Techniques for Specifying Bug Patterns

Daniel J. Quinlan, Richard W. 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 Datalog-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 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.3.4 [**Programming Languages**]: Processors—*compilers, debuggers*

Keywords

bug patterns, static analysis, Datalog, Message Passing Interface

General Terms

Languages, Verification

PADTAD'07, July 9, 2007, London, England, United Kingdom.

1. INTRODUCTION

Modern large-scale parallel applications in scientific computing may consist of a million or more lines of code and must run on hundreds of thousands of processors; yet, these codes 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 how to express bug patterns.

In this paper, we use ROSE, an open and extensible source-to-source compiler infrastructure, as a basis for exploring the problem bug pattern specification. ROSE is designed to enable source-based tool builders to develop a wide variety of customized analysis, transformation, and optimization tools [26, 29, 27]. ROSE currently processes C and C++ programs of a million or more lines of code, with support for Fortran 2003 in progress. The main intermediate representation (IR) in ROSE is an abstract syntax tree (AST) that preserves the detailed structure of the input source, including source file position and comment information. The AST's design enables source-based tool builders to accurately analyze and transform programs.

We report on two approaches to bug pattern specification, 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 using the BDDBDDB system [20, 34], which implements the query using binary decision diagram (BDD) techniques. The two approaches operate on the same AST, but enable different ways of specifying the bug patterns. In particular, the Datalog approach is declarative rather than imperative, and moreover, it is IR-neutral in principle even though we use the ROSE AST to populate the initial database here.

We present our 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, a catalog of common programming errors [25]. 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++, an infamous source of portability 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 [11], owing to the size of the MPI standard and the rich semantics of MPI's abstractions [23, 24]. 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 a user to specify MPI bug patterns in a way we believe is simple and natural, and leverages the extensibility and conciseness of the Datalog approach.

2. SWITCHES WITHOUT DEFAULTS

In this section, we use the two bug pattern specification approaches to find switch statements in a C or C++ program that does not have a 'default' case, as shown in Listing 1. 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 AST analysis. In this approach, we use only ROSE itself to build tools that find particular patterns.

2. *Declarative Datalog specification of a pattern in the AST.* We output the AST as relations in a database, use Datalog to specify a pattern search (*i.e.*, program analysis) on those relations. We use BDDBDDB to implement the Datalog program. The initial database is derived from the AST, so that the Datalog query program essentially uses the same program representation as the direct-search approach. In principle, the use of Datalog removes the dependence on the ROSE API present in the first approach.

2.1 Directly Searching the AST

Directly searching the AST in ROSE for this bug is relatively simple, but requires learning and using ROSE. Nevertheless, the interfaces in ROSE are designed to be easy to understand and use.

Listing 2 is an example of 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 29. The `exampleTraversal` object is an instance of the `visitorTraversal` class, which extends one of ROSE's traversal interfaces, the `AstSimpleProcessing` class. Here, we override the `visit()` method defined at line 5 to check whether a given node is a switch (lines 6–7), and if so, search through the list of its cases for a default (lines 8–18). This program works on any size AST, including a whole-program AST available in ROSE [28].

Listing 1: The switch-lacks-default bug.

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

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

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

We ran Listing 2 on a 200 KLOC file taken from the ROSE compiler itself. This file is automatically generated when building ROSE. To our surprise, our checker discovered two violations, which were indeed 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

Datalog is a declarative language for deductive databases, and is a subset of Prolog. Statements in Datalog are declarative inference rules and queries on an underlying relational database. We illustrate how to express bug patterns in terms of Datalog rules and queries that operate on a relational database generated from ROSE's AST.

Listing 3 shows the Datalog implementation of the direct-search checker in Listing 2 that finds switches lacking a default case. Listing 3 defines three rules (lines 1–14), and specifies one query (line 17).

