Techniques for Specifying Bug Patterns

Dan Quinlan, Richard Vuduc
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
7000 East Avenue, Livermore, CA USA
{dquinlan,richie}@llnl.gov

Ghassan Misherghi
Department of Computer Science
University of California, Davis
Davis, CA USA
ghassanm@cs.ucdavis.edu

ABSTRACT

We present our on-going work to develop techniques for specifying source code signatures of bug patterns. Specifically, we discuss two approaches. The first approach directly analyzes a program in the intermediate representation (IR) of the ROSE compiler infrastructure using ROSE's API. The second analyzes the program using the BDDBDDB system of Lam, Whaley, *et al.*. In this approach, we store the IR produced by ROSE as a relational database, express patterns as declarative inference rules on relations in the language Datalog, and BDDBDDB implements the Datalog programs using binary decision diagram (BDD) techniques. Both approaches readily apply to large-scale applications, since ROSE provides full type analysis, control flow, and other available analysis information. In this paper, we primarily consider bug patterns expressed with respect to the structure of the source code or the control flow, or both. More complex techniques to specify patterns that are functions of data flow properties may be addressed by either of the above approaches, but are not directly treated here.

Our BDDBDDB-based work includes explicit support for expressing patterns on the use of the Message Passing Interface (MPI) in parallel distributed memory programs. We show examples of this on-going work as well.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming—*parallel programming, distributed programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*distributed debugging, testing tools, tracing*

1. INTRODUCTION

Modern large-scale parallel applications in scientific computing, which often consist of a million or more lines of code and must run on hundreds of thousands of processors, are largely still written in serial languages, such as C, C++, and Fortran, and parallelized using complex library abstractions, such as the Message Passing Interface (MPI). The complexity of these codes, as well as the difficulty of testing and debugging on full-scale runs, demand automated mechanisms to address software quality concerns. These trends drive much of our general interest in automated static bug detection and our specific interest in bug pattern specification.

We are actively pursuing this work within ROSE, an open and extensible source-to-source compiler infrastructure for developing a wide variety of customized analysis, transformation, and optimization tools [23, 26, 24]. ROSE currently processes million-plus line C and C++ programs, with Fortran 2003 support in progress. The main intermediate program representation in ROSE is an abstract syntax tree (AST) that preserves the detailed structure of the input source (including source file position and comment information), thereby enabling source-based tool builders to accurately analyze and transform programs. In this paper, we use ROSE as a basis for exploring bug pattern specification.

Specifically, we report on two approaches, so that they may be contrasted. The first uses a direct pattern search specified on the AST, written using the interfaces in ROSE's IR, called SageIII. The second uses a declarative language, Datalog, to query a database of relations built from ROSE's AST. The database stores basic structural facts about the program, and the Datalog specification expresses a pattern (*i.e.*, a program analysis) as inference rules on those facts. We process the Datalog queries are processed using the BDDBDDB system [18, 30], which implements the query using binary decision diagram (BDD) techniques. Both mechanisms use the ROSE compiler infrastructure as a basis, and therefore essentially both operate on the same AST. However, they represent the specification of bug patterns differently from one other. In particular, the Datalog specification operates through a single level indirection over the AST.

We describe this work by a series of anecdotal examples. The first is a trivial example of a common bug pattern, namely detecting a switch statement in C or C++ that does not implement a default case (Section 2). This example is taken from SAMATE [22], a catalog of common programming errors. We show how to detect the switch-statement bug both using ROSE directly and using Datalog, explaining each approach in detail. Though our emphasis is not on the performance of these approaches, we present a result from processing a moderately sized 200K line code to make it clear that our work applies to realistic applications.

The second and third examples appear in Section 3, and are two non-trivial tests: searching for bugs related to static constructor initialization in C++, a famous source of porta-

Submitted to PADTAD-V, July 7–9, 2007, London, UK.

