Relational Program Reasoning Using Compiler IR

Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich

Karlsruhe Institute of Technology
Germany
moritz.kiefer@student.kit.edu
{klebanov, ulbrich}@kit.edu

Abstract. Relational program reasoning is concerned with formally comparing pairs of executions of programs. Prominent examples of relational reasoning are program equivalence checking (which considers executions from different programs) and detecting illicit information flow (which considers two executions of the same program).
The abstract logical foundations of relational reasoning are, in the meantime, sufficiently well understood. In this paper, we address some of the challenges that remain to make the reasoning *practicable*. Two major ones are dealing with the feature richness of programming languages such as C and with the weakly structured control flow that many real-world programs exhibit.
A popular approach to control this complexity is to define the analyses on the level of an intermediate program representation (IR) such as one generated by modern compilers. In this paper we describe the ideas and insights behind IR-based relational verification. We present a program equivalence checker for C programs operating on LLVM IR and demonstrate its effectiveness by automatically verifying equivalence of functions from different implementations of the standard C library.

1 Introduction

Relational program reasoning. Over the last years, there has been a growing interest in *relational* verification of programs, which reasons about the relation between the behavior of two programs or program executions – instead of comparing a single program or program execution to a more abstract specification. The main advantage of relational verification over standard functional verification is that there is no need to write and maintain complex specifications. Furthermore, one can exploit the fact that changes are often local and only affect a small portion of a program. The effort for relational verification often only depends on the difference between the programs respectively program executions and not on the overall size and complexity of the program(s).

Relational verification can be used for various purposes. An example is *regression verification* resp. *equivalence checking*, where the behavior of two different versions of a program is compared under identical input. Another example is checking for absence of *illicit information flow*, a security property, in which executions of the same program are compared for different inputs. For concreteness' sake, we focus in this paper on regression verification/equivalence checking of C programs, though the presented techniques readily apply to other instances of relational reasoning.

Regression verification. Regression verification is a formal verification approach intended to complement regression testing. The goal is to establish a formal proof of equivalence of two program versions (e.g., consecutive revisions during program evolution, or a program and a re-implementation). In its basic form, we are trying to prove that the two versions produce the same output for all inputs. In more sophisticated scenarios, we want to verify that the two versions are equivalent only on some inputs (*conditional equivalence*) or differ in a formally specified way (*relational equivalence*). Regression verification is not intended to replace testing, but when it is successful, it offers guaranteed coverage without requiring additional expenses to develop and maintain a test suite.

Challenges in making regression verification practicable. The abstract logical foundations of relational reasoning are, in the meantime, sufficiently well understood. For instance, in [8], we presented a method for regression verification that reduces the equivalence of two related C programs to Horn constraints over uninterpreted predicates. The reduction is automatic, just as the solvers (e.g., Z3 [15, 18] or ELDARICA [23]) used to solve the constraints. Our current work follows the same principles.

Yet, the calculus in [8] only defined rules for the basic, well-structured programming language constructs: assignment, if statement, while loop and function call. The RÊVE tool implemented the calculus together with a simple self-developed program parser.

While the tool could automatically prove equivalence of many intricate arithmetic-intensive programs, its limited programming language coverage hampered its practical application. The underlying calculus could not deal with `break`, `continue`, or `return` statements in a loop body, loop conditions with side effects, `for` or `do-while` loops, let alone arbitrary `goto` statements.

Contributions. The main contribution of this paper is a method for automated relational program reasoning that is significantly more practical than [8] or other state-of-the-art approaches. In particular, the method supports programs with arbitrary unstructured control flow without losing automation. The gained versatility is due to a completely redesigned reduction calculus together with the use of the LLVM compiler framework [17] and its intermediate program representation (IR).

Furthermore, the calculus we present in this paper is fine-tuned for the inference of *relational* predicates and deviates from straightforward encodings in crucial points: (a) Loops are not always reduced to tail recursion (see Sect. 4.6), (b) mutual function summaries are separated into two predicates for pre- and postcondition (see Sect. 4.5), and (c) control flow synchronization points can be placed by the user manually to enable more flexible synchronization schemes.

