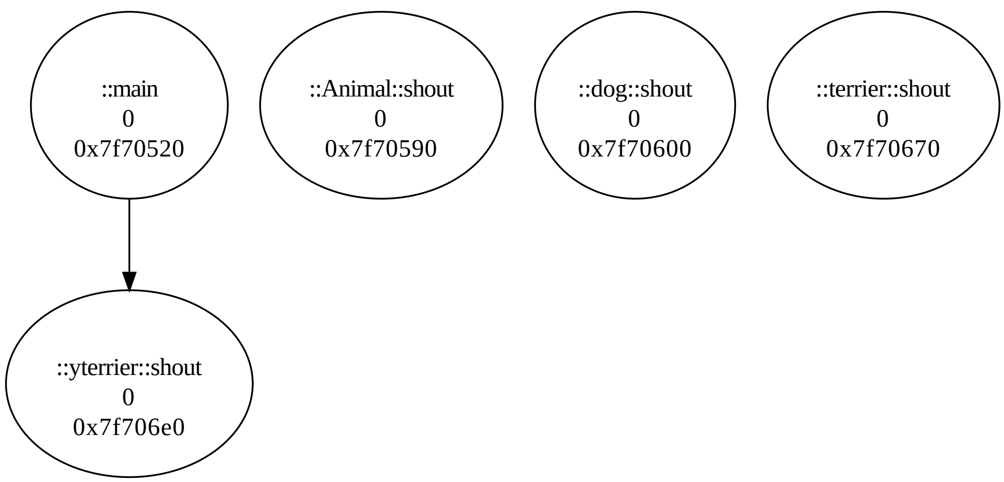


Figure 25.4: Call graph resulted from Virtual Function Analysis for input code in inputCode_vfa.C.

Chapter 26

Generating the Class Hierarchy Graph

```
#include "rose.h"
#include "CallGraph.h"
#include <boost/foreach.hpp>

#define foreach BOOST_FOREACH

using namespace std;

int main(int argc, char * argv[])
{
        SgProject* project = new SgProject(argc, argv);

        //Construct class hierarchy graph
        ClassHierarchyWrapper hier(project);

        //Display the ancestors of each class
        vector<SgClassDefinition*> allClasses = SageInterface::querySubTree<SgClassDefinition>(project, V_SgClassDefinitio
        foreach(SgClassDefinition* classDef, allClasses)
        {
                printf("\n%s subclasses: ", classDef->get_declaration()->get_name().str());
                foreach(SgClassDefinition* subclass, hier.getSubclasses(classDef))
                {
                        printf("%s, ", subclass->get_declaration()->get_name().str());
                }
        }

        return 0;
}
```

Figure 26.1: Example source code showing visualization of class hierarchy graph.

For C++, because of multiple inheritance, a class hierarchy graph is a directed graph with pointers from a class to a superclass. A superclass is a class which does not inherit from any other class. A class may inherit from a superclass by inheriting from another class which does rather than by a direct inheritance.

Figure 26 shows the code required to generate the class hierarchy graph for each class of an application. Using the input code shown in figure 26 the first function's call graph is shown in figure 26.3.

```
class A{};

class B : public A{};

class C : public B{};
```

Figure 26.2: Example source code used as input to build class hierarchy graph.

Figure 26.3 shows the class hierarchy graph for the classes in the input code in figure 26.

Figure 26.3: Class hierarchy graph in input code file: inputCode_ClassHierarchyGraph.C.

Chapter 27

Database Support

This chapter is specific to support in ROSE for persistent storage. ROSE uses the SQLite database and makes it simple to store data in the database for retrieval in later phases of processing large multiple file projects.

FIXME: *Need more information here.*

27.1 ROSE DB Support for Persistent Analysis

This section presents figure 27.3, a simple C++ source code using a template. It is used as a basis for showing how template instantiations are handled within ROSE. An example translator using a database connection to store function information is shown in Fig.27.1 and Fig.27.2. The output by the translator operating on the C++ source code is shown in Fig. 27.4.

27.2 Call Graph for Multi-file Application

This section shows an example of the use of the ROSE Database mechanism where information is stored after processing each file as part of generating the call graph for a project consisting of multiple files. The separate files are show in figures 27.3 and **??**. These files are processed using the translator in figure **??** to generate the final project call graph shown in figure **??**.

FIXME: *This example still needs to be implemented to use the new ROSE call graph generator.*

27.3 Class Hierarchy Graph

This section presents a translator in figure **??**, to generate the class hierarchy graph of the example shown in figure **??**. The input is a multi-file application show in figure **??** and figure **??**. *This example is incomplete.*

FIXME: *This example is still incomplete.*

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

#include "rose.h"

using namespace std;

// DQ (9/9/2005): Don't include the database by default
// TPS (01Dec2008): Enabled mysql and this fails.
// seems like it is not supposed to be included
#if 0
//#ifdef HAVE_MYSQL
   #include "GlobalDatabaseConnectionMYSQL.h"
#endif

int main( int argc, char * argv[] )
   {
// TPS (01Dec2008): Enabled mysql and this fails.
// seems like it is not supposed to be included
#if 0
     //#ifdef HAVE_MYSQL
  // Build the Data base
     GlobalDatabaseConnection *gDB;
     gDB = new GlobalDatabaseConnection( "functionNameDataBase" );
     gDB->initialize();
     string command = "";
     command = command + "CREATE TABLE Functions ( name TEXT, counter );";

     Query *q = gDB->getQuery();
     q->set( command );
     q->execute();

     if ( q->success() != 0 )
          cout << "Error creating schema: " << q->error() << "\n";
  // Alternative syntax, but does not permit access to error messages and exit codes
  // gDB->execute(command.c_str());
#endif

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

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

  // Build a list of functions within the AST
     Rose_STL_Container<SgNode*> functionDeclarationList =
          NodeQuery::querySubTree (project,V_SgFunctionDeclaration);

     int counter = 0;
     for (Rose_STL_Container<SgNode*>::iterator i = functionDeclarationList.begin();
                i != functionDeclarationList.end(); i++)
        {
       // Build a pointer to the current type so that we can call
       // the get_name() member function.
          SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(*i);
          ROSE_ASSERT(functionDeclaration != NULL);

          SgName func_name = functionDeclaration->get_name();
      // Skip builtin functions for shorter output, Liao 4/28/2008
          if (func_name.getString().find("__builtin",0)==0)
            continue;

       // output the function number and the name of the function
          printf ("function name #%d is %s at line %d \n",
               counter++,func_name.str(),
```

Figure 27.1: Example translator (part 1) using database connection to store function names.

```
                 functionDeclaration->get_file_info()->get_line());

            string functionName = functionDeclaration->get_qualified_name().str();

// TPS (01Dec2008): Enabled mysql and this fails.
// seems like it is not supposed to be included
#if 0
            //#ifdef HAVE_MYSQL
            command = "INSERT INTO Functions values(\"" + functionName + "\"," +
                    StringUtility::numberToString(counter) + ");";
         // Alternative interface
         // q->set( command );
         // cout << "Executing: " << q->preview() << "\n";
         // q->execute();
            gDB->execute(command.c_str());
#endif
          }

// TPS (01Dec2008): Enabled mysql and this fails.
// seems like it is not supposed to be included
#if 0
      //#ifdef HAVE_MYSQL
      command = "SELECT * from Functions;";

   // Alternative Interface (using query objects)
   // q << command;
      q->set(command);
      cout << "Executing: " << q->preview() << "\n";

   // execute and return result (alternative usage: "gDB->select()")
      Result *res = q->store();
      if ( q->success() != 0 )
           cout << "Error reading values: " << q->error() << "\n";
      else
         {
      // Read the table returned from the query
      // res->showResult();
         for ( Result::iterator i = res->begin(); i != res->end(); i++ )
            {
           // Alternative syntax is possible: "Row r = *i;"
              string functionName = (*i)[0].get_string();
              int counter = (*i)[1];
              printf ("functionName = %s counter = %d \n",functionName.c_str(),counter);
            }
         }

      gDB->shutdown();
#else
      printf ("Program compiled without data base connection support (add using ROSE configure option) \n");
#endif

      return 0;
    }
```

Figure 27.2: Example translator (part 2) using database connection to store function names.

```
// This example code is used to record names of functions into the data base.

class A
   {
     public:
          virtual int f1() = 0;
          virtual int f2() {}
          int f3();
          virtual int f4();
   };

int A::f3() { f1(); return f3();}
int A::f4() {}

class B : public A
   {
     public:
          virtual int f1();
          virtual int f2() {}
   };

int B::f1() {}

class C : public A
   {
     public:
          virtual int f1() {}
          int f3() {}
   };

class D : public B
   {
     public:
          virtual int f2() {}
   };

class E : public D
   {
     public:
          virtual int f1() { return 5; }
   };

class G : public E
   {
     public:
          virtual int f1();
   };

int G::f1() {}

class F : public D
   {
     public:
          virtual int f1() {}
          virtual int f2() {return 5;}
          int f3() {return 2;}
   };

class H : public C
   {
     public:
          virtual int f1() {}
          virtual int f2() {}
          int f3() {}
   };
```

Figure 27.3: Example source code used as input to database example.

```
function name #0 is __sync_lock_test_and_set at line 0
function name #1 is __sync_lock_release at line 0
function name #2 is f1 at line 6
function name #3 is f2 at line 7
function name #4 is f3 at line 8
function name #5 is f4 at line 9
function name #6 is f3 at line 12
function name #7 is f4 at line 13
function name #8 is f1 at line 18
function name #9 is f2 at line 19
function name #10 is f1 at line 22
function name #11 is f1 at line 27
function name #12 is f3 at line 28
function name #13 is f2 at line 34
function name #14 is f1 at line 40
function name #15 is f1 at line 46
function name #16 is f1 at line 49
function name #17 is f1 at line 54
function name #18 is f2 at line 55
function name #19 is f3 at line 56
function name #20 is f1 at line 62
function name #21 is f2 at line 63
function name #22 is f3 at line 64
Program compiled without data base connection support (add using ROSE configure option)
```

Figure 27.4: Output from processing input code through database example dataBaseTranslator27.1.

Chapter 28

Building Custom Graphs

What To Learn From This Example This example shows how to generate custom graphs using *SgGraph* class.

Rose provides a collection type *SgGraph* to store a graph. Two specific graphs are also provided which are derived from *SgGraph*: *SgIncidenceDirectedGraph* and *SgIncidenceUndirectedGraph*.

Nodes and edges in a *SgGraph* are represented by *SgGraphNode* and *SgGraphEdge* separately. A *SgGraph* is built by adding *SgGraphNode*s and *SgGraphEdge*s using its member function *addNode* and *addEdge*. You can get all nodes and edges of a *SgGraph* by calling its functions *computeNodeSet* and *computeEdgeSet* separately. More interfaces of *SgGraph* and its subclasses can be found in doxygen of Rose.

Since *SgGraph* is for Rose use, each node in it holds a pointer to *SgNode*, which is the default attribute of a *SgGraphNode*. If you want to add more attributes inside, you can use *SgGraphNode*'s member function *addNewAttribute* by providing a name and an *AstAttribute* object to add a new attribute to a node. Normally, you have to build your own attribute class which should be derived from class *AstAttribute*. Three attribute classes are provided by Rose: *AstRegExAttribute*, *AstTextAttribute*, and *MetricAttribute*. For more information about them, please refer to Rose's doxygen.

Part IV

Program Transformations and Optimizations

This part gives examples of building source-to-source program transformations and optimizations.

Chapter 29

Generating Unique Names for Declarations

There are many instances where a unique name must be generated for either a function or variable declaration. ROSE defines a mechanism to make the generation of unique names from all SgDeclarationStatment IR nodes and the SgInitializedName IR node. This simplifies ROSE-based applications that require this sort of mechanism. Our experience has found that a significant number of tools require such a mechanism and that its correct implementation can have subtle points.

The specific translator described in this chapter traverses an AST and outputs the unique names that can be generated for each declaration showing the use of the unique name generation mechanism. This tool is intended as an example of how to generate unique names using ROSE. Not all IR nodes can be used to generate a unique name. The generated names are unique under the following rules:

1. Any two generated names are the same if the declarations are the same.
   Declaration can be the same across files or within the same file. Declarations that are the same can have different location in the same file (be represented multiple times) or be in different files. Language constructs that are the same must follow the One-time Definition Rule (ODR) across files.

2. Declarations in different unnamed scopes (e.g. for loop bodies) will generate different names.

3. Names are the same when generated by different ROSE tools.
   Pointer values could be used to generate unique names of all IR nodes, but this would work only within a single invocation of the ROSE based tool. Generated names are not based on internal pointer values and are thus insensitive to pointer values. Generated names of the same declaration are thus the same even if generated from different tools. This allows multiple ROSE tools to inter-operate.

This unique name generation mechanism is only applicable to specific IR nodes, specifically:

- SgInitializedName

- SgDeclarationStatement IR nodes:
  - Obvious IR nodes supported:
    * SgClassDeclaration
    * SgFunctionDeclaration
    * SgEnumDeclaration
    * SgNamespaceDeclarationStatement
    * SgTypedefDeclaration
  - Less obvious IR nodes not supported (support for these would not make sense):
    * SgAsmStmt
    * SgCtorInitializerList
    * SgFunctionParameterList
    * SgNamespaceAliasDeclarationStatement
    * SgPragmaDeclaration
    * SgTemplateDeclaration (can this have a mangled name?)
    * SgTemplateInstantiationDirectiveStatement
    * SgUsingDeclarationStatement
    * SgUsingDirectiveStatement
    * SgVariableDeclaration
      Note that the SgVariableDeclaration contains a list of SgInitializedName nodes and the mangled names are best queried from each SgInitializedName instead of the SgVariableDeclaration.
    * SgVariableDefinition
- Un-named scopes
  A number of scopes are un-names and so there is an opportunity to generate non-unique names from declarations in such scopes. To fix this we generate names for each un-named scope to guarantee uniqueness. Nodes handled are:
  - SgForStatement
  - SgBasicBlock
  - SgIfStmt
  - get the complete list ...

Other language constructs can generate unique names as well, but their name could be invalid after certain transformation that move it structurally within the generated source code.

29.1 Example Code Showing Generation of Unique Names

29.2 Input For Examples Showing Unique Name Generation for Variables

Figure 29.1, shows an example translator demonstrating the generation of unique names from declarations in the AST. For each SgInitializedName we generate the mangled name. Figure 29.2

shows the input code and figure 29.3 shows the generated output from the translator (the mangled names from the AST associated with the input application).

29.3 Example Output Showing Unique Variable Names

29.4 Input For Examples Showing Unique Name Generation for Functions

Figure 29.1, shows an example translator demonstrating the generation of unique names from declarations in the AST. For each SgInitializedName we generate the mangled name. Figure 29.4 shows the input code and figure 29.5 shows the generated output from the translator (the mangled names from the AST associated with the input application).

29.5 Example Output Showing Unique Function Names

```
// This example shows the generation of unique names from declarations.

// Mangled name demo
//
// This translator queries the AST for all SgInitializedNames and
// SgFunctionDeclarations, and for each one prints (a) the source
// location, (b) the source name of the object, and (c) the mangled
// name.

#include <rose.h>

using namespace std;

// Returns a Sg_File_Info object as a display-friendly string, "[source:line]".
static string toString (const Sg_File_Info* info)
   {
     ostringstream info_str;
     if (info)
     info_str << '['
              << info->get_raw_filename ()
              << ":" << info->get_raw_line ()
              << ']';
     return info_str.str ();
   }

// Displays location and mangled name of an SgInitializedName object.
static void printInitializedName (const SgNode* node)
   {
     const SgInitializedName* name = isSgInitializedName (node);
     ROSE_ASSERT (name != NULL);

     if (name->get_file_info()->isCompilerGenerated() == false)
          cout // << toString (name->get_file_info ())
               // << " "
               << name->get_name ().str ()
               << " --> " << name->get_mangled_name ().str ()
               << endl;
   }

// Displays location and mangled name of an SgFunctionDeclaration object.
static void printFunctionDeclaration (const SgNode* node)
   {
     const SgFunctionDeclaration* decl = isSgFunctionDeclaration (node);
     ROSE_ASSERT (decl != NULL);

     if (decl->get_file_info()->isCompilerGenerated() == false)
          cout // << toString (decl->get_startOfConstruct ())
               // << " "
               << decl->get_qualified_name ().str ()
               << " --> " << decl->get_mangled_name ().str ()
               << endl;
   }

int main (int argc, char** argv)
   {
     SgProject* proj = frontend (argc, argv);

     cout << endl << "***** BEGIN initialized names *****" << endl;
     Rose_STL_Container<SgNode *> init_names = NodeQuery::querySubTree (proj, V_SgInitializedName);
     for_each (init_names.begin (), init_names.end (), printInitializedName);
     cout << "***** END initialized names *****" << endl;

     cout << endl << "***** BEGIN function declarations *****" << endl;
     Rose_STL_Container<SgNode *> func_decls = NodeQuery::querySubTree (proj, V_SgFunctionDeclaration);
     for_each (func_decls.begin (), func_decls.end (), printFunctionDeclaration);
     cout << "***** END function declarations *****" << endl;

     return backend (proj);
   }
```

Figure 29.1: Example source code showing the output of mangled name. The string represents the code associated with the subtree of the target IR node.

```
// Input file to test mangling of SgInitializedName objects.

int x;

// Global class
class A
   {
     private:
          int x;
       // Nested class
          class B
             {
               private:
                    int x;
               public:
                    void foo (int x_arg) { int x; }
             };
   };

template <typename T>
void
foo (T x_arg)
   {
     T x;
     for (x = 0; x < 10; x++)
        {
          T x = 0;
          do {
            // Local class
               class A
                  {
                    private:
                      // Nested class
                         class B
                            {
                              T x;
                            };
                    public:
                         void foo (T x) {}
                  };
               T x = 0;
             }
          while (x > 0);

          do {
               T x = 0;
             }
          while (x > 0);

       // Nested scope
             {
               T x = 0;
             }
        }
   }

template void foo<int> (int x);
template void foo<double> (double x);

void bar (void)
   {
     for (int x = 0; x != 0; x++)
          for (int x = 0; x != 0; x++)
               for (long x = 0; x != 0; x++)
                    ;
     try {
       for (int x = 0; x != 0; x++) ;
     }
     catch (int) {}
     catch (char x) {}
   }
```

Figure 29.2: Example source code used as input to program in codes showing debugging techniques shown in this section.

```

***** BEGIN initialized names *****
x --> x
x --> A__scope__x
x --> A__scope__B__scope__x
x_arg --> L2R__L3R__ARG1
x --> L2R__L3R__scope____SgSS2___scope__x
x_arg --> foo__tas__i__tae____Fb_v_Gb_i_Fe___ARG1
x_arg --> foo__tas__i__tae____Fb_v_Gb_i_Fe___ARG1
x_arg --> foo__tas__d__tae____Fb_v_Gb_d_Fe___ARG1
x_arg --> foo__tas__d__tae____Fb_v_Gb_d_Fe___ARG1
x --> bar___Fb_v_Gb__Fe___L4R__scope____SgSS2___scope____SgSS3___scope__x
x --> bar___Fb_v_Gb__Fe___L4R__scope____SgSS2___scope____SgSS3___scope____SgSS4___scope__x
x --> bar___Fb_v_Gb__Fe___L4R__scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS5___scope__x
x --> bar___Fb_v_Gb__Fe___L4R__scope____SgSS2___scope____SgSS6___scope____SgSS7___scope__x
 --> bar___Fb_v_Gb__Fe___L4R__scope____SgSS2___scope____SgSS8___scope__CATCHARG
x --> bar___Fb_v_Gb__Fe___L4R__scope____SgSS2___scope____SgSS10___scope__x
x_arg --> foo__tas__i__tae____Fb_v_Gb_i_Fe___ARG1
x --> foo__tas__i__tae____Fb_v_Gb_i_Fe___scope____SgSS2___scope__x
x --> foo__tas__i__tae____Fb_v_Gb_i_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope__x
x --> L0R__scope__B__scope__x
x --> L5R__L6R__ARG1
x --> foo__tas__i__tae____Fb_v_Gb_i_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS5___scope____SgSS6.
x --> foo__tas__i__tae____Fb_v_Gb_i_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS11___scope____SgSS1
x --> foo__tas__i__tae____Fb_v_Gb_i_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS13___scope__x
x_arg --> foo__tas__d__tae____Fb_v_Gb_d_Fe___ARG1
x --> foo__tas__d__tae____Fb_v_Gb_d_Fe___scope____SgSS2___scope__x
x --> foo__tas__d__tae____Fb_v_Gb_d_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope__x
x --> L1R__scope__B__scope__x
x --> L7R__L8R__ARG1
x --> foo__tas__d__tae____Fb_v_Gb_d_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS5___scope____SgSS6.
x --> foo__tas__d__tae____Fb_v_Gb_d_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS11___scope____SgSS1
x --> foo__tas__d__tae____Fb_v_Gb_d_Fe___scope____SgSS2___scope____SgSS3___scope____SgSS4___scope____SgSS13___scope__x
***** END initialized names *****

***** BEGIN function declarations *****
::A::B::foo --> L2R__L3R
::foo < int >  --> foo__tas__i__tae____Fb_v_Gb_i_Fe_
::foo < int >  --> foo__tas__i__tae____Fb_v_Gb_i_Fe_
::foo < double >  --> foo__tas__d__tae____Fb_v_Gb_d_Fe_
::foo < double >  --> foo__tas__d__tae____Fb_v_Gb_d_Fe_
::bar --> bar___Fb_v_Gb__Fe___L4R
::foo < int >  --> foo__tas__i__tae____Fb_v_Gb_i_Fe_
A::foo --> L5R__L6R
::foo < double >  --> foo__tas__d__tae____Fb_v_Gb_d_Fe_
A::foo --> L7R__L8R
***** END function declarations *****
```

Figure 29.3: Output of input code using generatingUniqueNamesFromDeclaration.C

```
// Input file to test mangling of SgFunctionDeclaration objects.


long foobar();
long foobar(int);
long foobar(int y);
long foobar(int x);
long foobar(int x = 0);
long foobar(int xyz)
   {
     return xyz;
   }

char foobarChar(char);
char foobarChar(char c);

// Input file to test mangling of SgFunctionDeclaration objects.

typedef int value0_t;
typedef value0_t value_t;
namespace N
   {
     typedef struct { int a; } s_t;
     class A { public: A () {} virtual void foo (int) {} };
     class B { public: B () {} void foo (value_t) const {} };
     class C : public A { public: C () {} void foo (int) {} void foo (const s_t&) {} };
     void foo (const s_t*) {}
   }

typedef N::s_t s2_t;
void foo (value_t);
void foo (s2_t) {}
void foo (float x[]) {}
void foo (value_t, s2_t);

template <typename T>
void foo (T) {}

namespace P
   {
     typedef long double type_t;
     namespace Q
        {
          template <typename T>
          void foo (T) {}

          class R
             {
               public:
                    R () {}
                    template <typename T>
                    void foo (T) {}
                    void foo (P::type_t) {}
                    template <typename T, int x>
                    int foo (T) { return x; }
             };
        }
   }

template <typename T, int x>
int foo (T) { return x; }

template void foo<char> (char);
template void foo<const value_t *> (const value_t *);
template void P::Q::foo<long> (long);
template void P::Q::R::foo<value_t> (value_t);
```

Figure 29.4: Example source code used as input to program in codes showing debugging techniques shown in this section.

```

***** BEGIN initialized names *****
 --> foobar___Fb_l_Gb_i_Fe___L4R__ARG1
y --> foobar___Fb_l_Gb_i_Fe___L4R__ARG1
x --> foobar___Fb_l_Gb_i_Fe___L4R__ARG1
x --> foobar___Fb_l_Gb_i_Fe___L4R__ARG1
xyz --> foobar___Fb_l_Gb_i_Fe___L4R__ARG1
 --> foobarChar___Fb_c_Gb_c_Fe___L5R__ARG1
c --> foobarChar___Fb_c_Gb_c_Fe___L5R__ARG1
a --> L2R__scope__a
 --> L6R__L7R__ARG1
 --> L8R__L9R__ARG1
 --> L10R__L11R__ARG1
 --> L12R__L13R__ARG1
 --> L14R__L15R__ARG1
 --> foo___Fb_v_Gb_L0R_Fe___L16R__ARG1
 --> foo___Fb_v_Gb_L3R_Fe___L17R__ARG1
x --> L18R__L19R__ARG1
 --> foo___Fb_v_Gb_L0R__sep__L3R_Fe___L20R__ARG1
 --> foo___Fb_v_Gb_L0R__sep__L3R_Fe___L20R__ARG2
 --> L21R__L22R__ARG1
 --> foo__tas__c__tae_____Fb_v_Gb_c_Fe___ARG1
 --> foo__tas__c__tae_____Fb_v_Gb_c_Fe___ARG1
 --> L23R__ARG1
 --> L23R__ARG1
 --> L24R__ARG1
 --> L24R__ARG1
 --> L25R__ARG1
 --> L25R__ARG1
 --> foo__tas__c__tae_____Fb_v_Gb_c_Fe___ARG1
 --> L23R__ARG1
 --> L24R__ARG1
 --> L25R__ARG1
***** END initialized names *****

***** BEGIN function declarations *****
::foobar --> foobar___Fb_l_Gb__Fe___L26R
::foobar --> foobar___Fb_l_Gb_i_Fe___L4R
::foobar --> foobar___Fb_l_Gb_i_Fe___L4R
::foobar --> foobar___Fb_l_Gb_i_Fe___L4R
::foobar --> foobar___Fb_l_Gb_i_Fe___L4R
::foobar --> foobar___Fb_l_Gb_i_Fe___L4R
::foobarChar --> foobarChar___Fb_c_Gb_c_Fe___L5R
::foobarChar --> foobarChar___Fb_c_Gb_c_Fe___L5R
::N::A::A --> L27R__L28R
::N::A::foo --> L6R__L7R
::N::B::B --> L29R__L30R
::N::B::foo --> L8R__L9R
::N::C::C --> L31R__L32R
::N::C::foo --> L10R__L11R
::N::C::foo --> L12R__L13R
::N::foo --> L14R__L15R
::foo --> foo___Fb_v_Gb_L0R_Fe___L16R
::foo --> foo___Fb_v_Gb_L3R_Fe___L17R
::foo --> L18R__L19R
::foo --> foo___Fb_v_Gb_L0R__sep__L3R_Fe___L20R
::P::Q::R::R --> L33R__L34R
::P::Q::R::foo --> L21R__L22R
::foo < char >  --> foo__tas__c__tae_____Fb_v_Gb_c_Fe_
::foo < char >  --> foo__tas__c__tae_____Fb_v_Gb_c_Fe_
::foo < const value_t * >  --> L23R
::foo < const value_t * >  --> L23R
::P::Q::foo < long >  --> L24R
::P::Q::foo < long >  --> L24R
::P::Q::R::foo < value_t >  --> L25R
::P::Q::R::foo < value_t >  --> L25R
::foo < char >  --> foo__tas__c__tae_____Fb_v_Gb_c_Fe_
::foo < const value_t * >  --> L23R
::P::Q::foo < long >  --> L24R
::P::Q::R::foo < value_t >  --> L25R
***** END function declarations *****
```

Figure 29.5: Output of input code using generatingUniqueNamesFromDeclaration.C

Chapter 30

Command-line Processing Within Translators

ROSE includes mechanism to simplify the processing of command-line arguments so that translators using ROSE can trivially replace compilers within makefiles. This example shows some of the many command-line handling options within ROSE and the ways in which customized options may be added for specific translators.

30.1 Commandline Selection of Files

Overview This example shows the optional processing of specific files selected after the call to the frontend to build the project. First the SgProject if build and *then* the files are selected for processing via ROSE or the backend compiler directly.

This example demonstrates the separation of the construction of a SgProject with valid SgFile objects for each file on the command line, but with an empty SgGlobal scope, and the call to the frontend, called for each SgFile in a separate loop over all the SgFile objects.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

using namespace std;

int
main ( int argc, char* argv[] )
   {
     Rose_STL_Container<string> l = CommandlineProcessing::generateArgListFromArgcArgv (argc,argv);
     printf ("Preprocessor (before): argv = \n%s \n",StringUtility::listToString(l).c_str());

  // Remove certain sorts of options from the command line
     CommandlineProcessing::removeArgs (l,"-edg:");
     CommandlineProcessing::removeArgs (l,"--edg:");
     CommandlineProcessing::removeArgsWithParameters (l,"-edg_parameter:");
     CommandlineProcessing::removeArgsWithParameters (l,"--edg_parameter:");

  // Add a test for a custom command line option
     int integerOptionForVerbose = 0;
     if ( CommandlineProcessing::isOptionWithParameter(l,"-myTranslator:","(v|verbose)",integerOptionForVerbos
        {
          printf ("Turning on my translator's verbose mode (set to %d) \n",integerOptionForVerbose);
        }

  // Adding a new command line parameter (for mechanisms in ROSE that take command lines)

     // printf ("argc = %zu \n",l.size());
     // l = CommandlineProcessing::generateArgListFromArgcArgv (argc,argv);
     printf ("l.size() = %zu \n",l.size());
     printf ("Preprocessor (after): argv = \n%s \n",StringUtility::listToString(l).c_str());

  // SgProject* project = frontend(argc,argv);
  // ROSE_ASSERT (project != NULL);
  // Generate the source code and compile using the vendor's compiler
  // return backend(project);

  // Build the AST, generate the source code and call the backend compiler ...
     frontend(l);
     return 0;
   }
```

Figure 30.1: Example source code showing simple command-line processing within ROSE translator.

```
Preprocessor (before): argv =
/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/ROSE-build/tutorial/.libs/lt
Turning on my translator's verbose mode (set to 42)
l.size() = 4
Preprocessor (after): argv =
/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/ROSE-build/tutorial/.libs/lt
```

Figure 30.2: Output of input code using commandlineProcessing.C

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// rose.C: Example (default) ROSE Preprocessor: used for testing ROSE infrastructure

#include "rose.h"

using namespace std;

int
main ( int argc, char* argv[] )
   {
     Rose_STL_Container<string> l = CommandlineProcessing::generateArgListFromArgcArgv (argc,argv);
     printf ("Preprocessor (before): argv = \n%s \n",StringUtility::listToString(l).c_str());

  // Remove certain sorts of options from the command line
     CommandlineProcessing::removeArgs (l,"-edg:");
     CommandlineProcessing::removeArgs (l,"--edg:");
     CommandlineProcessing::removeArgsWithParameters (l,"-edg_parameter:");
     CommandlineProcessing::removeArgsWithParameters (l,"--edg_parameter:");

  // Add a test for a custom command line option
     int integerOptionForVerbose = 0;
     if ( CommandlineProcessing::isOptionWithParameter(l,"-myTranslator:","(v|verbose)",integerOptionForVerbose,true) )
        {
          printf ("Turning on my translator's verbose mode (set to %d) \n",integerOptionForVerbose);
        }

  // Adding a new command line parameter (for mechanisms in ROSE that take command lines)

     // printf ("argc = %zu \n",l.size());
     // l = CommandlineProcessing::generateArgListFromArgcArgv (argc,argv);
     printf ("l.size() = %zu \n",l.size());
     printf ("Preprocessor (after): argv = \n%s \n",StringUtility::listToString(l).c_str());

  // SgProject* project = frontend(argc,argv);
  // ROSE_ASSERT (project != NULL);
  // Generate the source code and compile using the vendor's compiler
  // return backend(project);

  // Build the AST, generate the source code and call the backend compiler ...
     frontend(l);
     return 0;
   }
```

Figure 30.3: Example source code showing simple command-line processing within ROSE translator.

```
Preprocessor (before): argv =
/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/ROSE-build/tutorial/.libs/lt-commandli
Turning on my translator's verbose mode (set to 42)
l.size() = 4
Preprocessor (after): argv =
/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/ROSE-build/tutorial/.libs/lt-commandli
```

Figure 30.4: Output of input code using commandlineProcessing.C

Chapter 31

Tailoring The Code Generation Format

Figure 31.1 shows an example of how to use the mechanisms in ROSE to tailor the format and style of the generated code. This chapter presents an example translator that modifies the formatting of the code that is generated within ROSE.

The details of functionality are hidden from the user and a high level interface is provided that permits key parameters to be specified. This example will be made more sophisticated later, for now it just modifies the indentation of nested code blocks (from 2 spaces/block to 5 spaces/block).

31.1 Source Code for Example that Tailors the Code Generation

Figure 31.1 shows an example translator which calls the inliner mechanism. The code is designed to only inline up to ten functions. the list of function calls is recomputed after any function call is successfully inlined.

The input code is shown in figure 31.2, the output of this code is shown in figure 31.3.

31.2 Input to Demonstrate Tailoring the Code Generation

Figure 31.2 shows the example input used for demonstration of how to control the formatting of generated code.

31.3 Final Code After Tailoring the Code Generation

Figure 31.3 shows the results from changes to the formatting of generated code.

```
// This example will be made more sophisticated later, for now it just
// modifies the indentation of nested code blocks (from 2 spaces/block
// to 5 spaces/block).

#include "rose.h"
#include "unparseFormatHelp.h"

class CustomCodeFormat : public UnparseFormatHelp
   {
     public:
          CustomCodeFormat();
         ~CustomCodeFormat();

          virtual int getLine( SgLocatedNode*, SgUnparse_Info& info, FormatOpt opt);
          virtual int getCol ( SgLocatedNode*, SgUnparse_Info& info, FormatOpt opt);

       // return the value for indentation of code (part of control over style)
          virtual int tabIndent ();

       // return the value for where line wrapping starts (part of control over style)
          virtual int maxLineLength ();

     private:
          int defaultLineLength;
          int defaultIndentation;
   };


CustomCodeFormat::CustomCodeFormat()
   {
  // default values here!
     defaultLineLength = 20;
     defaultIndentation = 5;
   }

CustomCodeFormat::~CustomCodeFormat()
   {}

// return: > 0: start new lines; == 0: use same line; < 0:default
int
CustomCodeFormat::getLine( SgLocatedNode*, SgUnparse_Info& info, FormatOpt opt)
   {
  // Use default mechanism to select the line where to output generated code
     return -1;
   }

// return starting column. if < 0, use default
int
CustomCodeFormat::getCol( SgLocatedNode*, SgUnparse_Info& info, FormatOpt opt)
   {
  // Use default mechanism to select the column where to output generated code
     return -1;
   }

int
CustomCodeFormat::tabIndent()
   {
  // Modify the indentation of the generated code (trival example of tailoring code generation)
     return defaultIndentation;
   }

int
CustomCodeFormat::maxLineLength()
   {
     return defaultLineLength;
   }


int main (int argc, char* argv[])
   {
  // Build the project object (AST) which we will fill up with multiple files and use as a
  // handle for all processing of the AST(s) associated with one or more source files.
     SgProject* project = new SgProject(argc,argv);

     CustomCodeFormat* formatControl = new CustomCodeFormat();

     return backend(project,formatControl);
   }
```

Figure 31.1: Example source code showing how to tailor the code generation format.

```
extern int min(int ,int );

void dgemm(double *a,double *b,double *c,int n)
{
  int _var_1;
  int _var_0;
  int i;
  int j;
  int k;
  for (_var_1 = 0; _var_1 <= -1 + n; _var_1 += 16) {
    for (_var_0 = 0; _var_0 <= -1 + n; _var_0 += 16) {
      for (i = 0; i <= -1 + n; i += 1) {
        for (k = _var_1; k <= min(-1 + n,_var_1 + 15); k += 1) {
          int dummy_1 = k * n + i;
          for (j = _var_0; j <= min(n + -16,_var_0); j += 16) {
            int _var_2 = (j);
            c[j * n + i] = c[j * n + i] + a[k * n + i] * b[j * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
            _var_2 = 1 + _var_2;
            c[_var_2 * n + i] = c[_var_2 * n + i] + a[k * n + i] * b[_var_2 * n + k];
          }
          for (; j <= min(-1 + n,_var_0 + 15); j += 1) {
            c[j * n + i] = c[j * n + i] + a[k * n + i] * b[j * n + k];
          }
        }
      }
    }
  }
}
```

Figure 31.2: Example source code used as input to program to the tailor the code generation.

```
int min(int ,int );

void dgemm(double *a,double *b,double *c,int n)
{
     int _var_1;
     int _var_0;
     int i;
     int j;
     int k;
     for (_var_1 = 0; _var_1 <= (-1 + n); _var_1 += 16) {
          for (_var_0 = 0; _var_0 <= (-1 + n); _var_0 += 16) {
               for (i = 0; i <= (-1 + n); i += 1) {
                    for (k = _var_1; k <= min((-1 + n),(_var_1 + 15)); k += 1) {
                    int dummy_1 = ((k * n) + i);
                    for (j = _var_0; j <= min((n + -16),_var_0); j += 16) {
                    int _var_2 = j;
                    c[(j * n) + i] = (c[(j * n) + i] + (a[(k * n) + i] * b[(j * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    _var_2 = (1 + _var_2);
                    c[(_var_2 * n) + i] = (c[(_var_2 * n) + i] + (a[(k * n) + i] * b[(_var_2 * n) + k]));
                    }
                    for (; j <= min((-1 + n),(_var_0 + 15)); j += 1) {
                    c[(j * n) + i] = (c[(j * n) + i] + (a[(k * n) + i] * b[(j * n) + k]));
                    }
                    }
               }
          }
     }
}
```

Figure 31.3: Output of input code after changing the format of the generated code.

Chapter 32

AST Construction

AST construction is a fundamental operation needed for building ROSE source-to-source translators. Several levels of interfaces are available in ROSE for users to build AST from scratch. High level interfaces are recommended to use whenever possible for their simplicity. Low level interfaces can give users the maximum freedom to manipulate some details in AST trees.

This chapter uses several examples to demonstrate how to create AST fragments for common language constructs (such as variable declarations, functions, function calls, etc.) and how to insert them into an existing AST tree. More examples of constructing AST using high level interfaces can be found at *rose/tests/roseTests/astInterfaceTests*. The source files of the high level interfaces are located in *rose/src/frontend/SageIII/sageInterface*.

32.1 Variable Declarations

What To Learn Two examples are given to show how to construct a SAGE III AST subtree for a variable declaration and its insertion into the existing AST tree.

- Example 1. Building a variable declaration using the high level AST construction and manipulation interfaces defined in namespace SageBuilder and SageInterface.

  Figure 32.1 shows the high level construction of an AST fragment (a variable declaration) and its insertion into the AST at the top of each block. buildVariableDeclaration() takes the name and type to build a variable declaration node. prependStatement() inserts the declaration at the top of a basic block node. Details for parent and scope pointers, symbol tables, source file position information and so on are handled transparently.

- Example 2. Building the variable declaration using low level member functions of SAGE III node classes.

  Figure 32.2 shows the low level construction of the same AST fragment (for the same variable declaration) and its insertion into the AST at the top of each block. SgNode constructors and their member functions are used. Side effects for scope, parent pointers and symbol tables have to be handled by programmers explicitly.

```
// SageBuilder contains all high level buildXXX() functions,
// such as buildVariableDeclaration(), buildLabelStatement() etc.
// SageInterface contains high level AST manipulation and utility functions,
// e.g. appendStatement(), lookupFunctionSymbolInParentScopes() etc.
#include "rose.h"
using namespace SageBuilder;
using namespace SageInterface;

class SimpleInstrumentation:public SgSimpleProcessing
{
public:
  void visit (SgNode * astNode);
};

void
SimpleInstrumentation::visit (SgNode * astNode)
{
  SgBasicBlock *block = isSgBasicBlock (astNode);
  if (block != NULL)
    {
      SgVariableDeclaration *variableDeclaration =
                  buildVariableDeclaration ("newVariable", buildIntType ());
      prependStatement (variableDeclaration, block);
    }
}

int
main (int argc, char *argv[])
{
  SgProject *project = frontend (argc, argv);
  ROSE_ASSERT (project != NULL);

  SimpleInstrumentation treeTraversal;
  treeTraversal.traverseInputFiles (project, preorder);

  AstTests::runAllTests (project);
  return backend (project);
}
```

Figure 32.1: AST construction and insertion for a variable using the high level interfaces

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// Specifically it shows the design of a transformation to instrument source code, placing source code
// at the top and bottom of each basic block.
// Member functions of SAGE III AST node classes are directly used.
// So all details for Sg_File_Info, scope, parent, symbol tables have to be explicitly handled.

#include "rose.h"

class SimpleInstrumentation : public SgSimpleProcessing
   {
     public:
          void visit ( SgNode* astNode );
   };

void
SimpleInstrumentation::visit ( SgNode* astNode )
   {
     SgBasicBlock* block = isSgBasicBlock(astNode);
     if (block != NULL)
        {
       // Mark this as a transformation (required)
          Sg_File_Info* sourceLocation = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
          ROSE_ASSERT(sourceLocation != NULL);

          SgType* type = new SgTypeInt();
          ROSE_ASSERT(type != NULL);

          SgName name = "newVariable";

          SgVariableDeclaration* variableDeclaration = new SgVariableDeclaration(sourceLocation,name,type);
          ROSE_ASSERT(variableDeclaration != NULL);

          SgInitializedName* initializedName = *(variableDeclaration->get_variables().begin());

       // DQ (6/18/2007): The unparser requires that the scope be set (for name qualification to work).
          initializedName->set_scope(block);

       // Liao (2/13/2008): AstTests requires this to be set
          variableDeclaration->set_firstNondefiningDeclaration(variableDeclaration);

          ROSE_ASSERT(block->get_statements().size() > 0);

          block->get_statements().insert(block->get_statements().begin(),variableDeclaration);
          variableDeclaration->set_parent(block);

       // Add a symbol to the sybol table for the new variable
          SgVariableSymbol* variableSymbol = new SgVariableSymbol(initializedName);
          block->insert_symbol(name,variableSymbol);
        }
   }

int
main ( int argc, char * argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT(project != NULL);

     SimpleInstrumentation treeTraversal;
     treeTraversal.traverseInputFiles ( project, preorder );

     AstTests::runAllTests(project);
     return backend(project);
   }
```

Figure 32.2: Example source code to read an input program and add a new variable declaration at the top of each block.

```
int main()
   {
     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 32.3: Example source code used as input to the translators adding new variable.