The query (ending with `?`) asks for all values of the variable `S` that make the predicate `switch_without_default(S)` true. Our system is set up so that every value corresponds to a node in the source program's AST (*e.g.*, a switch statement, or `SgSwitchStatement` node); thus, asking for all values is equivalent to asking for all corresponding AST nodes.

The rule defined in lines 12–14 specifies that the predicate `switch_without_default(s)` will be true for `s` if both `switchS(s,c,b)` and `!switch_with_default(s)` hold. Here, `switchS(s,c,b)` is a fundamental predicate precomputed directly from the AST to be true for all values of `s`, `c`, and `b` (*i.e.*, all corresponding AST nodes) such that `s` is a switch statement and `c` and `b` are its condition and body, respectively. In this rule, we do not care about the particular condition and body, and so write the special value '_' to indicate a "don't care" (*i.e.*, "there exists"). The predicate, `switch_with_default(s)` is, as its name implies, a predicate that should be true for all `s` that are switch statements *with* a default case; the `!` operator means 'not.'

The rule defining the predicate `switch_with_default(s)` appears in lines 7–9 of Listing 3. Intuitively, we set this

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

```
#include "rose.h"

class visitorTraversal : public AstSimpleProcessing {
public:
  virtual void visit (SgNode* n) {
    SgSwitchStatement* s = isSgSwitchStatement (n);
    if (s) {
      SgStatementPtrList& cases =
        s->get_body ()->get_statements ();
      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) {
        if (isSgDefaultOptionStmt (*i))
          switch_has_default = true;
        ++i;
      }
      if (!switch_has_default)
        s->get_startOfConstruct ()
         ->display ("Error: switch without default case");
    }
  }
};

int main (int argc, char* argv[]) {
  SgProject* project = frontend (argc, argv); // Create AST
  visitorTraversal exampleTraversal;
  exampleTraversal.traverseInputFiles (project, preorder);
  return 0;
}
```

Listing 3: Datalog-based analogue of Listing 2.

```
# A block 'b' of statements has statement 's'.
block_has_stmt (b:node, s:node)
block_has_stmt (b, s) :- first_inB (s, b).
block_has_stmt (b, s) :- next_inB (s, _, b).

# Switch 's' has a default case.
switch_with_default (s:node)
switch_with_default (s) :- \
    switchS (s, _, b), defaultS (d, _), block_has_stmt (b, d).

# Switch 's' does not have a default case.
switch_without_default (s:node)
switch_without_default (s) :- \
    switchS (s, _, _), !switch_with_default (s).

# Query:
switch_without_default (S)?
```

predicate to be true for all switch statements `s` such that the body `b` contains a statement `d` that is a default case. This rule is the analogue of the direct-search code in lines 8–18 of Listing 2: the predicate `defaultS(d,_)` is precomputed from the AST to be true for all values `d` corresponding to the `SgDefaultOptionStmt` AST node, and the value `b` maps to the value of the AST node returned by `SgSwitchStatement::get_body()` when called on the corresponding node represented by the value of `s`. This rule shows how the direct-search and Datalog-based approaches relate.

Finally, lines 2–4 define `block_has_stmt(b,s)`, which is true if *either* of the two sub-rules (lines 3 and 4, respectively) is true. Here, `first_inB(s,b)` is true if the statement `s` is the first statement in block `b`; `next_inB(s,p,b)` is true if `s` is the statement in `b` following the statement `p` ("previous") that also appears in `b`. That is, the collection of `first_inB(_,b)` and `last_inB(_,_,b)` define the sequence (list) of statements that constitutes the block `b`. The body of the switch statement is represented by just such a list, as shown in lines 8–9 of Listing 2.

Taken together, the rules define the pattern which can be used to search the database of relations built from the AST. This technique is distinctly different from the ROSE-only direct-search of the AST, though with exactly the same results. We have Datalog to be simpler to code than direct pattern evaluation on the AST, though it takes a while for the use of Datalog to become natural.