Listing 1: The switch-lacks-default bug.

```
1  int
   main () {
       int x = 4, y = 0;

     // switch with default case
6      switch (x) {
           case 0: y = 5; break;
           case 1: y = 3; break;
           case 2: y = 7; break;
           default: y = -1;
11       }

     // switch without default case
       switch (x) {
           case 0: y = 5;
16         case 1: y = 3;
           case 2: y = 7;
       }
       return 0;
     }
```

bility bugs; and testing for null pointer dereferences in C or C++. These are *not* implemented redundantly using both proposed bug pattern specification approaches, but help illustrate interesting aspects of each approach.

The final examples are MPI-specific tests implemented using only Datalog. Correctly using MPI can be daunting [10], owing to the size of the MPI standard and the rich semantics of MPI's abstractions [20, 21]. To build the static checkers for these examples, we implemented a library of Datalog relations that capture MPI-specific abstractions, including data types, constants, and calls. Though still evolving, this library enables simple specification of MPI bug patterns.

2. FINDING SWITCHES THAT LACK DEFAULTS USING THE TWO APPROACHES

In this section, we use the two bug pattern specification approaches to find switch statements in a C or C++ program that do not have a 'default' case. The two approaches are as follows.

1. *Direct search for a pattern in the AST.* The ROSE compiler infrastructure provides an interface to the AST and tools to simplify the use of the AST for analysis. In this approach, we use only ROSE itself to build tools for finding a particular pattern.

2. *Declarative Datalog specification of a pattern in the AST.* We output the AST in the form of binary relations into a database, use Datalog to specify a pattern search (*i.e.*, program analysis) on the relations in the database, and use BDDBDDB to implement the Datalog program. Because the initial database is derived from the AST, the Datalog query program essentially uses an equivalent underlying program representation. Datalog represents a layer of indirection that, in principle, removes the dependence on the ROSE API present in the first approach.

Listing 1 shows an example of a switch statements with and without 'default' cases.

Listing 2: A ROSE-based program to search for switches lacking a default case.

```
#include "rose.h"

class visitorTraversal : public AstSimpleProcessing {
public:
5   virtual void visit (SgNode* n) {
      SgSwitchStatement* s = isSgSwitchStatement (n);
      if (s) {
        SgStatementPtrList& cases =
          s->get_body ()->get_statements ();
10      bool switch_has_default = false;

        // 'default' could be at any position in the list of cases.
        SgStatementPtrList::iterator i = cases.begin();
        while (i != cases.end () && !switch_has_default) {
15        if (isSgDefaultOptionStmt (*i))
            switch_has_default = true;
          ++i;
        }
        if (!switch_has_default)
20        s->get_startOfConstruct ()
            ->display ("Error: switch without default case");
      }
    }
};
25
int main (int argc, char* argv[]) {
  SgProject* project = frontend (argc, argv);

  // Build the traversal object
30 visitorTraversal exampleTraversal;

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

35 return 0;
}
```

2.1 Directly searching the AST

Directly searching the AST in ROSE for this bug is relatively simple, but is ROSE-specific and particularly explicit. Listing 2 is an example a ROSE-based program that implements a search for switches that lack a default case. Line 27 creates the AST, and the `exampleTraversal` object traverses all nodes of the AST on line 33. The `visit()` method defined at line 5 checks if a given node is a switch (lines 6–7), and if so, searches through the list of its cases for a default (lines 8–18). This program works on any size AST, including a whole program AST as supported in ROSE [25].

We ran Listing 2 on a 200 KLOC file taken from the ROSE compiler itself.[1] This file is automatically generated when building ROSE. Rather surprisingly, our checker discovered two violations (bugs). These bugs had existed in spite of previously having always compiled ROSE with all possible warnings enabled. Section 3 presents performance data for this direct AST handling using both this test and the one detailed in that section.