```

int main()
{
  int newVariable;
  for (int i = 0; i < 4; i++) {
    int newVariable;
    int x;
  }
  return 0;
}
```

Figure 32.4: Output of input to the translators adding new variable.

Figure 32.3 shows the input code used to test the translator. Figure 32.4 shows the resulting output.

32.2 Expressions

Figure 32.5 shows a translator using the high level AST builder interface to add an assignment statement right before the last statement in a main() function.

Figure 32.6 shows the input code used to test the translator. Figure 32.7 shows the resulting output.

```
// Expressions can be built using both bottomup (recommended ) and topdown orders.
// Bottomup: build operands first, operation later
// Topdown: build operation first, set operands later on.

#include "rose.h"
using namespace SageBuilder;
using namespace SageInterface;

int main (int argc, char *argv[])
{
  SgProject *project = frontend (argc, argv);
  // go to the function body
  SgFunctionDeclaration* mainFunc= findMain(project);

  SgBasicBlock* body= mainFunc->get_definition()->get_body();
  pushScopeStack(body);

 // bottomup: build operands first, create expression later on
 //  double result = 2 * (1 - gama * gama);
  SgExpression * init_exp =
            buildMultiplyOp(buildDoubleVal(2.0),
                 buildSubtractOp(buildDoubleVal(1.0),
                      buildMultiplyOp (buildVarRefExp("gama"),buildVarRefExp("gama")
                                      )));
  SgVariableDeclaration* decl = buildVariableDeclaration("result",buildDoubleType(),buildAssignInitializer(init_exp));

  SgStatement* laststmt = getLastStatement(topScopeStack());
  insertStatementBefore(laststmt,decl);

 // topdown: build expression first, set operands later on
 // double result2 = alpha * beta;
  SgExpression * init_exp2 = buildMultiplyOp();
  setLhsOperand(init_exp2,buildVarRefExp("alpha"));
  setRhsOperand(init_exp2,buildVarRefExp("beta"));

  SgVariableDeclaration* decl2 = buildVariableDeclaration("result2",buildDoubleType(),buildAssignInitializer(init_exp2));
  laststmt = getLastStatement(topScopeStack());
  insertStatementBefore(laststmt,decl2);

  popScopeStack();
  AstTests::runAllTests(project);

  //invoke backend compiler to generate object/binary files
   return backend (project);

}
```

Figure 32.5: Example translator to add expressions

```
int main()
{
  double alpha= 0.5;
  double beta = 0.1;
  double gama = 0.7;

  return 0;
}
```

Figure 32.6: Example source code used as input

```

int main()
{
  double alpha = 0.5;
  double beta = 0.1;
  double gama = 0.7;
  double result = 2.00000 * (1.00000 - gama * gama);
  double result2 = alpha * beta;
  return 0;
}
```

Figure 32.7: Output of the input

32.3 Assignment Statements

Figure 32.8 shows a translator using the high level AST builder interface to add an assignment statement right before the last statement in a main() function.

Figure 32.9 shows the input code used to test the translator. Figure 32.10 shows the resulting output.

```
// SageBuilder contains all high level buildXXX() functions,
// such as buildVariableDeclaration(), buildLabelStatement() etc.
// SageInterface contains high level AST manipulation and utility functions,
// e.g. appendStatement(), lookupFunctionSymbolInParentScopes() etc.
#include "rose.h"
using namespace SageBuilder;
using namespace SageInterface;

int main (int argc, char *argv[])
{
  SgProject *project = frontend (argc, argv);

  // go to the function body of main()
  // and push it to the scope stack
  SgFunctionDeclaration* mainFunc= findMain(project);
  SgBasicBlock* body= mainFunc->get_definition()->get_body();
  pushScopeStack(body);

  // build a variable assignment statement: i=9;
  // buildVarRefExp(string varName) will automatically search for a matching variable symbol starting
  // from the current scope to the global scope.
  SgExprStatement* assignStmt = buildAssignStatement(buildVarRefExp("i"),buildIntVal(9));

 // insert it before the last return statement
  SgStatement* lastStmt = getLastStatement(topScopeStack());
  insertStatementBefore(lastStmt,assignStmt);

  popScopeStack();

  //AstTests ensures there is no dangling SgVarRefExp without a mathing symbol
  AstTests::runAllTests(project);
  return backend (project);
}
```

Figure 32.8: Example source code to add an assignment statement

```
int main(int argc, char* argv[])
{
  int i;
  return 0;
}
```

Figure 32.9: Example source code used as input

```

int main(int argc,char *argv[])
{
  int i;
  i = 9;
  return 0;
}
```

Figure 32.10: Output of the input

32.4 Functions

This section shows how to add a function at the top of a global scope in a file. Again, examples for both high level and low level constructions of AST are given.

- Figure 32.11 shows the high level construction of a defining function (a function with a function body). Scope information is passed to builder functions explicitly when it is needed.

```
// This example shows how to construct a defining function (with a function body)
// using high level AST construction interfaces.
//
#include "rose.h"
using namespace SageBuilder;
using namespace SageInterface;

class SimpleInstrumentation : public SgSimpleProcessing
   {
     public:
          void visit ( SgNode* astNode );
   };

void
SimpleInstrumentation::visit ( SgNode* astNode )
   {
     SgGlobal* globalScope = isSgGlobal(astNode);
     if (globalScope != NULL)
        {
       // ********************************************************************
       // Create a parameter list with a parameter
       // ********************************************************************
          SgName var1_name = "var_name";
          SgReferenceType *ref_type = buildReferenceType(buildIntType());
          SgInitializedName *var1_init_name = buildInitializedName(var1_name, ref_type);
          SgFunctionParameterList* parameterList = buildFunctionParameterList();
          appendArg(parameterList,var1_init_name);

       // ****************************************************
       // Create a defining functionDeclaration (with a function body)
       // ****************************************************
          SgName func_name                    = "my_function";
          SgFunctionDeclaration * func        = buildDefiningFunctionDeclaration
                        (func_name, buildIntType(), parameterList,globalScope);
          SgBasicBlock*  func_body    = func->get_definition()->get_body();

       // ********************************************************
       // Insert a statement in the function body
       // *******************************************************

          SgVarRefExp *var_ref = buildVarRefExp(var1_name,func_body);
          SgPlusPlusOp *pp_expression = buildPlusPlusOp(var_ref);
          SgExprStatement* new_stmt = buildExprStatement(pp_expression);

       // insert a statement into the function body
          prependStatement(new_stmt,func_body);
          prependStatement(func,globalScope);

        }
   }

int
main ( int argc, char * argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT(project != NULL);

     SimpleInstrumentation treeTraversal;
     treeTraversal.traverseInputFiles ( project, preorder );

     AstTests::runAllTests(project);
     return backend(project);
   }
```

Figure 32.11: Addition of function to global scope using high level interfaces

- Figure 32.12 shows almost the same high level construction of the defining function, but with an additional scope stack. Scope information is passed to builder functions implicitly when it is needed.

```
// This example shows how to construct a defining function (with a function body)
// using high level AST construction interfaces.
// A scope stack is used to pass scope information implicitly to some builder functions
#include "rose.h"
using namespace SageBuilder;
using namespace SageInterface;

int
main ( int argc, char * argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT(project != NULL);
     SgGlobal *globalScope = getFirstGlobalScope (project);

 //push global scope into stack
     pushScopeStack (isSgScopeStatement (globalScope));

 // Create a parameter list with a parameter
    SgName var1_name = "var_name";
    SgReferenceType *ref_type = buildReferenceType(buildIntType());
    SgInitializedName *var1_init_name = buildInitializedName(var1_name, ref_type);
    SgFunctionParameterList* parameterList = buildFunctionParameterList();
    appendArg(parameterList,var1_init_name);

 // Create a defining functionDeclaration (with a function body)
    SgName func_name                    = "my_function";
    SgFunctionDeclaration * func        = buildDefiningFunctionDeclaration
                  (func_name, buildIntType(), parameterList);
    SgBasicBlock*  func_body    = func->get_definition()->get_body();

 // push function body scope into stack
    pushScopeStack(isSgScopeStatement(func_body));

 // build a statement in the function body
    SgVarRefExp *var_ref = buildVarRefExp(var1_name);
    SgPlusPlusOp *pp_expression = buildPlusPlusOp(var_ref);
    SgExprStatement* new_stmt = buildExprStatement(pp_expression);

 // insert a statement into the function body
    appendStatement(new_stmt);
//  pop function body off the stack
    popScopeStack();

//  insert the function declaration into the scope at the top of the scope stack
    prependStatement(func);
    popScopeStack();

    AstTests::runAllTests(project);
    return backend(project);
 }
```

Figure 32.12: Addition of function to global scope using high level interfaces and a scope stack

- The low level construction of the AST fragment of the same function declaration and its insertion is separated into two portions and shown in two figures (Figure 32.13 and Figure 32.14).

Figure 32.29 and Figure 32.30 give the input code and output result for the translators above.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// Specifically it shows the design of a transformation to instrument source code, placing source code
// at the top of the source file.

#include "rose.h"

#define TRANSFORMATION_FILE_INFO Sg_File_Info::generateDefaultFileInfoForTransformationNode()

class SimpleInstrumentation : public SgSimpleProcessing
   {
     public:
          void visit ( SgNode* astNode );
   };

void
SimpleInstrumentation::visit ( SgNode* astNode )
   {
     SgGlobal* globalScope = isSgGlobal(astNode);
     if (globalScope != NULL)
        {
       // ********************************************************
       // Create the functionDeclaration
       // ********************************************************
          SgType * func_return_type              = new SgTypeInt();
          SgName func_name                       = "my_function";
          SgFunctionType * func_type             = new SgFunctionType(func_return_type,false);
          SgFunctionDeclaration * func           = new SgFunctionDeclaration(TRANSFORMATION_FILE_INFO, func_name, func_type);
          SgFunctionDefinition *  func_def       = new SgFunctionDefinition(TRANSFORMATION_FILE_INFO, func);
          SgBasicBlock         * func_body       = new SgBasicBlock(TRANSFORMATION_FILE_INFO);

       // set the end source position as transformation generated
       // since the constructors only set the beginning source position by default
          func->set_endOfConstruct(TRANSFORMATION_FILE_INFO);
          func->get_endOfConstruct()->set_parent(func);

          func_def->set_endOfConstruct(TRANSFORMATION_FILE_INFO);
          func_def->get_endOfConstruct()->set_parent(func_def);

          func_body->set_endOfConstruct(TRANSFORMATION_FILE_INFO);
          func_body->get_endOfConstruct()->set_parent(func_body);

       // Sets the body into the definition
          func_def->set_body(func_body);
       // Sets the defintion's parent to the declaration
          func_def->set_parent(func);

       // DQ (9/8/2007): Fixup the defining and non-defining declarations
          ROSE_ASSERT(func->get_definingDeclaration() == NULL);
          func->set_definingDeclaration(func);
          ROSE_ASSERT(func->get_definingDeclaration()         != NULL);
          ROSE_ASSERT(func->get_firstNondefiningDeclaration() != func);

       // DQ (9/8/2007): We have not build a non-defining declaration, so this should be NULL.
          ROSE_ASSERT(func->get_firstNondefiningDeclaration() == NULL);

       // DQ (9/8/2007): Need to add function symbol to global scope!
          //printf ("Fixing up the symbol table in scope = %p = %s for function = %p = %s \n",globalScope,globalScope->class_name().c_str(),fu
          SgFunctionSymbol* functionSymbol = new SgFunctionSymbol(func);
          globalScope->insert_symbol(func->get_name(),functionSymbol);
          ROSE_ASSERT(globalScope->lookup_function_symbol(func->get_name()) != NULL);

       // ******************************************************************
       // Create the InitializedName for a parameter within the parameter list
       // ******************************************************************
          SgName var1_name = "var_name";

          SgTypeInt * var1_type     = new SgTypeInt();
          SgReferenceType *ref_type = new SgReferenceType(var1_type);
          SgInitializer * var1_initializer = NULL;

          SgInitializedName *var1_init_name = new SgInitializedName(var1_name, ref_type, var1_initializer, NULL);
          var1_init_name->set_file_info(TRANSFORMATION_FILE_INFO);

       // DQ (9/8/2007): We now test this, so it has to be set explicitly.
          var1_init_name->set_scope(func_def);

       // DQ (9/8/2007): Need to add variable symbol to global scope!
          //printf ("Fixing up the symbol table in scope = %p = %s for SgInitializedName = %p = %s \n",globalScope,globalScope->class_name().c
          SgVariableSymbol *var_symbol = new SgVariableSymbol(var1_init_name);
          func_def->insert_symbol(var1_init_name->get_name(),var_symbol);
```

Figure 32.13: Example source code shows addition of function to global scope (part 1).

```
            ROSE_ASSERT(func_def->lookup_variable_symbol(var1_init_name->get_name()) != NULL);
            ROSE_ASSERT(var1_init_name->get_symbol_from_symbol_table() != NULL);

         // Done constructing the InitializedName variable

         // Insert argument in function parameter list
            ROSE_ASSERT(func != NULL);
         // Sg_File_Info * parameterListFileInfo   = new Sg_File_Info();
         // Sg_File_Info* parameterListFileInfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
            SgFunctionParameterList* parameterList = new SgFunctionParameterList(TRANSFORMATION_FILE_INFO);
            ROSE_ASSERT(parameterList != NULL);
            func->set_parameterList(parameterList);
            ROSE_ASSERT(func->get_parameterList() != NULL);
            func->get_parameterList()->append_arg(var1_init_name);

         // ********************************************************
         // Insert a statement in the function body
         // *******************************************************

         // create a VarRefExp
         // SgVariableSymbol *var_symbol = new SgVariableSymbol(var1_init_name);
            SgVarRefExp *var_ref = new SgVarRefExp(TRANSFORMATION_FILE_INFO,var_symbol);
            var_ref->set_endOfConstruct(TRANSFORMATION_FILE_INFO);
            var_ref->get_endOfConstruct()->set_parent(var_ref);

         // create a ++ expression, 0 for prefix ++
            SgPlusPlusOp *pp_expression = new SgPlusPlusOp(TRANSFORMATION_FILE_INFO,var_ref,0);
            pp_expression->set_endOfConstruct(TRANSFORMATION_FILE_INFO);
            pp_expression->get_endOfConstruct()->set_parent(pp_expression);

         // create an expression statement
            SgExprStatement* new_stmt = new SgExprStatement(TRANSFORMATION_FILE_INFO,pp_expression);
            new_stmt->set_endOfConstruct(TRANSFORMATION_FILE_INFO);
            new_stmt->get_endOfConstruct()->set_parent(new_stmt);

#if 0
         // DQ (9/8/2007): This is no longer required, SgExpressionRoot is not longer used in the ROSE IR.
         // create an expression type
            SgTypeInt* expr_type = new SgTypeInt();

         // create an expression root
            SgExpressionRoot * expr_root = new SgExpressionRoot(TRANSFORMATION_FILE_INFO,pp_expression,expr_type);
            expr_root->set_parent(new_stmt);

         // DQ (11/8/2006): Modified to reflect use of SgExpression instead of SgExpressionRoot
            new_stmt->set_expression(expr_root);

            pp_expression->set_parent(new_stmt->get_expression());
#endif
            pp_expression->set_parent(new_stmt);

         // insert a statement into the function body
            func_body->prepend_statement(new_stmt);

         // setting the parent explicitly is not required since it would be done within AST post-processing
            func->set_parent(globalScope);

         // scopes of statments must be set explicitly since within C++ they are not guaranteed
         // to be the same as that indicated by the parent (see ChangeLog for Spring 2005).
            func->set_scope(globalScope);

         // ********************************************************
         // Insert the function declaration in the code
         // *******************************************************
            globalScope->prepend_declaration(func);

         // Required post processing of AST required to set parent pointers and fixup template names, etc.
         // temporaryAstFixes(globalScope);
            AstPostProcessing(globalScope);
          }
     }

int
main ( int argc, char * argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT(project != NULL);

     SimpleInstrumentation treeTraversal;
     treeTraversal.traverseInputFiles ( project, preorder );
```

Figure 32.14: Example source code shows addition of function to global scope (part 2).

```
int main()
   {
     for (int i=0; i < 4; i++)
        {
          int x;
        }

     return 0;
   }
```

Figure 32.15: Example source code used as input to translator adding new function.

```
int my_function(int &var_name)
{
  ++var_name;
}

int main()
{
  for (int i = 0; i < 4; i++) {
    int x;
  }
  return 0;
}
```

Figure 32.16: Output of input to translator adding new function.

32.5 Function Calls

Adding functions calls is a typical task for instrumentation translator.

- Figure 32.17 shows the use of the AST string based rewrite mechanism to add function calls to the top and bottom of each block within the AST.
- Figure 32.18 shows the use of the AST builder interface to do the same instrumentation work.

Figure 32.19 shows the input code used to get the translator. Figure 32.20 shows the resulting output.

Another example shows how to add a function call at the end of each function body. A utility function, *instrumentEndOfFunction()*, from SageInterface name space is used. The interface tries to locate all return statements of a target function and rewriting return expressions with side effects, if there are any. Figure 32.21 shows the translator code. Figure 32.22 shows the input code. The instrumented code is shown in Figure 32.23.

32.6 Creating a 'struct' for Global Variables

TODO: This tutorial el AST manipulation. d have a more concise ing SageInterface and SageBuilder functions.

This is an example written to support the Charm++ tool. This translator extracts global variables from the program and builds a structure to hold them. The support is part of a number of requirements associated with using Charm++ and AMPI.

Figure 32.24 shows repackaging of global variables within an application into a struct. All reference to the global variables are also transformed to reference the original variable indirectly through the structure. This processing is part of preprocessing to use Charm++.

This example shows the low level handling directly at the level of the IR.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// Specifically it shows the design of a transformation to instrument source code, placing source code
// at the top and bottome of each basic block.

#include "rose.h"

using namespace std;

class SimpleInstrumentation : public SgSimpleProcessing
   {
     public:
          void visit ( SgNode* astNode );
   };

void
SimpleInstrumentation::visit ( SgNode* astNode )
   {
     SgBasicBlock* block = isSgBasicBlock(astNode);
     if (block != NULL)
        {
          const unsigned int SIZE_OF_BLOCK = 1;
          if (block->get_statements().size() > SIZE_OF_BLOCK)
             {
            // It is up to the user to link the implementations of these functions link time
               string codeAtTopOfBlock    = "void myTimerFunctionStart(); myTimerFunctionStart();";
               string codeAtBottomOfBlock = "void myTimerFunctionEnd(); myTimerFunctionEnd();";

            // Insert new code into the scope represented by the statement (applies to SgScopeStatements)
               MiddleLevelRewrite::ScopeIdentifierEnum scope = MidLevelCollectionTypedefs::StatementScope;

            // Insert the new code at the top and bottom of the scope represented by block
               MiddleLevelRewrite::insert(block,codeAtTopOfBlock,scope,
                                          MidLevelCollectionTypedefs::TopOfCurrentScope);
               MiddleLevelRewrite::insert(block,codeAtBottomOfBlock,scope,
                                          MidLevelCollectionTypedefs::BottomOfCurrentScope);
             }
        }
   }

int
main ( int argc, char * argv[] )
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT(project != NULL);

     SimpleInstrumentation treeTraversal;
     treeTraversal.traverseInputFiles ( project, preorder );

     AstTests::runAllTests(project);
     return backend(project);
   }
```

Figure 32.17: Example source code to instrument any input program.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// Specifically it shows the design of a transformation to instrument source code, placing source code
// at the top and bottom of each basic block.

#include "rose.h"
using namespace std;
using namespace SageInterface;
using namespace SageBuilder;

class SimpleInstrumentation:public SgSimpleProcessing
{
public:
  void visit (SgNode * astNode);
};

void
SimpleInstrumentation::visit (SgNode * astNode)
{
  SgBasicBlock *block = isSgBasicBlock (astNode);
  if (block != NULL)
    {
      const unsigned int SIZE_OF_BLOCK = 1;
      if (block->get_statements ().size () > SIZE_OF_BLOCK)
        {
          SgName name1("myTimerFunctionStart");
          // It is up to the user to link the implementations of these functions link time
          SgFunctionDeclaration *decl_1 = buildNondefiningFunctionDeclaration
              (name1,buildVoidType(),buildFunctionParameterList(),block);
          ((decl_1->get_declarationModifier()).get_storageModifier()).setExtern();

          SgExprStatement* callStmt_1 = buildFunctionCallStmt
              (name1,buildVoidType(), buildExprListExp(),block);

          prependStatement(callStmt_1,block);
          prependStatement(decl_1,block);

          SgName name2("myTimerFunctionEnd");
          // It is up to the user to link the implementations of these functions link time
          SgFunctionDeclaration *decl_2 = buildNondefiningFunctionDeclaration
              (name2,buildVoidType(),buildFunctionParameterList(),block);
          ((decl_2->get_declarationModifier()).get_storageModifier()).setExtern();

          SgExprStatement* callStmt_2 = buildFunctionCallStmt
              (name2,buildVoidType(), buildExprListExp(),block);

          appendStatement(decl_2,block);
          appendStatement(callStmt_2,block);
        }
    }
}

int
main (int argc, char *argv[])
{
  SgProject *project = frontend (argc, argv);
  ROSE_ASSERT (project != NULL);

  SimpleInstrumentation treeTraversal;
  treeTraversal.traverseInputFiles (project, preorder);

  AstTests::runAllTests (project);
  return backend (project);
}
```

Figure 32.18: Example source code using the high level interfaces

```
// Overloaded functions for testing overloaded function resolution
void foo(double)
   {
     int x = 1;
     int y;

  // I think that this case fails currently
  // if (x) y = 1; else y = 2;
   }
```

Figure 32.19: Example source code used as input to instrumenting translator.

```
// Overloaded functions for testing overloaded function resolution

void foo(double )
{
  void myTimerFunctionStart();
  myTimerFunctionStart();
  int x = 1;
  int y;
  void myTimerFunctionEnd();;
  myTimerFunctionEnd();
// I think that this case fails currently
// if (x) y = 1; else y = 2;
}
```

Figure 32.20: Output of input to instrumenting translator.

```
/*! \brief  test instrumentation right before the end of a function
*/
#include "rose.h"
#include <iostream>
using namespace SageInterface;
using namespace SageBuilder;

int main (int argc, char *argv[])
{
  SgProject *project = frontend (argc, argv);

  // Find all function definitions we want to instrument
  std::vector<SgNode* > funcDefList =
    NodeQuery::querySubTree (project, V_SgFunctionDefinition);

 std::vector<SgNode*>::iterator iter;
 for (iter = funcDefList.begin(); iter!= funcDefList.end(); iter++)
 {
   SgFunctionDefinition* cur_def = isSgFunctionDefinition(*iter);
   ROSE_ASSERT(cur_def);
  SgBasicBlock* body = cur_def->get_body();
  // Build the call statement for each place
  SgExprStatement* callStmt1 = buildFunctionCallStmt("call1",
               buildIntType(),buildExprListExp() ,body);

  // instrument the function
  int i= instrumentEndOfFunction(cur_def->get_declaration(), callStmt1);
  std::cout<<"Instrumented "<<i<<" places. "<<std::endl;

 }  // end of instrumentation

  AstTests::runAllTests(project);
  // translation only
   project->unparse();
}
```

Figure 32.21: Example source code instrumenting end of functions

```
/* Example code:
 *  a function with multiple returns
 *     some returns have expressions with side effects
 *  a function without any return
 */
extern int foo();
extern int call1();
int main(int argc, char* argv[])
{
  if (argc>1)
    return foo();
  else
     return foo();
  return 0;
}

void bar()
{
  int i;
}
```

Figure 32.22: Example input code of the instrumenting translator for end of functions.

```
/* Example code:
 *  a function with multiple returns
 *     some returns have expressions with side effects
 *  a function without any return
 */
int foo();
int call1();

int main(int argc,char *argv[])
{
  if (argc > 1) {
    int rose_temp__1 = foo();
    call1();
    return rose_temp__1;
  }
  else {
    int rose_temp__2 = foo();
    call1();
    return rose_temp__2;
  }
  call1();
  return 0;
}

void bar()
{
  int i;
  call1();
}
```

Figure 32.23: Output of instrumenting translator for end of functions.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// Specifically it shows the design of a transformation to do a transformation specific for Charm++.

#include "rose.h"

using namespace std;

Rose_STL_Container<SgInitializedName*>
buildListOfGlobalVariables ( SgSourceFile* file )
   {
  // This function builds a list of global variables (from a SgFile).
     assert(file != NULL);

     Rose_STL_Container<SgInitializedName*> globalVariableList;

     SgGlobal* globalScope = file->get_globalScope();
     assert(globalScope != NULL);
     Rose_STL_Container<SgDeclarationStatement*>::iterator i = globalScope->get_declarations().begin();
     while(i != globalScope->get_declarations().end())
        {
          SgVariableDeclaration *variableDeclaration = isSgVariableDeclaration(*i);
          if (variableDeclaration != NULL)
             {
               Rose_STL_Container<SgInitializedName*> & variableList = variableDeclaration->get_variables();
               Rose_STL_Container<SgInitializedName*>::iterator var = variableList.begin();
               while(var != variableList.end())
                  {
                    globalVariableList.push_back(*var);
                    var++;
                  }
             }
          i++;
        }

     return globalVariableList;
   }

// This function is not used, but is useful for
// generating the list of all global variables
Rose_STL_Container<SgInitializedName*>
buildListOfGlobalVariables ( SgProject* project )
   {
  // This function builds a list of global variables (from a SgProject).

     Rose_STL_Container<SgInitializedName*> globalVariableList;

     const SgFilePtrList& fileList = project->get_fileList();
     SgFilePtrList::const_iterator file = fileList.begin();

  // Loop over the files in the project (multiple files exist
  // when multiple source files are placed on the command line).
     while(file != fileList.end())
        {
          Rose_STL_Container<SgInitializedName*> fileGlobalVariableList = buildListOfGlobalVariables(isSgSourceFile(*file));

       // DQ (9/26/2007): Moved from std::list to std::vector
       // globalVariableList.merge(fileGlobalVariableList);
          globalVariableList.insert(globalVariableList.begin(),fileGlobalVariableList.begin(),fileGlobalVariableList.end());

          file++;
        }

     return globalVariableList;
   }

Rose_STL_Container<SgVarRefExp*>
buildListOfVariableReferenceExpressionsUsingGlobalVariables ( SgNode* node )
   {
  // This function builds a list of "uses" of variables (SgVarRefExp IR nodes) within the AST.

  // return variable
     Rose_STL_Container<SgVarRefExp*> globalVariableUseList;

  // list of all variables (then select out the global variables by testing the scope)
     Rose_STL_Container<SgNode*> nodeList = NodeQuery::querySubTree ( node, V_SgVarRefExp );

     Rose_STL_Container<SgNode*>::iterator i = nodeList.begin();
     while(i != nodeList.end())
        {
          SgVarRefExp *variableReferenceExpression = isSgVarRefExp(*i);
```

Figure 32.24: Example source code shows repackaging of global variables to a struct (part 1).

```
            assert(variableReferenceExpression != NULL);

            assert(variableReferenceExpression->get_symbol() != NULL);
            assert(variableReferenceExpression->get_symbol()->get_declaration() != NULL);
            assert(variableReferenceExpression->get_symbol()->get_declaration()->get_scope() != NULL);

         // Note that variableReferenceExpression->get_symbol()->get_declaration() returns the
         // SgInitializedName (not the SgVariableDeclaration where it was declared)!
            SgInitializedName* variable = variableReferenceExpression->get_symbol()->get_declaration();

            SgScopeStatement* variableScope = variable->get_scope();

         // Check if this is a variable declared in global scope, if so, then save it
            if (isSgGlobal(variableScope) != NULL)
               {
                 globalVariableUseList.push_back(variableReferenceExpression);
               }
            i++;
          }

       return globalVariableUseList;
     }


SgClassDeclaration*
buildClassDeclarationAndDefinition (string name, SgScopeStatement* scope)
   {
  // This function builds a class declaration and definition
  // (both the defining and nondefining declarations as required).

  // Build a file info object marked as a transformation
     Sg_File_Info* fileInfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
     assert(fileInfo != NULL);

  // This is the class definition (the fileInfo is the position of the opening brace)
     SgClassDefinition* classDefinition   = new SgClassDefinition(fileInfo);
     assert(classDefinition != NULL);

  // Set the end of construct explictly (where not a transformation this is the location of the closing brace)
     classDefinition->set_endOfConstruct(fileInfo);

  // This is the defining declaration for the class (with a reference to the class definition)
     SgClassDeclaration* classDeclaration = new SgClassDeclaration(fileInfo,name.c_str(),SgClassDeclaration::e_struct,NULL,classDefinition);
     assert(classDeclaration != NULL);

  // Set the defining declaration in the defining declaration!
     classDeclaration->set_definingDeclaration(classDeclaration);

  // Set the non defining declaration in the defining declaration (both are required)
     SgClassDeclaration* nondefiningClassDeclaration = new SgClassDeclaration(fileInfo,name.c_str(),SgClassDeclaration::e_struct,NULL,NULL);
     assert(classDeclaration != NULL);
     nondefiningClassDeclaration->set_type(SgClassType::createType(nondefiningClassDeclaration));

  // Set the internal reference to the non-defining declaration
     classDeclaration->set_firstNondefiningDeclaration(nondefiningClassDeclaration);
     classDeclaration->set_type(nondefiningClassDeclaration->get_type());

  // Set the defining and no-defining declarations in the non-defining class declaration!
     nondefiningClassDeclaration->set_firstNondefiningDeclaration(nondefiningClassDeclaration);
     nondefiningClassDeclaration->set_definingDeclaration(classDeclaration);

  // Set the nondefining declaration as a forward declaration!
     nondefiningClassDeclaration->setForward();

  // Don't forget the set the declaration in the definition (IR node constructors are side-effect free!)!
     classDefinition->set_declaration(classDeclaration);

  // set the scope explicitly (name qualification tricks can imply it is not always the parent IR node!)
     classDeclaration->set_scope(scope);
     nondefiningClassDeclaration->set_scope(scope);

  // some error checking
     assert(classDeclaration->get_definingDeclaration() != NULL);
     assert(classDeclaration->get_firstNondefiningDeclaration() != NULL);
     assert(classDeclaration->get_definition() != NULL);

  // DQ (9/8/2007): Need to add function symbol to global scope!
     printf ("Fixing up the symbol table in scope = %p = %s for class = %p = %s \n",scope,scope->class_name().c_str(),classDeclaration,classDe
     SgClassSymbol* classSymbol = new SgClassSymbol(classDeclaration);
     scope->insert_symbol(classDeclaration->get_name(),classSymbol);
```

Figure 32.25: Example source code shows repackaging of global variables to a struct (part 2).

```
       ROSE_ASSERT(scope->lookup_class_symbol(classDeclaration->get_name()) != NULL);

       return classDeclaration;
     }



SgVariableSymbol*
putGlobalVariablesIntoClass (Rose_STL_Container<SgInitializedName*> & globalVariables, SgClassDeclaration* classDeclaration )
     {
    // This function iterates over the list of global variables and inserts them into the iput class definition

       SgVariableSymbol* globalClassVariableSymbol = NULL;

       for (Rose_STL_Container<SgInitializedName*>::iterator var = globalVariables.begin(); var != globalVariables.end(); var++)
          {
         // printf ("Appending global variable = %s to new globalVariableContainer \n",(*var)->get_name().str());
            SgVariableDeclaration* globalVariableDeclaration = isSgVariableDeclaration((*var)->get_parent());
            assert(globalVariableDeclaration != NULL);

         // Get the global scope from the global variable directly
            SgGlobal* globalScope = isSgGlobal(globalVariableDeclaration->get_scope());
            assert(globalScope != NULL);

            if (var == globalVariables.begin())
               {
              // This is the first time in this loop, replace the first global variable with
              // the class declaration/definition containing all the global variables!
              // Note that initializers in the global variable declarations require modification
              // of the preinitialization list in the class's constructor!  I am ignoring this for now!
                 globalScope->replace_statement(globalVariableDeclaration,classDeclaration);

              // Build source position informaiton (marked as transformation)
                 Sg_File_Info* fileInfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
                 assert(fileInfo != NULL);

              // Add the variable of the class type to the global scope!
                 SgClassType* variableType = new SgClassType(classDeclaration->get_firstNondefiningDeclaration());
                 assert(variableType != NULL);
                 SgVariableDeclaration* variableDeclaration = new SgVariableDeclaration(fileInfo,"AMPI_globals",variableType);
                 assert(variableDeclaration != NULL);

                 globalScope->insert_statement(classDeclaration,variableDeclaration,false);

                 assert(variableDeclaration->get_variables().empty() == false);
                 SgInitializedName* variableName = *(variableDeclaration->get_variables().begin());
                 assert(variableName != NULL);

              // DQ (9/8/2007): Need to set the scope of the new variable.
                 variableName->set_scope(globalScope);

              // build the return value
                 globalClassVariableSymbol = new SgVariableSymbol(variableName);

              // DQ (9/8/2007): Need to add the symbol to the global scope (new testing requires this).
                 globalScope->insert_symbol(variableName->get_name(),globalClassVariableSymbol);
                 ROSE_ASSERT(globalScope->lookup_variable_symbol(variableName->get_name()) != NULL);
               }
              else
               {
              // for all other iterations of this loop ...
              // remove variable declaration from the global scope
                 globalScope->remove_statement(globalVariableDeclaration);
               }

         // add the variable declaration to the class definition
            classDeclaration->get_definition()->append_member(globalVariableDeclaration);
          }

       return globalClassVariableSymbol;
     }


void
fixupReferencesToGlobalVariables ( Rose_STL_Container<SgVarRefExp*> & variableReferenceList, SgVariableSymbol* globalClassVariabl
     {
    // Now fixup the SgVarRefExp to reference the global variables through a struct
       for (Rose_STL_Container<SgVarRefExp*>::iterator var = variableReferenceList.begin(); var != variableReferenceList.end(); var
          {
            assert(*var != NULL);
```

Figure 32.26: Example source code shows repackaging of global variables to a struct (part 3).

```
        // printf ("Variable reference for %s \n",(*var)->get_symbol()->get_declaration()->get_name().str());

           SgNode* parent = (*var)->get_parent();
           assert(parent != NULL);

        // If this is not an expression then is likely a meaningless statement such as ("x;")
           SgExpression* parentExpression = isSgExpression(parent);
           assert(parentExpression != NULL);

        // Build the reference through the global class variable ("x" --> "AMPI_globals.x")

        // Build source position informaiton (marked as transformation)
           Sg_File_Info* fileInfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
           assert(fileInfo != NULL);

        // Build "AMPI_globals"
           SgExpression* lhs = new SgVarRefExp(fileInfo,globalClassVariableSymbol);
           assert(lhs != NULL);
        // Build "AMPI_globals.x" from "x"
           SgDotExp* globalVariableReference = new SgDotExp(fileInfo,lhs,*var);
           assert(globalVariableReference != NULL);

           if (parentExpression != NULL)
              {
             // Introduce reference to *var through the data structure

             // case of binary operator
                SgUnaryOp* unaryOperator = isSgUnaryOp(parentExpression);
                if (unaryOperator != NULL)
                   {
                     unaryOperator->set_operand(globalVariableReference);
                   }
                  else
                   {
                  // case of binary operator
                     SgBinaryOp* binaryOperator = isSgBinaryOp(parentExpression);
                     if (binaryOperator != NULL)
                        {
                       // figure out if the *var is on the lhs or the rhs
                          if (binaryOperator->get_lhs_operand() == *var)
                             {
                               binaryOperator->set_lhs_operand(globalVariableReference);
                             }
                            else
                             {
                               assert(binaryOperator->get_rhs_operand() == *var);
                               binaryOperator->set_rhs_operand(globalVariableReference);
                             }
                        }
                       else
                        {
                       // ignore these cases for now!
                          switch(parentExpression->variantT())
                             {
                            // Where the variable appers in the function argument list the parent is a SgExprListExp
                               case V_SgExprListExp:
                                  {
                                    printf ("Sorry not implemented, case of global variable in function argument list ... \n");
                                    assert(false);
                                    break;
                                  }
                               case V_SgInitializer:
                               case V_SgRefExp:
                               case V_SgVarArgOp:
                               default:
                                  {
                                    printf ("Error: default reached in switch
parentExpression = %p = %s \n",parentExpression,parentExpression->class_name().c_str());
                                    assert(false);
                                  }
                             }
                        }
                   }
              }
        }
   }

#define OUTPUT_NAMES_OF_GLOBAL_VARIABLES           0
#define OUTPUT_NAMES_OF_GLOBAL_VARIABLE_REFERENCES 0

void
```

Figure 32.27: Example source code shows repackaging of global variables to a struct (part 4).

```
transformGlobalVariablesToUseStruct ( SgSourceFile *file )
   {
     assert(file != NULL);

  // These are the global variables in the input program (provided as helpful information)
     Rose_STL_Container<SgInitializedName*> globalVariables = buildListOfGlobalVariables(file);

#if OUTPUT_NAMES_OF_GLOBAL_VARIABLES
     printf ("global variables (declared in global scope): \n");
     for (Rose_STL_Container<SgInitializedName*>::iterator var = globalVariables.begin(); var != globalVariables.end(); var++)
        {
          printf ("   %s \n",(*var)->get_name().str());
        }
     printf ("\n");
#endif

  // get the global scope within the first file (currently ignoring all other files)
     SgGlobal* globalScope = file->get_globalScope();
     assert(globalScope != NULL);

  // Build the class declaration
     SgClassDeclaration* classDeclaration = buildClassDeclarationAndDefinition("AMPI_globals_t",globalScope);

  // Put the global variables intothe class
     SgVariableSymbol* globalClassVariableSymbol = putGlobalVariablesIntoClass(globalVariables,classDeclaration);

  // Their associated symbols will be located within the project's AST
  // (where they occur in variable reference expressions).
     Rose_STL_Container<SgVarRefExp*> variableReferenceList = buildListOfVariableReferenceExpressionsUsingGlobalVariables(file);

#if OUTPUT_NAMES_OF_GLOBAL_VARIABLE_REFERENCES
     printf ("global variables appearing in the application: \n");
     for (Rose_STL_Container<SgVarRefExp*>::iterator var = variableReferenceList.begin(); var != variableReferenceList.end(); var
        {
          printf ("   %s \n",(*var)->get_symbol()->get_declaration()->get_name().str());
        }
     printf ("\n");
#endif

  // Fixup all references to global variable to access the variable through the class ("x" --> "AMPI_globals.x")
     fixupReferencesToGlobalVariables(variableReferenceList,globalClassVariableSymbol);
   }


void
transformGlobalVariablesToUseStruct ( SgProject *project )
   {
  // Call the transformation of each file (there are multiple SgFile
  // objects when multiple files are specfied on the command line!).
     assert(project != NULL);

     const SgFilePtrList& fileList = project->get_fileList();
     SgFilePtrList::const_iterator file = fileList.begin();
     while(file != fileList.end())
        {
          transformGlobalVariablesToUseStruct(isSgSourceFile(*file));
          file++;
        }
   }

// ******************************************
//                 MAIN PROGRAM
// ******************************************
int
main( int argc, char * argv[] )
   {
  // Build the AST used by ROSE
     SgProject* project = frontend(argc,argv);
     assert(project != NULL);

  // transform application as required
     transformGlobalVariablesToUseStruct(project);

  // Code generation phase (write out new application "rose_<input file name>")
     return backend(project);
   }
```

Figure 32.28: Example source code shows repackaging of global variables to a struct (part 5).

```
int x;
int y;
long z;
float pressure;

int main()
   {
     int a = 0;
     int b = 0;
     float density = 1.0;

     x++;
     b++;

     x = a + y;

     return 0;
   }
```

Figure 32.29: Example source code used as input to translator adding new function.

