UCRL-PROC-217691

An Extensible Open-Source Compiler Infrastructure for Testing

D. Quinlan, S. Ur, R. Vuduc

December 13, 2005

IBM Verification Conference 2005
Haifa, Israel
November 13, 2005 through November 16, 2005

This research was supported under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

An Extensible Open-Source Compiler Infrastructure for Testing

Dan Quinlan[*] Shmuel Ur Richard Vuduc

December 9, 2005

Abstract

Testing forms a critical part of the development process for large-scale software, and there is growing need for automated tools that can read, represent, analyze, and transform the application's source code to help carry out testing tasks. However, the support required to compile applications written in common general purpose languages is generally inaccessible to the testing research community. In this paper, we report on an extensible, open-source compiler infrastructure called ROSE, which is currently in development at Lawrence Livermore National Laboratory. ROSE specifically targets developers who wish to build source-based tools that implement customized analyses and optimizations for large-scale C, C++, and Fortran90 scientific computing applications (on the order of a million lines of code or more). However, much of this infrastructure can also be used to address problems in testing, and ROSE is by design broadly accessible to those without a formal compiler background. This paper details the interactions between testing of applications and the ways in which compiler technology can aid in the understanding of those applications. We emphasize the particular aspects of ROSE, such as support for the general analysis of whole programs, that are particularly well-suited to the testing research community and the scale of the problems that community solves.

1 Introduction

Testing software involves a number of formal processes (*e.g.*, coverage analysis, model checking, bug pattern analysis, code reviews, deducing errors), which require accurate characterizations of the behavior of the program being tested. Deriving such characterizations for modern large-scale applications increasingly requires automated tools that can read, represent, analyze, and possibly transform the source code directly. These tools in turn often depend on a robust underlying compiler infrastructure. In this paper, we present the ROSE source-to-source compiler infrastructure for C, C++, and Fortran90 [1–3]. We believe that ROSE is well-suited to support testing tool development because it is easy to use, robust with respect to large and complex applications, and preserves the structure of the input source (including source file position and comment information), thereby enabling accurate program characterization.

There are three aspects of ROSE particularly relevant to developers of program testing and checking tools. First, the ROSE infrastructure can process complex, large-scale, production-quality scientific applications on the order of a million lines of code or more, which are being developed throughout the U.S. Department of

[*] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, USA (dquinlan@llnl.gov).

Energy (DOE) laboratories. These applications use complex language and library features, and therefore possess all of the qualities of general large-scale industrial applications. Secondly, ROSE is designed to be accessible to developers who do not necessarily have a formal compiler background, enabling a broad community of tool builders to construct interesting tools quickly. Thirdly, ROSE is fully open-source, modularly designed, and extensible. This aspect is important because current and future static and dynamic analysis tools for testing have widely differing requirements of any underlying compiler infrastructure [4–6]. Collectively, these aspects make ROSE a suitable framework within which to build a variety of source-code analysis and transformation tools.

Automatic testing and defect (or bug) detection tools based on static analysis compose a large and growing body of recent work whose effectiveness demands accurate source code representation and analysis (see Section 5.2). For C++ in particular, accurately representing programs is particularly challenging due to complexity of both the language and commonly-used support libraries. The core ROSE intermediate representation (IR), used to form the abstract syntax tree (AST), is engineered to handle this complexity. Having been designed to support source-to-source translation, it preserves all the program information required to precisely reconstruct the original application in the same language (without language translation), and therefore represents the full type and structural representation of the original source, with only minor renormalization. As a result, ROSE has all of the information that may be required to support the kind of program analysis techniques needed, for instance, to keep a given tool's false positive reporting rates low.

The remainder of this paper reviews the interface between compilation and testing, with an emphasis on the ways in which compilers can support testing activities (Section 2). We discuss the ROSE infrastructure itself, highlighting the components of the infrastructure we believe will be particularly useful to testing tool builders (Section 3). We present a concrete example of an on-going collaboration between IBM and the ROSE project in the development of a testing tool for code coverage analysis, and discuss additional ideas for other testing tools that would require compiler support and be useful to the testing of large scale application (Section 4).

2 How Compilers Can Support Testing

Testing comprises many specific activities that can make use of compilation techniques. Although many useful tools can be built and used without using a full compiler infrastructure, compilers can provide stronger and deeper analysis to improve the quality of reports by, for instance, reducing the number of false positive reports in a defect detection tool. In this section, we review a number of important testing activities and show how a compiler infrastructure can assist.

2.1 Making programs fail

One important testing technique is to introduce instrumentation in a program in order to force more exhaustive coverage of control flow paths during execution. One example is ConTest, a tool developed at IBM, which introduces timing perturbation in multi-threaded applications at testing-time [7]. These perturbations force untested thread inter-leavings that might cause race conditions or deadlocks, for example. We are using ROSE to introduce this kind of instrumentation in C and C++ applications.