2.2 Specifying the bug using Datalog

Listing 4 shows the same test as Listing 2, but implemented using Datalog. Datalog is a declarative language for deductive databases, and is a subset of Prolog with better-

[1] ROSE_build/src/frontend/SageIII/Cxx_Grammar.C

Listing 3: A simple Datalog program to find switches lacking a default case. This example handles most instances of the bug.

```
switch_with_default (s:node)
switch_with_default (s) :- \
    switchS (s, _, _), \
    defaultS (d, _), \
    parent (s, d).

switch_without_default (s:node)
switch_without_default (s) :- \
    switchS (s, _, _), \
    !switch_with_default (s).

switch_without_default (S)?
```

defined termination within its specification. Statements in Datalog are declarative inference rules; these rules are converted into equivalent queries on a relational database matches to the bug patterns are reported as violations. There is a significant freedom as to how to express patterns in Datalog, in this case the code could be just that shown in Listing 3. To simplify the description we will explain the simpler case first and then say why the actual test is required to be more complex.

Listing 3 defines two rules. The first rule, on lines 1–5, specifies a rule `switch_with_default(s)` for a statement `s`. This rule evaluates to true on `s` if all of the following specific tests pass:

- (Line 3) `s` is a switch statement. This particular binary relation, `switchS(s,c,b)`, is precomputed from the AST and stored in a database as 'true' for all `s`, `c`, and `b`, such that `s` is a switch statement on condition `c` and with a body (list of cases) `b`. Here, `c` and `b` are specified as the special operator, _, which means "don't care" (at least 1).
- (Line 4) `d` is a default statement, also a precomputed binary relation.
- (Line 5) `d` is a parent of `s`; specifically it is in the structural chain of parents as defined recursively from `d` to the root of the AST.

The second rule is that `switch_without_default(s)` is a match if (all must pass):

- (Line 9) `s` is a switch statement.
- (Line 10) `switch_with_default(s)` (the first rule) is **not** a match.

Lastly, `switch_without_default(S)?` on line 12 is the query for all true instances of the rule.

These two rules taken together define the pattern which can be used to search the database of relations built from the AST. This technique is distinctly different from the direct search of the AST from the previous bug pattern specification technique; though with exactly same result. We have found this approach to be simpler than the direct pattern evaluation on the AST, though it takes a while for the use of Datalog to become natural.

This completes the simple test for a switch without a default except for the corner case of a nested switch statement

Listing 4: Handling nested switches. We extend Listing 3 to handle nested switch statements.

```
parent_noswitch( p:node, c:node )
parent_noswitch( p, c ) :- \
    !switchS( p, *, * ), \
    !switchS( c, *, * ), \
    se(p), se(c), \
    parent( p, c ).

anc_noswitch( a:node, c:node )
anc_noswitch( a, c ) :- parent_noswitch( a, c ).
anc_noswitch( a, c ) :- anc_noswitch( a, b ),
    parent_noswitch( b, c ).

switch_with_default( s:node )
switch_with_default( s ) :- \
    switchS( s, _, _ ), \
    defaultS( d, _ ), \
    parent( s, c ), \
    anc_noswitch( c, d ).

switch_without_default( s:node )
switch_without_default( s ) :- \
    switchS( s, _, _ ), \
    !switch_with_default( s ).

switch_without_default( S )?
```

with a default inside of the switch statement *not* having a default. To address this complexity, we augment Listing 3 with additional rules to avoid such a false negative as would otherwise be generated (because a default would be found but not matched properly to the correct switch). The complete, and a bit more complex, example is shown in Listing 4. However, Listing 4 could be made to be as simple as Listing 3 with additional work in, for example, defining the initial database of binary relations. Just as in most programming languages there are a lot of ways to implement the same idea in Datalog.

What is *not* shown are the numerous boolean relations that are computed to support this query and which are specific to the AST generated for any target application. The relations are general and numerous, and not specific to this query. All relations are computed as binary relations from an automated analysis of the AST. It is not clear how many relations are required for general queries, and our experience has been that while numerous queries can be built using a standard set of precomputed relations from the AST, a number of more complex Datalog queries have driven us to revisit what relations are required and add new precomputed relations. The performance of the Datalog queries are also dependent upon the number of relations computed and in ways that are rarely obvious due to the nature of the BDDs how they are combined. While simple and declarative, Datalog has caused us to frequently reevaluate what standard relations should be computed. We expect this to settle down over time as the approach and its use in ROSE matures. For these reasons, it is still difficult to evaluate this technique.

