SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs

Shuvendu K. Lahiri[1], Chris Hawblitzel[1], Ming Kawaguchi[2], and Henrique Rebêlo[3]

[1] Microsoft Research, Redmond, WA, USA
[2] University of California, San Diego
[3] Federal University of Pernambuco, Brazil

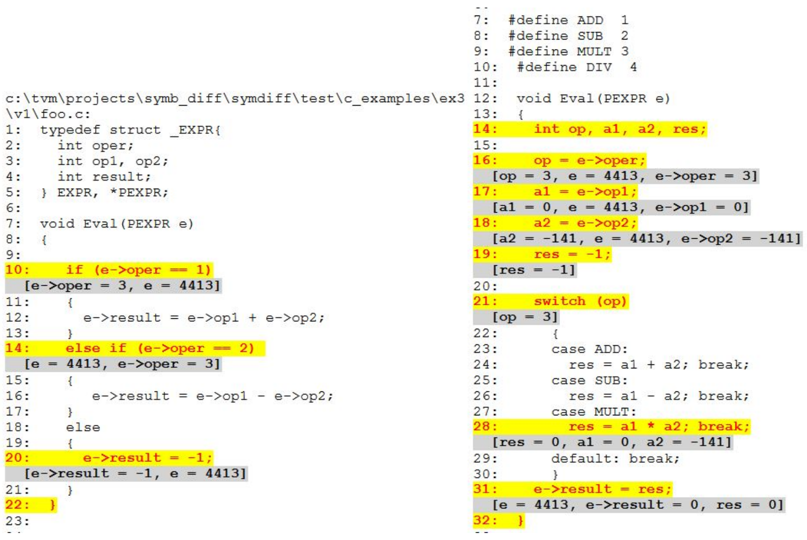
Abstract. In this paper, we describe SymDiff, a language-agnostic tool for equivalence checking and displaying semantic (behavioral) differences over imperative programs. The tool operates on an intermediate verification language Boogie, for which translations exist from various source languages such as C, C# and x86. We discuss the tool and the front-end interface to target various source languages. Finally, we provide a brief description of the front-end for C programs.

1 Introduction

An evolving software module undergoes several changes — refactoring, feature additions and bug fixes. Such changes may introduce compatibility bugs or regression bugs that are detected much later in the life-cycle of the software. There is a need for tools that can aid the developers understand the impact of changes faster. Such tools will complement "syntactic diff" tools such as WinDiff and GNU Diff on one hand, and regression testing based change validation (that provides limited coverage) on the other.

In this paper, we describe the design of SymDiff (**Sym**bolic **Diff**), a semantic differencing tool for imperative programs. Unlike most existing equivalence checking tools for software, the tool operates on an intermediate verification language called Boogie [1] (and hence language-agnostic). This provides a separation of concerns — the core analysis algorithms are independent of source language artifacts (object-orientation, generics, pointer arithmetic etc.) and are therefore reusable across different languages. To perform scalable equivalence checking, we leverage the modular program verifier in Boogie that exploits the Satisfiability Modulo Theories (SMT) solver Z3 [4]. A novel feature of the tool is that it displays *abstract* counterexamples for equivalence proofs by highlighting intra-procedural traces in the two versions (see Figure 1), a semantic extension to differing source lines highlighted by syntactic diff tools.