It may be possible to extend this idea further by building compiler-based tools to inject inputs automatically at various program input points, to force failures or perform security bug testing. Both the analysis to locate input points and the transformation to inject data at such points can best be supported using compiler-based tools that can generate correct code using an analysis of the context at an input point in the application.

2.2 Reproducing bugs

Reproducing bugs generated from complex execution environments is the first step in fixing them, and knowing all the steps that lead to the bug is critical in understanding the context for the bug. Post-mortem analysis can be helpful, but either the support for this analysis is woven into the application by hand or automated tools are developed to support such transformations and enable the generation of a post-mortem trace. A compiler-based tool with full transformation capabilities should be relatively easy to build so that it can record limited tracing information and dump it to a file upon detection of internal faults. Such automated tools then work with any product application during development and provide useful information from failures in the field. This strategy accepts that bugs happen in production software and, where possible, provides a mechanism to detect the root cause or history of steps the led to uncovering the bug.

2.3 Extracting simplifying examples to deduce bugs

We can ease the process of deducing bugs by building an automated mechanism to remove irrelevant code within proximity of where an error occurs. Such a technique requires substantial compiler and program analysis support (dependence analysis) and is part of a traditional compiler-based program slicing analysis. Recent work within ROSE now supports reverse static slicing.

2.4 Bug pattern analysis

The goal in a bug pattern analysis is to find potential bugs by specifying a "pattern" (*e.g.*, a syntactic template) of code to be identified within the program source of interest, and then searching for instances of the pattern. To operate robustly against the numerous naming and language constructs, such matching should be done directly on the type-checked AST. Full compiler support, with no loss of source information, is required to perform this step robustly and with minimal false positives. Compiler-based tools can be built to accept bug pattern specifications (the development of which is a research issue) using grammar rules (a parser technology) and construct a pattern matching test on the AST of any input application. There are numerous fast search algorithms that could be employed to implement such tests on whole application projects via global analysis and with patterns of quite arbitrary complexity.

Hovemeyer and Pugh observe that bugs are often due simply to misuse of subtle or complex language and library features, and so they propose bug pattern analysis to identify these erroneous uses [5]. Although such features enable general purpose languages to support a broad audience of developers, they may require specialized knowledge to be used correctly. Indeed, individual development groups often develop explicit style guides to control the language feature and library use. Compiler-aided bug pattern analysis could be used to enforce such explicit guidelines, in addition to identifying actual incorrect usage.

2.5 Coverage analysis

The main technique for demonstrating that testing has been thorough is called test coverage analysis [8]. Simply stated, the idea is to create, in some systematic fashion, a large and comprehensive list of tasks and check that each task is covered in the testing phase. Coverage can help in monitoring the quality of testing, assist in creating tests for areas that have not been tested before, and help with forming small yet comprehensive regression suites [9].

Instrumentation, in the source code or the object/byte code, is the foundation on which coverage tools are implemented. Coverage tools for different languages (*e.g.*, Java, C, C++) could have very similar front-ends to display the coverage results, but need different compiler-based tools for instrumentation. We are not aware of any easy-to-use coverage tool addressing the many different types of program coverage for large C++ applications, as are available for other languages such as Java; we are currently developing one based on ROSE.

2.6 Model checking

Model checking is a family of techniques, based on systematic and exhaustive state-space exploration, for verifying program properties. Properties are typically expressed as invariants (predicates) or formulas in a temporal logic. Model checkers are traditionally used to verify models of software expressed in special modeling languages, which are simpler and higher-level than general-purpose programming languages. Manually producing models of software is labor-intensive and error-prone, so a significant amount of research is focused on abstraction techniques for automatically or semi-automatically producing such models. We review related work along these lines in Section 5.

To simplify the models, and fight the state space explosion, abstractions are used. The abstractions used today are performed after the model has been transformed into a net-list. New types of abstractions based on program transformation techniques have been suggested [10]. In this way, program transformation techniques can have a greater role in getting model checking to work with larger programs.

2.7 Code reviews

Code reviews commonly use check lists to help reviewers express and address common problems.[1] For example, an item on the check list might be, “Should an array declaration containing a numerical value be replaced by a constant?” or perhaps, “If an if-statement contains an assignment, should the assignment actually be an equality comparison?” Though effective in practice, it can be very tedious to work with check lists and compare them against the source code to find where a given item is relevant. A better approach, suggested by Farchi [11], is to embed the review questions from the check lists in the code as comments. This requires two phases: one in which to check every item from the check list and see where it is reflected in the code (static analysis), and the second in which to annotate the code with the relevant comment (a simple code transformation). A compiler infrastructure is well-suited to both tasks.