3. MORE BUG PATTERN EXAMPLES

This section describes more interesting examples of bug patterns and their specification in the two approaches. We implement two specific tests. The first tests C++-specific

Listing 5: Detecting static constructor initialization. (Boiler-plate traversal code omitted.)

```
void Traversal::visit (SgNode* n) {
  SgVariableDeclaration* v = isSgVariableDeclaration (n);
  if (v) {
    // For each variable 'i' in declaration 'v'...
    SgInitializedNamePtrList::iterator i =
      v->get_variables ().begin ();
    while (i != v->get_variables ().end ()) {
      SgInitializedName* name = *i;
      // Check for a class type (strip typedefs).
      SgType* type = name->get_type ();
      SgClassType *class_type = isSgClassType (type->strip ());
      if (class_type) {
        // Check for a global variable or a static class member.
        SgScopeStatement* scope = v->get_scope ();
        if (isSgGlobal (scope)
            || (isSgClassDefinition (scope)
                && v->get_declarationModifier()
                    .get_storageModifier()
                    .isStatic()))
          print_position (v);
      }
      ++i; // Next variable in declaration...
    }
  }
}
```

Listing 6: Null pointer dereferences.

```
int main () {
  int* x_pointer;
  struct { int a; } *y_pointer;
  int (*function_ptr) ();

  *x_pointer = 7; // Normal deref through null
  y_pointer->a = 42; // Member access through null
  function_ptr (); // Function call through null

  return 0;
}
```

bug that effects the portability of applications between compilers. The second tests null pointer dereferences, which are common in Java, C, and C++ applications.

3.1 Directly using the AST to detection of static constructor initialization

Listing 5 demonstrates the search for static data members of a class type. Since the order of static initialization is compiler dependent, the use of static data members leads to constructors being called in different and unpredictable orders and often causes many subtle bugs in large scale applications used across different compilers. It is the experience in large scale numerical codes that they are worth while to eliminate, or at least clearly locate their use. However, they are difficult to locate in large scale codes because they are not part of the explicit control flow (the compiler calls them before executing `main`, so they are called implicitly) and have type names that are the same as existing types causing declarations of them to appear as normal data member declarations making them difficult to find using regular expressions. CPP macro handling can also make them more difficult to identify. For these reasons such static data member initialization is particularly difficult to locate and eliminate; worse yet then can continuously creep back into the application code unexpectedly. We present the specification of this pattern using the direct AST approach, this pattern is part of infrequent use on a number of large scale C++ applications at LLNL.

We ran Listing 5 and the simpler switch-lacks-default test from Section 2.1, Listing 2, on a 200K line single C++ file (one of the larger automatically generated files from ROSE itself). The performance of the compilation and test was just under 60 seconds, and the time of both tests were 1.5 seconds each. The compilation of both ROSE and the bug pattern test codes were unoptimized, and included internal debugging, but the performance of the traversal over the two million IR node AST was at a rate of 135K lines of code per second. Ultimately numerous tests will be required and traversals for each test may be impractical in the context of performing hundreds of tests. We expect significant efficiencies will come from combining tests so that the AST can be traversed in memory less frequently. Our demonstration of such simple tests on such a large single file is intended to reflect the size of AST that is formed from the whole program analysis capabilities within ROSE, which merges the ASTs from multiple files in a large project to form a single AST held in memory, with a size of 400 megabytes per million lines of code [25]. Performance results of the much newer Datalog based tests are however currently unavailable.

3.2 Datalog example

This example code demonstrates a test for the dereferencing of a null pointer. Any use of the pointer is considered a pointer dereference (e.g. normal variable dereference, function call from pointer to function, data member access from pointer to struct, etc.). A pointer is assumed to be null if, after its declaration, it has not be explicitly initialized at some point in the control flow upstream of its use. Listing 6 shows an example of a few types of null pointer dereferences.

Checking for null pointer dereferences is a large research area and we do not presume to handle all cases. Recent work has shown a number types of null pointer dereference bug just in Java [15, 16], and there are likely even more within C and C++. Here, we present a null dereference checker to show an example of combined analysis of the AST structure and the program's control flow.