One issue with the Datalog approach is the extent to which one should precompute basic relations, like `switchS` and `defaultS`, from automated AST analysis. Our implementation currently outputs a large number of basic relations, not only for the AST nodes, but also for other constructs such as the control-flow graph and for domain-specific libraries such as MPI. We do not yet understand how the performance of the Datalog queries depends on the number of relations, which is not obvious due to the nature of the underlying BDD technology. We are still evaluating these issues, and we expect more definitive answers as our experience with Datalog and BDDBDDB matures.

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 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 ROSE-based Static Constructor Checker

Listing 4 shows a ROSE-based static checker that finds static data members of a class type. The key point of this example is that we can, with relative ease, extract information about the structure and types of the input program through the ROSE IR. We have applied this checker to large-scale C++ applications at Lawrence Livermore National Laboratory, and removing instances of these members has eliminated portability bugs.

The use of such static members can lead to subtle bugs when porting to a new platform and/or compiler, because the order of initialization (*i.e.*, the order in which constructors are called) depends on the compiler. Developers are typically unaware of these issues for several reasons. First, static constructor initialization is not part of the explicit control flow—the compiler generates these calls, which "happen" before executing `main`. Secondly, these members have type names that are the same as existing types; thus, declarations of them 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. Moreover, static data member initialization may easily be re-introduced into the application code as it evolves.

We ran Listing 4 and the simpler switch-lacks-default test in Listing 2 on a 200K line single C++ file taken from ROSE

Listing 4: 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 5: Null pointer dereferences.

```
1  typedef struct {int a; char b;} my_type;
2  typedef int (*func_type) (void);
3  extern func_type get_func_ptr (void);
4
5  void foo (bool cond) {
6    int* xp;
7    my_type* yp;
8    int x;
9    func_type fp;
10
11   xp = new int;
12   yp = new my_type;
13   fp = get_func_ptr ();
14
15   x = *xp; // ERROR: xp may be null
16   if (cond) {
17     if (yp)
18       yp->b = 'a'; // OK: Guarded access
19     yp->a = 42; // ERROR: yp may be null
20   }
21   fp (); // ERROR: fp may be null
22 }
```

itself. Results for this particular file suggest the performance properties we might expect from checkers when run on fully merged whole-program ASTs, which in ROSE's IR consume roughly 400 MB per million lines of code [28]. 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 separate traversals for each test may be impractical when performing hundreds of tests, so we are actively designing efficient mechanisms to fuse separate traversals that preserve the simplicity of the existing traversal interfaces for tool builders. Performance results of the Datalog-based tests are however currently unavailable.

3.2 Datalog-based Null Dereference Checker

Next, we present a test for the dereferencing of a null pointer in Datalog. Our primary intent in this example is to show the relative ease with which one can express a combined analysis of the AST structure and the program's control flow in Datalog, given the appropriate initial relations.

We consider several possible uses of a pointer to be dereferences, including normal variable dereference, function call from pointer to function, and data member access from pointer to a `struct`. We also consider a relatively simple check: a pointer value may be null if it is the return value of a function call, a call to the `new` memory allocation operator, and is not guarded by an `if`-condition. Listing 5 shows several examples (lines 15, 19, and 21), including a guarded dereference which should be OK (line 18).

Checking for null pointer dereferences is a research problem and we do not presume to handle all cases. For instance, our checker looks for pointer assignment but does not reason about the value of the assignment; thus, if a pointer is assigned to an expression that evaluates to the constant 0 and later dereference, the checker would not flag it as a null dereference. Indeed, there has been interesting recent work on checking null dereferences generally in Java [17, 18], and we expect these ideas to extend to C and C++.

Listing 6 shows a Datalog program that checks for null pointer dereferences. This example relies on both AST and control-flow graph information; we summarize the relevant precomputed predicates in Table 1 and we highlight interesting aspects of this example below. We did not attempt to build a corresponding implementation using only ROSE, but know it would have been significantly more complex.