2.8 Capture and replay

Capture and replay tools enable test sessions to be recorded and then replayed. The test sessions might be edited, and then replayed repeatedly or with several test sessions running at the same time. Capture and reply is used in simulating load by helping to create many clones, in regression by automating re-execution of a test, and in debugging and testing of multi-threaded applications by helping to replay the scenario in which the bug appears. Most capture and replay tools enable the test sessions to be edited, parameterized, and generalized. Almost all of the tools

[1] *E.g.*, see: `http://ncmi.bcm.tmc.edu/homes/lpeng/psp/code/checklist.html`

have a facility to compare the expected results from a test run with those that actually occur.

One of the ways to implement capture and replay is by wrapping all the functions through which the applications communicate with the environment. Most commonly, this is done for GUI applications to record all the interactions of widgets. As another example, in scientific applications which use the Message Passing Interface (MPI) communication library, all the MPI calls could be to be recorded during the test execution; in the replay phase, the MPI calls would return the information collected during the testing session. It is expected that such techniques could greatly simplify isolating classes of bugs common to large parallel applications which occur only on tens of thousands of processors. Automatic wrapping of functions is another application for which a compiler framework is very suitable.

2.9 Identifying performance bugs

Of particular interest in high-performance scientific applications are *performance bugs*. These bugs do not represent computation errors, but rather result in unexpected performance degradation. Patterns of some performance bugs in parallel applications include synchronizing communication functions that can be identified from their placement along control paths with debugging code (often as redundant synchronization to simplify debugging) but without any use of their generated outputs. Control dependence analysis can be used to identify such bugs, but this analysis requires a full compiler support. Here the analysis of the AST must use the semantics of the communication library (*e.g.*, MPI, OpenMP) with the semantics of the language constructs to automatically deduce such bugs.

2.10 Aiding in configuration testing

One of the biggest problems in testing commercial code is *configuration testing*, *i.e.*, ensuring that the application will work not only on the test floor, but also at the customer site, which may have another combination of hardware and software. A configuration comprises many components, such as the type and number of processors, the type of communication hardware, the operating system and the compiler; testing all combinations is impractical.

Different compilers or even different versions of a compiler can vary in how they interact with a given application. For example, compilers often differ in what language features are supported, or how a particular feature is implemented when a language standard permits any freedom in the implementation. In addition, the same compiler can generate very different code depending on what compiler flags (*e.g.*, to select optimization levels) are specified. As any tester in the field is aware, the fact that the program worked with one compiler does not guarantee that it will work with the same compiler using a different set of flags, with another version of the same compiler, or with another compiler.

A compiler-based tool (or tools) could aid in compiler configuration testing. For example, it could try different compilers or, for a single compiler, different compile-time flags, and then report on configurations that lead to failures. As another example, in cases where the compiler has freedom in how a language feature is implemented, it could transform the program in different ways corresponding to different implementation decisions, thereby testing the sensitivity of the application to compiler implementation. Such a tool should help make the application more maintainable over time and across customer sites, and reduce time spent debugging compiler-related problems. We discuss specific related examples in Section 4.

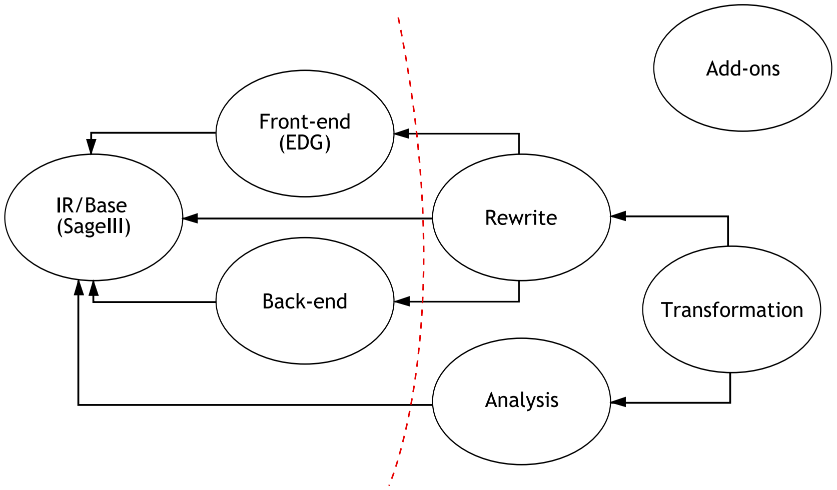

Figure 1: Major components the ROSE compiler infrastructure, and their dependencies

3 The ROSE Compiler Infrastructure

The ROSE Project is a U.S. Department of Energy (DOE) project to develop an open-source compiler infrastructure for optimizing large-scale (on the order of a million lines or more) DOE applications [1, 2]. The ROSE framework enables tool builders who do not necessarily have a compiler background to build their own source-to-source optimizers. The current ROSE infrastructure can process C and C++ applications, and we are extending it to support Fortran90 as part of ongoing collaborations with Rice University. For C and C++, ROSE fully supports all language features, and preserves all source information for use in analysis, and the rewrite system permits arbitrarily complex source-level transformations. Although research in the ROSE project emphasizes performance optimization, ROSE contains many of the components common to any compiler infrastructure, and is thus well-suited to addressing problems in testing and debugging.