```

struct AMPI_globals_t
{
  int x;
  int y;
  long z;
  float pressure;
}
;
struct AMPI_globals_t AMPI_globals;

int main()
{
  int a = 0;
  int b = 0;
  float density = 1.0;
  AMPI_globals.x++;
  b++;
  AMPI_globals.x = (a + AMPI_globals.y);
  return 0;
}
```

Figure 32.30: Output of input to translator adding new function.

Chapter 33

Parser Building Blocks

It is often needed to write a small parser to parse customized source code annotations in forms of C/C++ pragmas or Fortran comments. The parser is responsible for recognizing keywords, variables, expressions in the annotations and storing the recognized information in AST, often in a form of persistent AstAttribute. ROSE provides a set of helper functions which can be used to create such a simple parser using recursive descent parsing[1]. These functions are collected in the namespace named AstFromString, documented under the Namespace List of ROSE's online Doxygen web references.

A suggested workflow to build your own pragma (or comment) parser is:

- input: define the grammar of your pragma, including keywords, directives and optional clauses. Borrowing grammars of OpenMP is a good idea.

- output: define your own AstAttribute in order to store any possible pragma information generated by your parser. The attribute data structure should store the AST subtrees generated by a parser. The Attribute itself should be attached to relevant statement node (pragma declarations or others) in the original AST.

- parser: write your recursive descent pragma parser by leveraging the functions defined in *rose_sourcetree/src/frontend/SageIII/astFromString/AstFromString.h*. The parsing results will be used to generate your attribute which should be attached to your pragma declaration statement (or a following statement for a Fortran comment).

- use of parsing results: in another phase, write your traversal to recognize the attached attribute to do whatever you plan to do with the pragma information.

A full example is provided under *rose/projects/pragmaParsing* to demonstrate the entire workflow of using parser building blocks (helper functions within AstFromString) to create a parser to parse user-defined pragmas. We will use this example in this tutorial.

---

[1] Description of basic recursive descent parsing techniques can be found at http://en.wikipedia.org/wiki/Recursive_descent_parser

33.1 Grammar Examples

The first step of building a parser is to formally define the grammar of input strings. The online Doxygen web reference lists helper functions from the AstFromString namespace, with detailed information about the underneath grammars they try to recognize. These grammar can be used as example about how to prepare grammars.

For example bool afs_match_cast_expression () follows the grammar like: cast_expression : '(' type_name ')' cast_expression | unary_expression, which means a cast expression is either a unary expression or another cast expression prepended with a type name enclosed in a pair of parenthesis. Note that the grammar has a rule with a right recursion here (cast_expression : '(' type_name ')' cast_expression). The grammars should try to avoid left recursion (e.g., result : result something_else), which may leads infinite recursive calls in your parser. Again, a helper function in AstFromString often implements a grammar. Please take a look at some of them to see how grammars are written to facilitate building recursive descent parsers.

The pragma in the pragmaParsing project has the following grammar (documented in *rose/projects/pragmaParsing/hcpragma.h*):

```
---------- grammar begin -----------
% 'string' means literal string to be matched
% | means alternation
hcc_pragma = '#pragma' hc_simple_part | hc_cuda_part

hc_simple_part = 'hc' 'entry'| 'suspendable' | 'entry suspendable' | 'suspendable entry'

hc_cuda_part = 'CUDA' kernel_part| place_part

kernel_part = 'kernel'

% place could be an expression
% the grammar uses assignment_expression instead of expression to disallow comma expressions
% (list of expressions connected with comma) e.g. exp1, exp2 will be parsed to be (ex1, exp2)
% otherwise.
%
place_part = assignment_expression autodim_part | dim_part

% autodim(<dim1>[, <dim2>, <dim3>, <shared_size>])
%  [ ] means optional
% , means  ',' to be simple
% assignment_expression is used to avoid parsing exp1, exp2, exp3 to be one single comma
% expression ((exp1,exp2),exp3)
autodim_part = 'autodim' '('  assignment_expression [, assignment_expression
              [, assignment_expression [, assignment_expression ] ] ]  ')'

% dim(blocksPerGrid, threadsPerBlock[, shared_size])
dim_part = 'dim' '('  assignment_expression ,  assignment_expression ,
```

```
            [ , assignment_expression ]  ')'
```

The example grammar allows a list of expressions inside a pragma. A tricky part here is that C allows single comma expression like ((exp1,exp2),exp3). We use assignment_expression to avoid parsing exp1, exp2, exp3 to be one such single comma expression. The point here is that the terms in the grammar have to be accurately mapped to formal C grammar terms. Some familiarity of formal C grammar terms, as shown at http://www.antlr.org/grammar/1153358328744/C.g, is required since helper functions have names matching the formal terms in C grammar. The assignment expressions, not expressions, are what we care about in this particular simple grammar.

33.2 AstAttribute to Store results

Once the grammar is defined with terms matching helper functions of AstFromString, a data structure is needed to store the results of parsing. It is recommended to create your data structure by inheriting AstAttribute, which can be attached to any AST nodes with location information.

As in the pragmaParsing project, we define a few classes as the following:

```
class HC_PragmaAttribute: public AstAttribute
{
    public:
        SgNode * node;
            enum hcpragma_enum pragma_type;
... };

class HC_CUDA_PragmaAttribute: public  HC_PragmaAttribute
{
  public:
    SgExpression* place_exp;
...
};
class HC_CUDA_autodim_PragmaAttribute: public HC_CUDA_PragmaAttribute
{
  public:
    SgExpression* dim1_exp;
    SgExpression* dim2_exp;
    SgExpression* dim3_exp;
    SgExpression* shared_size_exp;
...
};
```

The point is that the class is inherited from AstAttribute and it contains fields to store all terms defined in the grammar.

33.3 The AstFromString Namespace

AstFromString has a few namespace scope variables, such as:

- char ∗ c_char: this indicates the current position of the input string being parsed.
- SgNode∗ c_sgnode: a SgNode variable storing the current anchor AST node, which servers as a start point to find enclosing scopes for resolving identifiers/symbols discovered by a parser.
- SgNode ∗ c_parsed_node: a SgNode variable storing the root of an AST subtree generated by the parser.

In general, your parser should initialize c_char with the first position of an input string to be parsed. It should also set c_sgnode to be the pragma statement when you parse pragma strings, or the immediate following statement when you parse Fortran comments. The results often are stored in c_parsed_node. Your parser should timely check the results and filling your AstAttribute structure with the AST substrees for identifiers, constants, expressions, etc.

Helper functions within AstFromString include functions to parse identifiers, types, substrings, expressions. AST pieces will be generated automatically as needed. So users can focus on building their grammar and parser without doing repetitive chores of parsing common language constructs.

Take bool afs_match_assignment_expression() as an example, this function will try to match an expression that satisfies a grammar rule like:
assignment_expression : lvalue assignment_operator assignment_expression | conditional_expression .
If a successful match is found, the function returns true. In the meantime, the function builds an AST subtree to represent the matched expression and stores the subtree into the variable SgNode∗ c_sgnode.

33.4 Write your parsers using parser building blocks

rose/src/frontend/SageIII/astFromString/AstFromString.cpp has the implementation of all parser building blocks (helper functions) for a wide range of grammar rules. They can serve as examples about how to hand-write additional functions to recognize your own grammars. For example, to implement a simple grammar rule like type_qualifier : 'const' | ' volatile ', we have the following helper function:

```
 /*
     type_qualifier
     : 'const'
     | 'volatile'
     ;

*/
  bool afs_match_type_qualifier()
  {
    bool result = false;
```

```
    const char* old_char = c_char;
    if (afs_match_substr("const"))
    {
      c_parsed_node = buildConstVolatileModifier (SgConstVolatileModifier::e_const);
      result = true;
    }
    else if (afs_match_substr("volatile"))
    {
      c_parsed_node = buildConstVolatileModifier (SgConstVolatileModifier::e_volatile);
      result = true;
    }
    if (result == false)   c_char = old_char;
    return result;
  }
```

Please note that the function above tries to undo any side effects if the parsing fails. If successful, the parsed result will be stored into c_parsed_node.

Here is another example with right recursion:

```
/* Grammar is
     cast_expression
     : '(' type_name ')' cast_expression
     | unary_expression
     ;

*/
  bool afs_match_cast_expression()
  {
    bool result = false;
    const char* old_char = c_char;

    if (afs_match_unary_expression())
    {
      if (isSgExpression(c_parsed_node))
      result = true;
    }
    else if (afs_match_char('('))
    {
      if (afs_match_type_name())
      {
        SgType* t = isSgType(c_parsed_node);
        assert (t!= NULL);
        if (afs_match_char(')'))
        {
          if (afs_match_cast_expression())
          {
            SgExpression* operand = isSgExpression(c_parsed_node);
```

```
          c_parsed_node = buildCastExp(operand, t);
          result = true; // must set this!!
        }
        else
        {
          c_char = old_char;
        }
      }
      else
      {
        c_char = old_char;
      }

    }
    else
    {
      c_char = old_char;
      result = false;
    }
  }

  if (result == false)   c_char = old_char;
  return result;
}
```

sourcetree/projects/pragmaParsing/hcpragma.C gives a full example of how to use helper functions inside your own parsing functions.

33.5 Limitations

Currently, the parser building blocks support C only. Expressions parsing functions are ready to be used by users. Statement parsing is still ongoing work.

Chapter 34

Handling Comments, Preprocessor Directives, And Adding Arbitrary Text to Generated Code

What To Learn From These Examples Learn how to access existing comments and CPP directives and modify programs to include new ones. Where such comments can be automated they can be used to explain transformations or for more complex transformations using other tools designed to read the generated comments. Also included is how to add arbitrary text to the generated code (often useful for embedded system programming to support back-end compiler specific functionality).

This chapter deals with comments and preprocessor directives. These are often dropped from compiler infrastructures and ignored by make tools. ROSE takes great care to preserve all comments and preprocessor directives. Where they exist in the input code we save them (note that EDG drops them from their AST) and weave them back into the AST.

Note that *#pragma* is not a CPP directive and is part of the C and C++ grammar, thus represented explicitly in the AST (see SgPragmaDeclaration).

34.1 How to Access Comments and Preprocessor Directives

Comments and CPP directives are treated identically within ROSE and are saved as special preprocessor attributes to IR nodes within the AST. Not all IR nodes can have these specific type of attributes, only SgLocatedNodes can be associated with such preprocessor attributes. The more general *persistent attribute mechanism* within ROSE is separate from this preprocessor attribute mechanism and is available on a wider selection of IR nodes.

34.1.1 Source Code Showing How to Access Comments and Preprocessor Directives

Figure 34.1 shows an example translator which access the comments and preprocessor directives on each statement. Note that in this example the AST is traversed twice, first header files are ignored and then the full AST (including header files) are traversed (generated additional comments).

The input code is shown in figure 34.2, the output of this code is shown in figure 34.3 for the source file only. Figure 34.4 shows the same input code processed to output comments and preprocessor directives assembled from the source file *and* all header files.

34.1.2 Input to example showing how to access comments and CPP directives

Figure 34.2 shows the example input used for demonstration of how to collect comments and CPP directives.

34.1.3 Comments and CPP Directives collected from source file (skipping headers)

Figure 34.3 shows the results from the collection of comments and CPP directives within the input source file only (without -rose:collectAllCommentsAndDirectives).

34.1.4 Comments and CPP Directives collected from source file and all header files

Figure 34.4 shows the results from the collection of comments and CPP directives within the input source file and all headers (with -rose:collectAllCommentsAndDirectives).

34.2 Collecting #define C Preprocessor Directives

This example shows how to collect the #define directives as a list for later processing.

34.2.1 Source Code Showing How to Collect #define Directives

Figure 34.5 shows an example translator which access the comments and preprocessor directives on each statement. Note that in this example the AST is traversed twice, first header files are ignored and then the full AST (including header files) are traversed (generated additional comments).

The input code is shown in figure 34.6, the output of this code is shown in Figure 34.7 shows the same input code processed to output comments and preprocessor directives assembled from the source file *and* all header files.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

// Class declaration
class visitorTraversal : public AstSimpleProcessing
   {
     public:
          virtual void visit(SgNode* n);
   };

void visitorTraversal::visit(SgNode* n)
   {
  // On each node look for any comments of CPP directives
     SgLocatedNode* locatedNode = isSgLocatedNode(n);
     if (locatedNode != NULL)
        {
          AttachedPreprocessingInfoType* comments = locatedNode->getAttachedPreprocessingInfo();

          if (comments != NULL)
             {
               printf ("Found attached comments (to IR node at %p of type: %s): \n",locatedNode,locatedNode->class_name()
               int counter = 0;
               AttachedPreprocessingInfoType::iterator i;
               for (i = comments->begin(); i != comments->end(); i++)
                  {
                    printf ("          Attached Comment #%d in file %s (relativePosition=%s): classification %s :\n%s\n",
                         counter++,(*i)->get_file_info()->get_filenameString().c_str(),
                         ((*i)->getRelativePosition() == PreprocessingInfo::before) ? "before" : "after",
                         PreprocessingInfo::directiveTypeName((*i)->getTypeOfDirective()).c_str(),
                         (*i)->getString().c_str());
                  }
             }
            else
             {
               printf ("No attached comments (at %p of type: %s): \n",locatedNode,locatedNode->sage_class_name());
             }
        }
   }

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

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
  // Traverse all header files and source file (the -rose:collectAllCommentsAndDirectives
  // commandline option controls if comments and CPP directives are separately extracted
  // from header files).
  // exampleTraversal.traverse(project,preorder);
     exampleTraversal.traverseInputFiles(project,preorder);

     return 0;
   }
```

Figure 34.1: Example source code showing how to access comments.

```

// #include<stdio.h>

#define SOURCE_CODE_BEFORE_INCLUDE_A
#define SOURCE_CODE_BEFORE_INCLUDE_B
#include<inputCode_collectComments.h>
#define SOURCE_CODE_AFTER_INCLUDE_A
#define SOURCE_CODE_AFTER_INCLUDE_B

// main program: collectComments input test code
int main()
   {
     return 0;
   }
```

Figure 34.2: Example source code used as input to collection of comments and CPP directives.

```
No attached comments (at 0x2b9719e80010 of type: SgGlobal):
Found attached comments (to IR node at 0x2b9719f8a908 of type: SgFunctionDeclaration):
          Attached Comment #0 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
// #include<stdio.h>

          Attached Comment #1 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
#define SOURCE_CODE_BEFORE_INCLUDE_A

          Attached Comment #2 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
#define SOURCE_CODE_BEFORE_INCLUDE_B

          Attached Comment #3 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
#include<inputCode_collectComments.h>

          Attached Comment #4 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
#define SOURCE_CODE_AFTER_INCLUDE_A

          Attached Comment #5 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
#define SOURCE_CODE_AFTER_INCLUDE_B

          Attached Comment #6 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/am
// main program: collectComments input test code

No attached comments (at 0x2b971a047628 of type: SgFunctionParameterList):
No attached comments (at 0x2b971a258010 of type: SgFunctionDefinition):
No attached comments (at 0x2b971a2a1010 of type: SgBasicBlock):
No attached comments (at 0x1ccde310 of type: SgReturnStmt):
No attached comments (at 0x1ccf3bf0 of type: SgIntVal):
```

Figure 34.3: Output from collection of comments and CPP directives on the input source file only.

34.2.2 Input to example showing how to access comments and CPP directives

Figure 34.6 shows the example input used for demonstration of how to collect comments and CPP directives.

```
No attached comments (at 0x2b70bbeca010 of type: SgGlobal):
Found attached comments (to IR node at 0x2b70bbfb3908 of type: SgFunctionDeclaration):
          Attached Comment #0 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
// #include<stdio.h>

          Attached Comment #1 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
#define SOURCE_CODE_BEFORE_INCLUDE_A

          Attached Comment #2 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
#define SOURCE_CODE_BEFORE_INCLUDE_B

          Attached Comment #3 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
#include<inputCode_collectComments.h>

          Attached Comment #4 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
#define SOURCE_CODE_AFTER_INCLUDE_A

          Attached Comment #5 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
#define SOURCE_CODE_AFTER_INCLUDE_B

          Attached Comment #6 in file /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/t
// main program: collectComments input test code

No attached comments (at 0x2b70bc070628 of type: SgFunctionParameterList):
No attached comments (at 0x2b70bc281010 of type: SgFunctionDefinition):
No attached comments (at 0x2b70bc2ca010 of type: SgBasicBlock):
No attached comments (at 0x905d4d0 of type: SgReturnStmt):
No attached comments (at 0x9072db0 of type: SgIntVal):
```

Figure 34.4: Output from collection of comments and CPP directives on the input source file and all header files.

34.2.3 Comments and CPP Directives collected from source file and all header files

Figure 34.7 shows the results from the collection of comments and CPP directives within the input source file and all headers (with -rose:collectAllCommentsAndDirectives).

34.3 Automated Generation of Comments

Figure 34.8 shows an example of how to introduce comments into the AST which will then show up in the generated source code. The purpose for this is generally to add comments to where transformations are introduced. If the code is read by the use the generated comments can be useful in identifying, marking, and/or explaining the transformation.

This chapter presents an example translator which just introduces a comment at the top of each function. The comment includes the name of the function and indicates that the comment is automatically generated.

Where appropriate such techniques could be used to automate the generation of documentation templates in source code that would be further filled in by the used. In this case the automatically generated templates would be put into the generated source code and a patch formed between the generated source and the original source. The patch could be easily inspected and applied to the original source code to place the documentation templates into the original source. The skeleton of the documentation in the source code could been be filled in by the use. The template would have all relevant information obtained by analysis (function parameters, system functions used, security information, side-effects, anything that could come from an analysis of the source code using ROSE).

34.3.1 Source Code Showing Automated Comment Generation

Figure 34.8 shows an example translator which calls the mechanism to add a comment to the IR node representing a function declaration (SgFunctionDeclaration).

The input code is shown in figure 34.9, the output of this code is shown in figure 34.10.

34.3.2 Input to Automated Addition of Comments

Figure 34.9 shows the example input used for demonstration of an automated commenting.

34.3.3 Final Code After Automatically Adding Comments

Figure 34.10 shows the results from the addition of comments to the generated source code.

34.4 Addition of Arbitrary Text to Unparsed Code Generation

This section is different from the comment generation (section 34.3) because it is more flexible and does not introduce any formatting. It also does not use the same internal mechanism, this mechanism supports the addition of new strings or the replacement of the IR node (where the string is attached) with the new string. It is fundamentally lower level and a more powerful mechanism to support generation of tailored output where more than comments, CPP directives, or AST transformation are required. It is also much more dangerous to use.

This mechanism is expected to be used rarely and sparingly since no analysis of the AST is likely to leverage this mechanism and search for code that introduced as a transformation here. Code introduced using this mechanism is for the most part unanalyzable since it would have to be reparsed in the context of the location in the AST were it is attached. (Technically this is possible and is the subject of the existing ROSE AST Rewrite mechanism, but that is a different subject).

Figure 34.11 shows an example of how to introduce arbitrary text into the AST for output by the unparser which will then show up in the generated source code. The purpose for this is generally to add backend compiler or tool specific code generation which don't map to any formal language constructs and so cannot be represented in the AST. However, since most tools that require specialized annotations read them as comments, the mechanism in the previous section 34.3 may be more appropriate. It is because this is not always that case that we have provide this more general mechanism (often useful for embedded system compilers).

34.4.1 Source Code Showing Automated Arbitrary Text Generation

Figure 34.11 shows an example translator which calls the mechanism to add a arbitrary text to the IR node representing a function declaration (SgFunctionDeclaration).

The input code is shown in figure 34.12, the output of this code is shown in figure 34.13.

34.4.2 Input to Automated Addition of Arbitrary Text

Figure 34.12 shows the example input used for demonstration of the automated introduction of text via the unparser.

34.4.3 Final Code After Automatically Adding Arbitrary Text

Figure 34.13 shows the results from the addition of arbitrary text to the generated source code.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

// Build a synthesized attribute for the tree traversal
class SynthesizedAttribute
   {
     public:
       // List of #define directives (save the PreprocessingInfo objects
       // so that we have all the source code position information).
          list<PreprocessingInfo*> accumulatedList;

          void display() const;
   };

void
SynthesizedAttribute::display() const
   {
     list<PreprocessingInfo*>::const_iterator i = accumulatedList.begin();
     while (i != accumulatedList.end())
        {
          printf ("CPP define directive = %s \n",(*i)->getString().c_str());
          i++;
        }
   }

class visitorTraversal : public AstBottomUpProcessing<SynthesizedAttribute>
   {
     public:
       // virtual function must be defined
          virtual SynthesizedAttribute evaluateSynthesizedAttribute (
                       SgNode* n, SynthesizedAttributesList childAttributes );
   };

SynthesizedAttribute
visitorTraversal::evaluateSynthesizedAttribute ( SgNode* n, SynthesizedAttributesList childAttributes )
   {
     SynthesizedAttribute localResult;

  // printf ("In evaluateSynthesizedAttribute(n = %p = %s) \n",n,n->class_name().c_str());

  // Build the list from children (in reverse order to preserve the final ordering)
     for (SynthesizedAttributesList::reverse_iterator child = childAttributes.rbegin(); child != childAttribut
        {
          localResult.accumulatedList.splice(localResult.accumulatedList.begin(),child->accumulatedList);
        }

  // Add in the information from the current node
     SgLocatedNode* locatedNode = isSgLocatedNode(n);
     if (locatedNode != NULL)
        {
          AttachedPreprocessingInfoType* commentsAndDirectives = locatedNode->getAttachedPreprocessingInfo();

          if (commentsAndDirectives != NULL)
             {
            // printf ("Found attached comments (to IR node at %p of type: %s): \n",locatedNode,locatedNode->c
            // int counter = 0;

            // Use a reverse iterator so that we preserve the order when using push_front to add each directiv
               AttachedPreprocessingInfoType::reverse_iterator i;
               for (i = commentsAndDirectives->rbegin(); i != commentsAndDirectives->rend(); i++)
                  {
                 // The different classifications of comments and directives are in ROSE/src/frontend/SageIII/
                    if ((*i)->getTypeOfDirective() == PreprocessingInfo::CpreprocessorDefineDeclaration)
                       {
#if 0
                         printf ("          Attached Comment #%d in file %s (relativePosition=%s): classificat
                              counter++,(*i)->get_file_info()->get_filenameString().c_str(),
                              ((*i)->getRelativePosition() == PreprocessingInfo::before) ? "before" : "after",
                              PreprocessingInfo::directiveTypeName((*i)->getTypeOfDirective()).c_str(),
                              (*i)->getString().c_str());
#endif
                      // use push_front() to end up with source ordering of final list of directives
                         localResult.accumulatedList.push_front(*i);
                       }
                  }
             }
        }

  // printf ("localResult after adding current node info \n");
  // localResult.display();
```

```

#define JUST_A_MACRO just_a_macro

#define ANOTHER_MACRO another_macro
```

Figure 34.6: Example source code used as input to collection of comments and CPP directives.

```
CPP define directive = #define max(a,b) ((a) > (b) ? (a) : (b))

CPP define directive = #define maxint(a,b) ({int _a = (a), _b = (b); _a > _b ? _a : _b; })

CPP define directive = #define SOURCE_CODE_BEFORE_INCLUDE_A

CPP define directive = #define SOURCE_CODE_BEFORE_INCLUDE_B

CPP define directive = #define SOURCE_CODE_AFTER_INCLUDE_A

CPP define directive = #define SOURCE_CODE_AFTER_INCLUDE_B
```

Figure 34.7: Output from collection of comments and CPP directives on the input source file and all header files.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

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

void visitorTraversal::visit(SgNode* n)
   {
     SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(n);
     if (functionDeclaration != NULL)
        {
          string comment = string("Auto-comment function name: ") +
                           functionDeclaration->get_name().str() +
                           " is now a commented function";

       // Note that this function will add the "//" or "/* */" comment syntax as required for C or C++, or Fo
          SageInterface::attachComment(functionDeclaration,comment);
        }

     SgValueExp* valueExp = isSgValueExp(n);
     if (valueExp != NULL)
        {
       // Check if there is an expression tree from the original unfolded expression.
       // This is a trivial example ouf the output of an analysis result.
          string comment = string("Auto-comment value: ") +
               ((valueExp->get_originalExpressionTree() != NULL) ?
                    " this IS a constant folded value" : " this is NOT a constant folded value");

          SageInterface::attachComment(valueExp,comment);
        }

   }

// Typical main function for ROSE translator
int main( int argc, char * argv[] )
   {
  // Build the AST used by ROSE
     SgProject* project = frontend(argc,argv);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project,preorder);

     return backend(project);
   }
```

Figure 34.8: Example source code showing how automate comments.

```

int
foo()
   {
     int x = 2;
     x += 3 + 4;
     return x;
   }
```

Figure 34.9: Example source code used as input to automate generation of comments.

```
// Auto-comment function name: foo is now a commented function

int foo()
{
  int x =
// Auto-comment value:  this is NOT a constant folded value
2;
  x +=
// Auto-comment value:  this is NOT a constant folded value
3 +
// Auto-comment value:  this is NOT a constant folded value
4;
  return x;
}
```

Figure 34.10: Output of input code after automating generation of comments.

```
// Example ROSE Translator: used within ROSE/tutorial

#include "rose.h"

using namespace std;

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

void visitorTraversal::visit(SgNode* n)
   {
     SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(n);
     if (functionDeclaration != NULL)
        {
       // This is an example of a XYZ tool specific annotation
          string compilerSpecificDirective = "\n#if XYZ_TOOL \n
\"builtin\"\n#endif\n";
          SageInterface::addTextForUnparser(functionDeclaration,compilerSpecificDirective,AstUnparseAttribute:
        }

     SgValueExp* valueExp = isSgValueExp(n);
     if (valueExp != NULL)
        {
       // Add a backend specific compiler directive
          string compilerSpecificDirective = "\n#if CRAY \n
cray_specific_attribute \n#endif\n";
          SageInterface::addTextForUnparser(valueExp,compilerSpecificDirective,AstUnparseAttribute::e_before);
        }

   }

// Typical main function for ROSE translator
int main( int argc, char * argv[] )
   {
  // Build the AST used by ROSE
     SgProject* project = frontend(argc,argv);

  // Build the traversal object
     visitorTraversal exampleTraversal;

  // Call the traversal starting at the project node of the AST
     exampleTraversal.traverseInputFiles(project,preorder);

     return backend(project);
   }
```

Figure 34.11: Example source code showing how automate the introduction of arbitrary text.

```
int
foo()
   {
     int x = 42;
     return x;
   }
```

Figure 34.12: Example source code used as input to automate generation of arbitrary text.

```
#if XYZ_TOOL
    "builtin"
#endif

int foo()
{
  int x =
#if CRAY
    cray_specific_attribute
#endif
42;
  return x;
}
```

Figure 34.13: Output of input code after automating generation of arbitrary text.

Chapter 35

Partial Redundancy Elimination (PRE)

Figure 35.1 shows an example of how to call the Partial Redundancy Elimination (PRE) implemented by Jeremiah Willcock. This transformation is useful for cleaning up code generated from other transformations (used in Qing's loop optimizations).

35.1 Source Code for example using PRE

Figure 35.1 shows an example translator which calls the PRE mechanism.

The input code is shown in figure 35.2, the output of this code is shown in figure 35.3.

```
// Example translator demontrating Partial Redundancy Elimination (PRE).

#include "rose.h"
#include "CommandOptions.h"

int main (int argc, char* argv[])
   {
  // Build the project object (AST) which we will fill up with multiple files and use as a
  // handle for all processing of the AST(s) associated with one or more source files.
     std::vector<std::string> l = CommandlineProcessing::generateArgListFromArgcArgv(argc, argv);

     CmdOptions::GetInstance()->SetOptions(argc, argv);
     SgProject* project = frontend(l);

     PRE::partialRedundancyElimination(project);

     return backend(project);
   }
```

Figure 35.1: Example source code showing how use Partial Redundancy Elimination (PRE).

35.2 Input to Example Demonstrating PRE

Figure 35.2 shows the example input used for demonstration of Partial Redundancy Elimination (PRE) transformation.

```
// Program, based on example in Knoop et al ("Optimal code motion: theory and
// practice", ACM TOPLAS 16(4), 1994, pp. 1117-1155, as cited in Paleri et al
// (see pre.C)), converted to C++

int unknown(); // ROSE bug: including body "return 0;" here doesn't work

void foo() {
  int a, b, c, x, y, z, w;

  if (unknown()) {
    y = a + b;
    a = c;
 // Added by Jeremiah Willcock to test local PRE
    w = a + b;
    a = b;
    x = a + b;
    w = a + b;
    a = c;
 // End of added part
    x = a + b;
  }

  if (unknown()) {
    while (unknown()) {y = a + b;}
  } else if (unknown()) {
    while (unknown()) {}
    if (unknown()) {y = a + b;} else {goto L9;} // FIXME: the PRE code crashes if this isn't in a block
  } else {
    goto L10;
  }

  z = a + b;
  a = c;

  L9: x = a + b;

  L10: 0; // ROSE bug: using return; here doesn't work
}

int unknown() {
  0; // Works around ROSE bug
  return 0;
}

int main(int, char**) {
  foo();
  return 0;
}
```

Figure 35.2: Example source code used as input to program to the Partial Redundancy Elimination (PRE) transformation.

35.3 Final Code After PRE Transformation

Figure 35.3 shows the results from the use of PRE on an the example input code.

```
// Program, based on example in Knoop et al ("Optimal code motion: theory and
// practice", ACM TOPLAS 16(4), 1994, pp. 1117-1155, as cited in Paleri et al
// (see pre.C)), converted to C++
// ROSE bug: including body "return 0;" here doesn't work
int unknown();

void foo()
{
// Partial redundancy elimination: cachevar__1 is a cache of (a + b)
  int cachevar__1;
  int a;
  int b;
  int c;
  int x;
  int y;
  int z;
  int w;
  if ((unknown())) {
    y = (a + b);
    a = c;
// Added by Jeremiah Willcock to test local PRE
    w = (a + b);
    a = b;
    cachevar__1 = (a + b);
    x = cachevar__1;
    w = cachevar__1;
    a = c;
// End of added part
    x = (a + b);
  }
  else {
  }
  if ((unknown())) {
    cachevar__1 = (a + b);
    while((unknown())){
      y = cachevar__1;
    }
  }
  else if ((unknown())) {
    while((unknown())){
    }
// FIXME: the PRE code crashes if this isn't in a block
    if ((unknown())) {
      cachevar__1 = (a + b);
      y = cachevar__1;
    }
    else {
      goto L9;
    }
  }
  else {
    goto L10;
  }
  z = cachevar__1;
  a = c;
  L9:
  x = (a + b);
// ROSE bug: using return; here doesn't work
  L10:
  0;
}

int unknown()
{
// Works around ROSE bug
  0;
  return 0;
}

int main(int ,char **)
{
  foo();
  return 0;
}
```

Figure 35.3: Output of input code after Partial Redundancy Elimination (PRE) transformation.

Chapter 36

Calling the Inliner

Figure 36.1 shows an example of how to use the inline mechanism. This chapter presents an example translator to to inlining of function calls where they are called. Such transformations are quite complex in a number of cases (one case is shown in the input code; a function call in a for loop conditional test). The details of functionality are hidden from the user and a high level interface is provided.

36.1 Source Code for Inliner

Figure 36.1 shows an example translator which calls the inliner mechanism. The code is designed to only inline up to ten functions. the list of function calls is recomputed after any function call is successfully inlined.

The input code is shown in figure 36.2, the output of this code is shown in figure 36.3.

36.2 Input to Demonstrate Function Inlining

Figure 36.2 shows the example input used for demonstration of an inlining transformation.

36.3 Final Code After Function Inlining

Figure 36.3 shows the results from the inlining of three function calls. The first two function calls are the same, and trivial. The second function call appears in the test of a for loop and is more complex.

```
// Example demonstrating function inlining (maximal inlining, up to preset number of inlinings).

#include "rose.h"

using namespace std;

// This is a function in Qing's AST interface
void FixSgProject(SgProject& proj);

int main (int argc, char* argv[])
   {
  // Build the project object (AST) which we will fill up with multiple files and use as a
  // handle for all processing of the AST(s) associated with one or more source files.
     SgProject* project = new SgProject(argc,argv);

  // DQ (7/20/2004): Added internal consistancy tests on AST
     AstTests::runAllTests(project);

     bool modifiedAST = true;
     int  count    = 0;

  // Inline one call at a time until all have been inlined.  Loops on recursive code.
     do {
          modifiedAST = false;

       // Build a list of functions within the AST
          Rose_STL_Container<SgNode*> functionCallList = NodeQuery::querySubTree (project,V_SgFunctionCallExp)

       // Loop over all function calls
       // for (list<SgNode*>::iterator i = functionCallList.begin(); i != functionCallList.end(); i++)
          Rose_STL_Container<SgNode*>::iterator i = functionCallList.begin();
          while (modifiedAST == false && i != functionCallList.end())
             {
               SgFunctionCallExp* functionCall = isSgFunctionCallExp(*i);
               ROSE_ASSERT(functionCall != NULL);

            // Not all function calls can be inlined in C++, so report if successful.
               bool sucessfullyInlined = doInline(functionCall);

               if (sucessfullyInlined == true)
                  {
                 // As soon as the AST is modified recompute the list of function
                 // calls (and restart the iterations over the modified list)
                    modifiedAST = true;
                  }
                 else
                  {
                    modifiedAST = false;
                  }

            // Increment the list iterator
               i++;
             }

       // Quite when we have ceased to do any inline transformations
       // and only do a predefined number of inline transformations
          count++;
        }
     while(modifiedAST == true && count < 10);

  // Call function to postprocess the AST and fixup symbol tables
     FixSgProject(*project);

  // Rename each variable declaration
     renameVariables(project);

  // Fold up blocks
     flattenBlocks(project);

  // Clean up inliner-generated code
     cleanupInlinedCode(project);

  // Change members to public
     changeAllMembersToPublic(project);

  // DQ (3/11/2006): This fails so the inlining, or the AST Interface
  // support, needs more work even though it generated good code.
  // AstTests::runAllTests(project);

     return backend(project);
   }
```

```
// This test code is a combination of pass1 and pass7, selected somewhat randomly
// from Jeremiah's test code of his inlining transformation from summer 2004.

int x = 0;

// Function it increment "x"
void incrementX()
   {
     x++;
   }

int foo()
   {
     int a = 0;
     while (a < 5)
        {
          ++a;
        }

     return a + 3;
   }

int main(int, char**)
   {
  // Two trival function calls to inline
     incrementX();
     incrementX();

  // Somthing more interesting to inline
     for (; foo() < 7;)
        {
          x++;
        }

     return x;
   }
```

Figure 36.2: Example source code used as input to program to the inlining transformation.

```
// This test code is a combination of pass1 and pass7, selected somewhat randomly
// from Jeremiah's test code of his inlining transformation from summer 2004.
int x = 0;
// Function it increment "x"

void incrementX()
{
  x++;
}

int foo()
{
  int a__0 = 0;
  while(a__0 < 5){
    ++a__0;
  }
  return a__0 + 3;
}

int main(int ,char **)
{
  x++;
  x++;
// Somthing more interesting to inline
  for (; true; ) {
    int a__1 = 0;
    while(a__1 < 5){
      ++a__1;
    }
    int rose_temp__7__0 = a__1 + 3;
    bool rose__temp__2 = (bool )(rose_temp__7__0 < 7);
    if (!rose__temp__2) {
      break;
    }
    else {
    }
    x++;
  }
  return x;
}
```

Figure 36.3: Output of input code after inlining transformations.

Chapter 37

Using the AST Outliner

Outlining is the process of replacing a block of consecutive statements with a function call to a new function containing those statements. Conceptually, outlining the inverse of inlining (Chapter 36). This chapter shows how to use the basic experimental outliner implementation included in the ROSE projects directory.

There are two basic ways to use the outliner. The first is a "user-level" method, in which you may use a special pragma to mark outline targets in the input program, and then call a high-level driver routine to process these pragmas. You can also use command line option to specify outlining targets using abstract handle strings (detailed in Chapter 46). The second method is to call "low-level" outlining routines that operate directly on AST nodes. After a brief example of what the outliner can do and a discussion of its limits (Sections 37.1–37.2), we discuss each of these methods in Sections 37.3 and 37.5, respectively.

37.1 An Outlining Example

Figure 37.1 shows a small program with a pragma marking the outline target, a nested for loop, and Figure 37.2 shows the result. The outliner extracts the loop and inserts it into the body of a new function, and inserts a call to that function. The outlined code's input and output variables are wrapped up as parameters to this function. We make the following observations about this output.

Placement and forward declarations. The function itself is placed, by default, at the end of the input file to guarantee that it has access to all of the same declarations that were available at the outline target site. The outliner inserts any necessary forward declarations as well, including any necessary `friend` declarations if the outline target appeared in a class member function.

Calling convention. The outliner generates a C-callable function (`extern ‘‘C’’`, with pointer arguments). This design choice is motivated by our need to use the outliner to extract code into external, dynamically loadable library modules.

37.2 Limitations of the Outliner

The main limitation of the outliner implementation is that it can only outline single SgStatement nodes. However, since an SgStatement node may be a block (*i.e.*, an SgBasicBlock node), a "single statement" may actually comprise a sequence of complex statements.

The rationale for restricting to single SgStatement nodes is to avoid subtly changing the program's semantics when outlining code. Consider the following example, in which we wish to outline the middle 3 lines of executable code.

```
int x = 5;
// START outlining here.
foo (x);
Object y (x);
y.foo ();
// STOP outlining here.
y.bar ();
```

This example raises a number of issues. How should an outliner handle the declaration of `y`, which constructs an object in local scope? It cannot just cut-and-paste the declaration of `y` to the body of the new outlined function because that will change its scope and lifetime, rendering the call to `y.bar()` impossible. Additionally, it may be unsafe to move the declaration of `y` so that it precedes the outlined region because the constructor call may have side-effects that could affect the execution of `foo(x)`. It is possible to heap-allocate `y` inside the body of the

```
namespace N
{
  class A
  {
    int foo (void) const { return 7; }
    int bar (void) const { return foo () / 2; }
  public:
    int biz (void) const
    {
      int result = 0;
#pragma rose_outline
      for (int i = 1; i <= foo (); i++)
        for (int j = 1; j <= bar (); j++)
          result += i * j;
      return result;
    }
  };
}

extern "C" int printf (const char* fmt, ...);

int main ()
{
  N::A x;
  printf ("%d\n", x.biz ()); // Prints '168'
  return 0;
}
```

Figure 37.1: inputCode_OutlineLoop.cc: Sample input program. The `#pragma` directive marks the nested for loop for outlining.

```
extern "C" void OUT__1__13785__ (int *resultp__, const void *this__ptr__p__);
namespace N
{

  class A
  {
  public:friend void ::OUT__1__13785__ (int *resultp__,
                                        const void *this__ptr__p__);

  private:  inline int foo () const
    {
      return 7;
    }

    inline int bar () const
    {
      return (this)->foo () / 2;
    }

  public:  inline int biz () const
    {
// //A declaration for this pointer
      const class A *this__ptr__ = this;
      int result = 0;
        OUT__1__13785__ (&result, &this__ptr__);
        return result;
    }
  }
   ;
}
extern "C"
{
  int printf (const char *fmt, ...);
}

int main ()
{
  class N::A x;
// Prints '168'
  printf ("%d\n", x.biz ());
  return 0;
}

extern "C" void OUT__1__13785__ (int *resultp__, const void *this__ptr__p__)
{
  int &result = *((int *) resultp__);
  const class N::A * &this__ptr__ = *((const class N::A **) this__ptr__p__);
  for (int i = 1; i <= this__ptr__->foo (); i++)
    for (int j = 1; j <= this__ptr__->bar (); j++)
      result += (i * j);
}
```

Figure 37.2: rose_outlined-inputCode_OutlineLoop.cc: The nested for loop of Figure 37.1 has been outlined.

outlined function so that it can be returned to the caller and later freed, but it is not clear if changing `y` from a stack-allocated variable to a heap-allocated one will always be acceptable, particularly if the developer of the original program has, for instance, implemented a customized memory allocator. Restricting outlining to well-defined SgStatement objects avoids these issues. It is possible to build a "higher-level" outliner that extends the outliner's basic infrastructure to handle these and other issues.

The outliner cannot outline all possible SgStatement nodes. However, the outliner interface provides a routine, `outliner::isOutlineable(s)`, for testing whether an SgStatement object `s` is known to satisfy the outliner's preconditions (see Section 37.5 for details).

37.3 User-Directed Outlining *via* Pragmas

Figure 37.3 shows the basic translator, `outline`, that produces Figure 37.2 from Figure 37.1. This translator extends the identity translator with an include directive on line 5 of Figure 37.3, and a call to the outliner on line 16. All outliner routines live in the `Outliner` namespace. Here, the call to `Outliner::outlineAll (proj)` on line 16 traverses the AST, looks for `#pragma rose_outline` directives, outlines the SgStatement objects to which each pragma is attached, and returns the number of outlined objects.

A slightly lower-level outlining primitive. The `Outliner::outlineAll()` routine is a wrapper around calls to a simpler routine, `Outliner::outline()`, that operates on pragmas:

```
Outliner::Result   Outliner::outline (SgPragmaDeclaration* s);
```

Given a pragma statement AST node `s`, this routine checks if `s` is a `rose_outline` directive, and if so, outlines the statement with which `s` is associated. It returns a `Outliner::Result` object, which is simply a structure that points to (a) the newly generated outlined function and (b) the statement corresponding to the new function call (*i.e.*, the outlined function call-site). See `Outliner.hh` or the ROSE Programmer's Reference for more details.

The `Outliner::outlineAll()` wrapper. The advantage of using the wrapper instead of the lower-level primitive is that the wrapper processes the pragmas in an order that ensures the outlining can be performed correctly in-place. This order is neither a preorder nor a postorder traversal, but in fact a "reverse" preorder traversal; refer to the wrapper's documentation for an explanation.

37.4 Outlining via Abstract Handles

The ROSE AST outliner also allows users to specify outlining targets using abstract handles (details are given in Chapter 46) without relying on planting pragmas into the source code. For the translator (e.g. named `outline`) built from the source shown in Figure 37.3, it accepts a command line option in a form of *-rose:outline:abstract_handle handle_string*. The `outline` program is able to locate a language construct matching the handle string within an input source file and then outline the construct.