We developed a tool implementing the approach, which can be tested online at `http://formal.iti.kit.edu/improve/reve/`. We have evaluated the tool by automatically proving equivalence of a number of string-manipulating functions from different implementations of the C standard library.

Main idea of our method. First, we employ the LLVM compiler framework to compile the C source code to LLVM IR. This step reduces all control flow in a program to branches (jumps) and function calls. Next, we divide the (potentially cyclic) control flow graph of the program into linear segments. For the points at which these segments are connected, we introduce relational abstractions represented by uninterpreted predicate symbols (instead of concrete formulas). The same applies for pairs of corresponding function calls. Finally, we generate constraints over these predicate symbols linking the linear segments with the corresponding state abstractions. The produced constraints are in Horn normal form.

The generation of constraints is automatic; the user does not have to supply coupling predicates, loop invariants, or function summaries. The constraints are passed to a constraint solver for Horn clauses (such as Z3 [15, 18] or ELDARICA [23]). The solver tries to find an instantiation of the uninterpreted abstraction predicates that would make the constraints true. If the solver succeeds in finding a solution, the programs are equivalent. Alternatively, the solver may show that no solution exists (i.e., disprove equivalence) or time out.

Advantages of using LLVM IR. There are several advantages to working on LLVM IR instead of on the source code level. The translation to LLVM IR takes care of preprocessing (resolving typedefs, expanding macros, etc.) and also eliminates many ambiguities in the C language such as the size of types (which is important when reasoning about pointers). Building an analysis for IR programs is much simpler as the IR language has fewer instruction types and only two control flow constructs, namely branches (jumps) and function calls. Furthermore, LLVM provides a constantly growing number of simplifying and canonicalizing transformations (*passes*) on the IR level. If the differences in the two programs are merely of a syntactical nature, these simplifications can often eliminate them completely. Also, it was easy to incorporate our own passes specifically geared towards our use case.

Challenges still remaining. Of course, using a compiler IR does not solve all challenges. Some of them, such as interpreting integers as unbounded or the inability to deal with general bit operations or floating-point arithmetic remain due to the limitations of the underlying solvers. Furthermore, we, as is common, assume that all considered programs are terminating. Verifying this property is delegated to the existing termination checking technology, such as [7, 9].

Listing 1: memchr(), dietlibc

```
#include <stddef.h>
extern int __mark(int);

void* memchr(const void *s,
             int c,
             size_t n) {
  const unsigned char *pc =
    (unsigned char *) s;
  for (;n--;pc++) {
    __mark(42);
    if (*pc == c)
      return ((void *) pc);
  }
  return 0;
}
```

Listing 2: memchr(), OpenBSD libc

```
#include <stddef.h>
extern int __mark(int);

void * memchr(const void *s,
              int c,
              size_t n) {
  if (n != 0) {
    const unsigned char *p = s;
    do {
      __mark(42);
      if (*p++ == (unsigned char)c)
        return ((void *)(p - 1));
    } while (--n != 0);
  }
  return (NULL);
}
```

2 Illustration

We tested our approach on examples from the C standard library (or libc). The interfaces and semantics of the library functions are defined in the language standard, while several implementations exist. GNU libc [10] and OpenBSD libc [21] are two mature implementations of the library. The diet libc (or dietlibc) [27] is an implementation that is optimized for small size of the resulting binaries.

Consider the two implementations of the `memchr()` function shown in Listings 1 and 2. The function scans the initial `n` bytes of the memory area pointed to by `s` for the first instance of `c`. Both `c` and the bytes of the memory area pointed to by `s` are interpreted as `unsigned char`. The function returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

In contrast to full functional verification, we are not asking whether each implementation conforms with this (yet to be formalized) specification. Instead, we are interested to find out whether the two implementations behave the same. Whether this is the case is not immediately obvious due to the terse programming style, subtle pointer manipulation, and the different control flow constructs used.