The ROSE infrastructure contains several components to build source-to-source translators, shown as ovals in Figure 1. At the core of ROSE is the intermediate representation (IR) of the abstract syntax tree (AST) for C and C++ programs, SageIII, which is based on Sage II and Sage++ [12]. A complete C++ front-end is available that generates the SageIII AST. The AST preserves the high-level C++ language representation so that no information about the structure of the original application (including comments) is lost. A C++ back-end can be used to unparse the AST and generate C++ code. These three components (IR, front-end, and back-end), which all appear to the left of the dashed vertical line, compose the basic ROSE infrastructure.

The user builds a "mid-end" to analyze or transform the AST. ROSE assists mid-end construction by providing a number of mid-end components, shown to the right of the dashed vertical line in Figure 1, including a extensible traversal mechanism based on an attribute grammar (with fixed attribute evaluation for simplicity), AST queries, transformation operators to restructure the AST, and predefined optimizations. ROSE also provides support for library annotations whether they be contained in pragmas, comments, or separate annotation files. Finally, ROSE provides a number of additional packages ("add-ons") for visualizing the AST, building GUI interfaces to ROSE-based tools, etc. The dependencies among all the major components form a directed acyclic graph, shown by arrows in Figure 1. Thus, a tool built using ROSE can use just the subset of the ROSE infrastructure required.

3.1 Front-end

We use the Edison Design Group C and C++ front-end (EDG) [13] to parse C and C++ programs. The EDG front-end generates an AST and performs a full type evaluation of the C++ program. The EDG AST is represented as a C data structure. Within the ROSE front-end, an initial phase translates this C AST into a different object-oriented abstract syntax tree, SageIII, based on Sage II and Sage++ [12]; additional phases in the ROSE front-end do further processing to handle template details, attach comments and C preprocessor directives to AST nodes, and perform AST verification tests. The EDG work is completely encapsulated as a binary which allows its general distribution (as a binary library) with the rest of the ROSE source code. ROSE source including EDG source is available to research groups who obtain the free EDG research license. Alternative language specific front-ends are possible within the ROSE front-end (e.g. Open64 front-end for Fortran90), abstracting the details of using language specific front-ends (e.g. EDG front-end). SageIII is used uniformly as *the* intermediate representation by the rest of the ROSE front-end (downstream of the EDG front-end), the mid-end, and the back-end. Full template support permits all templates to be instantiated, as required, in the AST. The AST passed to the mid-end represents the program and all the header files included by the program. The SageIII IR has approximately 240 types of IR nodes, as required to fully represent the original structure of the application as an AST.

3.2 Mid-end

The mid-end permits the analysis and restructuring of the AST for performance improving program transformations. Results of program analysis are accessible from AST nodes. The AST processing mechanism computes inherited and synthesized attributes on the AST. An AST restructuring operation specifies a location in the AST where code should be inserted, deleted, or replaced. Transformation operators can be built using the AST processing mechanism in combination with AST restructuring operations.

ROSE internally implements a number of forms of procedural and inter-procedural analysis. Much of this work is in current development. ROSE currently includes support for dependence, call graph, and control flow analysis.

To support whole-program analysis, ROSE has additional mechanisms to store analysis results persistently in a database (e.g., SQLite), to store ASTs in binary files, and to merge multiple ASTs from the compilation of different source files into a single AST (without losing project, file and directory structure).

3.3 Back-end

The back-end unparses the AST and generates C++ source code. Either all included (header) files or only source files may be unparsed; this feature is important when transforming user-defined data types, for example, when adding generated methods. Comments are attached to AST nodes (within the ROSE front-end) and unparsed by the back-end. Full template handling is included with any transformed templates output in the generated source code.

3.4 Features relevant to testing tools

Of the major components, the following features are particularly relevant to the design and implementation of ROSE-based testing tools.

Full C++ information in AST. ROSE maintains an AST with all information required to rebuild the original source code, including:

- comments and C Preprocessor (CPP) directive information;
- pragmas;
- all C++ constructs; the SageIII IR consists of 240 different types of nodes;
- templates, including all required instantiations of templates;
- source file positions for every statement, expression, and name;
- full constant expression trees and their constant-folded representations;
- all macro information, including what macros were expanded and where;
- floating point constants represented as both values and strings.