Line 1 of Listing 6 shows an include directive, which is the basic mechanism for defining libraries of relations. Indeed, the relations defined in this example could have been stored in a separate file to be re-used, customized, or even extended in other Datalog-based checkers. We revisit this technique when discussing our MPI checkers in Section 4.

Lines 4–23 declare various predicates. These forward declarations permit subsequent definitions to be mutually recursive, as is the case with `maybe_null_e` and `maybe_null_var` (*e.g.*, see lines 34 and 42).

Rules may be recursive. A simple example appears in the definition of `is_ptrT(t)` in lines 26–27. The effect of this rule is to strip away any typedefs when checking whether `t` is a pointer type. For example, consider the following code, which contains a cast expression:

```
typedef int* T1;
typedef T1 T2;
typedef T2 T3;
... (T3)x ...
```

Since the ROSE AST accurately preserves type information as it appears in the source, the type of the cast expression will be `T3`; if the Datalog variable `t` is the type of the cast expression, then `is_ptrT(t)` will be true.

The definitions of `maybe_null_e` and `maybe_null_var` consider various ways in which an expression or variable, re-

Listing 6: Datalog-based null dereference checker.

```
.include common

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

# The type 't' is a pointer type.
is_ptrT (t: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 ===
is_ptrT (t) :- ptrT (t, _).
is_ptrT (t) :- typedefT (t, b), is_ptrT (b).

ptr_exp (e) :- expType (e, t), is_ptrT (t).

maybe_null_e (e) :- newE (e).
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).
cfg_path_nocheck (s, si, t, ti, v) :- \
  cfgNext (s, si, m, _), \
  ifS (m, c, _, _), symbol_used (v, c), \
  cfgNext (_, _, s, si), cfgNext (s, 2, t, ti), si = 1.

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)?
```

Predicate	Meaning
addE(a,l,r)	The expression a adds l and r.
anc(a,e)	Node a is an ancestor in the AST of node e.
anyAssignE(a,l,r)	The expression a operator-assigns r to l, *i.e.*, a may be a plain assignment (=), add-assign (+ =), multiply-assign (∗ =), and so on.
arraySubscriptE (a,b,i)	Expression a is an array subscript reference of base pointer b and index i.
arrowE(a,b,f)	Expression a is an arrow-dereference expression of base pointer b and field f, *i.e.*, $b \rightarrow f$.
callE(c,f)	c is a function call to function reference expression f.
castE(c,e)	c is a casts expression e to another type.
cfgNext(s,s_i,t,t_i)	There is a control-flow graph edge from node s at point s_i to node t at point t_i. The "points" distinguish, *e.g.*, entry and exit of a node.
expType(e,t)	Expression e evaluates to type t.
ifS(i,c,t,f)	Statement i is an if-statement with condition c, true-branch body t and false-branch body f.
newE(n)	Expression n is a call to the `new` operator.
parent(p,c)	Node p is a parent in the AST of node c.
ptrDerefE(m,e)	Expression m dereferences the pointer expression e.
ptrT(t,b)	The type t is a pointer to base type b.
typedefT(t,b)	The type t is a typedef whose base type is b.
varE(v,s)	v references the variable whose symbol is s.

Table 1: Precomputed AST predicates, used in the null dereference example of Listing 6.

Listing 7: 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);
  }
```

Listing 8: Datalog to check MPI buffer types.