For example, a handle string *"ForStatement¡position,12¿"* will tell the outliner to outline the for loop at source position line 12.
Another handle, *"FunctionDeclaration¡name,initialize¿::ForStatement¡numbering,2¿"* indicates that the outlining target is the second loop within a function named *initializer*. Figure 37.5 shows the outlining results using the first handle(*"ForStatement¡position,12¿"*) from an input source file (shown in Figure 37.4). Figure 37.6 shows the results using the second handle string for the same input.

```
//! outline.cc: Demonstrates the pragma-interface of the Outliner.
#include <rose.h>
#include <iostream>

#include <Outliner.hh>
#include <vector>
#include <string>

using namespace std;

int
main (int argc, char* argv[])
{
  //! Accepting command line options to the outliner
  vector<string> argvList(argv, argv+argc);
  Outliner::commandLineProcessing(argvList);

  SgProject* proj = frontend (argvList);
  ROSE_ASSERT (proj);

  cerr << "[Outlining...]" << endl;
  size_t count = Outliner::outlineAll (proj);

  cerr << "  [Processed " << count << " outline directives.]" << endl;
  return backend (proj);
}
```

Figure 37.3: outline.cc: A basic outlining translator, which generates Figure 37.2 from Figure 37.1. This outliner relies on the high-level driver, `Outliner::outlineAll()`, which scans the AST for outlining pragma directives (`#pragma rose_outline`) that mark outline targets.

```
#define MSIZE 500
int n,m,mits;
double tol,relax=1.0,alpha=0.0543;
double u[MSIZE][MSIZE],f[MSIZE][MSIZE],uold[MSIZE][MSIZE];
double dx,dy;

void initialize( )
{
  int i,j, xx,yy;
  dx = 2.0 / (n-1);
  dy = 2.0 / (m-1);
  for (i=0;i<n;i++)
    for (j=0;j<m;j++)
    {
      xx =(int)( -1.0 + dx * (i-1));
      yy = (int)(-1.0 + dy * (j-1));
      u[i][j] = 0.0;
      f[i][j] = -1.0*alpha *(1.0-xx*xx)*(1.0-yy*yy)\
                - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy);
    }
}
```

Figure 37.4: inputCode_OutlineLoop2.c: Sample input program without pragmas.

```
#define MSIZE 500
int n;
int m;
int mits;
double tol;
double relax = 1.0;
double alpha = 0.0543;
double u[500UL][500UL];
double f[500UL][500UL];
double uold[500UL][500UL];
double dx;
double dy;
void OUT__1__11890__(int *ip__,int *jp__,int *xxp__,int *yyp__);

void initialize()
{
  int i;
  int j;
  int xx;
  int yy;
  dx = (2.0 / (n - 1));
  dy = (2.0 / (m - 1));
  OUT__1__11890__(&i,&j,&xx,&yy);
}

void OUT__1__11890__(int *ip__,int *jp__,int *xxp__,int *yyp__)
{
  for ( *ip__ = 0; *ip__ < n; ( *ip__)++)
    for ( *jp__ = 0; *jp__ < m; ( *jp__)++) {
       *xxp__ = ((int )(-1.0 + (dx * ( *ip__ - 1))));
       *yyp__ = ((int )(-1.0 + (dy * ( *jp__ - 1))));
      u[ *ip__][ *jp__] = 0.0;
      f[ *ip__][ *jp__] = (((((-1.0 * alpha) * (1.0 - ( *xxp__ *
*xxp__))) * (1.0 - ( *yyp__ * *yyp__))) - (2.0 * (1.0 - ( *xxp__ *
*xxp__)))) - (2.0 * (1.0 - ( *yyp__ * *yyp__))));
    }
}
```

Figure 37.5: rose_inputCode_OutlineLoop2.c: The loop at line 12 of Figure 37.12 has been outlined.

37.5 Calling Outliner Directly on AST Nodes

The preceding examples rely on the outliner's `#pragma` interface to identify outline targets. In this section, we show how to call the outliner directly on SgStatement nodes from within your translator.

Figure 37.7 shows an example translator that finds all `if` statements and outlines them. A sample input appears in Figure 37.8, with the corresponding output shown in Figure 37.9. Notice that valid preprocessor control structure is accounted for and preserved in the output.

The translator has two distinct phases. The first phase selects all *outlineable* if-statements, using the `CollectOutlineableIfs` helper class. This class produces a list that stores the targets in an order appropriate for outlining them in-place. The second phase iterates over the list of statements and outlines each one. The rest of this section explains these phases, as well as various aspects of the sample input and output.

```c
#define MSIZE 500
int n;
int m;
int mits;
double tol;
double relax = 1.0;
double alpha = 0.0543;
double u[500UL][500UL];
double f[500UL][500UL];
double uold[500UL][500UL];
double dx;
double dy;
void OUT__1__11890__(int i,int *jp__,int *xxp__,int *yyp__);

void initialize()
{
  int i;
  int j;
  int xx;
  int yy;
  dx = (2.0 / (n - 1));
  dy = (2.0 / (m - 1));
  for (i = 0; i < n; i++)
    OUT__1__11890__(i,&j,&xx,&yy);
}

void OUT__1__11890__(int i,int *jp__,int *xxp__,int *yyp__)
{
  for ( *jp__ = 0;  *jp__ < m; ( *jp__)++) {
     *xxp__ = ((int )(-1.0 + (dx * (i - 1))));
     *yyp__ = ((int )(-1.0 + (dy * ( *jp__ - 1))));
    u[i][ *jp__] = 0.0;
    f[i][ *jp__] = (((((-1.0 * alpha) * (1.0 - ( *xxp__ *  *xxp__))) * (1.0 - ( *yyp__ *
*yyp__))) - (2.0 * (1.0 - ( *xxp__ *  *xxp__)))) - (2.0 * (1.0 - ( *yyp__ *
*yyp__))));
  }
}
```

Figure 37.6: rose_inputCode_OutlineLoop2b.c: The 2nd loop within a function named initialize-from Figure 37.12 has been outlined.

37.5.1 Selecting the *outlineable* `if` statements

Line 45 of Figure 37.7 builds a list, `ifs` (declared on line 44), of outlineable if-statements. The helper class, `CollectOutlineableIfs` in lines 12–35, implements a traversal to build this list. Notice that a node is inserted into the target list only if it satisfies the outliner's preconditions; this check is the call to `Outliner::isOutlineable()` on line 28.

The function `Outliner::isOutlineable()` also accepts an optional second boolean parameter (not shown). When this parameter is true and the statement cannot be outlined, the check will print an explanatory message to standard error. Such messages are useful for discovering why the outliner will not outline a particular statement. The default value of this parameter is `false`.

37.5.2 Properly ordering statements for in-place outlining

Each call to `Outliner::outline(*i)` on line 50 of Figure 37.7 outlines a target if-statement `*i` in `if_targets`. However, in order for these statements to be outlined in-place, it is essential to

```
// outlineIfs.cc: Calls Outliner directly to outline if statements.
#include <rose.h>
#include <iostream>
#include <set>
#include <list>

#include <Outliner.hh>

using namespace std;

// Traversal to gather all outlineable SgIfStmt nodes.
class CollectOutlineableIfs : public AstSimpleProcessing
{
public:
  // Container of list statements in ``outlineable'' order.
  typedef list<SgIfStmt *> IfList_t;

  // Call this routine to gather the outline targets.
  static void collect (SgProject* p, IfList_t& final)
  {
    CollectOutlineableIfs collector (final);
    collector.traverseInputFiles (p, postorder);
  }

  virtual void visit (SgNode* n)
  {
    SgIfStmt* s = isSgIfStmt (n);
    if (Outliner::isOutlineable (s))
      final_targets_.push_back (s);
  }

private:
  CollectOutlineableIfs (IfList_t& final) : final_targets_ (final) {}
  IfList_t& final_targets_; // Final list of outline targets.
};

//===================================================================
int main (int argc, char* argv[])
{
  SgProject* proj = frontend (argc, argv);
  ROSE_ASSERT (proj);

#if 1
  // Build a set of outlineable if statements.
  CollectOutlineableIfs::IfList_t ifs;
  CollectOutlineableIfs::collect (proj, ifs);

  // Outline them all.
  for (CollectOutlineableIfs::IfList_t::iterator i = ifs.begin ();
       i != ifs.end (); ++i)
    Outliner::outline (*i);
#else
  printf ("Skipping outlining due to recent move from std::list to std::vector in ROSE \n");
#endif

  // Unparse
  return backend (proj);
}
```

Figure 37.7: outlineIfs.cc: A lower-level outlining translator, which calls `Outliner::outline()` directly on SgStatement nodes. This particular translator outlines all SgIfStmt nodes.

outline the statements in the proper order.

```
#include <iostream>

using namespace std;

#define LEAP_YEAR 0

int main (int argc, char* argv[])
{
  for (int i = 1; i < argc; ++i)
    {
      string month (argv[i]);
      size_t days = 0;
      if (month == "January"
          || month == "March"
          || month == "May"
          || month == "July"
          || month == "August"
          || month == "October"
          || month == "December")
        days = 31;
#if LEAP_YEAR
      else if (month == "February")
        days = 29;
#else
      else if (month == "February")
        days = 28;
#endif
      else if (month == "April"
               || month == "June"
               || month == "September"
               || month == "November")
        days = 30;
      cout << argv[i] << " " << days << endl;
    }
  return 0;
}
```

Figure 37.8: inputCode_Ifs.cc: Sample input program, without explicit outline targets specified using `#pragma rose_outline`, as in Figures 37.1 and 37.12.

The postorder traversal implemented by the helper class, `CollectOutlineableIfs`, produces the correct ordering. To see why, consider the following example code:

```
if (a) // [1]
{
  if (b) foo (); // [2]
}
else if (c) // [3]
{
  if (d) bar (); // [4]
}
```

The corresponding AST is (roughly)

```
        SgIfStmt:[1]
       /           \
      /             \
SgIfStmt:[2]    SgIfStmt:[3]
```

```
            |
      SgIfStmt:[4]
```

The postorder traversal—2, 4, 3, 1—ensures that child if-statements are outlined before their parents.

```
#include <iostream>
using namespace std;
#define LEAP_YEAR 0
extern "C" void OUT__1__11083__(void *monthp__,size_t *daysp__);
extern "C" void OUT__2__11083__(void *monthp__,size_t *daysp__);
extern "C" void OUT__3__11083__(void *monthp__,size_t *daysp__);

int main(int argc,char *argv[])
{
  for (int i = 1; i < argc; ++i) {
    std::string month(argv[i]);
    size_t days = 0;
    OUT__3__11083__(&month,&days);
    (( *(&std::cout)<<argv[i]<<" ") << days) << std::endl< char
, std::char_traits< char  >  > ;
  }
  return 0;
}

extern "C" void OUT__1__11083__(void *monthp__,size_t *daysp__)
{
  std::string &month =  *((std::string *)monthp__);
  size_t &days =  *((size_t *)daysp__);
  if (((month=="April" || month=="June") || month=="September") || month=="November")
    days = 30;
}

extern "C" void OUT__2__11083__(void *monthp__,size_t *daysp__)
{
  std::string &month =  *((std::string *)monthp__);
  size_t &days =  *((size_t *)daysp__);
#if 1 /* #if LEAP_YEAR ... #else */
;
  if (month=="February")
    days = 28;
  else
#endif
    OUT__1__11083__(&month,&days);
}

extern "C" void OUT__3__11083__(void *monthp__,size_t *daysp__)
{
  std::string &month =  *((std::string *)monthp__);
  size_t &days =  *((size_t *)daysp__);
  if ((((((month=="January" || month=="March") || month=="May") || month=="July") || month=="August") || month
    days = 31;
  else
#if LEAP_YEAR
#else
{
#endif /* #if LEAP_YEAR ... #else  */
    OUT__2__11083__(&month,&days);
  }
}
```

Figure 37.9: rose_inputCode_Ifs.cc: Figure 37.8, after outlining using the translator in Figure 37.7.

37.6 Outliner's Preprocessing Phase

Internally, the outliner implementation itself has two distinct phases. The first is a *preprocessing phase*, in which an arbitrary outlineable target is placed into a canonical form that is relatively simple to extract. The second phase then creates the outlined function, replacing the original target with a call to the outlined function. It is possible to run just the preprocessing phase, which is useful for understanding or even debugging the outliner implementation.

```
// outlinePreproc.cc: Shows the outliner's preprocessor-only phase.
#include <rose.h>
#include <iostream>

#include <Outliner.hh>

using namespace std;

int
main (int argc, char* argv[])
{
  SgProject* proj = frontend (argc, argv);
  ROSE_ASSERT (proj);

#if 1
  cerr << "[Running outliner's preprocessing phase only...]" << endl;
  size_t count = Outliner::preprocessAll (proj);
  cerr << "  [Processed " << count << " outline directives.]" << endl;
#else
  printf ("Skipping outlining due to recent move from std::list to std::vector in ROSE \n");
#endif

  cerr << "[Unparsing...]" << endl;
  return backend (proj);
}
```

Figure 37.10: outlinePreproc.cc: The basic translator of Figure 37.3, modified to execute the Outliner's preprocessing phase only. In particular, the original call to `Outliner::outlineAll()` has been replaced by a call to `Outliner::preprocessAll()`.

To call just the preprocessor, simply replace a call to `Outliner::outlineAll(s)` or `Outliner::outline(s)` with a call to `Outliner::preprocessAll(s)` or `Outliner::preprocess(s)`, respectively. The translator in Figure 37.10 modifies the translator in Figure 37.3 in this way to create a preprocessing-only translator.

The preprocessing phase consists of a sequence of initial analyses and transformations that the outliner performs in order to put the outline target into a particular canonical form. Roughly speaking, this form is an enclosing SgBasicBlock node, possibly preceded or followed by start-up and tear-down code. Running just the preprocessing phase on Figure 37.1 produces the output in Figure 37.11. In this example, the original loop is now enclosed in two additional SgBasicBlocks (Figure 37.11, lines 24–35), the outermost of which contains a declaration that shadows the object's `this` pointer, replacing all local references to `this` with the new shadow pointer. In this case, this initial transformation is used by the main underlying outliner implementation to explicitly identify all references to the possibly implicit references to `this`.

The preprocessing phase is more interesting in the presence of non-local control flow outside the outline target. Consider Figure 37.12, in which the outline target contains two `break`

```
namespace N
{

  class A
  {

  private:inline int foo () const
    {
      return 7;
    }

    inline int bar () const
    {
      return (this)->foo () / 2;
    }

  public:  inline int biz () const
    {
// //A declaration for this pointer
      const class A *this__ptr__ = this;
      int result = 0;

#pragma rose_outline
      {
        for (int i = 1; i <= this__ptr__->foo (); i++)
          for (int j = 1; j <= this__ptr__->bar (); j++)
            result += (i * j);
      }
      return result;
    }
  }
   ;
}

extern "C"
{
  int printf (const char *fmt, ...);
}

int main ()
{
  class N::A x;
// Prints '168'
  printf ("%d\n", x.biz ());
  return 0;
}
```

Figure 37.11: rose_outlined_pp-inputCode_OutlineLoop.cc: Figure 37.1 after outline preprocessing only, *i.e.*, specifying `-rose:outline:preproc-only` as an option to the translator of Figure 37.3.

statements, which require jumping to a regions of code outside the target. We show the preprocessed code in Figure 37.13. The original non-local jumps are first transformed into assignments to a flag, `EXIT_TAKEN__` (lines 18–20 and 26–29), and then relocated to a subsequent block of code (lines 38–53) with their execution controlled by the value of the flag. The final outlined result appears in Figure 37.14; the initial preprocessing simplifies this final step of extracting the outline target.

```
#include <iostream>

size_t factorial (size_t n)
{
  size_t i = 1;
  size_t r = 1;
  while (1)
    {
#pragma rose_outline
      if (i <= 1)
        break; // Non-local jump #1
      else if (i >= n)
        break; // Non-local jump #2
      else
        r *= ++i;
    }
  return r;
}

int main (int argc, char* argv[])
{
  std::cout << "7! == " << factorial (7) << std::endl; // Prints 5040
  return 0;
}
```

Figure 37.12: inputCode_OutlineNonLocalJumps.cc: Sample input program, with an outlining target that contains two non-local jumps (here, `break` statements).

```
#include <iostream>

size_t factorial (size_t n)
{
  size_t i = 1;
  size_t r = 1;
  while (1)
    {
#pragma rose_outline
      {
        int EXIT_TAKEN__ = 0;
        {
          if (i <= 1)
            {
              EXIT_TAKEN__ = 1;
              goto NON_LOCAL_EXIT__;
            }
          else if (i >= n)
            {
              EXIT_TAKEN__ = 2;
              goto NON_LOCAL_EXIT__;
            }
          else
            r *= ++i;
        NON_LOCAL_EXIT__:;
        }
        if (EXIT_TAKEN__ == 1)
          {
// Non-local jump #1
            break;
          }
        else
          {
            if (EXIT_TAKEN__ == 2)
              {
// Non-local jump #2
                break;
              }
            else
              {
              }
          }
      }
    }
  return r;
}

int main (int argc, char *argv[])
{
// Prints 5040
  ((*(&std::cout) << "7! == ") << factorial (7)) << std::endl < char,
    std::char_traits < char >>;
  return 0;
}
```

Figure 37.13: rose_outlined_pp-inputCode_OutlineNonLocalJumps.cc: The non-local jump example of Figure 37.12 after outliner preprocessing, but before the actual outlining. The non-local jump is handled by an additional flag, `EXIT_TAKEN__`, which indicates what non-local jump is to be taken.

```
#include <iostream>
extern "C" void OUT__1__14692__ (size_t * np__, size_t * ip__, size_t * rp__,
                                 int *EXIT_TAKEN__p__);

size_t factorial (size_t n)
{
  size_t i = 1;
  size_t r = 1;
  while (1)
    {
      {
        int EXIT_TAKEN__ = 0;
        OUT__1__14692__ (&n, &i, &r, &EXIT_TAKEN__);
        if (EXIT_TAKEN__ == 1)
          {
// Non-local jump #1
            break;
          }
        else
          {
            if (EXIT_TAKEN__ == 2)
              {
// Non-local jump #2
                break;
              }
            else
              {
              }
          }
      }
    }
  return r;
}

int main (int argc, char *argv[])
{
// Prints 5040
  ((*(&std::cout) << "7! == ") << factorial (7)) << std::endl < char,
    std::char_traits < char >>;
  return 0;
}

extern "C" void OUT__1__14692__ (size_t * np__, size_t * ip__, size_t * rp__,
                                 int *EXIT_TAKEN__p__)
{
  size_t & n = *((size_t *) np__);
  size_t & i = *((size_t *) ip__);
  size_t & r = *((size_t *) rp__);
  int &EXIT_TAKEN__ = *((int *) EXIT_TAKEN__p__);
  if (i <= 1)
    {
      EXIT_TAKEN__ = 1;
      goto NON_LOCAL_EXIT__;
    }
  else if (i >= n)
    {
      EXIT_TAKEN__ = 2;
      goto NON_LOCAL_EXIT__;
    }
  else
    r *= ++i;
NON_LOCAL_EXIT__:;
}
```

Figure 37.14: rose_outlined-inputCode_OutlineNonLocalJumps.cc: Figure 37.12 after outlining.

Chapter 38

Loop Optimization

This section is specific to loop optimization and show several tutorial examples using the optimization mechanisms within ROSE.

FIXME: *We might want to reference Qing's work explicitly since this is really just showing off here work.*

38.1 Example Loop Optimizer

Simple example translator showing use of pre-defined loop optimizations.

FIXME: *We are not running performance tests within this tutorial, but perhaps we could later.*

Figure 38.1 shows the code required to call some loop optimizations within ROSE. The translator that we build for this tutorial is simple and takes the following command line options to control which optimizations are done.

```
-ic1 :loop interchange for more reuses
-bk1/2/3 <blocksize> :block outer/inner/all loops
-fs1/2 :single/multi-level loop fusion for more reuses
-cp <copydim> :copy array
-fs0 : loop fission
-splitloop: loop splitting
-unroll [locond] [nvar] <unrollsize> : loop unrolling
-bs <stmtsize> : break up statements in loops
-annot <filename>:
     Read annotation from a file which defines side effects of functions
-arracc <funcname> :
     Use special function to denote array access (the special function can be replaced
     with macros after transformation). This option is for circumventing complex
     subscript expressions for linearized multi-dimensional arrays.
-opt <level=0> : The level of loop optimizations to apply (By default, only the outermost
              level is optimized).
-ta <int> : Max number of nodes to split for transitive dependence analysis (to limit the
         overhead of transitive dep. analysis)
-clsize <int> : set cache line size in evaluating spatial locality (affect decisions in
             applying loop optimizations)
-reuse_dist <int> : set maximum distance of reuse that can exploit cache (used to evaluate
```

```
temporal locality of loops)
```

```
// LoopProcessor:
//   Assume no aliasing
//   apply loop opt to the bodies of all function definitions

// =====================================

#include "rose.h"

#include <AstInterface_ROSE.h>
#include "LoopTransformInterface.h"
#include "CommandOptions.h"

using namespace std;

int
main ( int argc,  char * argv[] )
   {
     vector<string> argvList(argv, argv + argc);
     CmdOptions::GetInstance()->SetOptions(argvList);
     AssumeNoAlias aliasInfo;
     LoopTransformInterface::cmdline_configure(argvList);
     LoopTransformInterface::set_aliasInfo(&aliasInfo);

     SgProject* project = new SgProject(argvList);

  // Loop over the number of files in the project
     int filenum = project->numberOfFiles();
     for (int i = 0; i < filenum; ++i)
        {
          SgSourceFile* file = isSgSourceFile(project->get_fileList()[i]);
          SgGlobal *root = file->get_globalScope();
          SgDeclarationStatementPtrList& declList = root->get_declarations ();

       // Loop over the declaration in the global scope of each file
          for (SgDeclarationStatementPtrList::iterator p = declList.begin(); p != declList.end(); ++p)
             {
               SgFunctionDeclaration *func = isSgFunctionDeclaration(*p);
               if (func == NULL)
                    continue;
               SgFunctionDefinition *defn = func->get_definition();
               if (defn == NULL)
                    continue;

               SgBasicBlock *stmts = defn->get_body();
               AstInterfaceImpl faImpl(stmts);

            // This will do as much fusion as possible (finer grained
            // control over loop optimizations uses a different interface).
               LoopTransformInterface::TransformTraverse(faImpl, AstNodePtrImpl(stmts));

            // JJW 10-29-2007 Adjust for iterator invalidation and possible
            // inserted statements
               p = std::find(declList.begin(), declList.end(), func);
               assert (p != declList.end());
             }
        }

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

Figure 38.1: Example source code showing use of loop optimization mechanisms.

38.2 Matrix Multiply Example

Using the matrix multiply example code shown in figure 38.2, we run the loop optimizer in figure 38.1 and generate the code shown in figure 38.3.

```
// Example program showing matrix multiply
// (for use with loop optimization tutorial example)

#define N 50

int main()
   {
     int i,j, k;
     double a[N][N], b[N][N], c[N][N];

     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];
                  }
             }
        }

     return 0;
   }
```

Figure 38.2: Example source code used as input to loop optimization processor.

```
int min2(int a0,int a1)
{
  return a0 < a1?a0 : a1;
}
// Example program showing matrix multiply
// (for use with loop optimization tutorial example)
#define N 50

int main()
{
  int i;
  int j;
  int k;
  double a[50UL][50UL];
  double b[50UL][50UL];
  double c[50UL][50UL];
  int _var_0;
  int _var_1;
  for (_var_1 = 0; _var_1 <= 49; _var_1 += 16) {
    for (_var_0 = 0; _var_0 <= 49; _var_0 += 16) {
      for (k = 0; k <= 49; k += 1) {
        for (i = _var_1; i <= min2(49,_var_1 + 15); i += 1) {
          for (j = _var_0; j <= min2(49,_var_0 + 15); j += 1) {
            c[i][j] = (c[i][j] + (a[i][k] * b[k][j]));
          }
        }
      }
    }
  }
  return 0;
}
```

Figure 38.3: Output of loop optimization processor showing matrix multiply optimization (using options: `-bk1 -fs0`).

38.3 Loop Fusion Example

Using the loop fusion example code shown in figure 38.4, we run the loop optimizer in figure 38.1 and generate the code shown in figure 38.5.

```

main() {

 int x[30], i;

  for (i = 1; i <= 10; i += 1) {
    x[2 * i] = x[2 * i + 1] + 2;
  }
  for (i = 1; i <= 10; i += 1) {
    x[2 * i + 3] = x[2 * i] + i;
  }

}
```

Figure 38.4: Example source code used as input to loop optimization processor.

```

int main()
{
  int x[30UL];
  int i;
  for (i = 1; i <= 11; i += 1) {
    if (i <= 10) {
      x[2 * i] = (x[(2 * i) + 1] + 2);
    }
    else {
    }
    if (i >= 2) {
      x[(2 * (-1 + i)) + 3] = (x[2 * (-1 + i)] + (-1 + i));
    }
    else {
    }
  }
  return 0;
}
```

Figure 38.5: Output of loop optimization processor showing loop fusion (using options: `-fs2`).

38.4 Example Loop Processor (LoopProcessor.C)

This section contains a more detail translator which uses the command-line for input of specific loop processing options and is more sophisticated than the previous translator used to handle the previous two examples.

Figure 38.6 shows the code required to call the loop optimizations within ROSE. The translator that we build for this tutorial is simple and takes command line parameters to control which optimizations are done.

```
#include "rose.h"
#include <general.h>

#include "pre.h"
#include "finiteDifferencing.h"


// DQ (1/2/2008): I think this is no longer used!
// #include "copy_unparser.h"

#include "rewrite.h"
#include <CommandOptions.h>
#include <AstInterface_ROSE.h>
#include <LoopTransformInterface.h>
#include <AnnotCollect.h>
#include <OperatorAnnotation.h>

using namespace std;

#ifdef USE_OMEGA
#include <DepTestStatistics.h>

extern DepTestStatistics DepStats;
#endif

extern bool DebugAnnot();
extern void FixFileInfo(SgNode* n);
class UnparseFormatHelp;
class UnparseDelegate;
void unparseProject( SgProject* project, UnparseFormatHelp* unparseHelp /*= NULL*/, UnparseDelegate *repl
/*= NULL */);

void PrintUsage( char* name)
{
  cerr << name << " <options> " << "<program_name>" << "\n";
  cerr << "-gobj: generate object file\n";
  cerr << "-orig: copy non-modified statements from original file\n";
  cerr << "-splitloop: applying loop splitting to remove conditionals inside loops\n";
  cerr << ReadAnnotation::get_inst()->OptionString() << endl;
  cerr << "-pre:  apply partial redundancy elimination\n";
  cerr << "-fd:  apply finite differencing to array index expressions\n";
  LoopTransformInterface::PrintTransformUsage( cerr );
}
```

Figure 38.6: Detailed example source code showing use of loop optimization mechanisms (loopProcessor.C part 1).

```
bool GenerateObj()
{
  return CmdOptions::GetInstance()->HasOption("-gobj");
}


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

  if (argc <= 1) {
      PrintUsage(argv[0]);
      return -1;
  }
  vector<string> argvList(argv, argv + argc);
  CmdOptions::GetInstance()->SetOptions(argvList);
  AssumeNoAlias aliasInfo;
  LoopTransformInterface::cmdline_configure(argvList);
  LoopTransformInterface::set_aliasInfo(&aliasInfo);

#ifdef USE_OMEGA
  DepStats.SetFileName(buffer.str());
#endif

  OperatorSideEffectAnnotation *funcInfo =
         OperatorSideEffectAnnotation::get_inst();
  funcInfo->register_annot();
  ReadAnnotation::get_inst()->read();
  if (DebugAnnot())
     funcInfo->Dump();
  LoopTransformInterface::set_sideEffectInfo(funcInfo);
  SgProject *project = new SgProject ( argvList);

   int filenum = project->numberOfFiles();
   for (int i = 0; i < filenum; ++i) {
  // SgFile &sageFile = sageProject->get_file(i);
  // SgGlobal *root = sageFile.get_root();
     SgSourceFile* file = isSgSourceFile(project->get_fileList()[i]);
     SgGlobal *root = file->get_globalScope();
     SgDeclarationStatementPtrList& declList = root->get_declarations ();
     for (SgDeclarationStatementPtrList::iterator p = declList.begin(); p != declList.end(); ++p) {
```

Figure 38.7: loopProcessor.C source code (Part 2).

38.5 Matrix Multiplication Example (mm.C)

Using the matrix multiplication example code shown in figure 38.8, we run the loop optimizer in figure 38.6 and generate the code shown in figure 38.9.

```
#define N 50

void printmatrix( double x[][N]);
void initmatrix( double x[][N], double s);

main()
{
  int i,j, k;
  double a[N][N], b[N][N], c[N][N];

  double s;
  s = 235.0;
  initmatrix(a, s);
  s = 321.0;
  initmatrix(b, s);

  printmatrix(a);
  printmatrix(b);
  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];
       }
    }
  }

  printmatrix(c);
}
```

Figure 38.8: Example source code used as input to loopProcessor, show in figure 38.6.

```
int min2(int a0,int a1)
{
  return a0 < a1?a0 : a1;
}
#define N 50
void printmatrix(double x[][50UL]);
void initmatrix(double x[][50UL],double s);

int main()
{
  int i;
  int j;
  int k;
  double a[50UL][50UL];
  double b[50UL][50UL];
  double c[50UL][50UL];
  double s;
  int _var_0;
  int _var_1;
  s = 235.0;
  initmatrix(a,s);
  s = 321.0;
  initmatrix(b,s);
  printmatrix(a);
  printmatrix(b);
  for (_var_1 = 0; _var_1 <= 49; _var_1 += 16) {
    for (_var_0 = 0; _var_0 <= 49; _var_0 += 16) {
      for (k = 0; k <= 49; k += 1) {
        for (i = _var_1; i <= min2(49,_var_1 + 15); i += 1) {
          for (j = _var_0; j <= min2(49,_var_0 + 15); j += 1) {
            c[i][j] = (c[i][j] + (a[i][k] * b[k][j]));
          }
        }
      }
    }
  }
  printmatrix(c);
  return 0;
}
```

Figure 38.9: Output of loopProcessor using input from figure 38.8 (using options: `-bk1 -fs0`).

38.6 Matrix Multiplication Example Using Linearized Matrices (dgemm.C)

Using the matrix multiplication example code shown in figure 38.10, we run the loop optimizer in figure 38.6 and generate the code shown in figure 38.11.

```
// Function prototype
void dgemm(double *a, double *b, double *c, int n);

// Function definition
void dgemm(double *a, double *b, double *c, int n)
   {
     int i, j, k;

  // int n;

     for(k=0;k<n;k+=1){
        for(j=0;j<n;j+=1){
           for(i=0;i<n;i+=1){
              c[j*n+i]=c[j*n+i]+a[k*n+i]*b[j*n+k];
           }
        }
     }
   }
```

Figure 38.10: Example source code used as input to loopProcessor, show in figure 38.6.

```
int min2(int a0,int a1)
{
  return a0 < a1?a0 : a1;
}

int min2(int a0,int a1)
{
  return a0 < a1?a0 : a1;
}

int min2(int a0,int a1)
{
  return a0 < a1?a0 : a1;
}
// Function prototype
void dgemm(double *a,double *b,double *c,int n);
// Function definition

void dgemm(double *a,double *b,double *c,int n)
{
  int i;
  int j;
  int k;
  int _var_0;
  int _var_1;
  for (_var_1 = 0; _var_1 <= -1 + n; _var_1 += 16) {
    for (_var_0 = 0; _var_0 <= -1 + n; _var_0 += 16) {
      for (i = 0; i <= -1 + n; i += 1) {
        for (k = _var_1; k <= min2(-1 + n,_var_1 + 15); k += 1) {
          for (j = _var_0; j <= ::min2(n + -16,_var_0); j += 16) {
            c[(j * n) + i] = (c[(j * n) + i] + (a[(k * n) + i] * b[(j * n) + k]));
            c[((1 + j) * n) + i] = (c[((1 + j) * n) + i] + (a[(k * n) + i] * b[((1 + j) * n) + k]));
            c[((2 + j) * n) + i] = (c[((2 + j) * n) + i] + (a[(k * n) + i] * b[((2 + j) * n) + k]));
            c[((3 + j) * n) + i] = (c[((3 + j) * n) + i] + (a[(k * n) + i] * b[((3 + j) * n) + k]));
            c[((4 + j) * n) + i] = (c[((4 + j) * n) + i] + (a[(k * n) + i] * b[((4 + j) * n) + k]));
            c[((5 + j) * n) + i] = (c[((5 + j) * n) + i] + (a[(k * n) + i] * b[((5 + j) * n) + k]));
            c[((6 + j) * n) + i] = (c[((6 + j) * n) + i] + (a[(k * n) + i] * b[((6 + j) * n) + k]));
            c[((7 + j) * n) + i] = (c[((7 + j) * n) + i] + (a[(k * n) + i] * b[((7 + j) * n) + k]));
            c[((8 + j) * n) + i] = (c[((8 + j) * n) + i] + (a[(k * n) + i] * b[((8 + j) * n) + k]));
            c[((9 + j) * n) + i] = (c[((9 + j) * n) + i] + (a[(k * n) + i] * b[((9 + j) * n) + k]));
            c[((10 + j) * n) + i] = (c[((10 + j) * n) + i] + (a[(k * n) + i] * b[((10 + j) * n) + k]));
            c[((11 + j) * n) + i] = (c[((11 + j) * n) + i] + (a[(k * n) + i] * b[((11 + j) * n) + k]));
            c[((12 + j) * n) + i] = (c[((12 + j) * n) + i] + (a[(k * n) + i] * b[((12 + j) * n) + k]));
            c[((13 + j) * n) + i] = (c[((13 + j) * n) + i] + (a[(k * n) + i] * b[((13 + j) * n) + k]));
            c[((14 + j) * n) + i] = (c[((14 + j) * n) + i] + (a[(k * n) + i] * b[((14 + j) * n) + k]));
            c[((15 + j) * n) + i] = (c[((15 + j) * n) + i] + (a[(k * n) + i] * b[((15 + j) * n) + k]));
          }
          for (; j <= ::min2(-1 + n,15 + _var_0); j += 1) {
            c[(j * n) + i] = (c[(j * n) + i] + (a[(k * n) + i] * b[(j * n) + k]));
          }
        }
      }
    }
  }
}
```

Figure 38.11: Output of loopProcessor using input from figure 38.10 (using options: `-bk1 -unroll nvar 16`).

38.7 LU Factorization Example (lufac.C)

Using the LU factorization example code shown in figure 38.12, we run the loop optimizer in figure 38.6 and generate the code shown in figure 38.13.

```
double abs(double x) { if (x < 0) return -x; else return x; }

#define n 50
void printmatrix( double x[][n]);
void initmatrix( double x[][n], double s);

main(int argc, char* argv[]) {
int p[n], i, j, k;
double a[n][n], mu, t;

initmatrix(a, 5.0);
printmatrix(a);

for (k = 0; k<=n-2; k+=1) {
    p[k] = k;
    mu = abs(a[k][k]);
    for (i = k+1; i <= n-1; i+=1) {
      if (mu < abs(a[i][ k])) {
        mu = abs(a[i][k]);
        p[k] = i;
      }
    }

    for (j = k; j <= n-1; j+=1) {
       t = a[k][j];
       a[k][j] = a[p[k]][j];
       a[p[k]][j] = t;
    }

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

printmatrix(a);
}
```

Figure 38.12: Example source code used as input to loopProcessor, show in figure 38.6.

```
double abs(double x)
{
  if (x < 0)
    return -x;
  else
    return x;
}
#define n 50
void printmatrix(double x[][50UL]);
void initmatrix(double x[][50UL],double s);

int main(int argc,char *argv[])
{
  int p[50UL];
  int i;
  int j;
  int k;
  double a[50UL][50UL];
  double mu;
  double t;
  initmatrix(a,5.0);
  printmatrix(a);
  for (k = 0; k <= 48; k += 1) {
    p[k] = k;
    mu = abs(a[k][k]);
    for (i = 1 + k; i <= 49; i += 1) {
      if (mu < abs(a[i][k])) {
        mu = abs(a[i][k]);
        p[k] = i;
      }
    }
    for (j = k; j <= 49; j += 1) {
      t = a[k][j];
      a[k][j] = a[p[k]][j];
      a[p[k]][j] = t;
    }
    for (i = 1 + k; i <= 49; i += 1) {
      a[i][k] = (a[i][k] / a[k][k]);
    }
    for (j = 1 + k; j <= 49; j += 1) {
      for (i = 1 + k; i <= 49; i += 1) {
        a[i][j] = (a[i][j] - (a[i][k] * a[k][j]));
      }
    }
  }
  printmatrix(a);
  return 0;
}
```

Figure 38.13: Output of loopProcessor using input from figure 38.12 (using options: `-bk1 -fs0 -splitloop -annotation`).

38.8 Loop Fusion Example (tridvpk.C)

Using the loop fusion example code shown in figure 38.14, we run the loop optimizer in figure 38.6 and generate the code shown in figure 38.15.

```
#define n 100

  double a[n], b[n], c[n], d[n], e[n];
  double tot[n][ n];
  double dux[n][n][n], duy[n][n][n], duz[n][n][n];

main()
{
  int i,j,k;
      for ( j=0; j <=n-1; j+=1)
      for ( i=0; i<=n-1; i+=1)
         duz[i][j][0] = duz[i][j][0]*b[0];

      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];

}
```

Figure 38.14: Example source code used as input to loopProcessor, show in figure 38.6.

```
#define n 100
double a[100UL];
double b[100UL];
double c[100UL];
double d[100UL];
double e[100UL];
double tot[100UL][100UL];
double dux[100UL][100UL][100UL];
double duy[100UL][100UL][100UL];
double duz[100UL][100UL][100UL];

int main()
{
  int i;
  int j;
  int k;
  for (i = 0; i <= 99; i += 1) {
    for (j = 0; j <= 99; j += 1) {
      duz[i][j][0] = (duz[i][j][0] * b[0]);
      tot[i][j] = 0;
      for (k = 0; k <= 98; k += 1) {
        if (k >= 1) {
          duz[i][j][k] = ((duz[i][j][k] - (a[k] * duz[i][j][k - 1])) * b[k]);
        }
        else {
        }
        tot[i][j] = (tot[i][j] + (d[k] * duz[i][j][k]));
      }
      duz[i][j][100 - 1] = ((duz[i][j][100 - 1] - tot[i][j]) * b[100 - 1]);
      duz[i][j][100 - 2] = (duz[i][j][100 - 2] - (e[100 - 2] * duz[i][j][100 - 1]));
      for (k = 97; k >= 0; k += -1) {
        duz[i][j][k] = ((duz[i][j][k] - (c[k] * duz[i][j][k + 1])) - (e[k] * duz[i][j][100 - 1]));
      }
    }
  }
  return 0;
}
```

Figure 38.15: Output of loopProcessor input from figure 38.14 (using options: `-fs2 -ic1 -opt 1`).

Chapter 39

Parameterized Code Translation

This chapter gives examples of using ROSE's high level loop translation interfaces to perform parameterized loop transformations, including loop unrolling, interchanging and tiling. The motivation is to give users the maximized flexibility to orchestrate code transformations on the targets they want, the order they want, and the parameters they want. One typical application scenario is to support generating desired code variants for empirical tuning.

The ROSE internal interfaces (declared within the SageInterface namespace) to call loop transformations are:

- *bool loopUnrolling (SgForStatement *loop, size_t unrolling_factor)*: This function needs two parameters: one for the loop to be unrolled and the other for the unrolling factor.

- *bool loopInterchange (SgForStatement *loop, size_t depth, size_t lexicoOrder)*: The loop interchange function has three parameters, the first one to specify a loop which starts a perfectly-nested loop and is to be interchanged, the 2nd for the depth of the loop nest to be interchanged, and finally the lexicographical order for the permutation.

- *bool loopTiling (SgForStatement *loopNest, size_t targetLevel, size_t tileSize)* The loop tiling interface needs to know the loop nest to be tiled, which loop level to tile, and the tile size for the level.

For efficiency concerns, those functions only perform the specified translations without doing any legitimacy check. It is up to the users to make sure the transformations won't generate wrong code. We will soon provide interfaces to do the eligibility check for each transformation.

We also provide standalone executable programs (loopUnrolling,loopInterchange, and loopTiling under ROSE_INSTALL/bin) for the transformations mentioned above so users can directly use them via command lines and abstract handles (explained in Chapter 46) to orchestrate transformations they want.

39.1 Loop Unrolling

Figure 39.1 gives an example input code for loopUnrolling.

An example command line to invoke loop unrolling on the example can look like the following:

```
   int a[100][100];
2  int main(void)
   {
4    int j;
     for (int i=0;i<100;i++)
6      for (j=0;j<100;j++)
       {
8        int k=3;
         a[i][j]=i+j+k;
10     }
     return 0;
12 }
```

Figure 39.1: Example source code used as input to loopUnrolling

```
# unroll a for statement 5 times. The loop is a statement at line 6 within
# an input file.
loopUnrolling -c inputloopUnrolling.C \
-rose:loopunroll:abstract_handle "Statement<position,6>" -rose:loopunroll:factor 5
```

Two kinds of output can be expected after loop unrolling. One (Shown in Figure 39.2) is the case that the loop iteration count is known at compile-time and can be evenly divisible by the unrolling factor. The other case (Shown in Figure 39.3 is when the divisibility is unknown and a fringe loop has to be generated to run possible leftover iterations.

```
int a[100UL][100UL];

int
main ()
{
  int j;
  for (int i = 0; i < 100; i++)
    {
      for (j = 0; j <= 99; j += 5)
        {
          int k = 3;
          a[i][j] = ((i + j) + k);
          {
            int k = 3;
            a[i][(j + 1)] = ((i + (j + 1)) + k);
          }
          {
            int k = 3;
            a[i][(j + 2)] = ((i + (j + 2)) + k);
          }
          {
            int k = 3;
            a[i][(j + 3)] = ((i + (j + 3)) + k);
          }
          {
            int k = 3;
            a[i][(j + 4)] = ((i + (j + 4)) + k);
          }
        }
    }
  return 0;
}
```

Figure 39.2: Output for a unrolling factor which can divide the iteration space evenly

```
int a[100UL][100UL];

int
main ()
{
  int j;
  for (int i = 0; i < 100; i++)
    {
// iter_count = (ub-lb+1)%step ==0?(ub-lb+1)/step: (ub-lb+1)/step+1;
// fringe = iter_count%unroll_factor==0 ? 0:unroll_factor*step
      int _lu_fringe_1 = 3;
      for (j = 0; j <= 99 - _lu_fringe_1; j += 3)
        {
          int k = 3;
          a[i][j] = ((i + j) + k);
          {
            int k = 3;
            a[i][(j + 1)] = ((i + (j + 1)) + k);
          }
          {
            int k = 3;
            a[i][(j + 2)] = ((i + (j + 2)) + k);
          }
        }
      for (; j <= 99; j += 1)
        {
          int k = 3;
          a[i][j] = ((i + j) + k);
        }
    }
  return 0;
}
```

Figure 39.3: Output for the case when divisibility is unknown at compile-time

39.2 Loop Interchange

Figure 39.4 gives an example input code for loopInterchange.