While the dietlibc implementation on the left is relatively straightforward, the OpenBSD one on the right is more involved. The for loop on the left is replaced by a do-while loop wrapped in an if conditional on the right. This transformation known as *loop inversion* reduces the overall number of jumps by two (both in the branch where the loop is executed). The reduction increases performance by eliminating CPU pipeline stalls associated with jumps. The price of the transformation is the duplicate condition check increasing the size of the code. On the other hand, loop inversion makes further optimizations possible, such as eliminating the if statement if the value of the guard is known at compile time.

With one exception, the code shown is the original source code and can indeed be fed like that into our implementation LLRêve, which without further user interaction establishes the equivalence of the two implementations. The exception is the inclusion of the `__mark()` calls in the loop bodies. The calls identify *synchronization points* in the execution of two programs where the states of the two are most similar. The numerical arguments only serve to identify matching pairs of points. The user has to provide enough synchronization points to break all cycles in the control flow, otherwise the tool will abort with an error message. In [8], we used a simple heuristic to put default synchronization points automatically into loop bodies in their order of appearance, though this is not yet implemented in LLRÊVE.

Suppose that we are running the two implementations to look for the same character `c` in the same 100 byte chunk of memory. If we examine the values of variables at points in time when control flow reaches the `__mark(42)` calls for the first time, we obtain: for dietlibc $\mathrm{n} = 99, \mathrm{pc} = \mathrm{s}$, and for OpenBSD $\mathrm{n} = 100, \mathrm{p} = \mathrm{s}$. The second time: for dietlibc $\mathrm{n} = 98, \mathrm{pc} = \mathrm{s}+1$, and for OpenBSD $\mathrm{n} = 99, \mathrm{p} = \mathrm{s} + 1$. The values of `c`, `s`, and the whole heap remain the same. At this point, one could suspect that the following formula is an invariant relating the executions of the two implementations at the above-mentioned points:[1]

$$(\mathrm{n}_2 = \mathrm{n}_1 + 1) \wedge (\mathrm{p}_2 = \mathrm{pc}_1) \wedge (\mathrm{c}_2 = \mathrm{c}_1) \wedge \forall i.\, \mathit{heap}_1[i] = \mathit{heap}_2[i] \ . \qquad (*)$$

That our suspicion is correct can be established by a simple inductive argument. Once we have done that, we can immediately derive that both programs produce the same return value upon termination.

We call an invariant like $(*)$ for two loops a *coupling (loop) invariant*. A similar construct relating two function calls is called a *mutual (function) summary* [13, 14]. Together, they fall into the class of *coupling predicates*, inductive assertions allowing us to deduce the desired relation upon program termination. In [8], we have shown that coupling predicates witnessing equivalence of programs with `while` loops can be often automatically inferred by methods such as counterexample-guided abstraction refinement or property-directed reachability. In this paper, we present a method for doing this for programs with unstructured control flow.

3 Related Work

Our own previous work on relational verification of C programs [8] has already been discussed in the introduction.

Many code analysis and formal verification tools operate on LLVM IR, though none of them, to our knowledge, perform relational reasoning. Examples of non-relational verification tools building on LLVM IR are LLBMC [19] and SeaHorn [12]. The SeaHorn tool is related to our efforts in particular, since it processes safety properties of LLVM IR programs into Horn clauses over integers. An interesting recent development is the SMACK [22] framework for rapid prototyping of verifiers, a translator from the LLVM IR into the Boogie intermediate verification language (IVL) [2].

[1] To distinguish identifiers from the two programs, we add subscripts indicating the program to which they belong. We may also concurrently use the original identifiers without a subscript as long as the relation is clear from the context.

The term regression verification for equivalence checking of similar programs was coined by Godlin and Strichman [11]. In their approach, matching recursive calls are abstracted by the same uninterpreted function. The equivalence of functions (that no longer contain recursion) is then checked by the CBMC model checker. The technique is implemented in the RVT tool and supports a subset of ANSI C.

Verdoolaege et al. [26, 25] have developed an automatic approach to prove equivalence of static affine programs. The approach focuses on programs with array-manipulating `for` loops and can automatically deal with complex loop transformations such as loop interchange, reversal, skewing, tiling, and others. It is implemented in the ISA tool for the static affine subset of ANSI C.