```
.include common
.include libmpi.datalog

# 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), \
  mpiArgDatatype (t, c), \
  mpi_type_mismatch (p, t).

mpi_buftype_mismatch (C)?
```

spectively, may be null. For example, the `new` operator may return null (line 31), as might any other function call that returns a pointer (line 32). These rules propagate possible null values through various kinds of expressions, such as assignment and addition (lines 37 and 39, respectively).

The `cfg_path_nocheck(s,si,t,ti,v)` predicates incorporates control-flow information to detect when a dereference is guarded by a conditional that refers to the variable, to account for cases such as lines 17–18 of Listing 5. However, note that the treatment of conditionals is very simplistic: Listing 6 checks only that a condition refers to the subsequently dereferenced variable (*i.e.*, the `symbol_used` predicate), but does not attempt to evaluate the condition. Thus, had the condition in line 17 of Listing 5 been `!yp` rather than `yp`, line 18 would not be flagged as an error (*i.e.*, false negative). Nevertheless, this example is intended only to give a flavor of what a null dereference test might look like and how AST and control information could be used.

4. DETECTING BUGS IN MPI USAGE

We are particularly interested in building checkers for applications that use MPI. Toward this end, we have implemented a library of Datalog relations, precomputed 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. The first is a structural AST test that checks MPI buffer types, illustrating the basics of our MPI checking framework. The second checker extends the null-dereference example of Section 3.2 for MPI buffers, showing how easy it is to extend a Datalog-based checking library. We have tested these checkers on the IS (integer sort) benchmark, a small 1100 line C program that is part of the NAS Parallel Benchmarks suite [7] with errors introduced artifically.

4.1 Buffer Type Mismatches

Listing 7 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 datatype, int dest, int tag,
  MPI_Comm comm);
```

where `datatype` specifies the type of each element of `buf`, passed to `MPI_Send` through a `void` pointer. Listing 7, line 14, sends a `char` buffer but incorrectly specifies the type as `MPI_INT` instead of `MPI_CHAR`.

The Datalog program in Listing 8 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 refer to the parameters passed to such calls, such as the buffer argument (`mpiArgBuf`) or the buffer element type argument (`mpiArgDatatype`). These relations can in many cases 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.

Listing 9: Extending the null-dereference checker for MPI buffers.

```
.include common
.include null_deref.datalog # Import Listing 6
.include libmpi.datalog

# Extend a rule from Listing 6
null_deref (n) :- maybe_null_e (e), \
  mpiCall (n), mpiArgBuf (e, n).
```

4.2 Null MPI Buffers

We can easily extend the null dereference checkers in Listing 6 with a single additional inference rule to check for null MPI buffers, as shown in Listing 9. This listing uses the Datalog include directive to load existing rules, and then extends the null dereference rule accordingly (lines 6–7).

We are currently building checkers for a variety of other 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. We hope a domain-specific library of basic relations will encourage user-community contributions.

4.3 Building Domain-specific Relations

In general, deciding what relations to precompute and implementing the code to precompute them from the AST could be a time-consuming process. In our ROSE-Datalog implementation, we are building tools to help ease this process. For instance, we have a high-level Python-based scripting system for generating customized traversals to compute relations, as well as a utility to help build a domain-specific API from the header files and example source programs. We used initial implementations of these tools to create our MPI relation generator, and are extending these tools to handle additional constructs for other APIs. We will continue to extend and evaluate this in future work.

5. RELATED WORK

There are many commercial static analysis tools [2, 1, 4, 3], with at least 30 companies active in this area. Most are based on a parser technology and marginally handle the type analysis required to fully support C++, 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.

Recent work on static checking specifically for C++ includes OINK [35], the Orion system [10], and STLlint [15]. OINK, based on the robust Elsa C++ front-end [22], emphasizes analysis rather than source-level transformation tasks. Orion extends gcc, and so is also not focused on source-to-source transformations. STLlint is a static checker specifically for Standard Template Library (STL) usage, and is a good example of the utility of domain-specific checkers.