Listing 7 shows a Datalog program to specify a test for a number of types of null pointer dereferences. It is a bug pattern specification on the control flow and the AST; a corresponding example expressed using the AST directly would have been significantly more complex and has not been attempted. To support this test, additional relations were generated to build a more sophisticated database of precomputed relations for the Datalog queries. We believe that a large set of binary relations will at some point become clear and that these will become sufficient to express a wide range of Datalog queries without requiring additional relations to be defined as was required in this case. The current release of ROSE includes the internal tools for defining additional relations.

4. DETECTING BUGS IN MPI USAGE

We have implemented a library of Datalog relations, pre-

Listing 7: Datalog to check for null pointer dereferences.

```
.include common

# 'n' is a potential null dereference
null_deref( n:node )

# Expression 'e' is a pointer
ptr_exp( e:node )

# Expression 'e' may be null
maybe_null_e( e:node )

# Variable 'v' may be null at target node 't'
maybe_null_var( v:node, t:node )

# Path from 's' to 't' without 'v' in a conditional
cfg_path_nocheck( s:node, si:number,
    t:node, ti:number, v:node )

# Symbol 'v' is used in node 'c'
symbol_used( v:node, c:node )

# === Define the above rules ===
ptr_exp( e ) :- expType( e, t ), ptrT( t, _ ).

maybe_null_e( e ) :- ptr_exp(e), callE( e, _ ).
maybe_null_e( e ) :- maybe_null_e(s), castE(e,s).
maybe_null_e( e ) :-
    ptr_exp(e), varE(e,v), maybe_null_var(v,e).
maybe_null_e( e ) :-
    ptr_exp(e), addE(e,l,_), maybe_null_e(l).
maybe_null_e( e ) :-
    ptr_exp(e), anyAssignE(e,_,r), maybe_null_e(r).

maybe_null_var( v, t ) :- \
    anyAssignE(t,l,r), varE(l,v), maybe_null_e(r).
maybe_null_var( v, t ) :- \
    maybe_null_var( v, s ), \
    cfg_path_nocheck( s, 0, t, 0, v).

cfg_path_nocheck( s, si, t, ti, _ ) :- s=t, si=ti.
cfg_path_nocheck( s, si, t, ti, _ ) :-
   cfgNext( s, si, t, ti ).
cfg_path_nocheck( s, si, t, ti, v ) :- \
    cfgNext( s, si, m, mi ), \
    !ifS(m,*,*,*), \
    cfg_path_nocheck( m, mi, t, ti, v ).
cfg_path_nocheck( s, si, t, ti, v ) :- \
    cfgNext( s, si, m, mi ), \
    ifS(m,c,_,_), \
    !symbol_used( v, c ), \
    cfg_path_nocheck( m, mi, t, ti, v ).

symbol_used( v, c ) :- \
    varE( e, v ), \
    anc( c, e ).

null_deref( n ) :-
    maybe_null_e(e), ptrDerefE( m, e ), parent( n, m ).
null_deref( n ) :-
    maybe_null_e(e), arraySubscriptE( m, e, _ ), parent( n, m ).
null_deref( n ) :-
    maybe_null_e(e), arrowE( m, e, _ ), parent( n, m ).

null_deref( N )?
```

Listing 8: Mismatched buffer types in an MPI call.

```
#include <mpi.h>

void send_bufs (int* ibuf, char* cbuf, int d, int n) {
    int p; // My rank
    int np; // Total no. of procs.
    MPI_Comm_rank (MPI_COMM_WORLD, &p);
    MPI_Comm_size (MPI_COMM_WORLD, &np);

  // Send int buf to left neighbor (ok).
    MPI_Send (ibuf + d, n, MPI_INT,
              (p+np-1) % np, 1001, MPI_COMM_WORLD);

  // Send char buf to right neighbor (error).
    MPI_Send (cbuf + d, n, MPI_INT,
              (p+1) % np, 1002, MPI_COMM_WORLD);
 }
```

computed from the AST, to support statically checking MPI usage. These relations represent MPI data types, constants, and calls. In this section, we discuss two examples of simple MPI checkers, one that checks MPI buffer types (a structural AST test), and another that extends the null-dereference example for MPI buffers.