Whole program analysis. Recent supporting work in ROSE permits the whole application (spanning hundreds of source files) to be merged into a single AST held in memory. File I/O, supporting the AST, permits the fast reading and writing of the AST to disk to support analysis tools. This work avoids the recompilation of the application source to build the AST to support analysis or simple queries of the AST by ROSE-based tools.

GUI interface support for rapid tool development. ROSE supports QRose by Gabriel Coutinho at Imperial College, which implements a Qt interface for ROSE. QRose greatly simplifies the development of specialized analysis tools to select or present the AST or source code.

Robustness. ROSE has been tested on large-scale, production-quality, C++ scientific applications on the order of a millions lines of code. These applications make extensive use of C++ language and library features, including templates, the Standard Template Library (STL), and the Boost library, to name a few.

AST visualization tools. ROSE contains a number of graphical and non-graphical tools for visualizing the AST, which is useful when debugging ROSE-based tools or when trying to understand details of program structure.

Attribute-based tree traversals. ROSE has simple interfaces for traversing the AST and propagating context information through inherited and synthesized attributes. These interfaces are consistent among traversal and rewrite systems.

4 Examples of Testing Using ROSE

In this section, we show the interplay between compilers (specifically, ROSE) and testing through a series of specific concrete examples. The first three represent ongoing research and development work on building tools for automated testing and debugging, while the fourth relates a number of anecdotal examples to motivate the need for configuration testing of compilers as discussed in Section 2.10.

4.1 Coverage analysis and testing of multi-threaded code

We are developing tools to address problems of code coverage and automated bug testing of multi-threaded for C and C++ applications, in collaboration with IBM, Haifa. In particular, we are connecting ROSE to the IBM ConTest tools developed originally for Java [7]. Our ROSE-based tool carries out three transformation tasks needed to interface to the ConTest tools:

1. Changing specific function names and their arguments

2. Instrumenting functions and conditions (blocks) for coverage

3. Instrumenting shared variables for coverage

In each case, the ROSE-based tool modifies the AST, and then outputs source code corresponding to the modified AST. Each source-to-source translation phase was supported by less than 50 lines of code and can be used to operate on applications of any size, including whole applications at once using the whole program analysis and transformation mechanisms within ROSE. Such transformations take seconds on hundreds of thousands of lines of code.

4.2 Automatic checking of measurement units

An important source of errors in scientific applications are misuse of units of measurements, such as failing to convert a value in 'miles' to 'meters' before combining it with another value in 'meters' within some computation. Osprey is a type analysis-based tool for soundly and automatically detecting possible measurement unit errors in C programs, developed by Zhendong Su and Lingxiao Jiang at the University of California, Davis [14]. We are collaborating with Su and Jiang to extend this work to C++ programs.

Prior work on Osprey has shown that type inference mechanisms are an effective way to detect such errors. To be most effective, users need to introduce some lightweight annotations, in the form of user-defined type qualifiers, at a few places in the code to identify what variables have certain units. The type inference mechanism propagates this type information throughout the entire program to detect inconsistent usage. In ROSE, the type qualifiers can be expressed as comments or macros, since all comments and source file position information is preserved in the AST. Furthermore, the technique requires significant compiler infrastructure and program analysis support (inter-procedural control flow analysis and alias analysis).

4.3 Symbolic performance prediction

Another application demonstrating the use of ROSE is in the identification of performance bugs. In particular, we compute symbolic counts of program properties within loops and across function boundaries. The properties of interest include basic operation counts, numbers of memory accesses and function calls, and counts of global and point-to-point communication, among others. Our fully automated implementation uses basic traversals of the AST to gather raw symbolic terms, combined with calls to Maple, a popular commercial symbolic equation evaluation tool, to evaluate and to simplify sums and functions of these terms symbolically. Evaluated counts can then be inserted at various places in the AST as attached comments, so that the unparsed code includes comments indicating the symbolic counts. In connecting ROSE to Maple, we can thus make the performance of arbitrarily well-hidden high-level object-oriented abstractions transparent. Doing so permits the inspection of numerous aspects of program and subprogram complexity immediately obvious within general program documentation, desk-checking evaluation, or within the code review process.

4.4 Compiler-configuration testing

The main impact on the cost of a bug, which includes testing and debugging time, is the time between the time the bug was introduced and the time it was found [15]. As a program evolves through many versions, so does the compiler, and a future release may use different compiler or compiler options (*e.g.*, optimization level). In such cases, it is important to check the program under various options to detect if it is sensitive to specific compiler options or versions, and then either remove or carefully document such dependencies.

We encountered this particular problem recently in one of our projects when it stopped working for a client. It turned out that the client moved to a new compiler version that by default performed inlining, which changed location of objects in the heap, and caused our pointers to stop working. Automated "compiler-sensitivity testing" could have greatly simplified debugging. Understanding the root cause in the field was very difficult as it required copying the entire clients environment.