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 [20, 34]. The Java specific work is not made available, whereas 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 requires a compiler infrastructure that can save everything about the structure of the source code, and thus such a compiler 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, which typically perform some normalization, can cause false positives or false negatives when matching against user-specified bug patterns. Both SUIF [6] and Open64 [5] are substantial and respected open compiler infrastructures, but have intermediate representations which lose high level C++ constructs. OpenC++ [9] is an alternative open compiler infrastructure for C++, but lacks support for templates, which is a substantial limitation for modern C++ applications. Pivot [31] is a recent compiler infrastructure specifically for C++ and focused on source-to-source transformation being developed by Bjarne Stroupstrup and others at Texas A&M. A major goal of Pivot is 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, and in its use of the commercial EDG front-end. Like ROSE, Pivot has 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 [21], Umpire [32], MARMOT [19], the Intel Message Checker [11], and our own work on JitterBug [33].

MPI-CHECK statically locates certain classes of MPI errors [21]. However, many other kinds of errors require deeper program analysis and detailed knowledge of the semantics of MPI. 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 [11]. 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 [12, 8].

MPI-SPIN uses powerful model checking techniques to verify MPI programs [30], and has been applied to finding actual bugs in a widely-used textbook on MPI [13]. 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 pattern-based tools could be extended to support MPI as well, including those proposed by Farchi, *et al.*, in the context of code reviews [14], and frameworks like FindBugs for general programs [16]. FindBugs inspires our work, though in this paper we focus on on the mechanics of specifying the patterns themselves.

6. CONCLUSIONS AND FUTURE WORK

There is a recognized need for static checking systems that users can customize and extend for their particular applications or APIs. The promise of a simple bug pattern specification system is to enable the customization process for a wide variety of users who may not necessarily be experts in compilers or static analysis. In the parallel setting, libraries like MPI constitute important target domains for custom analyses that off-the-shelf checking tools might not provide.

This paper discusses two approaches to bug pattern specification: first, a direct access of the AST using ROSE dependent mechanisms, data structures, traversals; and secondly, a Datalog-based, compiler-independent approach to reason about the same AST. Our experience with the direct use of the AST is that it can be tedious for complex examples, 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, which suggests approaches like the use of Datalog. Our use of Datalog for building MPI checkers is an example of how one might use a Datalog-like system to define libraries of simple static checkers. Indeed, we expect libraries of relations to be developed and extended by the broader user-community over time.

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 the introduction of bugs in arbitrary codes (*i.e.*, *bug seeding*) may permit the automated evaluation the effectiveness of commercial and open source bug finding tools.

The original 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 results 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.

Acknowledgements

This work was partially produced at the University of California, Lawrence Livermore National Laboratory (UC LLNL) under contract no. W-7405-ENG-48 between the U.S. Department of Energy (DOE) and The Regents of the University of California (University) for the operation of UC LLNL.

We thank the anonymous referees for their thoughtful comments.

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

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[8] 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.

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

[10] D. Dams and K. Namjoshi. Orion: High-precision methods for static error analysis of C and C++ programs. Technical Report ITD-04-45263Z, Bell Labs, April 2004.

[11] 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.

[12] 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.

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

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

[15] D. Gregor and S. Schupp. STLlint: Lifting static checking from languages to libraries. *Software: Practice and Experience*, 2005.

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

[17] 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.

[18] 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.

[19] 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.

[20] M. Lam, J. Whaley, V. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proc. ACM Symposium on Principles of Database Systems*, pages 1–12, 2005.

[21] 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.

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

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

[24] 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.

[25] NIST. SAMATE - Software Assurance Metrics And Tool Evaluation, http://samate.nist.gov/index.php.

[26] 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*. Springer Verlag, 2000.

[27] 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.

[28] 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.

[29] 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.

[30] S. F. Siegel. Model checking nonblocking MPI programs. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Nice, France, January 2007.

[31] B. Stroustrop and G. D. Reis. Supporting SELL for high-performance computing. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, NY, USA, October 2005.

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

[33] 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.

[34] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. In *Proc. Programming Language Design and Implementation (PLDI)*, 2004.

[35] D. Wilkerson. OINK: A collection of composable C++ static analysis tools, 2005. http://freshmeat.net/projects/oink.