Listing 8 shows an example of an MPI buffer type mismatch error. The C-binding of `MPI_Send` is:

```
int MPI_Send (void* buf, int count,
              MPI_Datatype buf_elem_type,
              int dest, int tag, MPI_Comm comm);
```

where `buf_elem_type` specifies the type of each element of `buf`, passed to `MPI_Send` through a `void` pointer. Listing 8, line 14, sends a `char` buffer but incorrectly specifies the type as `MPI_INT` instead of `MPI_CHAR`.

The Datalog program in Listing 9 catches this bug. We provide precomputed relations for MPI data types, such as `mpiInt(t)`, which indicates a use of the constant, `MPI_INT`. Moreover, we provide relations for MPI calls and their arguments. For instance, the relation, `mpiCall(c)`, shown on line 15, is true if the function call `c` corresponds to any MPI call; alternatively, we can test for specific calls using relations such as, `mpiSend(c)`, `mpiIsend(c)`, `mpiRecv(c)`, and so on. We can "extract" parameters to calls, such as the buffer argument (`mpiArgBuf`) or the buffer element type argument (`mpiArgBufT`). These relations can easily be derived from more basic AST relations; however, providing a library of higher-level relations that map directly is convenient and suggests the overall extensibility of the Datalog approach.

As a second example, we can easily extend the null dereference checkers in Listing 7 with a single additional inference rule to check for null MPI buffers:

```
null_deref (n) :- maybe_null_e (e), \
  mpiCall (n), mpiArgBuf (e, n).
```

We are currently building checkers for a variety of MPI usage errors, including the use of uninitialized buffers, non-blocking operations without matching waits, and barriers and other collectives not called on all paths, among others.

5. RELATED WORK

Many commercial static analysis tools [2, 1, 4, 3], with at least 30 companies active in this area. Most are based on

Listing 9: Datalog to check MPI buffer types.

```
.include common

# Primitive C type 'p' does NOT match the MPI type 't'.
mpi_type_mismatch (p:node, t:node)
mpi_type_mismatch (p, t) :- !intT (p), mpiInt (t).
mpi_type_mismatch (p, t) :- !floatT (p), mpiFloat (t).
# ... etc. ...

# MPI call 'c' has a buffer type mismatch.
# E.g.,:  char* buf = ...;
#          ...
#         MPI_Send (buf, count, MPI_INT, dest, tag, comm);
mpi_buftype_mismatch (c:node)
mpi_buftype_mismatch (c) :- \
  mpiCall (c), \
  mpiArgBuf (b, c), expType (b, s), ptrT (s, p), \
  mpiArgBufT (t, c), \
  mpi_type_mismatch (p, t).

mpi_buftype_mismatch (C)?
```

a parser technology and marginally handle the type analysis required to support C++ fully, for instance, when overloaded functions must be resolved using the complex type evaluation rules in C++. Coverity and GrammaTech products are the only analysis tools we know of that are based on a full compiler front-end technology, with both using the Edison Design Group front-end (EDG) internally, just as ROSE does. Neither Coverity nor GrammaTech, to our knowledge, make the internal AST fully available for user-defined queries, though both permit limited degrees of bug pattern specification. The details of these interfaces are only available to customers and unavailable to us currently. Both of these products and others are quite widely used and our work does not compete with such commercial products. In contrast, our focus has been on the expression of user-defined bug patterns.

The BDDBDDB project has used their own work for both general program analysis (including an impressive pointer analysis demonstrated on a million lines of Java code) and recognition of general patterns in Java code [18, 30]. The Java specific work is not made available, where as BDDBDDB is an open source project. Because we use BDDBDDB, our work is related to theirs, though we are also keenly interested in C and C++.

