Differential Static Analysis: Opportunities, Applications, and Challenges

Shuvendu K. Lahiri
Microsoft Research
Redmond, USA
shuvendu@microsoft.com

Kapil Vaswani
Microsoft Research
Banglalore, India
kapilv@microsoft.com

C. A. R. Hoare
Microsoft Research
Cambridge, UK
thoare@microsoft.com

ABSTRACT

It is widely believed that program analysis can be more closely targeted to the needs of programmers if the program is accompanied by further redundant documentation. This may include regression test suites, API protocol usage, and code contracts. To this should be added the largest and most redundant text of all: the previous version of the same program. It is the differences between successive versions of a legacy program already in use which occupy most of a programmer's time. Although differential analysis in the form of equivalence checking has been quite successful for hardware designs, it has not received as much attention in the static program analysis community.

This paper briefly summarizes the current state of the art in differential static analysis for software, and suggests a number of promising applications. Although regression test generation has often been thought of as the ultimate goal of differential analysis, we highlight several other applications that can be enabled by differential static analysis. This includes equivalence checking, semantic diffing, differential contract checking, summary validation, invariant discovery and better debugging. We speculate that differential static analysis tools have the potential to be widely deployed on the developer's toolbox despite the fundamental stumbling blocks that limit the adoption of static analysis.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods, programming by contract*

General Terms

Reliability, verification

Keywords

static analysis, differential analysis, equivalence checking, semantic diff, regression testing

1. INTRODUCTION

Software evolves through the introduction of new features, performance optimizations, bug fixes and refactoring. For many large software modules that evolve over several years, the original developers are long gone. A developer or a tester for such a large legacy module has to rely on previous versions of the program and regression tests to understand the program. After each modification to such a codebase, most of a developers time is spent today in pondering on the following questions:

- Does the change add the desired functionality (in the case of a feature addition)?
- Does the change (not) add any undesirable behavior?
- Do the regression tests cover the interesting additional behaviors?
- What particular change was the cause of a regression failure?

Writing and running regression tests (tests that reveal an observable difference across a source change) is possibly the only option today to answer these questions, but it has several drawbacks.

1. It requires integration of the module into the overall system which can be executed — this can be fairly time consuming when the overall system is an operating system such as Windows or Linux.
2. The oracles for regression testing are the few runtime assertions (e.g. null dereferences, exceptions) in the code — this may not serve as a good indicator for detecting all desirable or undesirable behaviors.
3. Finally, the coverage is limited to the scenarios captured in the regression tests — in most cases they only cover a small fraction of behaviors of the program.

Static analysis involves analyzing the source code of a module at compile time to find or ensure the lack of a certain class of defects. The attractiveness of static analysis comes from two main factors:

- *Controllability*: it allows identifying defects without having to perform system integration and running system wide tests,
- *Coverage*: it can often provide a high coverage for the lack of a certain kind of defect at runtime.

These tools range from simple checkers for certain programming errors (e.g. certain naming convention for identifiers) to highly sophisticated verifiers that can ensure a specification.

In spite of the great progress made by static analysis, the adoption of static analysis based tools in the development life-cycle has been limited. An average developer only uses the syntax-directed, flow-insensitive type-checkers, or checkers for specific runtime errors (buffer overruns). This can be attributed to several reasons including but not limited to (a) need for property specifications, (b) intermediate contracts required to prove the assertions, (c) modeling environment, and (d) fundamental imprecision in static analyzers for reasoning about complex invariants. These factors result in a lot of false alarms from the use of a static analyzer which consumes valuable developer's time.

We believe static analysis aimed at exploiting the differential behavior of two versions of a program (broadly termed as differential static analysis) enjoys the following characteristics:

1. it is a more *restricted* problem than checking arbitrary contracts statically on a program,

2. if the complexity of an analysis depends on the *magnitude of change*, it can be applied cost-effectively to large examples without sacrificing precision [10, 4],

3. as we demonstrate in the next section, there are promising applications that can improve the productivity of a developer or tester, and

4. there are important challenges that need to be solved that make it an interesting research problem.

2. APPLICATIONS AND CHALLENGES

In this section, we highlight various applications that can be enabled by static differential reasoning. Some of these applications have received attention already, others are more speculative. Although regression test generation can be seen as the ultimate goal, we show that there are intermediate goals that can provide value. For each of the applications, we mention some existing works and envision the challenges that remain.