```
void OUT__1__6119__(int ri, double *rp, int stencil_size, int hypre__m, const double *Ap_0)
{
  int si , ii , jj , kk ;
  // the following 4-level loop nest is to be interchanged
  for (si = 0; si < stencil_size; si++)
    for (kk = 0; kk < hypre__m; kk++)
      for (jj = 0; jj < hypre__m; jj++)
        for (ii = 0; ii < hypre__m; ii++)
          rp[(ri + ii) + jj  + kk ] -= Ap_0[ii + jj  + kk];
}
```

Figure 39.4: Example source code used as input to loopInterchange

An example command line to invoke loop interchange:

```
# interchange a loop nest starting from the first loop within the input file,
# interchange depth is 4 and
# the lexicographical order is 1 (swap the innermost two levels)
loopInterchange -c inputloopInterchange.C -rose:loopInterchange:abstract_handle \
"ForStatement<numbering,1>" -rose:loopInterchange:depth 4 \
-rose:loopInterchange:order 1
```

Figure 39.5 shows the output.

```

void OUT__1__6119__(int ri,double *rp,int stencil_size,int hypre__m,const double *Ap_0)
{
  int si;
  int ii;
  int jj;
  int kk;
// the following 4-level loop nest is to be interchanged
  for (si = 0; si < stencil_size; si++)
    for (kk = 0; kk < hypre__m; kk++)
      for (ii = 0; ii < hypre__m; ii++)
        for (jj = 0; jj < hypre__m; jj++)
          rp[((ri + ii) + jj) + kk] -= Ap_0[(ii + jj) + kk];
}
```

Figure 39.5: Output for loop interchange

39.3 Loop Tiling

Figure 39.6 gives an example input code for loopTiling.

```
#define N 100
int i,j,k;
double a[N][N],b[N][N],c[N][N];

int main()
{
  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        c[i][j]= c[i][j]+a[i][k]*b[k][j];
  return 0;
}
```

Figure 39.6: Example source code used as input to loopTiling

An example command line to invoke loop tiling:

```
# Tile the loop with a depth of 3 within the first loop of the input file
# tile size is 5
loopTiling -c inputloopTiling.C -rose:loopTiling:abstract_handle \
"ForStatement<numbering,1>" -rose:loopTiling:depth 3 -rose:loopTiling:tilesize 5
```

Figure 39.7 shows the output.

```
#define N 100
int i;
int j;
int k;
double a[100UL][100UL];
double b[100UL][100UL];
double c[100UL][100UL];

int main()
{
  int _lt_var_k;
  for (_lt_var_k = 0; _lt_var_k <= 99; _lt_var_k += 5) {
    for (i = 0; i < 100; i++)
      for (j = 0; j < 100; j++)
        for (k = _lt_var_k; k <= (((99 < (_lt_var_k + 5 - 1))?99 : (_lt_var_k + 5 - 1))); k += 1) {
          c[i][j] = (c[i][j] + (a[i][k] * b[k][j]));
        }
  }
  return 0;
}
```

Figure 39.7: Output for loop tiling

Part V

Correctness Checking

Tutorials of using ROSE to help program correctness checking or debugging.

Chapter 40

Code Coverage

This translator is part of ongoing collaboration with IBM on the support of code coverage analysis tools for C, C++ and F90 applications. the subject of code coverage is much more complex than this example code would cover. The following web site: *http://www.bullseye.com/coverage.html* contains more information and is the source for the descriptions below. Code coverage can include:

- Statement Coverage
  This measure reports whether each executable statement is encountered.

- Decision Coverage
  This measure reports whether boolean expressions tested in control structures (such as the if-statement and while-statement) evaluated to both true and false. The entire boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Additionally, this measure includes coverage of switch-statement cases, exception handlers, and interrupt handlers.

- Condition Coverage
  Condition coverage reports the true or false outcome of each boolean sub-expression, separated by logical-and and logical-or if they occur. Condition coverage measures the sub-expressions independently of each other.

- Multiple Condition Coverage
  Multiple condition coverage reports whether every possible combination of boolean sub-expressions occurs. As with condition coverage, the sub-expressions are separated by logical-and and logical-or, when present. The test cases required for full multiple condition coverage of a condition are given by the logical operator truth table for the condition.

- Condition/Decision Coverage
  Condition/Decision Coverage is a hybrid measure composed by the union of condition coverage and decision coverage. This measure was created at Boeing and is required for aviation software by RCTA/DO-178B.

- Modified Condition/Decision Coverage
  This measure requires enough test cases to verify every condition can affect the result of its encompassing decision.

- Path Coverage
  This measure reports whether each of the possible paths in each function have been followed. A path is a unique sequence of branches from the function entry to the exit.

- Function Coverage
  This measure reports whether you invoked each function or procedure. It is useful during preliminary testing to assure at least some coverage in all areas of the software. Broad, shallow testing finds gross deficiencies in a test suite quickly.

- Call Coverage
  This measure reports whether you executed each function call. The hypothesis is that faults commonly occur in interfaces between modules.

- Linear Code Sequence and Jump (LCSAJ) Coverage
  This variation of path coverage considers only sub-paths that can easily be represented in the program source code, without requiring a flow graph. An LCSAJ is a sequence of source code lines executed in sequence. This "linear" sequence can contain decisions as long as the control flow actually continues from one line to the next at run-time. Sub-paths are constructed by concatenating LCSAJs. Researchers refer to the coverage ratio of paths of length n LCSAJs as the test effectiveness ratio (TER) n+2.

- Data Flow Coverage
  This variation of path coverage considers only the sub-paths from variable assignments to subsequent references of the variables.

- Object Code Branch Coverage
  This measure reports whether each machine language conditional branch instruction both took the branch and fell through.

- Loop Coverage
  This measure reports whether you executed each loop body zero times, exactly once, and more than once (consecutively). For do-while loops, loop coverage reports whether you executed the body exactly once, and more than once. The valuable aspect of this measure is determining whether while-loops and for-loops execute more than once, information not reported by others measure.

- Race Coverage
  This measure reports whether multiple threads execute the same code at the same time. It helps detect failure to synchronize access to resources. It is useful for testing multi-threaded programs such as in an operating system.

- Relational Operator Coverage
  This measure reports whether boundary situations occur with relational operators (¡, ¡=, ¿, ¿=). The hypothesis is that boundary test cases find off-by-one errors and mistaken uses of wrong relational operators such as ¡ instead of ¡=.

- Weak Mutation Coverage
  This measure is similar to relational operator coverage but much more general [Howden1982]. It reports whether test cases occur which would expose the use of wrong operators and also wrong operands. It works by reporting coverage of conditions derived by substituting (mutating) the program's expressions with alternate operators, such as "-" substituted for "+", and with alternate variables substituted.

- Table Coverage
  This measure indicates whether each entry in a particular array has been referenced. This is useful for programs that are controlled by a finite state machine.

The rest of this text must be changed to refer to the code coverage example within ROSE/-tutorial.

Figure 40 shows the low level construction of a more complex AST fragment (a function declaration) and its insertion into the AST at the top of each block. Note that the code does not handle symbol table issues, yet.

Building a function in global scope.

```
// ROSE is a tool for building preprocessors, this file is an example preprocessor built with ROSE.
// Specifically it shows the design of a transformation to instrument source code, placing source code
// at the top and bottome of each basic block.

#include "rose.h"

using namespace std;

/*
   Design of this code.
      Inputs: source code (file.C)
      Outputs: instrumented source code (rose_file.C and file.o)

   Properties of instrumented source code:
       1) added declaration for coverage support function
          (either forward function declaration or a #include to include a header file).
       2) Each function in the source program is instrumented to include a call to the
          coverage support function/
*/



// Global variables so that the global function declaration can be reused to build
// each function call expression in the AST traversal to instrument all functions.
SgFunctionDeclaration* globalFunctionDeclaration = NULL;
SgFunctionType*        globalFunctionType        = NULL;
SgFunctionSymbol* functionSymbol = NULL;

// Simple ROSE traversal class: This allows us to visit all the functions and add
// new code to instrument/record their use.
class SimpleInstrumentation : public SgSimpleProcessing
   {
     public:
       // required visit function to define what is to be done
          void visit ( SgNode* astNode );
   };


// Code to build function declaration: This declares Shmuel's function call which
// will be inserted (as a function call) into each function body of the input
// application.
void buildFunctionDeclaration(SgProject* project)
   {
  // *****************************************************
  // Create the functionDeclaration
  // *****************************************************

  // SgGlobal* globalScope = project->get_file(0).get_root();
     SgSourceFile* sourceFile = isSgSourceFile(project->get_fileList()[0]);
     ROSE_ASSERT(sourceFile != NULL);
     SgGlobal* globalScope = sourceFile->get_globalScope();
     ROSE_ASSERT(globalScope != NULL);

     Sg_File_Info * file_info            = Sg_File_Info::generateDefaultFileInfoForTransformationNode();
     SgType * function_return_type       = new SgTypeVoid();

     SgName function_name                         = "coverageTraceFunc1";
     SgFunctionType * function_type               = new SgFunctionType(function_return_type,false);
     SgFunctionDeclaration * functionDeclaration = new SgFunctionDeclaration(file_info, function_name, function_type);

  // DQ (9/8/2007): Fixup the defining and non-defining declarations
     ROSE_ASSERT(functionDeclaration->get_definingDeclaration() == NULL);
     functionDeclaration->set_definingDeclaration(functionDeclaration);
     ROSE_ASSERT(functionDeclaration->get_definingDeclaration()          != NULL);
     ROSE_ASSERT(functionDeclaration->get_firstNondefiningDeclaration() != functionDeclaration);

  // DQ (9/8/2007): We have not build a non-defining declaration, so this should be NULL.
     ROSE_ASSERT(functionDeclaration->get_firstNondefiningDeclaration() == NULL);

  // DQ (9/8/2007): Need to add function symbol to global scope!
     printf ("Fixing up the symbol table in scope = %p = %s for function = %p = %s \n",globalScope,globalScope->class_name().c_st
     functionSymbol = new SgFunctionSymbol(functionDeclaration);
     globalScope->insert_symbol(functionDeclaration->get_name(),functionSymbol);
     ROSE_ASSERT(globalScope->lookup_function_symbol(functionDeclaration->get_name()) != NULL);

  // *****************************************************************
  // Create the InitializedName for a parameter within the parameter list
  // *****************************************************************
     SgName var1_name = "textString";
```

Figure 40.1: Example source code shows instrumentation to call a test function from the top of each function body in the application (part 1).

```
      SgTypeChar * var1_type             = new SgTypeChar();
      SgPointerType *pointer_type        = new SgPointerType(var1_type);
      SgInitializer * var1_initializer   = NULL;
      SgInitializedName *var1_init_name = new SgInitializedName(var1_name, pointer_type, var1_initializer, NULL);
      var1_init_name->set_file_info(Sg_File_Info::generateDefaultFileInfoForTransformationNode());

   // Insert argument in function parameter list
      ROSE_ASSERT(functionDeclaration != NULL);
      ROSE_ASSERT(functionDeclaration->get_parameterList() != NULL);

      ROSE_ASSERT(functionDeclaration->get_parameterList() != NULL);
      functionDeclaration->get_parameterList()->append_arg(var1_init_name);

   // Set the parent node in the AST (this could be done by the AstPostProcessing
      functionDeclaration->set_parent(globalScope);

   // Set the scope explicitly (since it could be different from the parent?)
   // This can't be done by the AstPostProcessing (unless we relax some constraints)
      functionDeclaration->set_scope(globalScope);

   // If it is not a forward declaration then the unparser will skip the ";" at the end (need to fix this better)
      functionDeclaration->setForward();
      ROSE_ASSERT(functionDeclaration->isForward() == true);

   // Mark function as extern "C"
      functionDeclaration->get_declarationModifier().get_storageModifier().setExtern();
      functionDeclaration->set_linkage("C");  // This mechanism could be improved!

      globalFunctionType = function_type;
      globalFunctionDeclaration = functionDeclaration;

   // Add function declaration to global scope!
      globalScope->prepend_declaration(globalFunctionDeclaration);

   // functionSymbol = new SgFunctionSymbol(globalFunctionDeclaration);
   // All any modifications to be fixed up (parents etc)
   // AstPostProcessing(project); // This is not allowed and should be fixed!
      AstPostProcessing(globalScope);

    }

#if 0
// DQ (12/1/2005): This version of the visit function handles the special case of
// instumentation at each function (a function call at the top of each function).
// At IBM we modified this to be a version which instrumented every block.

void
SimpleInstrumentation::visit ( SgNode* astNode )
   {
     SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(astNode);
     SgFunctionDefinition*  functionDefinition  = functionDeclaration != NULL ? functionDeclaration->get_definition() : NULL;
     if (functionDeclaration != NULL && functionDefinition != NULL)
        {
       // It is up to the user to link the implementations of these functions link time
          string functionName = functionDeclaration->get_name().str();
          string fileName     = functionDeclaration->get_file_info()->get_filename();

       // Build a source file location object (for construction of each IR node)
       // Note that we should not be sharing the same Sg_File_Info object in multiple IR nodes.
          Sg_File_Info * file_info = Sg_File_Info::generateDefaultFileInfoForTransformationNode();

          SgFunctionSymbol* functionSymbol        = new SgFunctionSymbol(globalFunctionDeclaration);
          SgFunctionRefExp* functionRefExpression = new SgFunctionRefExp(file_info,functionSymbol,globalFunctionType);
          SgExprListExp* expressionList           = new SgExprListExp(file_info);

          string converageFunctionInput = functionName + string ("_in_") + fileName;
          SgStringVal* functionNameStringValue = new SgStringVal(file_info,(char*)converageFunctionInput.c_str());
          expressionList->append_expression(functionNameStringValue);
          SgFunctionCallExp* functionCallExp   = new SgFunctionCallExp(file_info,functionRefExpression,expressionList,globalFunctionType);

       // create an expression type
          SgTypeVoid* expr_type = new SgTypeVoid();

       // create an expression root
       // SgExpressionRoot * expr_root = new SgExpressionRoot(file_info,functionCallExp,expr_type);

       // create an expression statement
          SgExprStatement* new_stmt = new SgExprStatement(file_info,functionCallExp);

       // expr_root->set_parent(new_stmt);
```

Figure 40.2: Example source code shows instrumentation to call a test function from the top of each function body in the application (part 2).

```
        // new_stmt->set_expression_root(expr_root);
        // functionCallExp->set_parent(new_stmt->get_expression_root());

           functionCallExp->set_parent(new_stmt);

        // insert a statement into the function body
           functionDefinition->get_body()->prepend_statement(new_stmt);

#if 0
        // This shows the alternative use of the ROSE Rewrite Mechanism to do the same thing!
        // However, the performance is not as good as the version written above (more directly
        // building the IR nodes).

        // string codeAtTopOfBlock = "void printf(char*); printf (\"FUNCTION_NAME in FILE_NAME \\n\");";
           string codeAtTopOfBlock = "coverageTraceFunc1(\"FUNCTION_NAME_in_FILE_NAME\");";

           string functionTarget   = "FUNCTION_NAME";
           string fileTarget       = "FILE_NAME";

           codeAtTopOfBlock.replace(codeAtTopOfBlock.find(functionTarget),functionTarget.size(),functionName);
           codeAtTopOfBlock.replace(codeAtTopOfBlock.find(fileTarget),fileTarget.size(),fileName);

        // printf ("codeAtTopOfBlock = %s \n",codeAtTopOfBlock.c_str());
           printf ("%s_in_%s_\n",functionName.c_str(),fileName.c_str());

        // Insert new code into the scope represented by the statement (applies to SgScopeStatements)
           MiddleLevelRewrite::ScopeIdentifierEnum scope = MidLevelCollectionTypedefs::StatementScope;

           SgBasicBlock* functionBody = functionDefinition->get_body();
           ROSE_ASSERT(functionBody != NULL);

        // Insert the new code at the top of the scope represented by block
           MiddleLevelRewrite::insert(functionBody,codeAtTopOfBlock,scope,MidLevelCollectionTypedefs::TopOfCurrentScope);
#endif
         }
    }
#endif

void
SimpleInstrumentation::visit ( SgNode* astNode ) {
    SgBasicBlock *block = NULL;
    block = isSgBasicBlock(astNode);
    if (block != NULL) {
        // It is up to the user to link the implementations of these functions link time
        Sg_File_Info *fileInfo = block->get_file_info();
        string fileName = fileInfo->get_filename();
        int lineNum = fileInfo->get_line();

        // Build a source file location object (for construction of each IR node)
        // Note that we should not be sharing the same Sg_File_Info object in multiple IR nodes.
        Sg_File_Info * newCallfileInfo = Sg_File_Info::generateDefaultFileInfoForTransformationNode();

        ROSE_ASSERT(functionSymbol != NULL);
        SgFunctionRefExp* functionRefExpression = new SgFunctionRefExp(newCallfileInfo,functionSymbol,globalFunctionType);
        SgExprListExp* expressionList           = new SgExprListExp(newCallfileInfo);

        string codeLocation = fileName + "_" + StringUtility::numberToString(lineNum);
        SgStringVal* functionNameStringValue = new SgStringVal(newCallfileInfo,(char*)codeLocation.c_str());
        expressionList->append_expression(functionNameStringValue);
        SgFunctionCallExp* functionCallExp   = new SgFunctionCallExp(newCallfileInfo,functionRefExpression,expressionList,globalFu

     // create an expression type
     // SgTypeVoid* expr_type = new SgTypeVoid();
     // create an expression root
     // SgExpressionRoot * expr_root = new SgExpressionRoot(newCallfileInfo,functionCallExp,expr_type);

     // create an expression statement
        SgExprStatement* new_stmt = new SgExprStatement(newCallfileInfo,functionCallExp);

     // expr_root->set_parent(new_stmt);
     // new_stmt->set_expression(expr_root);
     // functionCallExp->set_parent(new_stmt->get_expression());
        functionCallExp->set_parent(new_stmt);

     // insert a statement into the function body
        block->prepend_statement(new_stmt);

    }
}
```

Figure 40.3: Example source code shows instrumentation to call a test function from the top of each function body in the application (part 3).

```
void foo()
   {
  // Should detect that foo IS called
  if (true) {
      int x = 3;
  }
  else {
      int x = 4;
   }
}

void foobar()
   {
      int y =4;
      switch (y) {
         case 1:
            //hello world
            break;
         case 2:

         case 3:

         default: {
            //
         }
      }

  // Should detect that foobar is NOT called
   }

int main()
   {
      if (true) {
      }
     foo();
     return 0;
   }
```

Figure 40.4: Example source code used as input to translator adding new function.

```
extern "C" void coverageTraceFunc1(char *textString);

void foo()
{
  coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutori
// Should detect that foo IS called
  if (true) {
    coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tuto
    int x = 3;
  }
  else {
    coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tuto
    int x = 4;
  }
}

void foobar()
{
  coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutori
  int y = 4;
  switch(y){
    coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tuto
    case 1:
{
      coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tu
//hello world
      break;
    }
    default:
{
      coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tu
{
        coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/
//
      }
    }
  }
// Should detect that foobar is NOT called
}

int main()
{
  coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutori
  if (true) {
    coverageTraceFunc1("/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tuto
  }
  foo();
  return 0;
}
```

Figure 40.5: Output of input to translator adding new function.

Chapter 41

Bug Seeding

Bug seeding is a technique used to construct example codes from existing codes which can be used to evaluate tools for the finding bugs in randomly selected existing applications. The idea is to seed an existing application with a known number of known bugs and evaluate the bug finding or security tool based on the percentage of the number of bugs found by the tool. If the bug finding tool can identify all known bugs then there can be some confidence that the tool detects all bugs of that type used to seed the application.

This example tutorial code is a demonstration of a more complete technique being developed in collaboration with NIST to evaluate tools for finding security flaws in applications. It will in the future be a basis for testing of tools built using ROSE, specifically Compass, but the techniques are not in any way specific to ROSE or Compass.

41.1 Input For Examples Showing Bug Seeding

Figure **??** shows the example input used for demonstration of bug seeding as a transformation.

```
void foobar()
   {
  // Static array declaration
     float array[10];

     array[0] = 0;

     for (int i=0; i < 5; i++)
        {
          array[i] = 0;
        }
   }
```

Figure 41.1: Example source code used as input to program in codes used in this chapter.

41.2 Generating the code representing the seeded bug

Figure 41.2 shows a code that traverses each IR node and for and modifies array reference index expressions to be out of bounds. The input code is shown in figure 41.1, the output of this code is shown in figure 41.3.

```
// This example demonstrates the seeding of a specific type
// of bug (buffer overflow) into any existing application to
// test bug finding tools.

#include "rose.h"
using namespace SageBuilder;
using namespace SageInterface;

namespace SeedBugsArrayIndexing {

class InheritedAttribute
   {
     public:
          bool isLoop;
          bool isVulnerability;
          InheritedAttribute() : isLoop(false),isVulnerability(false) {}
          InheritedAttribute(const InheritedAttribute & X) : isLoop(X.isLoop),isVulnerability(X.isVulnerability) {}
   };

class BugSeeding : public SgTopDownProcessing<InheritedAttribute>
   {
     public:
          InheritedAttribute evaluateInheritedAttribute (
             SgNode* astNode,
             InheritedAttribute inheritedAttribute );
   };

InheritedAttribute
BugSeeding::evaluateInheritedAttribute (
     SgNode* astNode,
     InheritedAttribute inheritedAttribute )
   {
  // Use this if we only want to seed bugs in loops
     bool isLoop = inheritedAttribute.isLoop           ||
                   (isSgForStatement(astNode) != NULL) ||
                   (isSgWhileStmt(astNode) != NULL)    ||
                   (isSgDoWhileStmt(astNode) != NULL);
  // Add Fortran support
     isLoop = isLoop || (isSgFortranDo(astNode) != NULL);

  // Mark future noes in this subtree as being part of a loop
     inheritedAttribute.isLoop = isLoop;

  // To test this on simple codes, optionally allow it to be applied everywhere
     bool applyEveryWhere = true;

     if (isLoop == true || applyEveryWhere == true)
        {
       // The inherited attribute is true iff we are inside a loop and this is a SgPntrArrRefExp.
          SgPntrArrRefExp *arrayReference = isSgPntrArrRefExp(astNode);
          if (arrayReference != NULL)
             {
            // Mark as a vulnerability
               inheritedAttribute.isVulnerability = true;

            // Now change the array index (to seed the buffer overflow bug)
               SgVarRefExp* arrayVarRef = isSgVarRefExp(arrayReference->get_lhs_operand());
               ROSE_ASSERT(arrayVarRef != NULL);
               ROSE_ASSERT(arrayVarRef->get_symbol() != NULL);
               SgInitializedName* arrayName = isSgInitializedName(arrayVarRef->get_symbol()->get_declaration());
               ROSE_ASSERT(arrayName != NULL);
               SgArrayType* arrayType = isSgArrayType(arrayName->get_type());
               ROSE_ASSERT(arrayType != NULL);
               SgExpression* arraySize = arrayType->get_index();

               SgTreeCopy copyHelp;
            // Make a copy of the expression used to hold the array size in the array declaration.
               SgExpression* arraySizeCopy = isSgExpression(arraySize->copy(copyHelp));
               ROSE_ASSERT(arraySizeCopy != NULL);

            // This is the existing index expression
               SgExpression* indexExpression = arrayReference->get_rhs_operand();
               ROSE_ASSERT(indexExpression != NULL);

            // Build a new expression: "array[n]" --> "array[n+arraySizeCopy]", where the arraySizeCopy is a size of "array"
               SgExpression* newIndexExpression = buildAddOp(indexExpression,arraySizeCopy);

            // Substitute the new expression for the old expression
               arrayReference->set_rhs_operand(newIndexExpression);
             }
        }

     return inheritedAttribute;
   }
}

int
main (int argc, char *argv[])
   {
     SgProject *project = frontend (argc, argv);
     ROSE_ASSERT (project != NULL);

     SeedBugsArrayIndexing::BugSeeding treeTraversal;
     SeedBugsArrayIndexing::InheritedAttribute inheritedAttribute;

     treeTraversal.traverseInputFiles (project,inheritedAttribute);

  // Running internal tests (optional)
     AstTests::runAllTests (project);

  // Output the new code seeded with a specific form of bug.
     return backend (project);
   }
```

Figure 41.2: Example source code showing how to seed bugs.

```
void foobar()
{
// Static array declaration
  float array[10UL];
  array[0 + 10UL] = 0;
  for (int i = 0; i < 5; i++) {
    array[i + 10UL] = 0;
  }
}
```

Figure 41.3: Output of input code using seedBugsExample_arrayIndexing.C

Part VI

Binary Support

Tutorials of using ROSE to handle binary executable files.

Chapter 42

Instruction Semantics

The Instruction Semantics layer in ROSE can be used to "evaluate" instructions and is controlled by a policy that defines the details of what "evaluate" means. For instance, given the following "xor" instruction, the X86InstructionSemantics class specifies that the value of the "eax" and "edx" registers are read, those two 32-bit values are xor'd together, and the 32-bit result is then written to the "eax" register. The policy defines what a 32-bit value is (it could be an integer, some representation of a constant, etc), how it is read and written to the registers, and how to compute an xor.

```
xor eax, edx
```

ROSE has a collection instruction semantic classes, one for each architecture. It also has a small collection of policies. This chapter briefly describes a policy that tracks constant values.

42.1 The FindConstantsPolicy Class

The FindConstantsPolicy is used to track constant values across an instruction trace. The basic idea is that ROSE "executes" the instructions one at a time in the instruction semantics layer, identifies constants, performs operations on those constants, and assigns constants to registers and memory locations. Each constant also maintains information about which instructions led to that particular constant's existence.

A "constant" is an abstract datum that has a known integer value, or a name corresponding to some unknown value, or a name and a known integer offset. Names take the form of the letter "v" (for "value") followed by a unique integer. Known values are represented as signed hexadecimal values in the output.

The findConstants.C program in the tests/roseTests/binaryTests directory (which is described herein) takes each function and processes the instructions of that function in address order. It makes no attempt to follow branches or any other kind of control flow, but serves to demonstrate a simple way to track constants.

```
#define \_\_STDC_FORMAT_MACROS
#include "rose.h"
```

```
#include "findConstants.h"
#include <inttypes.h>

/* Returns the function name if known, or the address as a string otherwise. */
static std::string
name\_or\_addr(const SgAsmFunction *f)
{
    if (f->get\_name()!="")
        return f->get\_name();

    char buf[128];
    SgAsmBlock *first\_bb = isSgAsmBlock(f->get\_statementList().front());
    sprintf(buf, "0x%"PRIx64, first\_bb->get\_id());
    return buf;
}

class AnalyzeFunctions : public SgSimpleProcessing {
  public:
    AnalyzeFunctions(SgProject *project) {
        traverse(project, postorder);
    }
    void visit(SgNode *node) {
        SgAsmFunction *func = isSgAsmFunction(node);
        if (func) {
            std::cout <<"===============================================\n"
                      <<"Constant propagation in function \"" <<name\_or\_addr(func) <<"\"\n"
                      <<"===============================================\n";
            FindConstantsPolicy policy;
            X86InstructionSemantics<FindConstantsPolicy, XVariablePtr> t(policy);
            std::vector<SgNode*> instructions = NodeQuery::querySubTree(func, V\_SgAsmx86Instruc
            for (size\_t i=0; i<instructions.size(); i++) {
                SgAsmx86Instruction *insn = isSgAsmx86Instruction(instructions[i]);
                ROSE\_ASSERT(insn);
                t.processInstruction(insn);
                RegisterSet rset = policy.currentRset;
                std::cout <<unparseInstructionWithAddress(insn) <<"\n"
                          <<rset;
            }
        }
    }
};

int main(int argc, char *argv[]) {
    AnalyzeFunctions(frontend(argc, argv));
}
```

Lines 30 through 40 are the main meat of the example. For each function, we construct a fresh policy. Since the policy holds the values of registers and memory, this resets them all to an initial state having unknown values. The instruction semantics engine depends on the policy, so we also create a new one for each function.

Then we loop over the instructions of the function in order of their addresses at lines 33 through 40. Each instruction is processed in turn by the X86InstructionSemantics object, adjusting the state of the associated policy.

Finally, the assembly language instruction is output followed by the values contained in the registers as a result of processing the instruction.

42.2 Sample Output

Here's some abbreviated output from running the "findConstants" test on a binary executable:

```
==============================================
Constant propagation in function "_init"
==============================================
0x80482c8:push   ebp
    ax = v62
    cx = v63
    dx = v64
    bx = v65
    sp = v66-0x4 [from 0x80482c8:push   ebp]
    bp = v67
    si = v68
    di = v69
    es = v70
    cs = v71
    ss = v72
    ds = v73
    fs = v74
    gs = v75
    cf = v76
    ?1 = v77
    pf = v78
    ?3 = v79
    af = v80
    ?5 = v81
    zf = v82
    sf = v83
    tf = v84
    if = v85
    df = v86
    of = v87
    iopl0 = v88
```

```
32      iopl1 = v89
33      nt = v90
34      ?15 = v91
35      memory = {
36          size=4; addr=v66-0x4 [from 0x80482c8:push   ebp]; value=v67
37      }
```

Line 4 indicates that the instruction “push ebp” is located at address 0x80482c8 and the following lines show the contents of registers and known memory addresses following execution of the “push.” One can readily see that each register has a unique constant of unknown value by virtue of each constant having a unique name. The stack pointer register (sp) has the constant “v66-0x4” obtained from this very instruction. If we had printed the registers prior to executing the “push” we would have seen that the original sp constant was “v66”. Therefore, this “push” instruction reduced the value of sp by four.

Line 36 indicates that the four bytes beginning at memory address “v66-0x4” (which happens to be the constant stored in the stack pointer register at line 9) contain the value “v67” (which is the constant stored in the bp register at line 10).

Therefore, it can be determined that “push ebp” decrements the stack pointer by four bytes, then copies a 32-bit value from the bp register to the memory pointed to by the new stack pointer.

```
0x80482c9:mov    ebp, esp
    ax = v62
    cx = v63
    dx = v64
    bx = v65
    sp = v66-0x4 [from 0x80482c8:push   ebp]
    bp = v66-0x4 [from 0x80482c8:push   ebp]
    si = v68
    di = v69
    es = v70
    cs = v71
    ss = v72
    ds = v73
    fs = v74
    gs = v75
    cf = v76
    ?1 = v77
    pf = v78
    ?3 = v79
    af = v80
    ?5 = v81
    zf = v82
    sf = v83
    tf = v84
    if = v85
```

```
    df = v86
    of = v87
    iopl0 = v88
    iopl1 = v89
    nt = v90
    ?15 = v91
    memory = {
        size=4; addr=v66-0x4 [from 0x80482c8:push   ebp]; value=v67
    }
```

The output after the “mov ebp, esp” instruction at address 0x80482c9 subsequent to the “push ebp” that we just saw, shows that the new stack pointer has been copied into the “bp” register and that nothing else has changed. A more interesting instruction follows...

```
0x80482cb:sub    esp, 0x8
    ax = v62
    cx = v63
    dx = v64
    bx = v65
    sp = v66-0xc [from 0x80482cb:sub    esp, 0x8]
    bp = v66-0x4 [from 0x80482c8:push   ebp]
    si = v68
    di = v69
    es = v70
    cs = v71
    ss = v72
    ds = v73
    fs = v74
    gs = v75
    cf = -v193-0x1 [from 0x80482cb:sub    esp, 0x8]
    ?1 = v77
    pf = -v187-0x1 [from 0x80482cb:sub    esp, 0x8]
    ?3 = v79
    af = -v191-0x1 [from 0x80482cb:sub    esp, 0x8]
    ?5 = v81
    zf = v190 [from 0x80482cb:sub    esp, 0x8]
    sf = v189 [from 0x80482cb:sub    esp, 0x8]
    tf = v84
    if = v85
    df = v86
    of = v197 [from 0x80482cb:sub    esp, 0x8]
    iopl0 = v88
    iopl1 = v89
    nt = v90
    ?15 = v91
    memory = {
```

```
        size=4; addr=v66-0x4 [from 0x80482c8:push   ebp]; value=v67
    }
```

The “sub esp, 0x8” subtracts eight from the value of the stack pointer register and then stores the result in the stack pointer register. This can be seen by the fact that the constant stored in the “sp” register has changed from “v66-0x4” to “v66-0xc.” One can also see that various flags have been modified, although we don’t know the values of any of them.

```
0x80482ce:call   0x8048364
    ax = v62
    cx = v63
    dx = v64
    bx = v65
    sp = v66-0x10 [from 0x80482ce:call   0x8048364]
    bp = v66-0x4 [from 0x80482c8:push   ebp]
    si = v68
    di = v69
    es = v70
    cs = v71
    ss = v72
    ds = v73
    fs = v74
    gs = v75
    cf = -v193-0x1 [from 0x80482cb:sub    esp, 0x8]
    ?1 = v77
    pf = -v187-0x1 [from 0x80482cb:sub    esp, 0x8]
    ?3 = v79
    af = -v191-0x1 [from 0x80482cb:sub    esp, 0x8]
    ?5 = v81
    zf = v190 [from 0x80482cb:sub    esp, 0x8]
    sf = v189 [from 0x80482cb:sub    esp, 0x8]
    tf = v84
    if = v85
    df = v86
    of = v197 [from 0x80482cb:sub    esp, 0x8]
    iopl0 = v88
    iopl1 = v89
    nt = v90
    ?15 = v91
    memory = {
        size=4; addr=v66-0x10 [from 0x80482ce:call   0x8048364]; value=0x80482d3 [from 0x80482ce
        size=4; addr=v66-0x4 [from 0x80482c8:push   ebp]; value=v67
    }
```

Now we get to our first branch-type instruction, a “call” to a particular address. Instruction semantics describe what the call instruction does to the registers and memory, but does not actually execute a call or process the called instructions. That means that a “ret” was not processed and thus we should see the return address sitting on the stack. In fact, we do: the stack pointer has been decremented by another four bytes and the memory address to which the stack pointer points contains the address of the instruction immediately after the "call".

42.3 Building on Instruction Semantics

The X86InstructionSemantics class and the policy classes can be extended to handle special cases. For instance, the X86InstructionSemantics class processes the "rep stosd" instruction in such a way that only one iteration of the "stosd" is considered. Sometimes it's more useful to process the entire repeated sequence in one step rather than iterating through the loop. Subclassing X86InstructionSemantics to override individual instructions or classes of instructions is simple. The subclass should redefine the "translate" method to do whatever is necessary for certain instructions while delegating to the superclass for all remaining instructions. For example:

```
/* Augments super::translate() to override rep\_stos instructions */
virtual void translate(SgAsmx86Instruction *insn) {
    switch (insn->get\_kind()) {
        case x86\_rep\_stosb: updateIP(insn); rep\_stos\_semantics<1>(insn); break;
        case x86\_rep\_stosw: updateIP(insn); rep\_stos\_semantics<2>(insn); break;
        case x86\_rep\_stosd: updateIP(insn); rep\_stos\_semantics<4>(insn); break;
        default: super::translate(insn); break;
    }
}
```

It's also possible to subclass the policies. For instance, if you need to do something special for binary AND operations on the stack pointer you could override the "and_" method in the policy.

Chapter 43

Binary Analysis

This chapter discusses the capabilities of ROSE to read, analyze and transform (transformations to the binary file format) binary executables.

Binary support in ROSE is currently based on a custom build *ROSE Disassembler* (for ARM, x86, and PowerPC).

The following code reads in a binary and creates a binary ROSE AST:

```
SgProject* project = frontend(argc,argv);
```

Similarily, one can unparse the AST to assembly using a call to the backend, cf. Figure **??**.

The best documentation for ROSE's binary analysis capabilities is found in the doxygen-generated API reference manual which can be found on the ROSE web site. Please refer to the following documented entities: class AsmUnparser, class Assembler, class BinaryLoader, namespace BinaryAnalysis, class Disassembler, class MemoryMap, class Partitioner, class SymbolicSemantics, class PartialSymbolicSemantics, class RegisterDictionary, and class X86InstructionSemantics.

43.1 The ControlFlowGraph

Based on a control flow traversal of the binary AST, a separate control flow graph is created that can be used for further analyses.

TODO: Describe recent work on binary CFG.

43.2 DataFlow Analysis

Based on the control flow many forms of dataflow analysis may be performed. Dataflow analyses available are:

43.2.1 Def-Use Analysis

Definition-Usage is one way to compute dataflow information about a binary program.

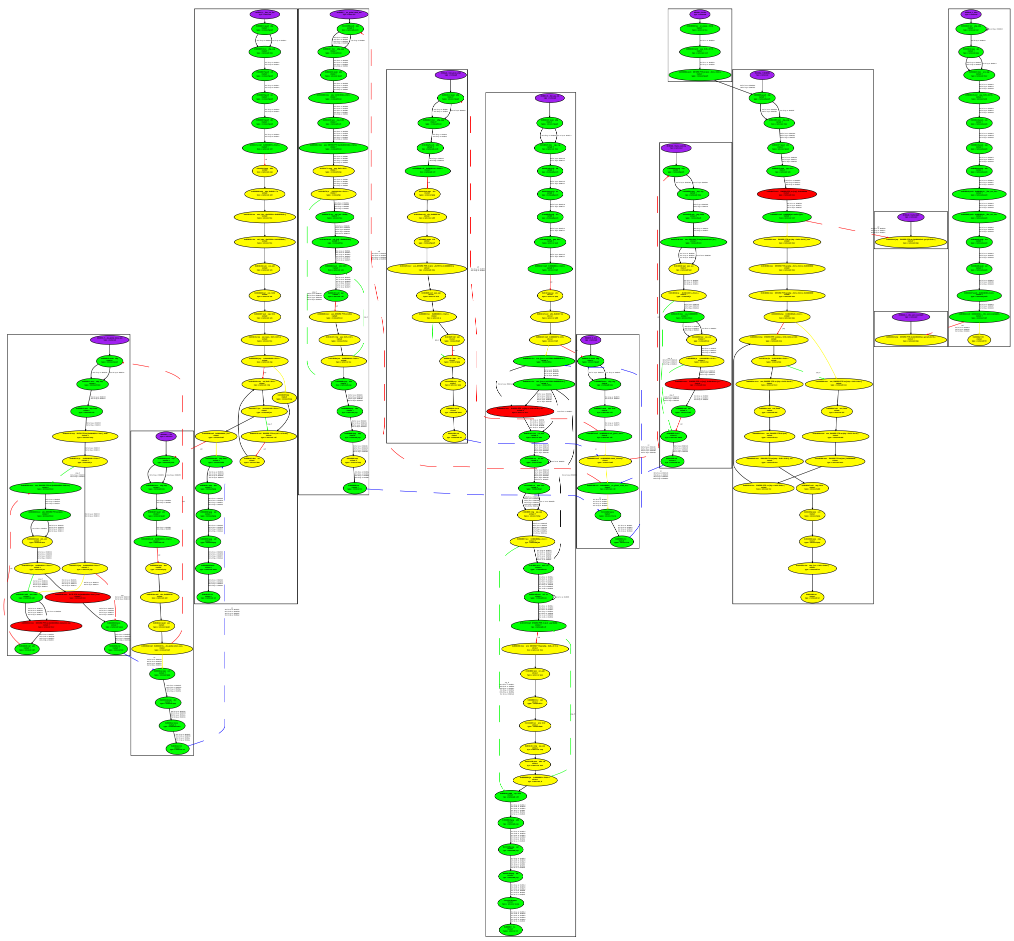


Figure 43.1: Dataflowflow graph for example program.

43.2.2 Variable Analysis

This analysis helps to detect different types within a binary. Currently, we use this analysis to detect interrupt calls and their parameters together with the def-use analysis. This allows us to track back the value of parameters to the calls, such as eax and therefore determine whether a interrupt call is for instance a write or read. Another feature is the buffer overflow analysis. By traversing the CFG, we can detect buffer overflows.

43.3 Dynamic Analysis

Recent work in ROSE has added support for dynamic analysis and for mixing of dynamic and static analysis using the Intel Pin framework. This optional support in ROSE requires a configure option (`--with-IntelPin=`< *path*>). The `path` in the configure option is the **absolute path** to the top level directory of the location of the Intel Pin distribution. This support for Intel Pin has only been tested on a 64bit Linux system using the most recent distribution of Intel Pin (version 2.6).

Note: The dwarf support in ROSE is currently incompatable with the dwarf support in Intel Pin. A message in the configuration of ROSE will detect if both support for Dwarf and Intel Pin are both specified and exit with an error message that they are incompatable options.

See `tutorial/intelPin` directory for examples using static and dynamic analysis. These example will be improved in the future, at the moment they just call the generation of the binary AST.

Note: We have added a fix to Intel Pin pin.H file:

```
// DQ (3/9/2009): Avoid letting "using" declarations into header files.
#ifndef REQUIRE_PIN_TO_USE_NAMESPACES
using namespace LEVEL_PINCLIENT;
#endif
```

so that the namespace of Intel Pin would not be a problem for ROSE. The development team have suggested that they may fix their use of "using" declarations for namespaces in their header files.

Also note that the path must be absolute since it will be the prefix for the **pin** executable to be run in the internal tests and anything else might be a problem if the path does not contain the current directory (**"."**). Or, perhaps we should test for this in the future.

Note 2: Linking to libdwarf.a is a special problem. Both ROSE and Intel Pin use libdwarf.a and both build shred libraries that link to the static version of the library (libdwarf.a). This is a problem im building Pin Tools since both the PinTool and librose.so will use a statically linked dwarf library (internally). This causes the first use of dwarf to fail, because there are then two versions of the same library in place. The solution is to force at most one static version of the library and let the other one be a shared library.

Alternatively both the Pin tool and librose.so can be built using the shared version of dwarf (libdwarf.so). There is a makefile rule in libdwarf to build the shared version of the library, but the default is to only build the static library (libdwarf.a), so use `make make libdwarf.so` to build the shared library. So we allow ROSE to link to the libdwarf.a (statically), which is how ROSE has always worked (this may be revisited in the future). And we force the Pin tool to link using the shared dwarf library (libdwarf.so). *Note: The specification of the location of libdwarf.so in the Intel Pin directory structure is problematic using* **rpath***, so for the case of using the Intel Pin package with ROSE please set the* **LD_LIBRARY_PATH** *explicitly (a better solution using* **rpath** *may be made available in the future).*

43.4 Analysis and Transformations on Binaries

This section demonstrates how the binary can be analyized and transformed via operations on the AST. In this tutorial example we will recognize sequences of NOP (No OPeration) instructions (both single byte and less common multi-byte NOPs) and replace them in the binary with as few multi-byte NOP instructions as required to overwrite the identified NOP sequence (also called a *nop sled*).

In the following subsections we will demonstrate three example codes that work together to demonstrate aspects of the binary analysis and transforamation using ROSE. All of these codes are located in the directory *tutorial/binaryAnalysis* of the ROSE distribution. We show specifically:

1. How to insert NOP sequences into binaries via source-to-source transformations. The tutorial example will demonstrate the trival insertion of random length nop sequences into all functions of an input source code application. The purpose of this is to really just generate arbitrary test codes upon which to use in sperately demonstrating (and testing) the binary analysis support and the binary transformation support (next). This example is shown in figure 43.2. The file name is:

2. How to identify sequences of NOP instructions in the binary (*nop sleds*). The tutorial example show how to identify arbitrary NOP sequences in the binary. Note that our approach looks only for single instructions that have no side-effect operations and thus qualify as a NOP, and specifically does not look at collections of multiple statements that taken togehter have no side-effect and thus could be considered a NOP sequence. Our test on each instruciton is isolated to the `SageInterface::isNOP(SgAsmInstruction*)` function. Initially this function only detects NOP instructions that are single and multi-byte Intel x86 suggested NOP instructions. The catagory of NOP instructions that have no side-effects is broader than this and will be addressed more completely in the future. This example is shown in figures 43.3, 43.4, and 43.5.

3. How transformations on the binary executable are done to support rewriting all identified NOP sequences as Intel x86 suggested NOP instructions. Importantly, this tutorial example of an AST rewrite does not change the size of any part of the final executable. This example is shown in figure 43.6.

43.4.1 Source-to-source transformations to introduce NOPs

This tutorial example (see figure 43.2) is a source-to-source trnasformation that is used to generate test codes for the binary NOP sequence detection example and the NOP sequence transformation example (the next two tutorial examples). This tutorial uses C language *asm* statements that are inserted into each function in the source code, the resulting generated source code (with

```
// This translator does a source-to-source transformation on
// the input source code to introduce asm statements of NOP
// instructions at the top of each function.