Recognition of language specific source code patterns places special constraints on a compiler infrastructure, and few handle these details. Such a compiler infrastructure must save everything about the structure of the source code, and thus must handle language-specific details. In general, most compiler infrastructures target binary executables and so ignore and lose much of the source information. The internal ASTs from such compilers infrastructure include normalization or lowering that lose fundamental information about the original source code and these can cause false positives or false negatives when matching against user specified bug patterns. Both SUIF [8] and Open64 [5] are substantial and respected open compiler infrastructures, but have intermediate representations which lose high level C++ constructs. OpenC++ [6] is an older compiler infrastructure (specific to C++), but lacks support for templates; as substantial limitation for use on modern C++ applications. Pivot [7] is a newer compiler infrastructure specifically for C++ and focused on source to source, it used EDG and is being developed by Stroustrup and others at Texas A&M to support experimental language research on C++0x (the next C++ standard) and a wide range of other work. Pivot is the closest compiler infrastructure to ROSE, that we know of, both in philosophy and design, both also use the same EDG front-end and share some internal collaboration, they have a high level IR design, even though the goals for each are subtly different and this results in numerous subtle issues being handled differently.

One of our goals is to support abstractions like MPI relevant to large-scale high-performance scientific computing. Indeed, a few such tools exist already. Several focus on dynamic error detection, including MPI-CHECK [19], Umpire [28], MARMOT [17], the Intel Message Checker [10], and our own work on JitterBug [29].

MPI-CHECK statically locates certain classes of MPI errors [19]. However, many other kinds of errors require deeper program analysis and detailed knowledge of the semantics of MPI to find many other kinds of MPI errors statically. For example, the control-flow of typical MPI programs depends on the unique rank of the process; this information could be used to help match calls, such as sends and receives, barriers or other collectives. Conversely, we could find errors due to improper or non-existing call matchings. Dependence analysis could trace the flow of data that passes through MPI, and thereby check for common buffer errors in MPI programs, such as buffer overruns, reading from a receive buffer before a non-blocking receive completes, and using unitialized buffers, among others [10]. Other analysis and model checking approaches could be used to verify temporal usage properties (*e.g.*, non-blocking sends followed by waits), similar to recent work for I/O, operating system kernel, and threading library abstractions [11, 9].

MPI-SPIN uses powerful model checking techniques to verify MPI programs [27], and has been applied to finding actual bugs in a widely-used textbook on MPI [12]. The examples we consider in this paper check lighter-weight program properties. However, our general work in ROSE is synergistic with the MPI-SPIN or other model checking efforts in the sense that we can provide the accurate representations of the source code used to drive and derive input models for the model checkers.

Other existing tools pattern-based tools could be extended to support MPI as well, including those proposed by Farchi, *et al.*, in the context of code reviews [13], and frameworks like FindBugs for general programs [14]. FindBugs is particularly aligned with the goals of the examples of our paper, with the focus of our paper on the mechanics of specifying the patterns themselves.

6. CONCLUSIONS AND FUTURE WORK

The specification of source code patterns form a basis for how numerous sorts of bugs can be identified in real code and how users can be expected to tailor domain-specific bug finders for their applications. The complexity of the process is critical to allowing users a mechanism to simply specify their own custom bug patterns.

This paper discusses two approaches: first, a direct access of the AST using ROSE dependent mechanisms, data structures, traversals; and secondly, a using Datalog forming a compiler infrastructure independent approach using

precomputed relations taken from the AST. The experience with the direct use of the AST, has been that it can be tedious and requires a moderate understanding the the AST interfaces to implement. In contrast, the use of Datalog results in a significantly simpler declarative specification, but in a language that may be unfamiliar to many users. Importantly, for there to be standards for bug patterns we seek a compiler infrastructure independent technique and this is an additional attractiveness of the Datalog approach. Clearly this would require a standard set of binary relations to be defined. Furthermore we believe that the bug pattern specifications may be more useful than for just driving the search for bugs in source code. For example, using the specifications to drive in introduction of bugs in arbitrary codes (bug seeding) may permit the automated evaluation the effectiveness of commercial and open source bug finding tools.

The Datalog approach is attractive due to its simple declarative nature, but Datalog is a foreign language to most users, so it is not clear how attractive it would be for end-user bug pattern specification. Our use of Datalog for building MPI checkers relies on additional precomputed relations, and we expect libraries of relations to grow over time. Toward that end, the ROSE-BDDBDDB work includes specialized tools for building relations from AST traversals, though these are not discussed in this paper.