Another potential source of problems is a dependence of an application on a particular compiler's implementation of some language construct. There are a number of examples for C++ [16]:

1. Casts are sometimes required (explicit) and sometimes optional (implicit) with many compilers (C and C++ in particular). It is common for the rules for explicit casts to be relaxed to make it easier to compile older code. Across multiple compilers which casts are required to be explicit vs. implicit can result in non-portable code, and sometime a bug.

2. The order in which static variables are initialized (especially for static objects) is compiler-dependent, and notoriously different for each compiler. Since the language does not specify an order, applications should seek to reduce dependences on a particular order.

3. Infrequently used language features are often a problem, as the level of support for such features can vary greatly across compilers. These features include variadic macros, compound literals, and case ranges.

4. The compiler has some degree of freedom in choosing how fields of a structure or class are aligned, depending on how access-privilege specifies are placed. In particular,

   ```
   class X {
      public: int x;
              int y;
   };
   ```

   is not the same as:

   ```
   class X {
      public: int x;
      public: int y;
   };
   ```

   The first example forces the layout to be sequential in x and y, whereas the second example permits the compiler to reorder the field values. This freedom is not likely implemented in many compilers, but can be an issue in both large scientific and systems applications.

In all of these cases, it can be very difficult to track down the problem if an application somehow depends on a particular implementation. We are developing support within ROSE through bug pattern analysis tools that can help identify such implementation dependencies.

5 Related Work

We review related work in the general area of alternative open compiler infrastructures for C++, and place our current research and development in the context of existing static defect detection tools. We also include a brief discussion of a developer-centric, non-analysis based technique that extends aspect-oriented programming for testing. The following discussion emphasizes source-level tools, and we do not mention related work on tools that operate on binaries, or the important class of dynamic testing tools.

5.1 Open compiler infrastructures for C++

Although there are a number of available open-source compiler infrastructures for C, Fortran, and Java [17, 18], there are relatively few for C++, including g++ [19], OpenC++ [20], MPC++ [21], and Simplicissimus [22]. ELSA is a robust C++ front-end based on the Elkhound GLR parser generator [23], but the IR is best-suited to analysis (and not source-level transformation) tasks. ROSE could use ELSA instead of EDG with appropriate IR translations, either directly or through some external C++ AST format such as IPR/XPR [24]. Like several of these infrastructures, we are developing ROSE to handle realistic large-scale (million lines or more) applications in use throughout the DOE laboratories. We distinguish ROSE by its emphasis on ease-of-use in building compiler-based tools for users who do not necessarily have a formal compiler background.

5.2 Automatic, static defect detection

There is a large and rapidly growing body of work on compile-time automatic software defect detection (*i.e.*, bug detection) systems. This research explores techniques that trade-off soundness (*i.e.*, finding all bugs), completeness (reporting no false positive errors), time and space complexity, and level of user interaction (*i.e.*, the degree to which user is required to provide manual annotations). Below, we provide just a sample of related projects, roughly grouped into four classes: bug pattern detectors, compiler-based program analyzers, model checkers, and formal verifiers based on automated theorem provers. (This categorization is somewhat artificial as many of the tools employ hybrid techniques.) Since no single class of techniques is ideal for all bugs of interest, our aim in ROSE is to provide an extensible, open infrastructure to support external research groups building equivalent tools for C++.

Bug pattern detectors. This class of tools uses a minimal, if any, amount of program analysis and user-supplied annotation. Bug pattern tools are particularly effective in finding errors in language feature and library API usage, but may also support diverse testing activities such as code reviews [11]. The classical example is the C LINT tool [25], which uses only lexical analysis of the source. Recent work on Splint (formerly, LClint) extends the LINT approach with lightweight annotations and basic program analysis for detecting potential security flaws [26]. More recently, Hovemeyer and Pugh have implemented the FINDBUGS framework for finding a variety of bug patterns in Java applications with basic program analysis, observing that many errors are due to a misunderstanding of language or API features [5]. This particular observation certainly applies to C++ applications, where many usage rules are well-documented [27], with some current progress toward automatic identification of STL usage errors (see STLlint, below). C++ in particular has additional challenges since the resolution of overloaded operators requires relatively complex type evaluation which can require more sophisticated compiler support.

Compiler-based static analyzers. Compared to bug pattern detectors, tools in this class use deeper analysis methods (*e.g.*, context-sensitive and flow-sensitive inter-procedural analysis for deadlock detection [28]) to improve analysis soundness while keeping the annotation burden small.