Because of the language-agnostic nature of the tool, one only needs a translator from the source language (such as C) to Boogie (many of which already exist). We describe the front-end interface required from such translators to target SymDiff. Finally, we briefly describe the implementation of one such front-end for C programs.



```
c:\tvm\projects\symb_diff\symdiff\test\c_examples\ex3
\v1\foo.c:
1:  typedef struct _EXPR{
2:    int oper;
3:    int op1, op2;
4:    int result;
5:  } EXPR, *PEXPR;
6:
7:  void Eval(PEXPR e)
8:  {
9:
10:    if (e->oper == 1)
  [e->oper = 3, e = 4413]
11:    {
12:      e->result = e->op1 + e->op2;
13:    }
14:    else if (e->oper == 2)
  [e = 4413, e->oper = 3]
15:    {
16:       e->result = e->op1 - e->op2;
17:    }
18:    else
19:    {
20:      e->result = -1;
  [e->result = -1, e = 4413]
21:    }
22:  }
23:
```

```
7:  #define ADD  1
8:  #define SUB  2
9:  #define MULT 3
10:  #define DIV  4
11:
12:  void Eval(PEXPR e)
13:  {
14:     int op, a1, a2, res;
15:
16:     op = e->oper;
  [op = 3, e = 4413, e->oper = 3]
17:     a1 = e->op1;
  [a1 = 0, e = 4413, e->op1 = 0]
18:     a2 = e->op2;
  [a2 = -141, e = 4413, e->op2 = -141]
19:     res = -1;
  [res = -1]
20:
21:     switch (op)
  [op = 3]
22:       {
23:       case ADD:
24:         res = a1 + a2; break;
25:       case SUB:
26:         res = a1 - a2; break;
27:       case MULT:
28:         res = a1 * a2; break;
  [res = 0, a1 = 0, a2 = -141]
29:       default: break;
30:       }
31:     e->result = res;
  [e = 4413, e->result = 0, res = 0]
32:  }
```

Fig. 1. Output of SymDiff for displaying semantic differences for C programs. The yellow source lines highlight a path, and the gray lines display values of some program expressions after each statement in the trace.

2 SymDiff

SymDiff operates on programs in an intermediate verification language Boogie [1]. Boogie is an imperative language consisting of assignments, assertions, control-flow and procedure calls. Variables (globals, locals and procedure parameters) and expressions can be either of a scalar type τ or a map type $[\tau']\tau$. We currently restrict SymDiff to the non-polymorphic subset of Boogie. A Boogie program may additionally contain symbolic constants, functions, and axioms over such constants and functions.

SymDiff takes as input two loop-free Boogie programs and a configuration file that *matches* procedures, globals, and constants from the two programs. Loops, if present, can be unrolled up to a user-specified depth, or may be extracted as tail-recursive procedures. The default configuration file matches procedures, parameters, returns and globals with the same name; the user can modify it to specify the appropriate configuration. The tool can (optionally) take a list of procedures that are assumed to be equivalent (e.g. procedures that do not call into any procedures with modifications). For each pair of matched procedures f_1 and f_2, SymDiff checks for *partial equivalence* — terminating executions of f_1 and f_2 under the same input states result in identical output states. The input state of a procedure consists of the value of parameters and globals (hereafter referred to as a single variable gl) on entry, and the output state consists of the value of the globals and returns on exit.

```
procedure Eq.f1.f2(x){
  var gl0;
  gl0 := gl; //copy the globals
  r1 := inline call f1(x);
  gl1 := gl; //store output globals
  gl  := gl0; //restore globals
  r2 := inline call f2(x);
  gl2 := gl; //store output globals
  assert (r1 == r2 && gl1 == gl2);
}
```

Fig. 2. Procedure for checking equivalence of f_1 and f_2

Given two programs P_1 and P_2 and a pairing of procedures over the two programs, the algorithm below checks for equivalence modularly. For each pair of paired procedures f_1 and f_2, we create a new Boogie procedure Eq.f_1.f_2 (Figure 2) that checks partial equivalence of f_1 and f_2. To enable modular checking, the procedure calls inside Eq.f_1.f_2 (i.e. callees of f_1 and f_2) are replaced by uninterpreted functions that update the modified globals and the return; the input to the functions are parameters and the globals that are read by the procedure. The resulting set of procedures $\{\text{Eq}.f_i.f_j \mid f_i \in P_1,\ f_j \in P_2\}$ (one for each matched pair f_i and f_j) are analyzed by the Boogie modular verifier using verification condition generation [1] and SMT solver Z3. We omit details of verification condition generation here; it suffices to know that it transforms a program (a set of annotated procedures) to a single logical formula whose validity implies that the program does not fail. In our case, if *all* the Eq.f_i.f_j procedures are verified, then the matched procedure pairs in P_1 and P_2 are partially equivalent. On the other hand, if the assertion in any Eq.f_i.f_j procedure cannot be proved by Boogie, we extract a set of intraprocedural paths through f_i and f_j and report them to the user. We modified Boogie to produce multiple (up to a user-specified limit) counterexample traces for the same assertion.