The work on BDDBDDB includes the development of a specification language, PQL, based on the concrete syntax of Java, and it is likely that this approach would work well for C, C++, and other languages.

We mention anecdotal performance results in Section 3 for the direct approach on a moderately-sized (200 KLOC, 2 million IR node) example. In future work, we will carry out much more extensive performance comparisons of the direct AST handling to the Datalog representation (these result were unavailable for this paper).

Looking forward, we plan to extend our work and experiments not just to source pattern analysis, but also to binaries. Recent work in ROSE to include binary analysis, specifically the disassembled instruction sequence representation of a binary in an AST form will permit these identical techniques to be applied to pattern matching of instructions on the binary. Significant forms of binary analysis consist of the identification and synthesis of subtle patterns of instructions; these approaches may be significant in this future work.

7. REFERENCES

[1] Coverity - source code analysis, http://www.coverity.com.

[2] Fortify - source code analysis, http://www.fortifysoftware.com.

[3] Grammatech - source code analysis, http://www.grammatech.com.

[4] Klockwork - source code analysis, http://www.klockwork.com.

[5] Open64, http://www.open64.net.

[6] Openc++, http://www.csg.is.titech.ac.jp/ chiba/openc++.html.

[7] Pivot, http://www-unix.mcs.anl.gov/workshops/dslopt/talks/dosreis.pdf.

[8] Suif, http://suif.stanford.edu.

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

[10] J. DeSouza, B. Kuhn, and B. R. de Supinski. Automated, scalable debugging of MPI programs with the Intel Message Checker. In *Proc. 2nd Intl. Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, MO, USA, May 2005.

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

[12] M. S. et. al. *MPI—The Complete Reference*. MIT Press, 1996.

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

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

[15] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of Program Analysis for Software Tools and Engineering (PASTE05)*, 2005.

[16] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, New York, NY, USA, 2005. ACM Press.

[17] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI analysis and checking tool. In *Proc. Parallel Computing: Software Technology, Algorithms, Architectures, and Applications*, pages 493–500. Elsevier, 2004.

[18] M. Lam, J. Whaley, V. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries, 2005.

[19] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A tool for checking Fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15:93–100, 2003.

[20] Message Passing Interface Forum (MPIF). MPI: A Message-Passing Interface Standard. Technical Report, University of Tennessee, Knoxville, June 1995. http://www.mpi-forum.org/.

[21] Message Passing Interface Forum (MPIF). MPI-2: Extensions to the Message Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[22] NIST. Samate - software assurance metrics and tool evaluation, http://samate.nist.gov/index.php.

[23] D. Quinlan. Rose: Compiler support for object-oriented frameworks. In *Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France*, volume 10 of *Parallel Processing Letters*, pages ??–?? Springer Verlag, 2000.

[24] D. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In H. G. Dietz, editor, *Languages and Compilers for Parallel Computing, 14th International Workshop, LCPC 2001, Revised Papers*, volume 2624 of *Lecture Notes in Computer Science*, pages 570–578. Springer Verlag, 2003.

[25] D. Quinlan, R. Vuduc, T. Panas, J. Härdtlein, and A. Sæbjørnsen. Support for whole-program analysis and verification of the One-Definition Rule in C++. In *Proc. Static Analysis Summit*, Gaithersburg, MD, USA, June 2006. National Institute of Standards and Technology Special Publication.

[26] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *JMLC'03: Joint Modular Languages Conference*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer Verlag, Aug. 2003.

[27] S. F. Siegel. Model checking nonblocking MPI programs. In

Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI), Nice, France, January 2007.

[28] J. S. Vetter and B. R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proc. Supercomputing*. ACM/IEEE, 2000.

[29] R. Vuduc, M. Schulz, D. Quinlan, and B. de Supinski. Improving distributed memory applications testing by message perturbation. In *Proc. Int'l Symp. on Software Testing and Analysis (ISSTA), 4th Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD-IV)*, Portland, ME, USA, July 2006.

[30] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams, 2004.