Type checkers constitute an important subclass, since a number of defect detection problems can be mapped to equivalent type analysis problems. The CQual tool uses constraint-based type inference to propagate user-specified type qualifiers to detect Y2K bugs, user- and kernel-space trust errors, correct usage of locks, and format string vulnerabilities, among others, in C programs [29]. CQual analyses are sound, but require some user annotation for best results. Work with Zhendong Su and Lingxiao Jiang are integrating these ideas into ROSE via Osprey, a type qualifier-based system for checking the usage of scientific measurement units [14].

Researchers have used or extended classical program analysis in a variety of ways to create customized, lightweight static analyzers. This body of work includes meta-level compilation (MC[30] and the commercial version that became a basis for Coverity Inc.[31]), symbolic execution techniques as embodied in PREfix [32, 33] and STLlint [34], property simulation as in ESP [35], and reduction of program property checking to boolean satisfiability [36, 37]. Another interesting example in this class of tools is the highly regarded commercial tool, JTest, which combines static and dynamic (*e.g.*, coverage, test execution) analysis techniques [38]. The ROSE infrastructure supports the development of similar tools for C++ by providing an interface to a robust C++ front-end, and we (and others) are extending the available analysis in ROSE through direct implementation as well as interfacing to external analysis tools like OpenAnalysis [39].

Software model checkers. We consider, as members of this class, tools which explore the state-space of some model of the program. These tools include FeaVer for C programs [40], the Java PathFinder [41, 42], Bandera (and Bogor) for Java [43, 44], MOPS for C [45], SLAM for C [46], and BLAST for C [47]. These tools generally employ program analysis or theorem provers to extract a model, and then apply model checkers to verify the desired properties of the model. It is also possible to apply model checking in some sense directly to the source, as suggested in work on VeriSoft [48].

Although model checkers constitute powerful tools for detecting defects, they can be challenging to implement effectively for at least two reasons. First, their effectiveness is limited by how accurately the abstract models represent the source code. Secondly, they may require whole-program analysis. In ROSE, we have tried to address the first issue by maintaining an accurate representation of input program, including all high-level abstractions and their uses, thereby potentially enabling accurate mappings of the C++ source code to models. To address the second, we provide support for whole-program analysis, as outlined in Section 3.

Formal verifiers. Verification tools such as ESC [49], which are based on automated theorem provers, are extremely powerful but typically require users to provide more annotations than with the above approaches. ESC specifically was originally developed for Modula-3 but has been recently extended to apply to Java. ROSE preserves all comments and represents C/C++ pragmas directly, thus providing a way to express these annotations directly in the source.

5.3 Aspect-oriented testing

Aspect-oriented programming (AOP) permits the creation of generic instrumentation [50]. The central idea in AOP is that although the hierarchical modularity mechanisms of object-oriented languages are useful, they are unable to modularize all concerns of interest in complex systems. In the implementation of any complex system, there will be concerns that inherently cross-cut the natural modularity of the rest of the implementation. AOP provides language mechanisms that explicitly capture cross-cutting structures. This makes it possible to program cross-cutting concerns in a modular way, and achieve the usual benefits of improved modularity [51]. The most common use of AOP is to implement logging.

However, because AOP frameworks were designed with simplicity of mind, they do not permit context-dependent instrumentation. For example, one may use aspects to insert instrumentation at the beginning of all methods (possibly of a specific type), but cannot limit the instrumentation based on some attribute of the method. However, AOP has more recently also been shown to be useful in testing [52, 53].

6 Conclusions

The need for an open, extensible compiler infrastructure for testing is motivated simultaneously by (1) the desire to automate or semi-automate the many kinds of activities required for effective testing, (2) the fact that each activity has its own unique analysis and transformation requirements from a compiler infrastructure, and (3) that each testing team will require its own set of customized tools. Our goals in developing ROSE are to facilitate the development of all these kinds of existing and future testing tools by providing a robust, modular, complete, and easy-to-use foundational infrastructure. We are currently pursuing a number of different projects to support testing specifically, as discussed in Section 4. In both our current and future work, we are extending the analysis and transformation capabilities of ROSE, in part by interfacing ROSE with external tools to leverage their capabilities and also working with other research groups.

References

[1] Dan Quinlan, Markus Schordan, Qing Yi, and Andreas Saebjornsen. Classification and utilization of abstractions for optimization. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods*, Paphos, Cyprus, October 2004.

[2] Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *Proc. Joint Modular Languages Conference*, 2003.

[3] Qing Yi and Dan Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, September 2004.

[4] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. In *Proc. Conference on the Future of Software Engineering (International Conference on Software Engineering)*, pages 133–145, Limerick, Ireland, 2000.

[5] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices (Proceedings of Onward! at OOPSLA 2004)*, December 2004.

[6] Dawson Engler and Mandanlal Musuvathi. Static analysis versus software model checking for bug finding. In *Proc.International Conference on Verification, Model Checking, and Abstract Interpretation*, Venice, Italy, 2004.