#include "rose.h"

using namespace std;
using namespace SageInterface;
using namespace SageBuilder;

class NopTransform : public SgSimpleProcessing
   {
     public:
          NopTransform() { srand ( time(NULL) ); }
          void visit ( SgNode* n );
   };

void NopTransform::visit ( SgNode* n )
   {
     SgFunctionDefinition* functionDefinition = isSgFunctionDefinition(n);
     if (functionDefinition != NULL)
        {
       // Introduce NOP's into function definition
          SgBasicBlock* functionBody = functionDefinition->get_body();
          ROSE_ASSERT(functionBody != NULL);

       // Use a multi-byte NOP just for added fun.
       // SgAsmStmt* nopStatement = buildAsmStatement("nop");

       // Generate a single random multi-byte NOP (1-9 bytes long including no NOP, when n == 0)
          int n = rand() % 10;
          if (n > 0)
             {
               printf ("Introducing a multi-byte NOP instruction (n = %d) at the top of function %s \n",n,functionDefiniti
               SgAsmStmt* nopStatement = buildMultibyteNopStatement(n);

            // Add to the front o the list of statements in the function body
               prependStatement(nopStatement,functionBody);
             }
        }
   }

int main( int argc, char * argv[] )
   {
  // Generate the ROSE AST.
     SgProject* project = frontend(argc,argv);

  // AST consistency tests (optional for users, but this enforces more of our tests)
     AstTests::runAllTests(project);

     NopTransform t;
     t.traverse(project,preorder);

  // regenerate the original executable.
     return backend(project);
   }
```

Figure 43.2: Source-to-source transformation to introduce NOP assemble instructions in the generated binary executable.

asm statements) is then compiled to generate inputs to use in our detection and transformation of NOP sequences. Although it is possible, the backend compiler (at least the one we are using), does not optimize away these *asm* statements that represent NOP instructions. In general all C and C++ compilers turn off optimizations upon encountering the *asm* language construct, so it is easy to use this approach to build binaries from arbitrary source code that have a range of properties. Seeding the source code with such properties causes the binary to have the similar properties. We include this example in the tutorial because it is a cute example of how we can combine source code analsys with binary analysis (in this case we are only supporting testing of the binary analysis). Much more interesting example of the connection of source code and binary analysis are available.

```
class CountTraversal : public SgSimpleProcessing
   {
     public:
       // Local Accumulator Attribute
          int count;
          bool previousInstructionWasNop;
          SgAsmInstruction* nopSequenceStart;
          std::vector<std::pair<SgAsmInstruction*,int> > nopSequences;

          CountTraversal() : count(0), previousInstructionWasNop(false) {}
          void visit ( SgNode* n );
   };
```

Figure 43.3: Header file for the traversal used to identify the NOP sequences in the binary executable.

43.4.2 Detection of NOP sequences in the binary AST

The tutorial example (see figures 43.3, 43.4, and 43.5) shows the detection of NOP sequences and is separated into three figures (the header file, the source code, and the main program). The header file and source file will be reused in the next tutorial example to demonstrate the transformation of the NOP sequences that this tutorial identifies. The transformation will be done on the AST (and a new modified executable generated by unparsing the AST).

Using a simple preorder traversal over the AST we identify independent sequences of NOP instructions. This traversal save the sequence so that it can be used in both the tutorial example that detects the NOP sequences and also the tutorial example that will transform them to be more compact multi-byte NOP sequences that will use the suggested Intel x86 multi-byte NOP instructions.

In this example, the sequence of a NOP sequence is saved as the address of the starting NOP instruction and the number of instructions in the sequence. The function `SageInterface::isNOP(SgAsmInstruction*)` is used to test of an instruction is a NOP. Presently this test only identifies single or multi-byte NOP instrucitons that have been suggested for use as single and multi-byte NOP instructions by Intel. Other instruction may semantically be equivalent to a NOP instruction and they are not identified in this initial work. Later work may include this capability and for this purpose we have isolated out the `SageInterface::isNOP(SgAsmInstruction*)`. Note also that sequences of instructions may be semantically equivalent to a NOP sequence and we also do not attempt to identify these (separate interface functions in the `SageInterface` namespace have been defined to support this work; these functions are presently unimplemented.

43.4.3 Transformations on the NOP sequences in the binary AST

Using the results from the previously presented traversal to identify NOP sequences, we demonstrate the transformation to change these locations of the binary (in the AST) and then regenerate the executable to have a different representation. The new representation is a more clear (obvious within manual binary analysis) and likely more compressed representation using only the suggested Intel x86 multi-byte NOP.

43.4.4 Conclusions

FIXME: *Not sure if the resulting executable will use the assembler on each instruction in the generated binary from the unparsing within ROSE. This might require some more work. So at the moment the AST is transformed and I have to look into if the binary sees the effect of the transformation in the AST.*

The identification and transformation have been deomonstrated on the AST and can be expressed in the binary binary by calling the *backend()* function in ROSE; which will regenerate the executable. The capability to regenerate the executable will not always result in a properly formed executable that can be executed, this depends upon the transformations done and does ROSE does not have specialized support for this beyond regnerating the binary executable from the AST. In the case of this NOP transformation we have not changed the size of any part of the binary so any relative offsets need not be fixed up. Assumeing we have not accedentaly interpreted data that had values that matched parts of the opcodes that were transformed, then resulting binary executable should execute without problems. This transformation however makes clear how critical it is that data not be interpreted as instructions (which could randomly be interpreted as NOPs in the case of this tutorial example).

```
#include "rose.h"

#include "sageInterfaceAsm.h"
#include "detectNopSequencesTraversal.h"
#include "stringify.h"

using namespace std;
using namespace SageInterface;

void CountTraversal::visit ( SgNode* n )
   {
     SgAsmInstruction* asmInstruction = isSgAsmInstruction(n);
     if (asmInstruction != NULL)
        {
       // Use the new interface support for this (this detects all multi-byte nop instructions).
          if (SageInterface::isNOP(asmInstruction) == true)
             {
               if (previousInstructionWasNop == true)
                  {
                 // Increment the length of the identified NOP sequence
                    count++;
                  }
                 else
                  {
                    count = 1;
                 // Record the starting address of the NOP sequence
                    nopSequenceStart = asmInstruction;
                  }

               previousInstructionWasNop = true;
             }
            else
             {
               if (count > 0)
                  {
                 // Report the sequence when we have detected the end of the sequence.
                    SgAsmFunction* functionDeclaration = getAsmFunction(asmInstruction);
                    printf ("Reporting NOP sequence of length %3d at address %zu in function %s (reason for t
                         count,nopSequenceStart->get_address(),functionDeclaration->get_name().c_str(),
                            functionDeclaration->get_reason(),
                            stringifySgAsmFunctionFunctionReason(functionDeclaration->get_reason()).c_str());

                    nopSequences.push_back(pair<SgAsmInstruction*,int>(nopSequenceStart,count));

                    SgAsmBlock* block = isSgAsmBlock(nopSequenceStart->get_parent());
                    ROSE_ASSERT(block != NULL);
                    SgAsmStatementPtrList & l = block->get_statementList();

                 // Now iterate over the nop instructions in the sequence and report the lenght of each (can
                    SgAsmStatementPtrList::iterator i = find(l.begin(),l.end(),nopSequenceStart);
                    ROSE_ASSERT(i != l.end());
                    int counter = 0;
                    while ( (*i != asmInstruction) && (i != l.end()) )
                       {
                         printf ("--- NOP #%2d is length = %2d \n",counter++,(int)isSgAsmInstruction(*i)->get
                         i++;
                       }
                  }

               count = 0;
               previousInstructionWasNop = false;
             }
        }
   }
```

Figure 43.4: Example code to identify the NOP sequences in the binary executable.

```
// This example ROSE translator just does an analysis of the input binary.
// The existence of NOP sequences are detected and reported.  For each
// NOP in the sequence the size of the NOP instruction is reported.
// Note that all multi-byte NOP instructions are detected and so the
// reported size of each instruction in the NOP sequence can vary.
// Intel multi-byte NOP instructions can be 1-9 bytes long.

#include "rose.h"
#include "detectNopSequencesTraversal.h"

int main( int argc, char * argv[] )
   {
  // Generate the ROSE AST.
     SgProject* project = frontend(argc,argv);

  // AST consistency tests (optional for users, but this enforces more of our tests)
     AstTests::runAllTests(project);

     CountTraversal t;
     t.traverse(project,preorder);

  // regenerate the original executable.
     return backend(project);
   }
```

Figure 43.5: Main program using the traversal to identify the NOP sequences in the binary executable.

```
// This tutorial example show how to introduce
// transformations on the input binary exectutable.

#include "rose.h"

#include "sageInterfaceAsm.h"

#include "detectNopSequencesTraversal.h"

using namespace std;
using namespace SageInterface;
using namespace SageBuilderAsm;


class NopReplacementTraversal : public SgSimpleProcessing
   {
     public:
       // Local Accumulator Attribute
          std::vector<std::pair<SgAsmInstruction*,int> > & nopSequences;
          std::vector<std::pair<SgAsmInstruction*,int> >::iterator listIterator;
          SgAsmStatementPtrList deleteList;

          NopReplacementTraversal(std::vector<std::pair<SgAsmInstruction*,int> > & X) : nopSequences(X)
             {
               listIterator = nopSequences.begin();
             }

          void visit ( SgNode* n );
   };


void NopReplacementTraversal::visit ( SgNode* n )
   {
     SgAsmInstruction* asmInstruction = isSgAsmInstruction(n);
     if (asmInstruction != NULL && asmInstruction == listIterator->first)
        {
       // This is the instruction in the AST that matches the start of one of the NOP sequences.
          SgAsmBlock* block = isSgAsmBlock(asmInstruction->get_parent());
          int numberOfOriginalNopInstructions = listIterator->second;

          printf ("numberOfOriginalNopInstructions = %d \n",numberOfOriginalNopInstructions);

       // SgAsmBlock* block = isSgAsmBlock(asmInstruction->get_parent());
          ROSE_ASSERT(block != NULL);
          SgAsmStatementPtrList & l = block->get_statementList();

       // Now iterate over the nop instructions in the sequence and report the lenght of each (can be multi-byte nop instructions
          SgAsmStatementPtrList::iterator i = find(l.begin(),l.end(),asmInstruction);
          ROSE_ASSERT(i != l.end());
          int nop_sled_size = 0;
          for (int j = 0; j < numberOfOriginalNopInstructions; j++)
             {
               ROSE_ASSERT(i != l.end());
               printf ("--- NOP #%2d is length = %2d \n",j,(int)isSgAsmInstruction(*i)->get_raw_bytes().size());
               nop_sled_size += (int)isSgAsmInstruction(*i)->get_raw_bytes().size();
               i++;
             }

          printf ("nop_sled_size = %d \n",nop_sled_size);

       // From the size of the NOP sled, compute how many multi-byte NOPs we will want to use to cover the same space in the bina
       // This code is specific to x86 supporting multi-byte NOPs in sized 1-9 bytes long.
          const int MAX_SIZE_MULTIBYTE_NOP = 9;
          int numberOfNopSizeN [MAX_SIZE_MULTIBYTE_NOP+1];

       // 0th element of numberOfNopSizeN is not used.
          numberOfNopSizeN[0] = 0;
          for (int i = MAX_SIZE_MULTIBYTE_NOP; i > 0; i--)
             {
            // int numberOfNopSize9 = nop_sled_size / 9;
               numberOfNopSizeN[i] = nop_sled_size / i;
               nop_sled_size -= numberOfNopSizeN[i] * i;
               if (numberOfNopSizeN[i] > 0)
                    printf ("numberOfNopSizeN[%d] = %d \n",i,numberOfNopSizeN[i]);
             }

       // Now rewrite the AST to use the number of multi-byte NOPS specified
       // in the numberOfNopSizeN array for each size of multi-byte NOP.

          printf ("Rewrite the AST here! \n");

       // Ignore the 0th element of numberOfNopSizeN (since a 0 length NOP does not make sense).
          for (int i = 1; i <= MAX_SIZE_MULTIBYTE_NOP; i++)
             {
            // Build this many bytes of this size
               for (int j = 0; j < numberOfNopSizeN[i]; j++)
                  {
                 // We want to build a (binary AST node) SgAsmInstruction object instead of a (source code AST node) SgAsmStmt.
                 // SgAsmStmt* nopStatement = buildMultibyteNopStatement(i);
                 // SgAsmInstruction* multiByteNop = makeInstruction(x86_nop, "nop", modrm);
                    SgAsmInstruction* multiByteNopInstruction = buildMultibyteNopInstruction(i);

                 // Add to the front o the list of statements in the function body
                 // prependStatement(nopStatement,block);
                    insertInstructionBefore(/*target*/ asmInstruction,/*new instruction*/ multiByteNopInstruction);
                  }
             }

       // Now iterate over the nop instructions in the sequence and report the lenght of each (can be multi-byte nop instructions
          i = find(l.begin(),l.end(),asmInstruction);
          ROSE_ASSERT(i != l.end());
          for (int j = 0; j < numberOfOriginalNopInstructions; j++)
             {
               ROSE_ASSERT(i != l.end());
            // printf ("Deleting original NOP instruction #%2d is length = %2d \n",j,(int)isSgAsmInstruction(*i)->get_raw_bytes(

            // Removing the original NOP instruction.
            // removeStatement(*i);
            // removeInstruction(*i);
               deleteList.push_back(*i);
```

Chapter 44

Binary Construction

ROSE is normally used in such a way that a file (source code or binary) is parsed to construct an AST, then operations are performed on the AST, and the modified AST is unparsed to create a new source or binary file. However, it is also possible to construct an AST explicitly without parsing and then use that AST to generate the output. The AST construction interface for binary files was designed so that working files could be created simply, while still providing methods to control the finer details of the resulting file.

The example in this chapter shows how to construct a statically linked ELF executable containing a small ".text" section that simply causes the process to exit with a specific non-zero value.

44.1 Constructors

The AST node constructors are designed to construct the tree from the root down, and thus generally take the parent node as an argument. Nodes that refer to other nodes as prerequisites also take those prerequisites as arguments. For instance, an ELF Relocation Section is a child of the ELF File Header but also needs an ELF Symbol Table and therefore takes both objects as constructor arguments.

44.2 Read-Only Data Members

When two or more file formats have a similar notion then that notion is represented in a base class. However, part of the information may continue to survive in the specialized class. In these situations modifications to the specilized data will probably be overridden by the generic values from the base class. For instance, all formats have a notion of byte order which is represented in the base class SgAsmGenericHeader as little- or big-endian (an enumeration constant). The ELF specification provides an 8-bit unsigned field to store the byte order and therefore has potentially more than two possibilities. Any value assigned to the ELF-specific byte order will likely be overwritten by the generic byte order before the AST is unparsed.

A similar situation arises with section offsets, sizes, memory mappings, permissions, etc. The SgAsmGenericSection class represents ELF Sections, ELF Segments, PE Objects, and other contiguous regions of a file and has methods for obtaining/setting these values. In addition, many of the formats have some sort of table that describes these sections and which also contains similar information (e.g., the ELF Segment Table, a.k.a., the ELF Program Header Table). As above, the generic representation of these notions (stored in SgAsmGenericSection) override the format-specific values (stored in SgAsmElfSegmentEntry).

ROSETTA doesn't make a distinction between data members that can be user-modified and data members that should be modified only by the parser. Therefore it is up to the user to be aware that certain data members will have their values computed or copied from other locations in the AST during the unparsing phase.

44.3 Constructing the Executable File Container

All executable files are stored as children of an SgAsmGenericFile node. The children are file format headers (SgAsmGenericHeader) such as an ELF File Header (SgAsmElfFileHeader). This design allows a single executable file to potentially contain more than one executable and is used by formats like Windows-PE where the file contains a DOS File Header as well as a PE File Header.

For the purposes of this example the SgAsmGenericFile node will serve as the root of the AST and therefore we do not need to specify a parent in the constructor argument list.

```
SgAsmGenericFile *ef = new SgAsmGenericFile;
```

44.4 Constructing the ELF File Header

The ELF File Header is the first thing in the file, always at offset zero. File headers are always children of an SgAsmGenericFile which is specified as the constructor argument.

The section constructors (a file header is a kind of section) always create the new section so it begins at the current end-of-file and contains at least one byte. This ensures that each section has a unique starting address, which will be important when file memory is actually allocated and sections need to be moved around–the allocator needs to know the relative positions of the sections in order to correctly relocate them.

If we were parsing an ELF file we would usually use ROSE's frontend() method. However, one can also parse the file by first constructing the SgAsmElfFileHeader and then invoking its parse() method, which parses the ELF File Header and everything that can be reached from that header.

We use the typical 0x400000 as the virtual address of the main LOAD segment, which occupies the first part of the file up through the end of the ".text" section (see below). ELF File Headers don't actually store a base address, so instead of assigning one to the SgAsmElfFileHeader we'll leave the header's base address at the default zero and add base_va explicitly whenever we need to.

```
SgAsmElfFileHeader *fhdr = new SgAsmElfFileHeader(ef);
fhdr->get_exec_format()->set_word_size(8);                  /* default is 32-bit; we want 64-bit */
fhdr->set_isa(SgAsmExecutableFileFormat::ISA_X8664_Family); /* instruction set architecture; default
rose_addr_t base_va = 0x400000;                             /* base virtual address */
```

44.5 Constructing the ELF Segment Table

ELF executable files always have an ELF Segment Table (also called the ELF Program Header Table), which usually appears immediately after the ELF File Header. The ELF Segment Table describes contiguous regions of the file that should be memory mapped by the loader. ELF Segments don't have names–names are imparted to the segment by virtue of the segment also being described by the ELF Section Table, which we'll create later.

Being a contiguous region of the file, an ELF Segment Table (SgAsmElfSegmentTable) is derived from SgAsmGenericSection. All non-header sections have a header as a parent, which we supply as an argument to the constructor. Since all ELF Segments will be children of the ELF File Header rather than children of the ELF Segment Table, we could define the ELF Segment Table at the end rather than here. But defining it now is an easy way to get it located in its usuall location immediately after the ELF File Header.

```
SgAsmElfSegmentTable *segtab = new SgAsmElfSegmentTable(fhdr);
```

44.6 Constructing the .text Section

ROSE doesn't treat a ".text" section as being anything particularly special–it's just a regular SgAsmElfSection, which derives from SgAsmGenericSection. However, in this example, we want to make sure that our ".text" section gets initialized with some instructions. The easiest way to do that is to specialize SgAsmElfSection and override or augment a few of the virtual functions.

We need to override two functions. First, the calculate_sizes() function should return the size we need to store the instructions. We'll treat the instructions as an array of entries each entry being one byte of the instruction stream. In other words, each "entry" is one byte in length consisting of one required byte and no optional bytes.

We need to also override the unparse() method since the base class will just fill the ".text" section with zeros. The SgAsmGenericSection::write method we use will write the instructions starting at the first byte of the section.

Finally, we need to augment the reallocate() method. This method is reponsible for allocating space in the file for the section and performing any other necessary pre-unparsing actions. We don't need to allocate space since the base class's method will take care of that in conjuction with our version of calculate_sizes(), but we do need to set a special ELF flag (SHF_ALLOC) in the ELF Segment Table entry for this section. There's a few ways to accomplish this. We do it this way because the ELF Section Table Entry is not created until later and we want to demonstrate how to keep all .text-related code in close proximity.

```
class TextSection : public SgAsmElfSection {
public:
    TextSection(SgAsmElfFileHeader *fhdr, size_t ins_size, const unsigned char *ins_bytes)
: SgAsmElfSection(fhdr), ins_size(ins_size), ins_bytes(ins_bytes)
{}
    virtual rose_addr_t calculate_sizes(size_t *entsize, size_t *required, size_t *optional, size_t *entcount)
if (entsize)  *entsize  = 1;                          /* an "entry" is one byte of instruction */
if (required) *required = 1;                          /* each "entry" is also stored in one byte of the file */
if (optional) *optional = 0;                          /* there is no extra data per instruction byte */
```

```
if (entcount) *entcount = ins_size;                /* number of "entries" is the total instruction
return ins_size;                                   /* return value is section size required */
    }
    virtual bool reallocate() {
bool retval = SgAsmElfSection::reallocate();       /* returns true if size or position of any secti
SgAsmElfSectionTableEntry *ste = get_section_entry();
ste->set_sh_flags(ste->get_sh_flags() | 0x02); /* set the SHF_ALLOC bit */
return retval;
    }
    virtual void unparse(std::ostream &f) const {
write(f, 0, ins_size, ins_bytes);                  /* Write the instructions at offset zero in sect
    }
    size_t ins_size;
    const unsigned char *ins_bytes;
};
```

The section constructors and reallocators don't worry about alignment issues–they always allocate from the next available byte. However, instructions typically need to satisfy some alignment constraints. We can easily adjust the file offset chosen by the constructor, but we also need to tell the reallocator about the alignment constraint. Even if we didn't ever resize the ".text" section the reallocator could be called for some other section in such a way that it needs to move the ".text" section to a new file offset.

For the purpose of this tutorial we want to be very picky about the location of the ELF Segment Table. We want it to immediately follow the ELF File Header without any intervening bytes of padding. At the current time, the ELF File Header has a size of one byte and will eventually be reallocated. When we reallocate the header the subsequent sections will need to be shifted to higher file offsets. When this happens, the allocator shifts them all by the same amount taking care to satisfy all alignment constraints, which means that an alignment constraint of byte bytes on the ".text" section will induce a similar alignment on the ELF Segment Table. Since we don't want that, the best practice is to call reallocate() now, before we create the ".text" section.

```
ef->reallocate();                                          /* Give existing sections a chance to al
static const unsigned char instructions[] = {0xb8, 0x01, 0x00, 0x00, 0x00, 0xbb, 0x56, 0x00, 0x00, 0
SgAsmElfSection *text = new TextSection(fhdr, NELMTS(instructions), instructions);
text->set_purpose(SgAsmGenericSection::SP_PROGRAM);        /* Program-supplied text/data/etc. */
text->set_offset(ALIGN_UP(text->get_offset(), 4));         /* Align on an 8-byte boundary */
text->set_file_alignment(4);                               /* Tell reallocator about alignment cons
text->set_mapped_alignment(4);                             /* Alignment constraint for memory mappi
text->set_mapped_rva(base_va+text->get_offset());          /* Mapped address is based on file offse
text->set_mapped_size(text->get_size());                   /* Mapped size is same as file size */
text->set_mapped_rperm(true);                              /* Readable */
text->set_mapped_wperm(false);                             /* Not writable */
text->set_mapped_xperm(true);                              /* Executable */
```

At this point the text section doesn't have a name. We want to name it ".text" and we want those characters to eventually be stored in the ELF file in a string table which we'll provide later. In ELF, section names are represented by the section's entry in the ELF Section Table as an offset into an ELF String Table for a NUL-terminated ASCII string. ROSE manages strings using the SgAsmGenericString class, which has two subclasses: one for strings that aren't stored in the executable file (SgAsmBasicString) and one for strings that are stored in the file (SgAsmStoredString). Both are capable of string an std::string value and querying its byte offset (although SgAsmBasicString::get_offset() will always return SgAsmGenericString::unallocated). Since we haven't added the ".text" section to the ELF Section Table yet the new section has an SgAsmBasicString name. We can assign a string to the name now and the string will be allocated in the ELF file when we've provided further information.

```
text->get_name()->set_string(".text");
```

The ELF File Header needs to know the virtual address at which to start running the program. In ROSE, virtual addresses can be attached to a specific section so that if the section is ever moved the address is automatically updated. Some formats allow more than one entry address which is why the method is called add_entry_rva() rather than set_entry_rva(). ELF, however, only allows one entry address.

```
rose_rva_t entry_rva(text->get_mapped_rva(), text);
fhdr->add_entry_rva(entry_rva);
```

44.7 Constructing a LOAD Segment

ELF Segments define parts of an executable file that should be mapped into memory by the loader. A program will typically have a LOAD segment that begins at the first byte of the file and continues through the last instruction (in our case, the end of the ".text" section) and which is mapped to virtual address 0x400000.

We've already created the ELF Segment Table, so all we need to do now is create an ELF Segment and add it to the ELF Segment Table. ELF Segments, like ELF Sections, are represented by SgAsmElfSection. An SgAsmElfSection is an ELF Section if it has an entry in the ELF Section Table, and/or it's an ELF Segment if it has an entry in the ELF Segment Table. The methods get_section_entry() and get_segment_entry() retrieve the actual entries in those tables.

Recall that the constructor creates new sections located at the current end-of-file and containing one byte. Our LOAD segment needs to have a different offset and size.

```
SgAsmElfSection *seg1 = new SgAsmElfSection(fhdr);          /* ELF Segments are represented by SgAsmElfSection
seg1->get_name()->set_string("LOAD");                       /* Segment names aren't saved (but useful for debu
seg1->set_offset(0);                                        /* Starts at beginning of file */
seg1->set_size(text->get_offset() + text->get_size());      /* Extends to end of .text section */
seg1->set_mapped_rva(base_va);                              /* Typically mapped by loader to this memory addres
seg1->set_mapped_size(seg1->get_size());                    /* Make mapped size match size in the file */
seg1->set_mapped_rperm(true);                               /* Readable */
seg1->set_mapped_wperm(false);                              /* Not writable */
seg1->set_mapped_xperm(true);                               /* Executable */
segtab->add_section(seg1);                                  /* Add definition to ELF Segment Table */
```

44.8 Constructing a PAX Segment

This documentation shows how to construct a generic ELF Segment, giving it a particular file offset and size. ELF Segments don't have names stored in the file, but we can assign a name to the AST node to aid in debugging–it just won't be written out. When parsing an ELF file, segment names are generated based on the type stored in the entry of the ELF Segment Table. For a PAX segment we want this type to be PT_PAX_FLAGS (the default is PT_LOAD).

```
SgAsmElfSection *pax = new SgAsmElfSection(fhdr);
pax->get_name()->set_string("PAX Flags");                    /* Name just for debugging */
pax->set_offset(0);                                          /* Override address to be at zero rather
pax->set_size(0);                                            /* Override size to be zero rather than
segtab->add_section(pax);                                    /* Add definition to ELF Segment Table *
pax->get_segment_entry()->set_type(SgAsmElfSegmentTableEntry::PT_PAX_FLAGS);
```

44.9 Constructing a String Table

An ELF String Table always corresponds to a single ELF Section of class SgAsmElfStringSection and thus you'll often see the term "ELF String Section" used interchangeably with "ELF String Table" even though they're two unique but closely tied classes internally.

When the ELF String Section is created a corresponding ELF String Table is also created under the covers. Since string tables manage their own memory in reponse to the strings they contain, one should never adjust the size of the ELF String Section (it's actually fine to enlarge the section and the new space will become free space in the string table).

ELF files typically have multiple string tables so that section names are in a different section than symbol names, etc. In this tutorial we'll create the section names string table, typically called ".shstrtab", but use it for all string storage.

```
SgAsmElfStringSection *shstrtab = new SgAsmElfStringSection(fhdr);
shstrtab->get_name()->set_string(".shstrtab");
```

44.10 Constructing an ELF Section Table

We do this last because we want the ELF Section Table to appear at the end of the file and this is the easiest way to achieve that. There's really not much one needs to do to create the ELF Section Table other than provide the ELF File Header as a parent and supply a string table.

The string table we created above isn't activated until we assign it to the ELF Section Table. The first SgAsmElfStringSection added to the SgAsmElfSectionTable becomes the string table for storing section names. It is permissible to add other sections to the table before adding the string table.

```
SgAsmElfSectionTable *sectab = new SgAsmElfSectionTable(fhdr);
sectab->add_section(text);                                   /* Add the .text section */
sectab->add_section(shstrtab);                               /* Add the string table to store section
```

44.11 Allocating Space

Prior to calling unparse(), we need to make sure that all sections have a chance to allocate space for themselves, and perform any other operations necessary. It's not always possible to determine sizes at an earlier time, and most constructors would have declared sizes of only one byte.

The reallocate() method is defined in the SgAsmGenericFile class since it operates over the entire collection of sections simultaneously. In other words, if a section needs to grow then all the sections located after it in the file need to be shifted to higher file offsets.

```
ef->reallocate();
```

The reallocate() method has a shortcoming (as of 2008-12-19) in that it might not correctly update memory mappings in the case when the mapping for a section is inferred from the mapping of a containing segment. This can happen in our example since the ".text" section's memory mapping is a function of the LOAD Segment mapping. The work-around is to adjust mappings for these situations and then call reallocate() one final time. This final reallocate() call won't move any sections, but should always be the last thing to call before unparsing() (it gives sections a chance to update data dependencies which is not possible during unparse() due to its const nature).

```
text->set_mapped_rva(seg1->get_mapped_rva()+(text->get_offset()-seg1->get_offset()));
ef->reallocate(); /*won't resize or move things this time since we didn't modify much since the last call to re
```

44.12 Produce a Debugging Dump

A debugging dump can be made with the following code. This dump will not be identical to the one produced by parsing and dumping the resulting file since we never parsed a file (a dump contains some information that's parser-specific).

```
ef->dump(stdout);
SgAsmGenericSectionPtrList all = ef->get_sections(true);
for (size_t i=0; i<all.size(); i++) {
    fprintf(stdout, "Section %zu:\n", i);
    all[i]->dump(stdout, "    ", -1);
}
```

44.13 Produce the Executable File

The executable file is produced by unparsing the AST.

```
std::ofstream f("a.out");
ef->unparse(f);
```

Note that the resulting file will not be created with execute permission–that must be added manually.

Chapter 45

Dwarf Debug Support

DWARF is a widely used, standardized debugging data format. DWARF was originally designed along with ELF, although it is independent of object file formats. The name is a pun on "ELF" that has no official meaning but "may be an acronym for 'Debug With Attributed Record Format'". See Wikipedia for more information about the Dwarf debug format.

This chapter presents the support in ROSE for Dwarf 3 debug information; its representation in the AST and its use in binary analysis tools. This work is part of general work to support as much information as possible about binaries.

In the following sections we use a small example (see figure 45.1) that demonstrates various features of Dwarf. The source code of our binary example is:

```
// Test code to demonstration of dwarf support.
// Designed to be small because Dwarf is verbose.

namespace example_namespace
   {
     int a;
   };


int main()
   {
     int x = 42;

  // Loops are not recognised in Dwarf...
     for (int i = 0; i < 10; i++)
        {
          x = (x % 2) ? x * 2 : x + 1;
        }

     return 0;
   }
```

Figure 45.1: Example source code used to generate Dwarf AST for analysis.

Much larger binaries can be analyzed, but such larger binary executables are more difficult to present (in this tutorial).

45.1 ROSE AST of Dwarf IR nodes

ROSE tools process the binary into an AST that is used to represent all the information in the binary executable. Figure 45.2 shows the subset of that AST (which includes the rest of the binary file format and the disassembled instructions) that is specific to Dwarf. A command line option (*-rose:visualize_dwarf_only*) is used to restrict the generated dot file visualization to just the Dwarf information. This option is used in combination with *-rose:read_executable_file_format_only* to process only the binary file format (thus skipping the instruction disassembly).

45.2 Source Position to Instruction Address Mapping

One of the valuable parts of Dwarf is the mapping between the source lines and the instruction addresses at a statement level (provided in the *.debug_line* section). Even though Dwarf does not represent all statements in the source code, the mapping is significantly finer granularity than that provided at only the function level by the symbol table (which identifies the functions by instruction address, but not the source file line numbers; the later requires source code analysis).

The example code in 45.3 shows the mapping between the source code lines and the instruction addresses.

XME: *This need to be o correctly return sets nd sets of addresses in be a second interface.*

Output from the compilation of this test code and running it with the example input results in the output shown in figure 45.4. This output shows the binary executables instruction address range (binary compiled on Linux x86 system), the range of lines of source code used by the binary executable, the mapping of a source code range of line numbers to the instruction addresses, and the mapping of a range of instruction addresses to the source code line numbers.

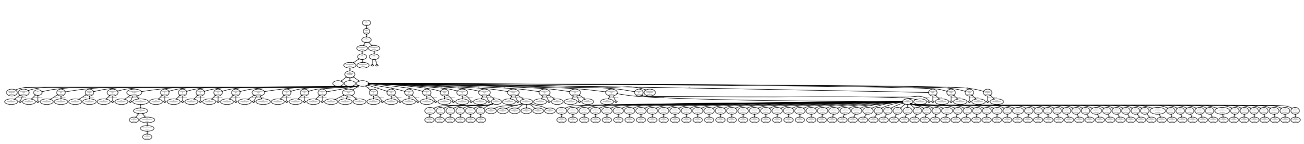


Figure 45.2: Dwarf AST (subset of ROSE binary AST).

```cpp
#include "rose.h"

int
main(int argc, char** argv)
   {
     SgProject* project = frontend(argc,argv);
     ROSE_ASSERT (project != NULL);

// This is controled by using the --with-dwarf configure command line option.
#if USE_ROSE_DWARF_SUPPORT

  // The input file is the binary file...
     int binary_file_id = project->get_fileList()[0]->get_file_info()->get_file_id();

  // Increment to get the next file id (for the source file, instead of the binary file)
     int source_file_id = binary_file_id + 1;

     std::string binaryFilename = Sg_File_Info::getFilenameFromID(binary_file_id);
     printf ("file_id = %d binaryFilename = %s \n",binary_file_id,binaryFilename.c_str());

     std::string sourceFilename = Sg_File_Info::getFilenameFromID(source_file_id);
     printf ("file_id = %d sourceFilename = %s \n",source_file_id,sourceFilename.c_str());

  // Compute the source line range from the instructions in the binary executable
     std::pair<LineColumnFilePosition,LineColumnFilePosition> sourceFileRange;
     sourceFileRange = SgAsmDwarfLineList::sourceCodeRange( source_file_id );

     printf ("\nSource file line number range for:\n -- file = %s (id = %d)\n -- [(line=%d, col=%d), (line=%d,
          sourceFilename.c_str(),source_file_id,
          sourceFileRange.first.first, sourceFileRange.first.second,
          sourceFileRange.second.first, sourceFileRange.second.second);

  // Compute the binary executable instruction address range
     std::pair<uint64_t,uint64_t> addressRange = SgAsmDwarfLineList::instructionRange();
     printf ("\nBinary instruction address range = (0x%lx, 0x%lx) \n",addressRange.first,addressRange.second);

     int minLine = sourceFileRange.first.first;
     int maxLine = sourceFileRange.second.first;
     int columnNumber = -1;

     printf ("\nInstruction addresses computed from source positions: \n");
  // Iterate over line numbers to map back to instruction addresses
     for (int lineNumber = minLine - 2; lineNumber <= maxLine + 2; lineNumber++)
        {
       // Out of range values generate the next address or NULL.
          FileIdLineColumnFilePosition s(source_file_id,std::pair<int,int>(lineNumber,columnNumber));
          uint64_t instructionAddress = SgAsmDwarfLineList::sourceCodeToAddress(s);
          printf ("   sourceCodeToAddress(%d,%d,%d) = 0x%lx \n",s.first,s.second.first,s.second.second,instruc
        }

     uint64_t minInstructionAddress = addressRange.first;

  // Iterate over the addresses of the binary and compute the source code line numbers (limit the range so the
     printf ("\nSource lines computed from address range (truncated to keep output short): \n");
     for (uint64_t address = minInstructionAddress - 1; address < minInstructionAddress + 25; address++)
        {
          FileIdLineColumnFilePosition s_map = SgAsmDwarfLineList::addressToSourceCode(address);
          printf ("   addressToSourceCode: address 0x%lx = (%d,%d,%d) \n",address,s_map.first,s_map.second.fir
        }
#else
     printf ("\n\nROSE must be configured with --with-dwarf=<path to libdwarf> to use Dwarf support. \n\n");
#endif

     printf ("\nProgram Terminated Normally! \n\n");

  // Skip call to backend since this is just an analysis.
     return backend(project);
   }
```

Figure 45.3: Example source code (typical for reading in a binary or source file).

```

ROSE must be configured with --with-dwarf=<path to libdwarf> to use Dwarf support.


Program Terminated Normally!
```

Figure 45.4: Example source code (typical for reading in a binary or source file).

Part VII

Interacting with Other Tools

How to build interoperable tools using ROSE.

Chapter 46

Abstract Handles to Language Constructs

This chapter describes a reference design and its corresponding implementation for supporting abstract handles to language constructs in source code and optimization phases. It can be used to facilitate the interoperability between compilers and tools. We define an abstract handle as a representation for a unique language construct in a specific program. Interoperability between tools is achieved by writing out the abstract handles as strings and reading them within other tools to build the equivalent abstract handle. [1]

The idea is to define identifiers for unique statements, loops, functions, and other language constructs in source code. Given the diverse user requirements, an ideal specification should include multiple forms to specify a language construct.

Currently, we are interested in the following forms for specifying language constructs:

- Source file position information including path, filename, line and column number etc. GNU standard source position from http://www.gnu.org/prep/standards/html_node/Errors.html presents some examples.
- Global or local numbering of specified language construct in source file (e.g. 2nd "do" loop in a global scope). The file is itself specified using an abstract handle (typically generated from the file name).
- Global or local names of constructs. Some language constructs, such as files, function definitions and namespace, have names which can be used as their handle within a context.
- Language-specific label mechanisms. These include named constructs in Fortran, numbered labels in Fortran, and statement labels in C and C++, etc.

[1] Abstract Handles are not appropriate for program analysis since they are not intended to be used to capture the full structure of a program. Instead, Abstract Handles represent references to language constructs in a program, capturing only a program's local structure; intended to support interoperability between source based tools (including compilers). We don't advise the use of abstract handles in an aggressive way to construct an alternative intermediate representation (IR) for a full program.

In addition to human-readable forms, compilers and tools can generate internal IDs for language constructs. It is up to compiler/tool developers to provide a way to convert their internal representations into human-readable formats.

Abstract Handles can have any of the human-readable or machine-generated forms. A handle can be used alone or combined with other handles to specify a language construct. A handle can also be converted from one form to another (e.g. from a compiler specific form to an human readable form relative to the source position; filename, line number, etc.). Abstract handles can have different lifetimes depending on their use and implementation. An abstract handle might be required to be persistent if it is used to reference a language construct that would be optimized over multiple executions of one or more different tools. Where as an abstract-handle might be internally generated only for purposes of optimizations used in a single execution (e.g. optimization within a compiler).

46.1 Use Case

A typical use can for Abstract Handles might be for a performance tool to identify a collection of loops in functions that are computationally intensive and to construct Abstract Handles that refer to this specific loops. Then pass the Abstract Handles to a second tool that might analyze the source code and/or the binary executable to evaluate if the computational costs are reasonable or if optimizations might be possible. The specific goal of the Abstract Handles is to support these sorts of uses within autotuning using diverse tools used and/or developed as part of autotuning research within the DOE SciDAC PERI project.

46.2 Syntax

A possible grammar for abstract handles could be:

```
/* a handle is a single handle item or a link of them separated by ::, or
other delimiters */
handle ::= handle_item | handle '::' handle_item

/* Each handle item consists of construct_type and a specifier.
Or it can be a compiler generated id of any forms. */

handle_item ::= construct_type specifier | compiler_generated_handle

/*
Construct types are implementation dependent.
An implementation can support a subset of legal constructs or all of them.
We define a minimum set of common construct type names here and
will grow this list as needed.
*/
construct_type ::= Project|SourceFile|FunctionDeclaration|ForStatement|...