In addition to the purely modular approach, SymDiff offers various options for inlining callees (to improve precision at the cost of scalability) for the case of non-recursive programs. There are options for either inlining (a) every callee, (b) only the callees that can't be proved equivalent, or (c) only behaviors in callees that can't be proved equivalent (*differential inlining* [8]). These options require a bottom-up traversal of the call graph of procedures.

3 Interface for Source Languages

In this section, we briefly describe the important considerations for adapting SymDiff for a source imperative language such as C, C#, or x86. First, one needs a translator for the language (say C) that performs two tasks: (i) represents the state of a program (e.g. variables, pointers and the heap) explicitly in terms of scalar and map variables in Boogie, and (ii) translates each statement in the source language to a sequence of statements in Boogie. The precision and soundness of the resulting tool will be parameterized by how faithful the translator is. Many such translators exist today with various precision and soundness trade-offs. For example, HAVOC [3] translates C programs to Boogie; Spec# [2] converts C# programs to Boogie; there have also been translators from binary (x86) programs to Boogie [5].

When two procedures cannot be proven equivalent, SymDiff generates counterexample traces on the source programs (Figure 1). The counterexample contains an intra-procedural trace for each procedure and values of "relevant" program expressions (of scalar type) for each statement. We have found this to be the most useful feature of the tool when applied to real examples. This feature requires two pieces of (optional) additional information in the translated Boogie programs, for *each* source line translated:

– The source file and the line number have to be provided as attributes.
– For each scalar valued program expression e to be displayed in the trace (e.g. `e->oper` in Line 10 of the first program in Figure 1), associate the corresponding expression in Boogie.

Note that this requires only a *one time* change to the translator for the source language to Boogie.

Non-deterministic Statements. In the presence of non-deterministic statements (such as the Boogie statement `havoc x` that scrambles a variable `x`), a procedure may not be equivalent to itself. Source language translators often use non-deterministic statements such as `havoc` to model allocation, effect of I/O methods such as `scanf`, calls to external APIs etc. To use SymDiff effectively, we require that the translators use *deterministic* statements to model such cases. We provide an example of deterministic modeling of allocation for C programs in the next section. SymDiff also models external procedures as deterministic (in their parameters) transformers using uninterpreted functions.

3.1 C Front End

In this section, we briefly describe the implementation of the front-end for C programs. The tool takes two directories (for the two versions) containing a set of .c files and a makefile. We use the HAVOC [3] tool to translate C programs into Boogie programs. HAVOC uses maps to model the heap of the C program [3], where pointer and field dereferences are modeled as select or updates of a map. By default, HAVOC assumes that the input C programs are *field safe* (i.e. different field names cannot alias) and maintains a map per word-valued (scalar or pointer) field and type. For example, the statement `x->f := *(int*)y;` is modeled as `f[x+4] := T_int[y];` using two maps `f` and `T_int` of type `[int]int` (assuming offset of `f` is 4 inside `x`).

We modified HAVOC to incorporate deterministic modeling of allocation (for `malloc` and `free`) and I/O methods (such as `scanf`, `getc`) [8]. Here we sketch the modeling of allocation: we maintain a (ghost) global variable `allocvar`, which can be modified by calls to `malloc` and `free`. `malloc` is modeled as follows (in Boogie) : `malloc(n:int) returns (r:int) {r := allocvar; allocvar := newAlloc(allocvar, n);}`, where `newAlloc` is an uninterpreted function. The specification for `free` is similar. The modeling ensures that two identical sequences of `malloc` and `free` return the same (but arbitrary) sequence of pointers. This suffices for many examples, but can be incomplete in the presence

Table 1. Results on Siemens benchmarks. "Proc" stands for procedures, "Time" is the time taken by SymDiff to analyze all the "Changed" procedures.