[7] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Testing multithreaded Java programs. *IBM Systems Journal: Special Issue on Software Testing*, February 2002.

[8] Shmuel Ur and A. Ziv. Off-the-shelf vs. custom made coverage models, which is the one for you? In *Proc. International Conference on Software Testing Analysis and Review*, May 1998.

[9] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[10] Gil Ratsaby, Baruch Sterin, and Shmuel Ur. Improvements in coverability analysis. In *FME*, pages 41–56, 2002.

[11] Eitan Farchi and Bradley R. Harrington. Assisting the code review process using simple pattern recognition. In *Proc. IBM Verification Conference*, Haifa, Israel, November 2005.

[12] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[13] Edison Design Group. EDG front-end. `www.edg.com`.

[14] Lingxiao Jiang and Zhendong Su. Osprey: A practical type system for validating the correctness of measurement units in C programs, 2005. (*submitted*; `wwwcsif.cs.ucdavis.edu/∼jiangl/research.html`).

[15] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report Planning Report 02–3, National Institute of Standards and Technology, May 2002.

[16] Bjarne Stroustrop. *The C++ programming language.* Addison-Wesley, 3rd edition, 2000.

[17] M. S. Lam S. P. Amarasinghe, J. M. Anderson and C. W. Tseng. The suif compiler for scalable parallel machines. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing*, Feb 1995.

[18] G.-A. Silber and A. Darte. The Nestor library: A tool for implementing Fortran source to source transformations. *Lecture Notes in Computer Science*, LNCSD9(1593), 1999.

[19] Free Software Foundation. GNU Compiler Collection, 2005. `gcc.gnu.org`.

[20] Shigeru Chiba. Macro processing in object-oriented languages. In *TOOLS Pacific '98, Technology of Object-Oriented Languages and Systems*, 1998.

[21] Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jorg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and implementation of metalevel architecture in C++—MPC++ approach. In *Proc. Reflection '96 Conference*, April 1996.

[22] Sibylle Schupp, Douglas Gregor, David Musser, and Shin-Ming Liu. Semantic and behavioral library transformations. *Information and Software Technology*, 44(13):797–810, April 2002.

[23] Scott McPeak and George C. Necula. Elkhound: A fast, practical GLR parser generator. In *Proc. Conference on Compiler Construction*, Barcelona, Spain, April 2004.

[24] Bjarne Stroustrop and Gabriel Dos Reis. Supporting SELL for high-performance computing. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, USA, October 2005.

[25] S. C. Johnson. Lint, a C program checker, April 1986.

[26] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–51, Jan 2002.

[27] Scott Meyers. *Effective C++: 50 specific ways to improve your programs and design*. Addison-Wesley, 2nd edition, 1997.

[28] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005.

[29] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.

[30] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[31] Coverity Inc. Coverity source code security tool. `www.coverity.com`.

[32] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice and Experience*, 30:775–802, 2000.

[33] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Warsaw, Poland, April 2003.

[34] Douglas Gregor and Sibylle Schupp. STLlint: Lifting static checking from languages to libraries. *Software: Practice and Experience*, 2005. (*to appear*).

[35] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[36] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI C programs. In *Proc. International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume LNCS 2988, Barcelona, Spain, March 2004.

[37] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proc. Principles of Programming Languages*, Long Beach, CA, USA, January 2005.

[38] Parasoft Corporation. Jtest, 2005. `www.parasoft.com`.

[39] Michelle Mills Strout, John Mellor-Crummey, and Paul D. Hovland. Representation-independent program analysis. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 2005.

[40] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[41] Willem Visser, Klaus Havelund, Guillame Brat, Seung-Joon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2002.

[42] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[43] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, and Robby. Bandera: Extracting finite-state models from Java source code. In *Proc.International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, 2000.

[44] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proc. Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, March 2003.

[45] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proc. Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2004.

[46] Thomas A. Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. Principles of Programming Languages*, January 2002.

[47] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *Proc. 10th SPIN Workshop on Model Checking Software*, volume LNCS 2648, pages 235–239. Springer-Verlag, 2003.

[48] Patrice Godefroid. Software model checking: the VeriSoft approach. Technical Report ITD-03-44189G, Bell Labs, 2003.

[49] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report SRC-159, Compaq Systems Research Center, December 18 1998.

[50] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[52] Daniel Hughes and Philip Greenwood. Aspect testing framework. In *Proceedings of the Formal Methods for Open Object-based Distributed Systems and Distributed Applications and Interoperable Systems Student Workshop*, Paris, France, November 2003.

[53] Shady Copty and Shmuel Ur. Multi-threaded testing with AOP is easy, and it finds bugs! In *Euro-Par*, pages 740–749, 2005.