2.1 Equivalence checking

Equivalence checking for combinational circuit implementations has been one of the widely adopted tools in hardware design industry, and many commercial tools exist today (e.g. Formality [2]). Just as the advances in Boolean Satisfiability (SAT) and (Ordered) Binary Decision Diagrams [1] accounted for the growth of these equivalence checkers in hardware, we envision recent advances in Satisfiability Modulo Theories (SMT) [13] and better formulation of efficient program logics [9] to provide a solid basis in the context of software. Unlike equivalence checking of Boolean circuits, checking equivalence for software will need to account for dynamic memory allocation, loops, recursion and other rich constructs present in modern programming languages.

In *regression verification* [4], the (partial) equivalence checking problem of two closely related versions of a program is performed by checking (partial) equivalence of two versions of the individual procedures modularly using SMT solvers. They use *uninterpreted functions* as abstractions to summarize procedures in the two versions that are partially equivalent. This leads to efficient checking when the two versions are similar. This can be useful when the changes involve *refactoring* or performance optimizations, to ensure that there are no observable behavior changes.

The approach above is mostly restricted to scalar programs; the presence of heap with recursive datatypes can make checking equivalence challenging. Similarly, more global transformations such as the example below may require more refined abstractions than uninterpreted functions.

```
void F(...){                  void F'(...){
   ....                          ....
   i = n;                        i = 0;
   while (i > 0)                 while (i < n)
       a[--i] = i;                   a[i] = i++;
 }                            }
```

2.2 Semantic diff

Most software changes involving a bug fix, feature addition, actually modify the behavior of a program. In the case when the set of behaviors actually change, a tool that displays the *effect* of a change can be extremely valuable. Tools such as Windiff merely provide syntactic differences between two versions of a program. However, these tools do not provide any insight about *semantic changes* caused by these syntactic changes.

Jackson and Ladd [6] proposed using the "dependency" between the input and output variables of a procedure as the semantic notion of change. A change in the dependency would indicate that there is a change in behavior; however, it will be hard to ensure the lack of change. Recently, Person et al. [11] have proposed *differential symbolic execution* to summarize the effect of program paths in the two versions and comparing the symbolic summaries for differences using a theorem prover. *Symbolic Diff* [7] uses *differential inlining* to inline the differential paths to propagate the changes interprocedurally; it uses SMT solvers lazily to enumerate intraprocedural paths that may differ. Figure 1 shows the output of the tool, where intraprocedural paths are shown to highlight behavioral differences between two versions of a program. Such an utility could provide valuable feedback to the developer at the time of source checkin into a repository.

In spite of the progress, several challenges need to be addressed. Some of them are:

- propagate changes interprocedurally to the public APIs of a module,

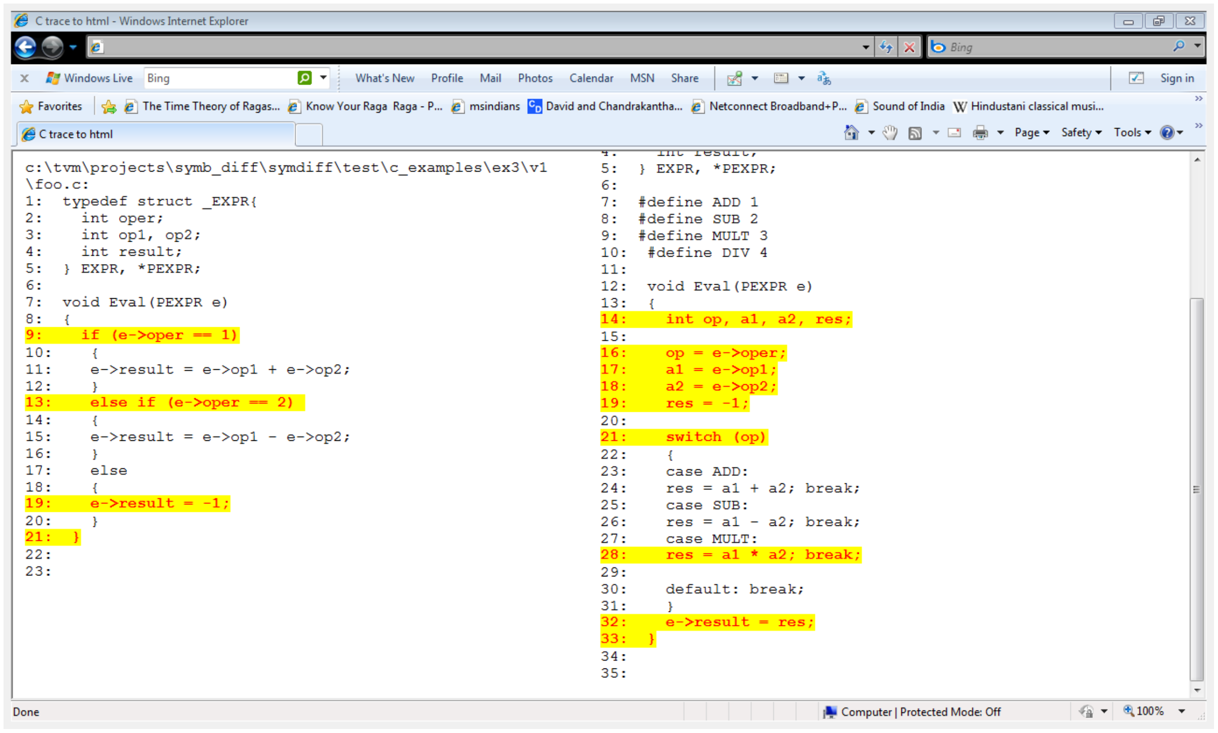
- succinctly represent the changes and the conditions under which the module does not have any behavioral changes,

- localizing the changes in the presence of deep heap updates, and

- allowing the user to specify a vocabulary that limits the space of expected changes [7]. This could be seen as a generalization of the work to identify structural changes in object-oriented programs [8].

2.3 Differential contract checking

Contract checking involves checking various assertions in a program. The contracts are expressed as preconditions,


C trace to html - Windows Internet Explorer
C trace to html
Done
Computer | Protected Mode: Off
100%

```
c:\tvm\projects\symb_diff\symdiff\test\c_examples\ex3\v1
\foo.c:
typedef struct _EXPR{
  int oper;
  int op1, op2;
  int result;
} EXPR, *PEXPR;

void Eval(PEXPR e)
{
  if (e->oper == 1)
   {
   e->result = e->op1 + e->op2;
   }
   else if (e->oper == 2)
   {
   e->result = e->op1 - e->op2;
   }
   else
   {
   e->result = -1;
   }
}
```

```
  int result;
} EXPR, *PEXPR;

#define ADD 1
#define SUB 2
#define MULT 3
 #define DIV 4

 void Eval(PEXPR e)
 {
   int op, a1, a2, res;

   op = e->oper;
   a1 = e->op1;
   a2 = e->op2;
   res = -1;

   switch (op)
   {
   case ADD:
   res = a1 + a2; break;
   case SUB:
   res = a1 - a2; break;
   case MULT:
   res = a1 * a2; break;

   default: break;
   }
   e->result = res;
 }
```

Figure 1: Output of the symbolic diff tool [7]. The highlighted statements indicate a path in the two versions of a program, where the procedure has different side effects.

postconditions, loop invariants for procedures and object invariants. Dynamic contract checking helps add additional runtime assertions to check, which can provide more observability at runtime. However, checking user-defined contracts statically is not cost-effective beyond a specific class of properties (buffer overruns, locking protocols). Although great progress has been in mechanizing the proofs of contracts on large modules, proving these contracts can have very high overhead. The overhead could be in terms of writing intermediate contracts, or worse still writing proofs scripts within expressive provers.

Let us define the problem of *differential contract checking* as follows:

> Given two versions P and P' of a program with a set of contracts, is there an input such that a contract C evaluates differently in the two versions?