/* A specifier is used to locate a particular construct
```

```
  e.g: <name, "foo">
*/

specifier::= '<' specifier_type ',' specifier_value '>'

/* tokens for specifier types could be name, position,numbering, label, etc.
specifier type is necessary to avoid ambiguity for specifier values,
because a same value could be interpreted in different specifier types otherwise
*/

specifier_type::= name | position | numbering | label

/* Possible values for a specifier */

specifier_value::= string_lit|int_lit|position_value| label_value

/*A label could be either integer or string */
label_value::= int_lit | string_lit

/* Start and end source line and column information
e.g.: 13.5-55.4,  13,  13.5 , 13.5-55 */
position_value:: = line_number[ '.' column_number][ '-' line_number[ '.' column_number]]

/* Integer value: one or more digits */
int_lit ::= [0-9]+

/* String value: start with a letter, followed by zero or more letters or digits */
string_lit ::= [a-z][a-z0-9]*
```

46.3 Examples

We give some examples of language handles using the syntax mentioned above. Canonical AST's node type names are used as the construct type names. Other implementations can use their own construct type names.

- A file handle consisting of only one handle item:

```
SourceFile<name,"/home/PERI/test111.f">
```

- A function handle using a named handle item, combined with a parent handle using a name also:

```
SourceFile<name,"/home/PERI/test111.f">::FunctionDeclaration<name,"foo">
```

- A function handle using source position(A function starting at line 12, column 1 till line 30, column 1 within a file):

```
SourceFile<name,"/home/PERI/test111.f">::FunctionDeclaration<position,"12.1-30.1">
```

- A function handle using numbering(The first function definition in a file):

```
SourceFile<name,/home/PERI/test111.f">::FunctionDeclaration<numbering,1>
```

- A return statement using source position (A return statement at line 100):

```
SourceFile<name,/home/PERI/test222.c>::ReturnStatement<position,"100">
```

- A loop using numbering information (The second loop in function main()):

```
SourceFile<name,"/home/PERI/test222.c">::FunctionDeclaration<name,"main">::
ForStatement<numbering,2>
```

- A nested loop using numbering information (The first loop inside the second loop in function main()):

```
SourceFile<name,"/home/PERI/test222.c">::FunctionDeclaration<name,"main">::
ForStatement<numbering,2>::ForStatement<numbering,1>
```

46.4 Reference Implementation

Abstract Handles are fundamentally compiler and tool independent, however to clarify the concepts, provide meaningful examples, a working reference implementation we have provided a reference implementation in ROSE. The source files are located in *src/midend/abstractHandle* in the ROSE distribution. A generic interface (abstract_handle.h and abstract_handle.cpp) provides data structures and operations for manipulating abstract handles using source file positions, numbering, or names. Any compilers and tools can have their own implementations using the same interface.

46.4.1 Connecting to ROSE

A ROSE adapter (roseAdapter.h and roseAdapter.cpp) using the interface is provided as a concrete implementation for the maximum capability of the implementation (within a source-to-source compiler). Figure 46.1 shows the code (using ROSE) to generate abstract handles for loops in an input source file (as in Figure 46.2). Abstract handle constructors generate handles from abstract nodes, which are implemented using ROSE AST nodes. Source position is used by default to generate a handle item. Names or numbering are used instead when source position information is not available. The Constructor can also be used to generate a handle item using a specified handle type (numbering handles in the example). Figure 46.3 is the output showing the generated handles for the loops.

```
/*
Example code to generate abstract handles for language constructs

by Liao, 10/6/2008
*/
#include "rose.h"
#include <iostream>
#include "abstract_handle.h"
#include "roseAdapter.h"
#include <string.h>

using namespace std;
using namespace AbstractHandle;

// a global handle for the current file
static abstract_handle* file_handle = NULL;

class visitorTraversal : public AstSimpleProcessing
{
  protected:
    virtual void visit(SgNode* n);
};

void visitorTraversal::visit(SgNode* n)
{
  SgForStatement* forloop = isSgForStatement(n);

  if (forloop)
  {
    cout<<"Creating handles for a loop construct..."<<endl;
    //Create an abstract node
    abstract_node* anode= buildroseNode(forloop);

    //Create an abstract handle from the abstract node
    //Using source position specifiers by default
    abstract_handle * ahandle = new abstract_handle(anode);
    cout<<ahandle->toString()<<endl;

    // Create handles based on numbering specifiers within the file
    abstract_handle * bhandle = new abstract_handle(anode,e_numbering,file_handle);
    cout<<bhandle->toString()<<endl<<endl;
  }
}

int main(int argc, char * argv[])
{
  SgProject *project = frontend (argc, argv);

  //Generate a file handle
  abstract_node * file_node = buildroseNode((project->get_fileList())[0]);
  file_handle = new abstract_handle(file_node);

  //Generate handles for language constructs
  visitorTraversal myvisitor;
  myvisitor.traverseInputFiles(project,preorder);

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

Figure 46.1: Example 1: Generated handles for loops: using constructors with or without a specified handle type.

```
/* test input for generated abstract handles */
int a[100][100][100];

void foo()
{
  int i,j,k;
  for (i=0;i++;i<100)
    for (j=0;j++;j<100)
      for (k=0;k++;k<100)
        a[i][j][k]=i+j+k;

  for (i=0;i++;i<100)
    for (j=0;j++;j<100)
      for (k=0;k++;k<100)
        a[i][j][k]+=5;
}
```

Figure 46.2: Example 1: Example source code with some loops, used as input.

```
Creating handles for a loop construct...
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<position,7.3-10.25>
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<numbering,1>

Creating handles for a loop construct...
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<position,8.5-10.25>
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<numbering,2>

Creating handles for a loop construct...
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<position,9.7-10.25>
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<numbering,3>

Creating handles for a loop construct...
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<position,12.3-15.22>
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<numbering,4>

Creating handles for a loop construct...
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<position,13.5-15.22>
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<numbering,5>

Creating handles for a loop construct...
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<position,14.7-15.22>
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle1.cpp>::ForStatement<numbering,6>
```

Figure 46.3: Example 1: Abstract handles generated for loops.

A second example (shown in Figure 46.4) demonstrates how to create handles using user-specified strings representing handle items for language constructs within a source file (shown in Figure 46.5). This is particularly useful to grab internal language constructs from handles provided by external software tools. The output of the example is given in Figure 46.6.

```
/*
Example code to generate language handles from input strings about
* source position information
* numbering information

by Liao, 10/9/2008
*/
#include "rose.h"
#include <iostream>
#include <string.h>
#include "abstract_handle.h"
#include "roseAdapter.h"

using namespace std;
using namespace AbstractHandle;

int main(int argc, char * argv[])
{
  SgProject *project = frontend (argc, argv);

  // Generate a file handle from the first file of the project
  abstract_node* file_node= buildroseNode((project->get_fileList())[0]);
  abstract_handle* handle0 = new abstract_handle(file_node);
  cout<<"Created a file handle:\n"<<handle0->toString()<<endl<<endl;;

  //Create a handle to a namespace given its name and parent handle
  string input1="NamespaceDeclarationStatement<name,space1>";
  abstract_handle* handle1 = new abstract_handle(handle0,input1);
  cout<<"Created a handle:\n"<<handle1->toString()<<endl<<endl;
  cout<<"It points to:\n"<<handle1->getNode()->toString()<<endl<<endl;

  // Create a handle within the file, given a string specifying
  // its construct type (class declaration) and source position
  string input="ClassDeclaration<position,4.3-9.2>";
  abstract_handle* handle2 = new abstract_handle(handle0,input);

  cout<<"Created a handle:\n"<<handle2->toString()<<endl<<endl;
  cout<<"It points to:\n"<<handle2->getNode()->toString()<<endl<<endl;;

  // find the second function declaration within handle2
  abstract_handle handle3(handle2,"FunctionDeclaration<numbering,2>");

  cout<<"Created a handle:\n"<<handle3.toString()<<endl<<endl;
  cout<<"It points to:\n"<<handle3.getNode()->toString()<<endl;

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

Figure 46.4: Example 2: Generated handles from strings representing handle items.

```
void bar(int x);
namespace space1
{
  class A
  {
  public:
    void bar(int x);
    void foo();
  };
}
```

Figure 46.5: Example 2: Source code with some language constructs.

```
Created a file handle:
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle2.cpp>

Created a handle:
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle2.cpp>::NamespaceDeclarationStatement<name,space1>

It points to:
namespace space1{class A {public: void bar(int x);void foo();};}

Created a handle:
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle2.cpp>::ClassDeclaration<position,4.3-9.2>

It points to:
class A {public: void bar(int x);void foo();};

Created a handle:
Project<numbering,1>::FileList<numbering,1>::SourceFile<name,/export/tmp.hudson-
rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCod
e_AbstractHandle2.cpp>::ClassDeclaration<position,4.3-9.2>::MemberFunctionDeclar
ation<numbering,2>

It points to:
public: void foo();
```

Figure 46.6: Example 2: Handles generated from string and their language constructs.

46.4.2 Connecting to External Tools

A third example is provided to demonstrate how to use the abstract interface with any other tools, which may have less features in terms of supported language constructs and their correlations compared to a compiler. Assume a tool operating on some simple for-loops within an arbitrary source file (the input file is not shown in this example). Such a tool might have an internal data structure representing loops; such as that in given in Figure 46.7. We will show how the tool specific data structure for loops can be used to generate abstract handles and output as strings that can be used by other tools which use abstract handles (which would generate the abstract handles by reading the strings).

```
/*
 * A toy loop data structure demonstrating a thin client of abstract handles:
 * A simplest loop tool which keeps a tree of loops in a file
 */
#ifndef my_loop_INCLUDED
#define my_loop_INCLUDED

#include <string>
#include <vector>
class MyLoop
{
public:
  std::string sourceFileName;
  size_t line_number;
  std::vector<MyLoop*> children;
  MyLoop* parent;
};

#endif
```

Figure 46.7: Example 3: A simple data structure used to represent a loop in an arbitrary tool.

An adapter (loopAdapter.h and loopAdapter.cpp) using the proposed abstract handle interface is given in *src/midend/abstractHandle*. It provides a concrete implementation for the interface for the simple loops and adds a node to support file nodes (Compared to a full-featured IR for a compiler, the file node is an additional detail for tools without data structures to support files). The test program is given in Figure 46.8. Again, it creates a top level file handle first. Then a loop handle (*loop_handle1*) is created within the file handle using its relative numbering information. The *loop_handle2* is created from from its string format using file position information (using GNU standard file position syntax). The *loop_handle3* uses its relative numbering information within *loop_handle1*.

The output of the program is shown in Figure 46.9. It demonstrates the generated strings to represent the abstract handles in the arbitrary code operated upon by the tool. Interoperability is achieved by another tool reading in the generated string representation to generate an abstract handle to the same source code language construct.

```cpp
#include <iostream>
#include <string>
#include <vector>
#include "abstract_handle.h"
#include "myloop.h"
#include "loopAdapter.h"

using namespace std;
using namespace AbstractHandle;

int main()
{
  //-------------Preparing the internal loop representation---------
  // declare and initialize a list of loops using MyLoop
  // The loop tool should be able to generate its representation from
  // source code somehow. We fill it up manually here.
  vector <MyLoop* > loops;

  MyLoop loop1, loop2, loop3;
  loop1.sourceFileName="file1.c";
  loop1.line_number = 7;
  loop1.parent = NULL;
  loop2.sourceFileName="file1.c";
  loop2.line_number = 8;
  loop2.parent=&loop1;
  loop1.children.push_back(&loop2);
  loop3.sourceFileName="file1.c";
  loop3.line_number = 12;
  loop3.parent=NULL;
  loops.push_back(&loop1);
  loops.push_back(&loop3);

  //------------------ using abstract handles-------------
  //Generate the abstract handle  for the source file
  fileNode* filenode = new fileNode("file1.c");
  filenode->setMLoops(loops);
  abstract_handle* file_handle = new abstract_handle(filenode);
  cout<<"Created a file handle:"<<endl<<file_handle->toString()<<endl;

  //Create a loop handle within the file using numbering info.
  abstract_node* loop_node1= new loopNode(&loop1);
  abstract_handle* loop_handle1= new abstract_handle(loop_node1,e_numbering,file_handle);
  cout<<"Created a loop handle:"<<endl<<loop_handle1->toString()<<endl;

  //Create another loop handle within a file using its source position information
  string input1("ForStatement<position,12>");
  abstract_handle* loop_handle2= new abstract_handle(file_handle,input1);
  cout<<"Created a loop handle:"<<endl<<loop_handle2->toString()<<endl;

  //Create yet another loop handle within a loop using its relative numbering information
  string input2("ForStatement<numbering,1>");
  abstract_handle* loop_handle3= new abstract_handle(loop_handle1,input2);
  cout<<"Created a loop handle:"<<endl<<loop_handle3->toString()<<endl;

  return 0;
}
```

Figure 46.8: Example 3: A test program for simple loops' abstract handles.

```
bash-3.00: ./testMyLoop
Created a file handle:
SourceFile<name,file1.c>
Created a loop handle:
SourceFile<name,file1.c>::ForStatement<numbering,1>
Created a loop handle:
SourceFile<name,file1.c>::ForStatement<position,12>
Created a loop handle:
SourceFile<name,file1.c>::ForStatement<numbering,1>::ForStatement<numbering,1>
```

Figure 46.9: Example 3: Output of the test program for simple loops' abstract handles (as strings).

46.5 Summary

Abstract handles are low level mechanisms to support multiple tools to exchange references to source code. Several examples are used to present the different features of abstract handles. Importantly, the specification of abstract handles is tool independent. A reference implementation is provided and is publically available within the ROSE compiler framework. We encourage debate on the pros and cons of this concept to support interoperability of tools which must pass references to source code between them. This work is expected to a small piece of the infrastructure to suport autotuning research.

Chapter 47

ROSE-HPCToolKit Interface

ROSE-HPCToolKit is designed to read in performance data generated by HPCToolkit (and recently GNU gprof) and annotate ROSE AST with performance metrics. It is included in the ROSE distribution and enabled by default if an existing installation of the Gnome XML library, libxml2 (http://xmlsoft.org) can be detected by ROSE's configure script. Or it can be enabled explicitly by specifying the `--enable-rosehpct` option when running configure.

The HPCToolkit (http://www.hipersoft.rice.edu/hpctoolkit) is a set of tools for analyzing the dynamic performance behavior of applications. It includes a tool that instruments a program's binary, in order to observe CPU hardware counters during execution; additional post-processing tools attribute the observed data to statements in the original source code. HPCToolkit stores this data and the source attributions in XML files. In this chapter, we give an overview of simple interfaces in ROSE that can read this data and attach it to the AST.

GNU gprof is a basic but easy to use profiling tool. It produces an execution profile of applications. gprof's output is a flat profile for each function by default, which is not very interesting to us. We use a line-by-line output with full file path information generated by using option `-l -L` with gprof.

47.1 An HPCToolkit Example Run

Consider the sample source program shown in Figures 47.1–47.2. This program takes an integer n on the command line, and has a number of loops whose flop and memory-operation complexity are either $\Theta(n)$ or $\Theta(n^2)$. For this example, we would expect the loop nest at line 56, which has $O(n^2)$ cost, to be the most expensive loop in the program for large n.

Suppose we use the HPCToolkit to profile this program, collecting cycle counts and floating-point instructions.[1] HPCToolkit will generate one XML file for each metric.

A schema specifying the format of these XML files appears in Figure 47.3. In essence, this schema specifies that the XML file will contain a structured, abstract representation of the program in terms of abstract program entities such as "modules," "procedures," "loops," and "statements." Each of these entities may have line number information and a "metric" value.

[1] In this example, HPCToolkit uses the PAPI to read CPU counters (http://icl.cs.utk.edu/papi).

(Refer to the HPCToolkit documentation for more information.) This schema is always the first part of an HPCToolkit-generated XML profile data file.

We ran HPCToolkit on the program in Figures 47.1–47.2, and collected cycle and flop counter data. The actual XML files storing this data appear in Figures 47.4 and 47.5. By convention, these metrics are named according to their PAPI symbolic name, as shown on line 67 in both Figures 47.4 and 47.5. According to the cycle data on line 90 of Figure 47.4, the most expensive statement in profiled.c is line 62 of Figure 47.1, as expected.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef double val_t;

static val_t *
gen (size_t n)
{
  val_t* x = (val_t *) malloc (sizeof(val_t) * n);
  if (x != NULL)
    {
      size_t i;
      for (i = 0; i < n; i++)
          x[i] = (val_t)1.0 / n;
    }
  return x;
}

static val_t
dot (size_t n, const val_t* x, const val_t* y)
{
  size_t i;
  val_t sum;
  for (i = 0, sum = 0.0; i < n; i++)
    sum += x[i] * y[i];
  return sum;
}

static size_t
max (size_t n, const val_t* x)
{
  size_t i;
  size_t i_max;
  val_t v_max;

  if (n <= 0)
    return 0;

  v_max = x[0];
  i_max = 0;
  for (i = 1; i < n; i++)
    if (x[i] > v_max)
      {
        v_max = x[i];
        i_max = i;
      }
  return i_max;
}

static void
mv (size_t n, const val_t* A, const val_t* x, val_t* y)
{
  size_t j;
  memset (y, 0, sizeof(val_t) * n);
  for (j = 0; j < n; j++)
    {
      const val_t* Ap;
      register val_t xj = x[j];
      size_t i;
      for (i = 0, Ap = A + j*n; i < n; i++, Ap++)
        y[i] += Ap[0] * xj;
    }
}
```

Figure 47.1: profiled.c (part 1 of 2): Sample input program, profiled using the HPCToolkit.

```
int
main (int argc, char* argv[])
{
  size_t n;
  val_t* x;
  val_t* y;
  val_t* A;
  size_t i;

  /* outputs */
  val_t sum;
  size_t i_max;
  val_t y_max;

  if (argc != 2)
    {
      fprintf (stderr, "usage: %s <n>\n", argv[0]);
      return 1;
    }
  n = atoi (argv[1]);
  if (n <= 0)
    return 1;

  A = gen (n * n);
  x = gen (n);
  y = gen (n);

  if (A == NULL || x == NULL || y == NULL)
    {
      fprintf (stderr, "*** Out of memory ***\n");
      return 1;
    }

  sum = 0;
  for (i = 0; i < n; i++)
    sum += dot (n, x, y);
  mv (n, A, x, y);
  i_max = max (n, y);
  y_max = y[i_max];

  printf ("%g %lu %g\n", sum, (unsigned long)i_max, y_max);

  return 0;
}

/* eof */
```

Figure 47.2: profiled.c (part 2 of 2): Sample input program, profiled using the HPCToolkit.

```
<?xml version="1.0"?>
<!DOCTYPE PROFILE [
<!-- Profile: correlates profiling info with program structure. -->
<!ELEMENT PROFILE (PROFILEHDR, PROFILEPARAMS, PROFILESCOPETREE)>
<!ATTLIST PROFILE
        version CDATA #REQUIRED>
  <!ELEMENT PROFILEHDR (#PCDATA)>
  <!ELEMENT PROFILEPARAMS (TARGET, METRICS)>
    <!ELEMENT TARGET EMPTY>
    <!ATTLIST TARGET
        name CDATA #REQUIRED>
    <!ELEMENT METRICS (METRIC)+>
    <!ELEMENT METRIC EMPTY>
    <!ATTLIST METRIC
        shortName   CDATA #REQUIRED
        nativeName  CDATA #REQUIRED
        period      CDATA #REQUIRED
        units       CDATA #IMPLIED
        displayName CDATA #IMPLIED
        display     (true|false) #IMPLIED>
  <!ELEMENT PROFILESCOPETREE (PGM)*>
    <!-- This is essentially the PGM dtd with M element added. -->
    <!ELEMENT PGM (G|LM|F|M)+>
    <!ATTLIST PGM
        n CDATA #REQUIRED>
    <!-- Groups create arbitrary sets of other elements except PGM. -->
    <!ELEMENT G (G|LM|F|P|L|S|M)*>
    <!ATTLIST G
        n CDATA #IMPLIED>
    <!-- Runtime load modules for PGM (e.g., DSOs, exe) -->
    <!ELEMENT LM (G|F|M)*>
    <!ATTLIST LM
        n CDATA #REQUIRED>
    <!-- Files contain procedures and source line info -->
    <!ELEMENT F (G|P|L|S|M)*>
    <!ATTLIST F
        n CDATA #REQUIRED>
    <!-- Procedures contain source line info
         n: processed name; ln: link name -->
    <!ELEMENT P (G|L|S|M)*>
    <!ATTLIST P
        n  CDATA #REQUIRED
        ln CDATA #IMPLIED
        b CDATA #IMPLIED
        e CDATA #IMPLIED>
    <!-- Loops -->
    <!ELEMENT L (G|L|S|M)*>
    <!ATTLIST L
        b CDATA #IMPLIED
        e CDATA #IMPLIED>
    <!-- Statement/Statement range -->
    <!ELEMENT S (M)*>
    <!ATTLIST S
        b CDATA #REQUIRED
        e CDATA #IMPLIED
        id CDATA #IMPLIED>
    <!ELEMENT M EMPTY>
    <!ATTLIST M
        n CDATA #REQUIRED
        v CDATA #REQUIRED>
]>
```

Figure 47.3: XML schema for HPCToolkit data files: This schema, prepended to each of the HPCToolkit-generated XML files, describes the format of the profiling data. This particular schema was generated by HPCToolkit 1.0.4.

```
<PROFILE version="3.0">
<PROFILEHDR></PROFILEHDR>
<PROFILEPARAMS>
  <TARGET name="example"/>
  <METRICS>
    <METRIC shortName="0" nativeName="PAPI_TOT_CYC" displayName="PAPI_TOT_CYC" period="32767" units="PAPI_even
  </METRICS>
</PROFILEPARAMS>
<PROFILESCOPETREE>
<PGM n="example">
  <LM n="/home/vuduc2/projects/ROSE/tmp/xml/xerces-c/hpctif/examples/data/01/example">
    <F n="./profiled.c">
      <P n="main">
        <S b="15">
          <M n="0" v="163835"/>
        </S>
        <S b="25">
          <M n="0" v="65534"/>
        </S>
        <S b="26">
          <M n="0" v="65534"/>
        </S>
      </P>
      <P n="mv">
        <S b="61">
          <M n="0" v="65534"/>
        </S>
        <S b="62">
          <M n="0" v="327670"/>
        </S>
      </P>
    </F>
  </LM>
</PGM>
</PROFILESCOPETREE>
</PROFILE>
```

Figure 47.4: PAPI_TOT_CYC.xml: Sample cycle counts observed during profiling, generated from running the HPCToolkit on profiled.c (Figures 47.1–47.2.) These lines would appear after the schema shown in Figure 47.3.

```
<PROFILE version="3.0">
<PROFILEHDR></PROFILEHDR>
<PROFILEPARAMS>
  <TARGET name="example"/>
  <METRICS>
    <METRIC shortName="0" nativeName="PAPI_FP_OPS" displayName="PAPI_FP_OPS" period="32767" units="PAPI_events"/>
  </METRICS>
</PROFILEPARAMS>
<PROFILESCOPETREE>
<PGM n="example">
  <LM n="/home/vuduc2/projects/ROSE/tmp/xml/xerces-c/hpctif/examples/data/01/example">
    <F n="./profiled.c">
      <P n="main">
        <S b="25">
          <M n="0" v="32767"/>
        </S>
        <S b="26">
          <M n="0" v="98301"/>
        </S>
      </P>
      <P n="mv">
        <S b="61">
          <M n="0" v="131068"/>
        </S>
        <S b="62">
          <M n="0" v="262136"/>
        </S>
      </P>
    </F>
  </LM>
</PGM>
</PROFILESCOPETREE>
</PROFILE>
```

Figure 47.5: PAPI_FP_OPS.xml: Sample flop counts observed during profiling, generated from running the HPCToolkit on profiled.c (Figures 47.1–47.2.) These lines would appear after the schema shown in Figure 47.3.

47.2 Attaching HPCToolkit Data to the ROSE AST

To attach the data of Figures 47.4 and 47.5 to the AST, we augment a basic ROSE translator with two additional calls, as shown in Figure 47.6, lines 47–48 and 54. We describe these calls below.

47.2.1 Calling ROSE-HPCT

We must first include rosehpct/rosehpct.hh, as shown on line 6 of Figure 47.6. All ROSE-HPCT routines and intermediate data structures reside in the RoseHPCT namespace.

Next, lines 47–48 of Figure 47.6 store the contents of the raw XML file into an intermediate data structure of type RoseHPCT::ProgramTreeList_t. The RoseHPCT::loadProfilingFiles() routine processes command-line arguments, extracting ROSE-HPCT-specific options that specify the files. We discuss these options in Section 47.4.

Line 54 of Figure 47.6, attaches the intermediate profile data structure to the ROSE AST. The RoseHPCT::attachMetrics() routine creates new persistent attributes that store the counter data.[2] The attributes are named using the metric name taken from the XML file (see lines 67 of Figures 47.4–47.5); in this example, the attributes are named PAPI_TOT_CYC and PAPI_FP_OPS. Following the conventions of persistent attribute mechanism as described in Chapter 7, the attributes themselves are of type RoseHPCT::MetricAttr, which derives from the AstAttribute type.

47.2.2 Retrieving the attribute values

We retrieve the attribute values as described in Chapter 7. In particular, given a located node with cycle and flop attribute values, the printFlopRate() routine defined in lines 11–42 of Figure 47.6 prints the source position, AST node type, and estimated flops per cycle. We call printFlopRate() for each expression statement (SgExpressionStmt), for-initializer (SgForInitStatement), and for-statement (SgForStatement) in lines 59–66 of Figure 47.6. The output is shown in Figure 47.7.

Inspecting the output carefully, you may notice seeming discrepancies between the values shown and the values that appear in the XML files, or other values which seem unintuitive. We explain how these values are derived in Section 47.2.3.

This example dumps the AST as a PDF file, as shown on line 68 of Figure 47.6. You can inspect this file to confirm where attributes have been attached. We show an example of such a page in Figure 47.8. This page is the SgExprStatement node representing the sum-accumulate on line 26 of Figure 47.1.

47.2.3 Metric propagation

The example program in Figure 47.6 dumps metric values at each expression statement, for-initializer, and for-statement, but the input XML files in Figure 47.4–47.5 only attribute the profile data to "statements" that are not loop constructs. (The `<S ...>` XML tags refer to statements, intended to be "simple" non-scoping executable statements; a separate `<L ...>` tag

[2]The last parameter to RoseHPCT::attachMetrics() is a boolean that, when true, enables verbose (debugging) messages to standard error.

would refer to a loop.) Since the XML file specifies statements only by source line number, RoseHPCT::attachMetrics() attributes measurements to AST nodes in a heuristic way.

For example, lines 78–80 of Figure 47.4 indicate that all executions of the "simple statements" of line 25 of the original source (Figure 47.1) accounted for 65534 observed cycles, and that line 26 accounted for an additional 65534 cycles. In the AST, there are multiple "statement" and expression nodes that occur on line 25; indeed, Figure 47.7 lists 4 such statements.

The ROSE-HPCT modules uses a heuristic which only assigns `<S ...>` metric values to non-scoping nodes derived from SgStatement. When multiple SgStatement nodes occur at a particular source line, ROSE-HPCT simply attaches the metric to each of them. But only one of them will be used for propagating metrics to parent scopes.

How is the measurement of 65534 cycles attributed to all of the AST nodes corresponding to line 25 of Figure 47.1? Indeed, line 25 actually "contains" four different SgStatement nodes: an SgForStatement representing the whole loop on lines 25–26, an SgForInitStatement (initializer), and two SgExprStatements (one which is a child of the SgForInitStatement, and another for the for-loop's test expression). The loop's increment is stored in the SgForStatement node as an SgExpression, not an SgStatement. The SgForStatement node is a scoping statement, and so it "receives" none of the 65534 cycles. Since the increment is not a statement and one of the SgExprStatements is a child of the initializer, this leaves only two direct descendants of the SgForStatement—the initializer and the test expression statement—among which to divide the 65534 cycles. Thus, each receives 32767 cycles. The initializer's SgExprStatement child gets the same 32767 as its parent, since the two nodes are equivalent (see first two cases of Figure 47.7).

For the entire loop on lines 25–26 of Figure 47.1, the original XML files attribute 65534 cycles to line 25, and another 65534 cycles to line 26 (see Figure 47.4). Moreover, the XML files do not attribute any costs to this loop *via* an explicit `<L ...>` tag. Thus, the best we can infer is that the entire for-statement's costs is the sum of its immediate child costs; in this case, 131068 cycles. The RoseHPCT::attachMetrics() routine will heuristically accumulate and propagate metrics in this way to assign higher-level scopes approximate costs.

The RoseHPCT::attachMetrics() routine automatically propagates metric values through parent scopes. A given metric attribute, RoseHPCT::MetricAttr* x, is "derived" through propagation if x->isDerived() returns true. In fact, if you call x->toString() to obtain a string representation of the metric's value, two asterisks will be appended to the string as a visual indicator that the metric is derived. We called RoseHPCT::MetricAttr::toString() on lines 27 and 29 of Figure 47.6, and all of the SgForStatement nodes appearing in the output in Figure 47.7 are marked as derived.

Alternatively, you cann call RoseHPCT::attachMetricsRaw(), rather than calling RoseHPCT::attachMetrics(). The "raw" routine takes the same arguments but only attaches the raw data, *i.e.*, without attempting to propagate metric values through parent scopes.

47.3 Working with GNU gprof

ROSE-HPCT can also accept the line-by-line profiling output generated by GNU gprof. Currently, we only use the self seconds associated with each line and attach them to ROSE AST as AST attributes named `WALLCLK`.

A typical session to generate compatible gprof profiling file for ROSE-HPCT is given below:

```
[liao@codes]$ gcc -g seq-pi.c -pg
```

```
[liao@codes]$ ./a.out
[liao@codes]$ gprof -l -L a.out gmon.out &>profile.result
```

`-l` tells gprof to output line-by-line profiling information and `-L` causes gprof to output full file path information.

An excerpt of an output file looks like the following:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 38.20      8.84     8.84                             jacobi (/home/liao6/temp/jacobi.c:193 @ 804899c)
 36.43     17.27     8.43                             jacobi (/home/liao6/temp/jacobi.c:196 @ 8048a3f)
 11.00     19.82     2.54                             jacobi (/home/liao6/temp/jacobi.c:188 @ 804893e)
  5.66     21.12     1.31                             jacobi (/home/liao6/temp/jacobi.c:187 @ 8048968)
  3.93     22.04     0.91                             jacobi (/home/liao6/temp/jacobi.c:197 @ 8048a71)
  3.24     22.79     0.75                             jacobi (/home/liao6/temp/jacobi.c:191 @ 8048a7f)
  0.95     23.00     0.22                             jacobi (/home/liao6/temp/jacobi.c:186 @ 8048976)
  0.50     23.12     0.12                             jacobi (/home/liao6/temp/jacobi.c:187 @ 8048935)
  0.09     23.14     0.02                             jacobi (/home/liao6/temp/jacobi.c:190 @ 8048a94)
  0.00     23.14     0.00        1     0.00     0.00  driver (/home/liao6/temp/jacobi.c:91 @ 8048660)
  0.00     23.14     0.00        1     0.00     0.00  error_check (/home/liao6/temp/jacobi.c:220 @ 8048b7c)
  0.00     23.14     0.00        1     0.00     0.00  initialize (/home/liao6/temp/jacobi.c:116 @ 8048722)
  0.00     23.14     0.00        1     0.00     0.00  jacobi (/home/liao6/temp/jacobi.c:160 @ 8048892)
```

47.4 Command-line options

The call to RoseHPCT::loadProfilingFiles() on line 49 of Figure 47.6 processes and extracts ROSE-HPCT-specific command-line options. To generate the output in this chapter, we invoked Figure 47.6 with the following command-line:

```
./attachMetrics \
  -rose:hpctprof ../../../tutorial/roseHPCT/PAPI_TOT_CYC.xml \
  -rose:hpctprof ../../../tutorial/roseHPCT/PAPI_FP_OPS.xml \
  -rose:hpcteqpath .=/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-lin
  -c ../../../tutorial/roseHPCT/profiled.c
```

The main option is `-rose:hpct:prof <file>`, which specifies the HPCToolkit-generated XML file containing metric data. Here, we use this option twice to specify the names of the cycle and flop data files (Figures 47.4–47.5). To accept gprof output file, please use another option `-rose:gprof:linebyline <file>`. This option cannot be used with `-rose:hpct:prof <file>` currently.

We need the other option, `-rose:hpct:eqpath <A>=<B>`, to specify how paths in the HPCToolkit XML files can be mapped to file paths in the ROSE AST. This option allows users to generate performance files on one machine and analyze the results on another machine. In this example, the XML files specify the source file as, "./profiled.c" (line 73 of Figures 47.4 and 47.5); the "eqpath" command-line option above remaps the relative path "." to an absolute path as it would appear in the ROSE AST. Another example is to use the same performance file even after the original source tree is moved to another location. ROSE-HPCT can still correctly match performance data if the root source paths are given as `-rose:hpct:eqpath <oldRootPath>=<newRootPath>`.

Yet another option `-rose:hpct:enable_debug` is provided to display runtime debugging information such as metrics reading, attaching, and propagating. It also adds performance metrics into the ROSE output source file as source comments as shown below. Users can examine the source comments to make sure performance metrics are attached and propagated properly. As we can see, ROSE-HPCT attaches each performance metric to each matching statement. If there are multiple statements showing in the same line, the same metric will be attached to each of them. The metric propagation step will only propagate one of them to upper-level language constructs to ensure the correctness.

```
/* ROSE-HPCT propagated metrics WALLCLK:18.95[SgForStatement at 0xb7beb218] */
  for (
/* ROSE-HPCT raw data: Statement  WALLCLK:0.02@File jacobi.c  190-0 -> SgForInitStatement 0x94e8d08 at 190 */
i = 1;
/* ROSE-HPCT raw data: Statement  WALLCLK:0.02@File jacobi.c  190-0 -> SgExprStatement 0x94516d8 at 190 */
i < (n - 1); i++)
/* ROSE-HPCT propagated metrics WALLCLK:18.93[SgForStatement at 0xb7beb29c] */
    for (
/* ROSE-HPCT raw data: Statement  WALLCLK:0.75@File jacobi.c  191-0 -> SgForInitStatement 0x94e8d38 at 191 */
j = 1;
/* ROSE-HPCT raw data: Statement  WALLCLK:0.75@File jacobi.c  191-0 -> SgExprStatement 0x9451728 at 191 */
j < (m - 1); j++)
/* ROSE-HPCT propagated metrics WALLCLK:18.18[SgBasicBlock at 0x93f60b4] */
{
/* ROSE-HPCT raw data: Statement  WALLCLK:8.84@File jacobi.c  193-0 -> SgExprStatement 0x9451750 at 193 */
      resid = (((((ax * (((( *uold)[i - 1])[j]) + ((( *uold)[i + 1])[j]))) + (ay * (((( *uold)[i])[j - 1])
               + ((( *uold)[i])[j + 1])))) + (b * ((( *uold)[i])[j]))) - ((( *f)[i])[j])) / b);
/* ROSE-HPCT raw data: Statement  WALLCLK:8.43@File jacobi.c  196-0 -> SgExprStatement 0x9451778 at 196 */
      (( *u)[i])[j] = (((( *uold)[i])[j]) - (omega * resid));
/* ROSE-HPCT raw data: Statement  WALLCLK:0.91@File jacobi.c  197-0 -> SgExprStatement 0x94517a0 at 197 */
      error = (error + (resid * resid));
    }
```

```
// attachMetrics.cc -- Sample translator showing how to attach
// HPCToolkit data to the ROSE AST.

#include "rose.h"
#include <iostream>
#include <list>
#include <rosehpct/rosehpct.hh>

using namespace std;

// Prints the estimated flops/cycle at a located node.
static void printFlopRate (const SgNode* n)
{
  const SgLocatedNode* n_loc = isSgLocatedNode (n);
  if (n_loc
      && n_loc->attributeExists ("PAPI_TOT_CYC")
      && n_loc->attributeExists ("PAPI_FP_OPS"))
    {
      // Extract attributes.
      const RoseHPCT::MetricAttr* cycles_attr =
        dynamic_cast<const RoseHPCT::MetricAttr *> (n->getAttribute ("PAPI_TOT_CYC"));
      const RoseHPCT::MetricAttr* flops_attr =
        dynamic_cast<const RoseHPCT::MetricAttr *> (n->getAttribute ("PAPI_FP_OPS"));
      ROSE_ASSERT (cycles_attr && flops_attr);

      // Get the metric values, as doubles and strings.
      double cycles = cycles_attr->getValue ();
      string cycles_s = const_cast<RoseHPCT::MetricAttr *> (cycles_attr)->toString ();
      double flops = flops_attr->getValue ();
      string flops_s = const_cast<RoseHPCT::MetricAttr *> (flops_attr)->toString ();

      // Print node pointer/type, parent, estimated flop rate, and source position.
      const SgNode* n_par = n_loc->get_parent ();
      cout << (const void *)n_loc << ":<" << n_loc->class_name () << ">"
           << " (Par=" << (const void *)n_par << ":<" << n_par->class_name () << ">)"
           << " = (" << flops_s << " flops)"
           << " / (" << cycles_s << " cy)"
           << " ~= " << flops / cycles << " flops/cy" << endl
           << "  [" << n_loc->get_startOfConstruct ()->get_raw_filename ()
           << ':' << n_loc->get_startOfConstruct ()->get_raw_line () << "]"
           << endl;
    }
}

int main (int argc, char* argv[])
{
  vector<string> argvList(argv, argv+argc);
  cerr << "[Loading HPCToolkit or Gprof profiling data...]" << endl;
  RoseHPCT::ProgramTreeList_t profiles
    = RoseHPCT::loadProfilingFiles(argvList);

  cerr << "[Building the ROSE AST...]" << endl;
  SgProject* proj = frontend (argvList);

  cerr << "[Attaching metrics to the AST...]" << endl;
  RoseHPCT::attachMetrics (profiles, proj, proj->get_verbose () > 0);

  cerr << "[Estimating flop execution rates...]" << endl;
  typedef Rose_STL_Container<SgNode *> NodeList_t;

  NodeList_t estmts = NodeQuery::querySubTree (proj, V_SgExprStatement);
  for_each (estmts.begin (), estmts.end (), printFlopRate);

  NodeList_t for_inits = NodeQuery::querySubTree (proj, V_SgForInitStatement);
  for_each (for_inits.begin (), for_inits.end (), printFlopRate);

  NodeList_t fors = NodeQuery::querySubTree (proj, V_SgForStatement);
  for_each (fors.begin (), fors.end (), printFlopRate);

  cerr << "[Dumping a PDF...]" << endl;
  generatePDF (*proj);

// DQ (1/2/2008): This output appears to have provided enough synchronization in
// the output to allow -j32 to work. Since I can't debug the problem further for
// now I will leave it.
  cerr << "[Calling backend...]" << endl;
  return backend (proj);
}
```

Figure 47.6: attachMetrics.cc: Sample translator to attach HPCToolkit metrics to the AST.

Figure 47.7: Sample output, when running attachMetrics.cc (Figure 47.6) with the XML inputs in Figures 47.4–47.5. Here, we only show the output sent to standard output (*i.e.*, cout and not cerr).

pointer:0x2b24264ca010
SgVariableDeclaration
/export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/roseHPCT
IsTransformation:0
IsOutputInCodeGeneration:0
Declaration mangled name: _variable_declaration__variable_type_val_td__typedef_declaration_variable_

Click here to go to the parent node

SgNode* p_parent : 0x2b2426706560
bool p_isModified : 1
$CLASSNAME* p_freepointer : 0xffffffffffffffff
Sg_File_Info* p_startOfConstruct : 0x5d66fb8
Sg_File_Info* p_endOfConstruct : 0x5d67000
AttachedPreprocessingInfoType* p_attachedPreprocessingInfoPtr : 0
AstAttributeMechanism* p_attributeMechanism : 0x5fee380
SgLabelRefExp* p_numeric_label : 0
int p_source_sequence_value : 1737
unsigned int p_decl_attributes : 0

Figure 47.8: Sample PDF showing attributes.

Chapter 48

TAU Instrumentation

Tau is a performance analysis tool from University of Oregon. They have mechanisms for automating instrumentation of the source code's text file directly, but these can be problematic in the present of macros. We present an example of instrumentation combined with code generation to provide a more robust means of instrumentation for source code. This work is preliminary and depends upon two separate mechanisms for the rewrite of the AST (one high level using strings and one low level representing a more direct handling of the AST at the IR level).

48.1 Input For Examples Showing Information using Tau

Figure 48.1 shows the example input used for demonstration of Tau instrumentation.

48.2 Generating the code representing any IR node

Figure 48.2 shows a code that traverses each IR node and for a SgInitializedName of SgStatement output the scope information. The input code is shown in figure 48.2, the output of this code is shown in figure 48.3.