Example	#LOC	#Proc	#Versions	#Changed procs (Avg)	Time (sec) (Avg)	# Paths (Avg)	Enum Time (sec) (Avg)
tcas	173	9	42	1.2	0.64	26.72	1.19
print_tokens	727	18	7	1.4	1.30	357.43	2.87
print_tokens2'	569	19	10	1.1	0.90	169.36	1.25
replace	563	21	32	1.1	0.96	20.38	2.11
schedule	412	18	9	1.1	0.94	16.00	1.99
schedule2	373	16	10	1	0.83	10.60	0.99
print_tokens2_n4	569	19	1	1	1.13	7560	56.83
print_tokens2_n6	569	19	1	1	1.30	>10,000	160.63
print_tokens2_e4	569	19	1	1	3.50	4200	24.02
print_tokens2_e6	569	19	1	1	32.96	>10,000	150.61

of different allocation orders. We also added an option to generate the information required to display the counterexample traces. For each statement, we generate any pointer or scalar subexpression in the trace. For example, for the C statement `x->f.g->h = y[i] + z.f;` we add the expressions {`x, x->f.g, x->f.g->h, y[i], z.f`} whose values will be displayed in the trace. For procedure calls, we add the expressions in the arguments and the return. Figure 1 shows the semantic diff as a pair of traces over two programs. The second program performs some *refactoring* and *feature addition* (case for `MULT`). A syntactic diff tool gets confused by the refactoring and offers little idea about the change in behavior.

Table 1 describes an evaluation of SymDiff on a set of medium-sized C programs representing the Siemens benchmarks from the SIR repository [10]. Each program comes with multiple versions representing injection of real and seeded faults. The benchmark `print_tokens2'` represents a slightly altered version of the `print_token2` benchmark, where we change a constant loop iterating 80 times to one over a symbolic constant n. The experiments were performed on a 3GHz Windows 7 machine with 16GB of memory. We used a loop unroll depth of 2 for the examples. The runtime of SymDiff ("Time") does not include the time required to generate Boogie files. The number of intraprocedural paths ("Paths") correspond to the number of feasible paths inside Eq.$f_1.f_2$ that reach the return statement, and "Enum Time" is the time inside Z3 to enumerate them using an ALL-SAT procedure. The first few rows indicate that the tool scales well for finding differences when the number of intraprocedural paths is less than 1000. To investigate the effect of large number of paths, we created two sets of examples `print_token2_n<k>` and `print_tokens2_e<k>` (with loop unrolling of `k`), for semantically different and equivalent procedures respectively. The results indicate that the tool scales better on semantically different procedures, perhaps due to the large number of paths leading to a difference. For the equivalent cases too, the scalability appears to be better than the approach of enumerating paths outside Z3. In addition to these examples, the tool has been successfully applied to C programs several thousand lines large.

4 Conclusion and Related Work

In this paper, we describe the design of a language-agnostic semantic differencing tool for imperative programs. We have currently developed a front-end for C programs. We have also built a preliminary front-end for x86 programs that we have applied to perform compiler validation. We are also developing a front-end for .NET programs using a variation of the Spec# tool chain. We are currently working on making a binary release of the tool along with the C front end at `http://research.microsoft.com/projects/symdiff/`.

Related Work. There have been a few recent static tools for performing semantic diff for programs. Jackson and Ladd [7] use dependencies between input and the output variables of a procedure — it does not use any theorem provers. The approach of regression verification [6] uses SMT solvers to check equivalence of C programs in the presence of mutual recursion, without requiring all procedures to be equivalent. This is the work closest to ours[1], and the main difference lies in the language agnostic nature of our tool, generation of abstract counterexamples, and the modeling of the heap. Differential symbolic execution [9] uses symbolic execution to enumerate paths to check for equivalence. Our preliminary experience shows that the use of verification conditions instead of path enumeration often helps SymDiff scale to procedures with several thousand intraprocedural paths.

References

1. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
2. Barnett, M., M. Leino, K.R., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Condit, J., Hackett, B., Lahiri, S.K., Qadeer, S.: Unifying type checking and property checking for low-level code. In: POPL, pp. 302–314 (2009)
4. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Godefroid, P., Lahiri, S.K., Rubio-González, C.: Statically Validating Must Summaries for Incremental Compositional Dynamic Test Generation. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 112–128. Springer, Heidelberg (2011)
6. Godlin, B., Strichman, O.: Regression verification. In: DAC, pp. 466–471 (2009)
7. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: ICSM, pp. 243–252 (1994)
8. Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research (2010)
9. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: SIGSOFT FSE, pp. 226–237 (2008)
10. Software-artifact Infrastructure Repository, `http://sir.unl.edu/portal/index.html`

[1] Ofer Strichman has helped incorporate the algorithm into SymDiff.