The motivation of the problem lies in the following belief: although most program changes will affect the behavior of a program, the changes should at least not affect the contracts. We believe that this problem can be handled precisely in spite of imprecision in the underlying static analysis. Although the analysis may be imprecise for a given input, it may still be able to say whether the contract evaluates differently for this input for the two versions.

```
void F(int x){              void F'(...){
   A *r = G(x, &flag);         A *r = G(x, &flag);
   if (*flag)                  if (*flag)
      r->f = 5;                   r->f = 5;
   if (r)                      //if (r)
      r->h = 10;                  r->h = 10;
 }                           }
```

For example, the two version of F (F and F') dereference the fields f and h. Proving the absence of all null-dereferences in each version require capturing the postcondition of the procedure G; the absence of which may lead to false alarms. However, one can guarantee that the dereference of r->f is safe in F' if and only if it was safe in F as the procedure G has the same effect on both versions. A differential contract checker will only report that the dereference of r->h is suspicious in the later version. We believe that this will provide more confidence than pure dynamic contract checking for richer contracts, with low false alarms.

2.4 Specification documentation

Internal specifications of modules can serve as valuable documentation for codebases undergoing frequent changes in the hand of multiple developers. However, there is no recipe for discovering specifications, and the ones that are important to document. For example, certain invariants may be needed as intermediate assertions to prove a given specification. This requires the presence of enough specifications for the code base, which is unreasonable to expect.

We conjecture that the specifications required to prove equivalence of procedures modularly are an important source of internal invariants that should *at least* be documented. This is because these invariants are not property specific and justify the code changes.

In the following example:

```
void F(...){                  void F'(...){
      ...                           ...
      G(..);                        H'(..);
      H(..);                        G'(..);
}                             }
```

The code change inverts the order of calls to procedures G and H. If the developer does not expect the code to change, he/she should document the fact that the procedure G and H commute. This may be easy to infer if they do not depend or modify the same global, but may be difficult to infer in the presence of deep heap updates inside the procedures.

2.5 Debugging

One of the harsh realities of software development today is that programmers often spend disproportionately large amounts of time debugging code, finding the root cause of failures. While debugging in general is more art than science, the debugging process often involves asking questions such as "what has changed in this library since the last release" and "how does the behavior of the failure inducing input compare across versions of the program". Existing debugging tools provide little help in answering such questions.

Several of these questions can be naturally reduced to the problem of identifying semantic differences between versions of the software artifact. For example, one approach for finding the root cause of a failure is to systematically roll back changes until a change the eliminate the failure is found. Differential analysis can be used to automatically generate new inputs that are similar to a failing input but do not induce a failure [12, 5]. Comparing such inputs with the failing input often provides critical clues to the cause of failure. Debugging tools can also assist programmers during a debugging session by using differential analysis to find and highlight functions exercised by the failing input that exhibit different behavior since the last version.

In general, a correct version of an application significantly simplifies the debugging problem significantly by serving as a baseline with which to compare program behavior against. Furthermore, tools that employ differential analysis as a debugging aid are likely to be more scalable because they only need to reason about a small number of code fragments that have changed and/or a small number of inputs.

2.6 Regression testing

The problem of *regression test selection*, leveraging differential information to prune regression tests that are not affected by the code change has been widely studied [14]. This is also widely used in industries for testing large modules with several million test cases. However, the problem of generating new regression test cases that will cover the differential behaviors has not being studied extensively; a recent work in this area [15] tries to leverage the change propagation to guide dynamic symbolic simulation [3] for test generation. Although obtaining 100% path coverage is infeasible for large modules, we can expect to provide or generate regression tests that cover 100% of the differential paths for small changes (such as bug fixes). One of the main challenges here is to provide a metric for coverage that ensures that all the differential paths generated by a static tool have been covered.

3. REFERENCES

[1] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[2] Formality. Available at http://www.synopsys.com/Tools/Verification/ FormalEquivalence/Pages/Formality.aspx.

[3] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI '05)*, pages 213–223. ACM, 2005.

[4] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[5] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI*, 2009.

[6] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM*, pages 243–252, 1994.

[7] M. Kawaguchi, S. K. Lahiri, and H. Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.

[8] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[9] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL '08)*, pages 171–182, 2008.

[10] D. Notkin. Longitudinal program analysis. In *PASTE*, page 1. ACM, 2002.

[11] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In *SIGSOFT FSE*, pages 226–237, 2008.

[12] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: an approach for debugging evolving programs. In *ESEC/SIGSOFT FSE*, 2009.

[13] Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.

[14] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA*, pages 97–106, 2002.

[15] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *ICSE Companion*, pages 311–314. IEEE, 2009.