```
// #include <math.h>
// #include <stdlib.h>

double foo(double x)
   {
     double theValue = x;
     theValue*= x;
     return theValue;
   }

int main(int argc, char* argv[])
   {
     int j,i;
     double tSquared,t;
     t = 1.0;

     tSquared = t*t;

     i = 1000;
     for( j=1; j < i; j += 2)
        {
          tSquared += 1.0;
          tSquared += foo(2.2);
        }

     return 0;
   }
```

Figure 48.1: Example source code used as input to program in codes used in this chapter.

```
// Demonstration of instrumentation using the TAU performance monitoring tools (University of Oregon)

#include "rose.h"

using namespace std;

#define NEW_FILE_INFO Sg_File_Info::generateDefaultFileInfoForTransformationNode()

SgClassDeclaration*
getProfilerClassDeclaration (SgProject* project)
   {
  // Note that it would be more elegant to look this up in the Symbol table (do this next)

     SgClassDeclaration* returnClassDeclaration = NULL;
     Rose_STL_Container<SgNode*> classDeclarationList = NodeQuery::querySubTree (project,V_SgClassDeclaration);
     for (Rose_STL_Container<SgNode*>::iterator i = classDeclarationList.begin(); i != classDeclarationList.end(); i++)
        {
       // Need to cast *i from SgNode to at least a SgStatement
          SgClassDeclaration* classDeclaration = isSgClassDeclaration(*i);
          ROSE_ASSERT (classDeclaration != NULL);

       // printf ("In getProfilerClassDeclaration(): classDeclaration->get_name() = %s \n",classDeclaration->get_name().s

          if (classDeclaration->get_name() == "Profiler")
               returnClassDeclaration = classDeclaration;
        }

     ROSE_ASSERT(returnClassDeclaration != NULL);
     return returnClassDeclaration;
   }


int
main( int argc, char * argv[] )
   {
  // This test code tests the AST rewrite mechanism to add TAU Instrumention to the AST.

     SgProject* project = frontend(argc,argv);

  // Output the source code file (as represented by the SAGE AST) as a PDF file (with bookmarks)
  // generatePDF(*project);

  // Output the source code file (as represented by the SAGE AST) as a DOT file (graph)
  // generateDOT(*project);

  // Allow ROSE translator options to influence if we transform the AST
     if ( project->get_skip_transformation() == false )
        {
       // NOTE: There can be multiple files on the command line and each file has a global scope

          MiddleLevelRewrite::ScopeIdentifierEnum   scope
= MidLevelCollectionTypedefs::StatementScope;
          MiddleLevelRewrite::PlacementPositionEnum locationInScope = MidLevelCollectionTypedefs::TopOfCurrentScope;

       // Add a TAU include directive to the top of the global scope
          Rose_STL_Container<SgNode*> globalScopeList = NodeQuery::querySubTree (project,V_SgGlobal);
          for (Rose_STL_Container<SgNode*>::iterator i = globalScopeList.begin(); i != globalScopeList.end(); i++)
             {
            // Need to cast *i from SgNode to at least a SgStatement
               SgGlobal* globalScope = isSgGlobal(*i);
               ROSE_ASSERT (globalScope != NULL);

            // TAU does not seem to compile using EDG or g++ (need to sort this out with Brian)
            // MiddleLevelRewrite::insert(globalScope,"#define PROFILING_ON \n#include<TAU.h> \n",scope,locationInScope);
            // MiddleLevelRewrite::insert(globalScope,"#include<TAU.h> \n",scope,locationInScope);
               MiddleLevelRewrite::insert(globalScope,"#define PROFILING_ON 1 \n#define TAU_STDCXXLIB 1 \n#include<TAU.h>
             }
#if 1
       // Now get the class declaration representing the TAU type with which to build variable declarations
          SgClassDeclaration* tauClassDeclaration = getProfilerClassDeclaration(project);
          ROSE_ASSERT(tauClassDeclaration != NULL);
          SgClassType* tauType = tauClassDeclaration->get_type();
          ROSE_ASSERT(tauType != NULL);

       // Get a constructor to use with the variable declaration (anyone will due for code generation)
          SgMemberFunctionDeclaration* memberFunctionDeclaration = SageInterface::getDefaultConstructor(tauClassDeclaratio
          ROSE_ASSERT(memberFunctionDeclaration != NULL);

       // Add the instrumentation to each function
          Rose_STL_Container<SgNode*> functionDeclarationList = NodeQuery::querySubTree (project,V_SgFunctionDeclaration)
          for (Rose_STL_Container<SgNode*>::iterator i = functionDeclarationList.begin(); i != functionDeclarationList.en
             {
               SgFunctionDeclaration* functionDeclaration = isSgFunctionDeclaration(*i);
```

```
Test Failed!
```

Figure 48.3: Output of input code using tauInstrumenter.C

Chapter 49

The Haskell Interface

ROSE's Haskell interface allows the user to analyse and transform the Sage III IR from Haskell, a statically typed pure functional programming language. See http://www.haskell.org/.

The interface exposes almost all Sage III APIs to Haskell, allowing the user to call whichever APIs are required. The interface also supports an AST traversal mechanism inspired by Haskell's *scrap your boilerplate* design pattern.

The Haskell interface also provides a convenient mechanism for a user to rapidly experiment with the ROSE IR. GHC's command-line interpreter `ghci` can be used to explore the IR interactively by invoking API methods at will.

The Haskell interface relies on the Glasgow Haskell Compiler (GHC). It is auto-configured so long as the GHC binaries are in your `$PATH`. If not, you will need to supply the path to the binaries at configure time with the option `--with-haskell=`*bindir*, where *bindir* is the path to GHC's `bin` directory.

After installation, ROSE is available as a standard Haskell package named `rose`. This means that you can supply the flag `-package rose` to the Haskell compiler in order to make the extension available for use.

To understand the usage of the interface, it is crucial to grasp how the concept of *monads* works in Haskell. For a useful tutorial on monads, the reader is referred to the "All About Monads" tutorial found at http://www.haskell.org/all_about_monads/.

The simplest Haskell-based ROSE program is the identity translator, whose code is listed in Figure 49.1.

```haskell
module Main where

import ROSE
import System

main = do
        project <- frontend =<< getArgs
        exitWith =<< backend project
```

Figure 49.1: Haskell version of identity translator.

49.1 Traversals

As previously mentioned, the traversal mechanism is inspired by the scrap-your-boilerplate pattern. Our implementation of the scrap-your-boilerplate pattern provides both *transformation* and *query* traversals. A transformation traversal applies a global transformation to a tree by applying a given function to each tree node, whereas a query traversal derives a result from a tree using a function that produces a result from a node together with a *combinator* which combines the results from several nodes (for example in a summation query, the combinator may be the addition function).

In order to carry out a traversal, two steps are necessary. Firstly one must build a *type extension*, a type-generic function built from one or more type-specific functions. Secondly one must employ a *generic traversal combinator* which applies the type extension throughout the program.

In our interface type extensions for transformations are built using the functions `mkMn`, which builds a type extension from a type-specific function, and `extMn`, which extends an existing type extension with a type-specific function. Likewise `mkMqn` and `extMqn` for queries. These functions perform static and dynamic type checking such that they will only call the type-specific functions when it is safe to do so.

The two generic traversal combinators are `everywhereMc` and `everythingMc`. They take two arguments: the type extension and the tree to be traversed. `everywhereMc` returns the transformed tree, and `everythingMc` the result of the query.

Tying everything together, Figure 49.2 shows an example of a simple constant folding transformation.

49.2 Further Reading

Reference documentation for the interface is available on ROSE's website at:

`http://www.rosecompiler.org/ROSE_HaskellAPI/`

```
module Main where

import Data.Maybe
import Control.Monad
import System
import Data.DataMc
import ROSE
import ROSE.Sage3
import Time

simplifyAddOp :: SgAddOp () -> IO (SgExpression ())
simplifyAddOp = simplify (+)

simplifySubtractOp :: SgSubtractOp () -> IO (SgExpression ())
simplifySubtractOp = simplify (-)

simplifyMultiplyOp :: SgMultiplyOp () -> IO (SgExpression ())
simplifyMultiplyOp = simplify (*)

simplifyDivideOp :: SgDivideOp () -> IO (SgExpression ())
simplifyDivideOp = simplify div

simplify op n | n == nullSgNode = return (upSgExpression n)
              | otherwise = do
  lhs <- binaryOpGetLhsOperand n
  rhs <- binaryOpGetRhsOperand n
  lhsInt <- isSgIntVal lhs
  rhsInt <- isSgIntVal rhs
  if isJust lhsInt && isJust rhsInt then do
    lhsVal <- intValGetValue (fromJust lhsInt)
    rhsVal <- intValGetValue (fromJust rhsInt)
    let sum = lhsVal `op` rhsVal
    fi <- sgNullFile
    liftM upSgExpression (newIntVal fi sum (show sum))
   else
    return (upSgExpression n)

main :: IO ()
main = do
  time1 <- getClockTime
  prj <- frontend =<< getArgs
  time2 <- getClockTime
  putStrLn ("Frontend took " ++ show (diffClockTimes time2 time1))
  everywhereMc (mkMn simplifyAddOp `extMn` simplifySubtractOp
        `extMn` simplifyMultiplyOp `extMn` simplifyDivideOp) prj
  time3 <- getClockTime
  putStrLn ("Traversal took " ++ show (diffClockTimes time3 time2))
  exitWith =<< backend prj
```

Figure 49.2: Haskell version of constant folding transformation.

Part VIII

Parallelism

Topics relevant to shared or distributed parallel computing using ROSE.

Chapter 50

Shared-Memory Parallel Traversals

Besides the traversal classes introduced in Chapter 7, ROSE also includes classes to run multi-threaded traversals to make use of multi-CPU systems with shared memory (such as typical multicore desktop systems). These shared memory traversals are like the combined traversal classes in that they run several small traversal classes simultaneously; the difference is that here different visit functions may be executed concurrently on different CPUs, while the combined traversals always execute visit functions sequentially.

Because of this similarity, the public interface for the parallel traversals is a superset of the combined traversal interface. For each Ast*Processing class there is an AstSharedMemoryParallel*Processing class that provides an interface for adding traversals to its internal list, getting a reference to the internal list, and for starting the combined traversal. The `traverse()` method performs the same combined traversal as in the corresponding AstCombined*Processing class, and the new `traverseInParallel()` method (with the same signature as `traverse()`) must be used to start a parallel traversal. (We currently do not provide `traverseWithinFileInParallel()` and `traverseInputFilesInParallel()` that would be needed to make the parallel processing classes a fully-featured drop-in replacement for other classes.)

A example of how to use the parallel traversal classes is given in Figure 50.1 (note the similarity to Figure 7.24 on page 61). A group of traversal objects is executed first in combined mode and then in parallel threads.

It is the user's responsibility to make sure that the actions executed in the parallel traversal are thread-safe. File or terminal I/O may produce unexpected results if several threads attempt to write to the same stream at once. Allocation of dynamic memory (including the use of ROSE or standard C++ library calls that allocate memory) may defeat the purpose of multi-threading as such calls will typically be serialized by the library.

Two member functions in each AstSharedMemoryParallel*Processing class are available to tune the performance of the parallel traversals. The first is `void set_numberOfThreads(size_t threads)` which can be used to set the number of parallel threads. The default value for this parameter is 2. Our experiments suggest that even on systems with more than two CPUs, running more than two traversal threads in parallel does not typically increase performance

```cpp
#include <rose.h>

class NodeTypeTraversal: public AstSimpleProcessing {
public:
    NodeTypeTraversal(enum VariantT variant, std::string typeName)
        : myVariant(variant), typeName(typeName) {
    }

protected:
    virtual void visit(SgNode *node) {
        if (node->variantT() == myVariant) {
            std::cout << "Found " << typeName;
            if (SgLocatedNode *loc = isSgLocatedNode(node)) {
                Sg_File_Info *fi = loc->get_startOfConstruct();
                if (fi->isCompilerGenerated()) {
                    std::cout << ": compiler generated";
                } else {
                    std::cout << ": " << fi->get_filenameString()
                              << ":" << fi->get_line();
                }
            }
            std::cout << std::endl;
        }
    }

private:
    enum VariantT myVariant;
    std::string typeName;
};

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

    std::cout << "combined execution of traversals" << std::endl;
    AstSharedMemoryParallelSimpleProcessing parallelTraversal(5);
    parallelTraversal.addTraversal(new NodeTypeTraversal(V_SgForStatement, "for loop"));
    parallelTraversal.addTraversal(new NodeTypeTraversal(V_SgIntVal, "int constant"));
    parallelTraversal.addTraversal(new NodeTypeTraversal(V_SgVariableDeclaration, "variable declaration"));
    parallelTraversal.traverse(project, preorder);
    std::cout << std::endl;

    std::cout << "shared-memory parallel execution of traversals" << std::endl;
    parallelTraversal.traverseInParallel(project, preorder);
}
```

Figure 50.1: Example source showing the shared-memory parallel execution of traversals.

because the memory bandwidth is saturated.

The second function is `void set_synchronizationWindowSize(size_t windowSize)`. This sets a parameter that corresponds to the size of a 'window' of AST nodes that the parallel threads use to synchronize. The value is, in effect, the number of AST nodes that are visited by each thread before synchronizing. Smaller values may in theory result in more locality and therefore better cache utilization at the expense of more time spent waiting for other threads. In practice, synchronization overhead appears to dominate caching effects, so making this parameter too small inhibits performance. The default value is 100000; any large values will result in comparable execution times.

```
combined execution of traversals
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found for loop: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_Exa
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found variable declaration: compiler generated
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found for loop: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_Exa
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_

shared-memory parallel execution of traversals
Found for loop: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_Exa
Found for loop: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_Exa
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found int constant: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/inputCode_
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found variable declaration: compiler generated
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
Found variable declaration: /export/tmp.hudson-rose/hudson/workspace/a90-ROSE-daily-release/label/amd64-linux/tutorial/in
```

Figure 50.2: Output of input file to the shared-memory parallel traversals. Output may be garbled depending on the multi-threaded behavior of the underlying I/O libraries.

Chapter 51

Distributed-Memory Parallel Traversals

ROSE provides an experimental distributed-memory AST traversal mechanism meant for very large scale program analysis. It allows you to distribute expensive program analyses among a distributed-memory system consisting of many processors; this can be a cluster or a network of workstations. Different processes in the distributed system will get different parts of the AST to analyze: Each process is assigned a number of defining function declarations in the AST, and a method implemented by the user is invoked on each of these. The parts of the AST outside of function definitions are shared among all processes, but there is no guarantee that all function definitions are visible to all processes.

The distributed memory analysis framework uses the MPI message passing library for communicating attributes among processes. You will need an implementation of MPI to be able to build and run programs using distributed memory traversals; consult your documentation on how to run MPI programs. (This is often done using a program named `mpirun`, `mpiexecute`, or similar.)

Distributed memory analyses are performed in three phases:

1. A top-down traversal (the 'pre-traversal') specified by the user runs on the shared AST outside of function definitions. The inherited attributes this traversal computes for defining function declaration nodes in the AST are saved by the framework for use in the next phase.

2. For every defining function declaration, the user-provided `analyzeSubtree()` method is invoked; these calls run concurrently, but on different function declarations, on all processors. It takes as arguments the AST node for the function declaration and the inherited attribute computed for this node by the pre-traversal. Within `analyzeSubtree()` any analysis features provided by ROSE can be used. This method returns the value that will be used as the synthesized attribute for this function declaration in the bottom-up traversal (the 'post-traversal').

   However, unlike normal bottom-up traversals, the synthesized attribute is not simply copied in memory as the AST is distributed. The user must therefore provide the methods `serializeAttribute()` and `deserializeAttribute()`. These compute a serialized representation of a synthesized attribute, and convert such a representation back to the user's synthesized attribute type, respectively. A serialized attribute is a pair of an integer specifying the size of the attribute in bytes and a pointer to a region of memory of that size that will be copied byte by byte across the distributed system's communication network. Attributes from different parts of the AST may have different sizes. As serialization of attributes will often involve dynamic memory allocation, the user can also implement the `deleteSerializedAttribute()` method to such dynamic memory after the serialized data has been copied to the communication subsystem's internal buffer.

Within the `analyzeSubtree()` method the methods `numberOfProcesses()` and `myID()` can be called. These return the total number of concurrent processes, and an integer uniquely identifying the currently running process, respectively. The ID ranges from 0 to one less than the number of processes, but has no semantics other than that it is different for each process.

3. Finally, a bottom-up traversal is run on the shared AST outside of function definitions. The values returned by the distributed analyzers in the previous phase are used as synthesized attributes for function definition nodes in this traversal.

After the bottom-up traversal has finished, the `getFinalResults()` method can be invoked to obtain the final synthesized attribute. The `isRootProcess()` method returns true on exactly one designated process and can be used to perform output, program transformations, or other tasks that are not meant to be run on each processor.

Figure 51.1 gives a complete example of how to use the distributed memory analysis framework. It implements a trivial analysis that determines for each function declaration at what depth in the AST it can be found and what its name is. Figure 51.2 shows the output produced by this program when running using four processors on some input files.

```
// This is a small example of how to use the distributed memory traversal mechanism. It computes a list of function
// definitions in a program and outputs their names, their depth in the AST, and the ID of the process that found it.

#include <rose.h>
#include "DistributedMemoryAnalysis.h"

// The pre-traversal runs before the distributed part of the analysis and is used to propagate context information down
// to the individual function definitions in the AST. Here, it just computes the depth of nodes in the AST.
class FunctionNamesPreTraversal: public AstTopDownProcessing<int>
{
protected:
    int evaluateInheritedAttribute(SgNode *, int depth)
    {
        return depth + 1;
    }
};

// The post-traversal runs after the distributed part of the analysis and is used to collect the information it
// computed. Here, the synthesized attributes computed by the distributed analysis are strings representing information
// about functions. These strings are concatenated by the post-traversal (and interleaved with newlines where necessary).
class FunctionNamesPostTraversal: public AstBottomUpProcessing<std::string>
{
protected:
    std::string evaluateSynthesizedAttribute(SgNode *node, SynthesizedAttributesList synAttributes)
    {
        std::string result = "";
        SynthesizedAttributesList::iterator s;
        for (s = synAttributes.begin(); s != synAttributes.end(); ++s)
        {
            std::string &str = *s;
            result += str;
            if (str.size() > 0 && str[str.size()-1] != '\n')
                result += "\n";
        }
        return result;
    }

    std::string defaultSynthesizedAttribute()
    {
        return "";
    }
};

// This is the distributed part of the analysis. The DistributedMemoryTraversal base class is a template taking an
// inherited and a synthesized attribute type as template parameters; these are the same types used by the pre- and
// post-traversals.
class FunctionNames: public DistributedMemoryTraversal<int, std::string>
{
protected:
 // The analyzeSubtree() method is called for every defining function declaration in the AST. Its second argument is the
 // inherited attribute computed for this node by the pre-traversal, the value it returns becomes the synthesized
 // attribute used by the post-traversal.
    std::string analyzeSubtree(SgFunctionDeclaration *funcDecl, int depth)
    {
        std::string funcName = funcDecl->get_name().str();
        std::stringstream s;
        s << "process " << myID() << ": at depth " << depth << ": function " << funcName;
        return s.str();
    }

 // The user must implement this method to pack a synthesized attribute (a string in this case) into an array of bytes
 // for communication. The first component of the pair is the number of bytes in the buffer.
    std::pair<int, void *> serializeAttribute(std::string attribute) const
    {
        int len = attribute.size() + 1;
        char *str = strdup(attribute.c_str());
        return std::make_pair(len, str);
    }

 // This method must be implemented to convert the serialized data to the application's synthesized attribute type.
    std::string deserializeAttribute(std::pair<int, void *> serializedAttribute) const
    {
        return std::string((const char *) serializedAttribute.second);
    }

 // This method is optional (the default implementation is empty). Its job is to free memory that may have been
 // allocated by the serializeAttribute() method.
    void deleteSerializedAttribute(std::pair<int, void *> serializedAttribute) const
    {
        std::free(serializedAttribute.second);
    }
};
```

```
----- found the following functions: ------
process 0: at depth 3: function il
process 0: at depth 5: function head
process 0: at depth 5: function eq
process 1: at depth 3: function headhead
process 1: at depth 3: function List
process 1: at depth 3: function find
process 1: at depth 3: function head
process 2: at depth 3: function operator!=
process 2: at depth 3: function find
process 2: at depth 3: function head
process 2: at depth 3: function fib
process 3: at depth 3: function xform
process 3: at depth 3: function func
process 3: at depth 3: function f
process 3: at depth 3: function g
process 3: at depth 3: function deref
-------------------------------------------
```

Figure 51.2: Example output of a distributed-memory analysis running on four processors.

Chapter 52

Parallel Checker

This Chapter is about the project *DistributedMemoryAnalysisCompass*, which runs Compass Checkers in Parallel, i.e. shared, combined and distributed.

52.1 Different Implementations

The project contains the following files:

- parallel_functionBased_ASTBalance contains the original implementation, which is based on an AST traversal that is balanced based on the number of nodes in each function. Then the functions are distributed over all processors. It contains as well the original interfaces to the shared and combined traversal work.
- parallel_file_compass distributed on the granularity level of files.
- parallel_functionBased_dynamicBalance is the implementation of dynamically scheduling functions across processors. In addition, this algorithm weights the functions first and then sorts them in descending order according to their weight.
- parallel_compass performs dynamic scheduling based on nodes. The nodes are weighted and then sorted. This algorithm allows the greatest scalability.

52.2 Running through PSUB

The following represents a typical script to run parallel_compass on 64 processors using CXX_Grammer. CXX_Grammar is a binary ROSE AST representation of a previously parsed program. We specify 65 processors because processor 0 does only communication and no computation. Furthermore, we ask for 17 nodes of which each has 8 processors giving us a total of 136 possible processes. We only need 65 but still want to use this configuration. This will average out our 65 processes over 17 nodes, resulting in about 4 processors per node. This trick is used because the AST loaded into memory takes about 400 MB per process. We end up with 1600MB per node.

```
#!/bin/bash
#  Sample LCRM script to be submitted with psub
#PSUB -r ncxx65 # sets job name (limit of 7 characters)
#PSUB -b nameofbank  # sets bank account
#PSUB -ln 17 # == defines the amount of nodes needed
#PSUB -o ~/log.log
#PSUB -e ~/log.err
#PSUB -tM 0:05  # Max time 5 min runtime
#PSUB -x  # export current env var settings
#PSUB -nr # do NOT rerun job after system reboot
#PSUB -ro # send output log directly to file
#PSUB -re # send err log directly to file
#PSUB -mb # send email at execution start
#PSUB -me # send email at execution finish
#PSUB -c zeus
#PSUB # no more psub commands
# job commands start here
set echo
echo LCRM job id = $PSUB_JOBID
cd ~/new-src/build-rose/projects/DistributedMemoryAnalysisCompass/
srun -n 65 ./parallel_compass -load ~/CXX_Grammar.ast
echo "ALL DONE"
```

There are a few tricks that could be considered. Prioritization is based on the amount of time and nodes requested. If less time is specified, it is more likely that a job runs very soon, as processing time becomes available.

To submit the job above, use *psub file-name*. To check the job in the queue, use *squeue* and to cancel the job use *mjobctl -c job-number*.

Chapter 53

Reduction Recognition

Figures 53.1 shows a translator which finds the first loop of a main function and recognizes reduction operations and variables within the loop. A reduction recognition algorithm (ReductionRecognition()) is implemented in the SageInterface namespace and follows the C/C++ reduction restrictions defined in the OpenMP 3.0 specification.

```
// Test reduction recognition
#include "rose.h"
#include <iostream>
#include <set>

using namespace std;

int main(int argc, char * argv[])
{
  SgProject *project = frontend (argc, argv);
  //Find main() function
  SgFunctionDeclaration* func = SageInterface::findMain(project);
  ROSE_ASSERT(func != NULL);
  SgBasicBlock* body = func->get_definition()->get_body();

  //Find the first loop
  Rose_STL_Container<SgNode*> node_list = NodeQuery::querySubTree(body,V_SgForStatement);
  SgForStatement* loop = isSgForStatement(*(node_list.begin()));
  ROSE_ASSERT(loop != NULL);

  //Collect reduction variables and operations
  std::set< std::pair <SgInitializedName*, VariantT> > reductions;
  std::set< std::pair <SgInitializedName*, VariantT> >::const_iterator iter;
  SageInterface::ReductionRecognition(loop, reductions);

  // Show the results
  cout<<"Reduction recognition results:"<<endl;
  for (iter=reductions.begin(); iter!=reductions.end(); iter++)
  {
    std::pair <SgInitializedName*, VariantT> item = *iter;
    cout<<"\t variable: "<<item.first->unparseToString()<<"\t operation:"<<getVariantName(item.second)<<endl;;
  }

  return backend(project);
}
```

Figure 53.1: Example source code showing reduction recognition.

Using this translator we can compile the code shown in figure 53.2. The output is shown in figure 53.3.

```
int a[100], sum;
int  main()
{
  int i,sum2,yy,zz;
  sum = 0;
  for (i=0;i<100;i++)
  {
    int xx;
    a[i]=i;
    sum = a[i]+ sum;
    xx++;
    yy=0;
    yy--;
    zz*=a[i];
  }
  return 0;
}
```

Figure 53.2: Example source code used as input to loop reduction recognition processor.

```
Reduction recognition results:
         variable: sum    operation:SgAddOp
         variable: zz     operation:SgMultAssignOp
```

Figure 53.3: Output of input to reduction recognition processor.

Part IX

Tutorial Summary

Summary of the ROSE tutorials.

Chapter 54

Tutorial Wrap-up

This tutorial has shown the construction and simple manipulation of the AST as part of the construction of the source-to-source translators using ROSE. Much more complex translators are possible using ROSE, but they are not such that they present well as part of a tutorial with short example programs. The remaining chapters of the tutorial include examples of translators built using ROSE as part of active collaborations with external research groups.

FIXME: *Reference the User Manual, HTML Doxygen generated documentation, unresolved issues, etc. Reference other work currently using ROSE (ANL, Cornell in the future), academic collaborations.*

Appendix

This appendix includes information useful in supporting the ROSE Tutorial.

54.1 Location of To Do List

This was an older location for the Tutorial Tod List. We now keep the Tod list in the `ROSE/docs/testDoxygen/ProjectToDoList.docs` in the section called: `ROSE Tutorial Todo List`.

54.2 Abstract Grammar

In this section we show an abstract grammar for the ROSE AST. The grammar generates the set of all ASTs. On the left hand side of a production we have a non-terminal that corresponds to an inner node of the class hierarchy. On the right hand side of a production we have either one non-terminal or one terminal. The terminal corresponds to a leaf-node where the children of the respective node are listed as double-colon separated pairs, consisting of an access name (= name for get function) and a name that directly corresponds to the class of the child. Details like pointers are hidden. The asterisk shows where lists of children (containers) are used in the ROSE AST. For each terminal, a name followed by '(' and ')', a variant exists in ROSE with the prefix `V_` that can be obtained by using the function variantT() on a node. Note, that concrete classes of AST nodes directly correspond to terminals and base classes to non-terminals.

```
START:SgNode
SgNode : SgSupport
 | SgLocatedNode
 | SgSymbol
 ;

SgSupport : SgName()
 | SgSymbolTable()
 | SgInitializedName ( initptr:SgInitializer )
 | SgFile ( root:SgGlobal ( declarations:SgDeclarationStatement* ) )
 | SgProject ( fileList:SgFile ( root:SgGlobal ( declarations:SgDeclarationStatement* ) ) )
 | SgOptions()
 | SgBaseClass ( base_class:SgClassDeclaration )
 | SgTemplateParameter ( expression:SgExpression, defaultExpressionParameter:SgExpression,
```

```
                         templateDeclaration:SgTemplateDeclaration(),
                         defaultTemplateDeclarationParameter:SgTemplateDeclaration() )
 | SgTemplateArgument ( expression:SgExpression,
                        templateInstantiation:SgTemplateInstantiationDecl
                           ( definition:SgClassDefinition ) )
 | SgFunctionParameterTypeList()
 | SgAttribute
 | SgModifier
 ;

SgAttribute : SgPragma()
 | SgBitAttribute
 ;

SgBitAttribute : SgFuncDecl_attr()
 | SgClassDecl_attr()
 ;

SgModifier : SgModifierNodes()
 | SgConstVolatileModifier()
 | SgStorageModifier()
 | SgAccessModifier()
 | SgFunctionModifier()
 | SgUPC_AccessModifier()
 | SgSpecialFunctionModifier()
 | SgElaboratedTypeModifier()
 | SgLinkageModifier()
 | SgBaseClassModifier()
 | SgDeclarationModifier()
 ;

SgLocatedNode : SgStatement
 | SgExpression
 ;

SgStatement : SgExprStatement ( expression_root:SgExpressionRoot ( operand_i:SgExpression ) )
 | SgLabelStatement()
 | SgCaseOptionStmt ( key_root:SgExpressionRoot ( operand_i:SgExpression ),
                      body:SgBasicBlock ( statements:SgStatement* ) )
 | SgTryStmt ( body:SgBasicBlock ( statements:SgStatement* ),
               catch_statement_seq_root:SgCatchStatementSeq ( catch_statement_seq:SgStatement* ) )
 | SgDefaultOptionStmt ( body:SgBasicBlock ( statements:SgStatement* ) )
 | SgBreakStmt()
 | SgContinueStmt()
 | SgReturnStmt ( expression_root:SgExpressionRoot ( operand_i:SgExpression ) )
 | SgGotoStatement()
```

```
 | SgSpawnStmt ( the_func_root:SgExpressionRoot ( operand_i:SgExpression ) )
 | SgForInitStatement ( init_stmt:SgStatement* )
 | SgCatchStatementSeq ( catch_statement_seq:SgStatement* )
 | SgClinkageStartStatement()
 | SgDeclarationStatement
 | SgScopeStatement
 ;

SgDeclarationStatement :
   SgVariableDeclaration ( variables:SgInitializedName ( initptr:SgInitializer ) )
 | SgVariableDefinition ( vardefn:SgInitializedName ( initptr:SgInitializer ),
                          bitfield:SgUnsignedLongVal() )
 | SgEnumDeclaration()
 | SgAsmStmt ( expr_root:SgExpressionRoot ( operand_i:SgExpression ) )
 | SgTemplateDeclaration()
 | SgNamespaceDeclarationStatement ( definition:SgNamespaceDefinitionStatement
                                        ( declarations:SgDeclarationStatement* )
                                   )
 | SgNamespaceAliasDeclarationStatement()
 | SgUsingDirectiveStatement()
 | SgUsingDeclarationStatement()
 | SgFunctionParameterList ( args:SgInitializedName ( initptr:SgInitializer ) )
 | SgCtorInitializerList ( ctors:SgInitializedName ( initptr:SgInitializer ) )
 | SgPragmaDeclaration ( pragma:SgPragma() )
 | SgClassDeclaration
 | SgFunctionDeclaration
 ;

SgClassDeclaration : SgTemplateInstantiationDecl ( definition:SgClassDefinition )
 ;

SgFunctionDeclaration :
   SgTemplateInstantiationFunctionDecl ( parameterList:SgFunctionParameterList
                                          ( args:SgInitializedName ( initptr:SgInitializer ) ),
                                            definition:SgFunctionDefinition
                                               ( body:SgBasicBlock ( statements:SgStatement* ) )
                                       )
 | SgMemberFunctionDeclaration
 ;

SgMemberFunctionDeclaration :
    SgTemplateInstantiationMemberFunctionDecl ( parameterList:SgFunctionParameterList
                                                  ( args:SgInitializedName ( initptr:SgInitializer ) ),
                                                definition:SgFunctionDefinition
                                                  ( body:SgBasicBlock ( statements:SgStatement* ) ),
                                                CtorInitializerList:SgCtorInitializerList
```

```
                                            ( ctors:SgInitializedName ( initptr:SgInitializer
                                        )
 ;

SgScopeStatement : SgGlobal ( declarations:SgDeclarationStatement* )
 | SgBasicBlock ( statements:SgStatement* )
 | SgIfStmt ( conditional:SgStatement,
              true_body:SgBasicBlock ( statements:SgStatement* ),
              false_body:SgBasicBlock ( statements:SgStatement* ) )
 | SgForStatement ( for_init_stmt:SgForInitStatement ( init_stmt:SgStatement* ),
                    test_expr_root:SgExpressionRoot ( operand_i:SgExpression ),
                    increment_expr_root:SgExpressionRoot ( operand_i:SgExpression ),
                    loop_body:SgBasicBlock ( statements:SgStatement* ) )
 | SgFunctionDefinition ( body:SgBasicBlock ( statements:SgStatement* ) )
 | SgWhileStmt ( condition:SgStatement, body:SgBasicBlock ( statements:SgStatement* ) )
 | SgDoWhileStmt ( condition:SgStatement, body:SgBasicBlock ( statements:SgStatement* ) )
 | SgSwitchStatement ( item_selector_root:SgExpressionRoot ( operand_i:SgExpression ),
                       body:SgBasicBlock ( statements:SgStatement* ) )
 | SgCatchOptionStmt ( condition:SgVariableDeclaration
                                      ( variables:SgInitializedName ( initptr:SgInitializer ) ),
                                        body:SgBasicBlock ( statements:SgStatement* )
                     )
 | SgNamespaceDefinitionStatement ( declarations:SgDeclarationStatement* )
 | SgClassDefinition
 ;

SgClassDefinition : SgTemplateInstantiationDefn ( members:SgDeclarationStatement* )
 ;

SgExpression : SgExprListExp ( expressions:SgExpression* )
 | SgVarRefExp()
 | SgClassNameRefExp()
 | SgFunctionRefExp()
 | SgMemberFunctionRefExp()
 | SgFunctionCallExp ( function:SgExpression, args:SgExprListExp ( expressions:SgExpression* ) )
 | SgSizeOfOp ( operand_expr:SgExpression )
 | SgConditionalExp ( conditional_exp:SgExpression,
                      true_exp:SgExpression,
                      false_exp:SgExpression )
 | SgNewExp ( placement_args:SgExprListExp ( expressions:SgExpression* ),
              constructor_args:SgConstructorInitializer(
                                             args:SgExprListExp(expressions:SgExpression*)
                                                        ),
              builtin_args:SgExpression )
 | SgDeleteExp ( variable:SgExpression )
 | SgThisExp()
```

```
 | SgRefExp()
 | SgVarArgStartOp ( lhs_operand:SgExpression, rhs_operand:SgExpression )
 | SgVarArgOp ( operand_expr:SgExpression )
 | SgVarArgEndOp ( operand_expr:SgExpression )
 | SgVarArgCopyOp ( lhs_operand:SgExpression, rhs_operand:SgExpression )
 | SgVarArgStartOneOperandOp ( operand_expr:SgExpression )
 | SgInitializer
 | SgValueExp
 | SgBinaryOp
 | SgUnaryOp
 ;

SgInitializer :
   SgAggregateInitializer ( initializers:SgExprListExp ( expressions:SgExpression* ) )
 | SgConstructorInitializer ( args:SgExprListExp ( expressions:SgExpression* ) )
 | SgAssignInitializer ( operand_i:SgExpression )
 ;

SgValueExp : SgBoolValExp()
 | SgStringVal()
 | SgShortVal()
 | SgCharVal()
 | SgUnsignedCharVal()
 | SgWcharVal()
 | SgUnsignedShortVal()
 | SgIntVal()
 | SgEnumVal()
 | SgUnsignedIntVal()
 | SgLongIntVal()
 | SgLongLongIntVal()
 | SgUnsignedLongLongIntVal()
 | SgUnsignedLongVal()
 | SgFloatVal()
 | SgDoubleVal()
 | SgLongDoubleVal()
 ;

SgBinaryOp : SgArrowExp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgDotExp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgDotStarOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgArrowStarOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgEqualityOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgLessThanOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgGreaterThanOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgNotEqualOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgLessOrEqualOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
```

```
 | SgGreaterOrEqualOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgAddOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgSubtractOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgMultiplyOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgDivideOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgIntegerDivideOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgModOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgAndOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgOrOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgBitXorOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgBitAndOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgBitOrOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgCommaOpExp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgLshiftOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgRshiftOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgPntrArrRefExp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgScopeOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgPlusAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgMinusAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgAndAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgIorAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgMultAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgDivAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgModAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgXorAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgLshiftAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 | SgRshiftAssignOp ( lhs_operand_i:SgExpression, rhs_operand_i:SgExpression )
 ;

SgUnaryOp : SgExpressionRoot ( operand_i:SgExpression )
 | SgMinusOp ( operand_i:SgExpression )
 | SgUnaryAddOp ( operand_i:SgExpression )
 | SgNotOp ( operand_i:SgExpression )
 | SgPointerDerefExp ( operand_i:SgExpression )
 | SgAddressOfOp ( operand_i:SgExpression )
 | SgMinusMinusOp ( operand_i:SgExpression )
 | SgPlusPlusOp ( operand_i:SgExpression )
 | SgBitComplementOp ( operand_i:SgExpression )
 | SgCastExp ( operand_i:SgExpression )
 | SgThrowOp ( operand_i:SgExpression )
 ;

SgSymbol : SgVariableSymbol()
 | SgClassSymbol ( declaration:SgClassDeclaration )
 | SgTemplateSymbol ( declaration:SgTemplateDeclaration() )
```

```
 | SgEnumSymbol ( declaration:SgEnumDeclaration() )
 | SgEnumFieldSymbol()
 | SgLabelSymbol ( declaration:SgLabelStatement() )
 | SgDefaultSymbol()
 | SgNamespaceSymbol ( declaration:SgNamespaceDeclarationStatement
                       ( definition:SgNamespaceDefinitionStatement
                          ( declarations:SgDeclarationStatement* )
                       )
                     )
 | SgFunctionSymbol
 ;

SgFunctionSymbol : SgMemberFunctionSymbol ( declaration:SgFunctionDeclaration )
 ;

SgPartialFunctionType :
   SgPartialFunctionModifierType ( ref_to:SgReferenceType, ptr_to:SgPointerType,
                                   modifiers:SgModifierNodes(), typedefs:SgTypedefSeq,
                                   return_type:SgType, orig_return_type:SgType )
 ;
```

This grammar was generated with GRATO, a grammar transformation tool, written by Markus Schordan. The input is a representation generated by ROSETTA. Several other versions of the grammar can be generated as well, such as eliminating nested tree nodes by introducing auxiliary non-terminals, introducing base types as non-terminals etc. Additionally from that grammar we can also generate grammars that can be used with yacc/bison, Coco, and other attribute grammar tools, as well as tree grammar based tools such as burg (requires a transformation to a binary tree).

Glossary

FIXME: *Define the following terms: IR node, Inherited Attribute, Synthesized Attribute, Accumulator Attribute, AST Traversal*

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

- **AST** Abstract Syntax Tree. A very basic understanding of an AST is the entry level into ROSE.
- **Attribute** User defined information (objects) associated with IR nodes. Forms of attributes include: accumulator, inherited, persistent, and synthesized. Both inherited and synthesized attributes are managed automatically on the stack within a traversal. Accumulator attributes are typically something semantically equivalent to a global variable (often a static data member of a class). Persistent attributes are explicitly added to the AST and are managed directly by the user. As a result, they can persist across multiple traversals of the AST. Persistent attributes are also saved in the binary file I/O, but only if the user provides the attribute specific `pack()` and `unpack()` virtual member functions. See the ROSE User Manual for more information, and the ROSE Tutorial for examples.
- **CFG** As used in ROSE, this is the Control Flow Graph, not Context Free Grammar or anything else.
- **EDG** Edison Design Group (the commercial company that produces the C and C++ front-end that is used in ROSE).
- **IR** Intermediate Representation (IR). The IR is the set of classes defined within SAGE III that allow an AST to be built to define any application in C, C++, and Fortran application.
- **Query** (as in AST Query) Operations on the AST that return answers to questions posed about the content or context in the AST.
- **ROSE** A project that covers both research in optimization and a specific infrastructure for handling large scale C, C++, and Fortran applications.
- **Rosetta** A tool (written by the ROSE team) used within ROSE to automate the generation of the SAGE III IR.
- **SAGE++ and SAGE II** An older object-oriented IR upon which the API of SAGE III IR is based.
- **Semantic Information** What abstractions mean (short answer). (This might be better as a description of what kind of semantic information ROSE could take advantage, not a definition.)

- **Telescoping Languages** A research area that defines a process to generate domain-specific languages from a general purpose languages.
- **Transformation** The process of automating the editing (either reconfiguration, addition, or deletion; or some combination) of input application parts to build a new application. In the context of ROSE, all transformations are source-to-source.
- **Translator** An executable program (in our context built using ROSE) that performs source-to-source translation on an existing input application source to generate a second (generated) source code file. The second (generated) source code is then typically provided as input to a vendor provided compiler (which generates object code or an executable program).
- **Traversal** The process of operating on the AST in some order (usually pre-order, post-order, out of order [randomly], depending on the traversal that is used). The ROSE user builds a traversal from base classes that do the traversal and execute a function, or a number of functions, provided by the user.