Mutual function summaries have been prominently put forth by Hawblitzel et al. in [13] and later developed in [14]. The concept is implemented in the equivalence checker SymDiff [16], where the user supplies the mutual summary. Loops are encoded as recursion. The tool uses Boogie as the intermediate language, and the verification conditions are discharged by the Boogie tool. A frontend for C programs is available.

The BCVerifier tool for proving backwards compatibility of Java class libraries by Welsch and Poetzsch-Heffter [28] has a similar pragmatics as SymDiff. The tool prominently features a language for defining synchronization points.

Balliu et al. [1] present a relational calculus and reasoning toolchain targeting information flow properties of unstructured machine code. Coupling loop invariants are supplied by the user.

Barthe et al. [3] present a calculus for reasoning about relations between programs that is based on pure program transformation. The calculus offers rules to merge two programs into a single *product program*. The merging process is guided by the user and facilitates proving relational properties with the help of any existing safety verification tool. We are not aware of an implementation of the transformation.

Beringer [4] defines a technique for deriving soundness arguments for relational program calculi from arguments for non-relational ones. In particular, one of the presented relational calculi contains a loop rule similar to ours. The rule targets so-called *dissonant* loops, i.e., loops not proceeding in lockstep.

Ulbrich [24] introduces a framework and implementation for relational verification on an unstructured intermediate verification language (similar to Boogie), mainly targeted at conducting refinement proofs. Synchronization points are defined and used similar to this work. However, the approach is limited to fully synchronized programs and requires user-provided coupling predicates.

4 The Method

4.1 From Source Code to LLVM IR

LLVM's intermediate representation is an abstract, RISC-like assembler language for a register machine with an unbounded number of registers. A program in LLVM-IR consists of type definitions, global variable declarations, and the program itself, which is represented as a set of functions, each consisting of a graph of basic blocks. Each basic block in turn is a list of instructions with acyclic control flow and a single exit point.

The branch instructions between basic blocks induce a graph on the basic blocks, called the *control flow graph* (CFG), in which edges are annotated with the condition under which the transition between the two basic blocks is taken. Programs in LLVM IR are in *static single assignment* (SSA) form, i.e., each (scalar) variable is assigned exactly once in the static program. Assignments to scalar variables can thus be treated as logical equivalences.

To obtain LLVM IR programs from C source code, we first compile the two programs separately using the Clang compiler. Next, we apply a number of standard and custom-built transformation passes that:

– eliminate load and store instructions (generated by LLVM) for stack-allocated variables in favor of register operations. While we do support the general load and store instructions, they increase deduction complexity.
– propagate constants and eliminate unreachable code.
– eliminate conditional branching between blocks in favor of conditional assignments (similar to the ternary operator `?` in C). This step reduces the number of distinct paths through the program. As we are considering a product of all paths, this step is important.
– inline function calls where desired by the user.

4.2 Synchronization Points and Breaking Control Flow Cycles

If the compiled program contained loops or iteration formulated using goto statements, the resulting CFG is cyclic. Cycles are a challenge for deductive verification because the number of required iterations is, in general, not known beforehand.

We break up cycles in the control flow by defining *synchronization points*, at which we will abstract from the program state by means of predicates. The paths between synchronization points are then cycle-free and can be handled easily. Synchronization points are defined by labeling basic blocks of the CFG with unique numbers $n \in \mathbb{N}$. Additionally, the entry and the exit of a function are considered special synchronization points labeled with B and E. If every cycle in the CFG contains at least one synchronization point, the CFG can be considered as the set of all *linear paths* leading from one synchronization point directly to another. A linear path is a sequence of basic blocks together with the transition conditions between them. Formally, it is a triple $\langle n, \pi, m \rangle$ in which n and m denote the beginning and end synchronization point of the segment and $\pi(x, x')$ is the two-state transition predicate between the synchronization points in which x are the variables before and x' after the transition. Since basic blocks are in SSA form, the transition predicate defined by a path is the conjunction of all traversed assignments (as equalities) and transition conditions. The treatment of function invocation is explained in Sect. 4.5.

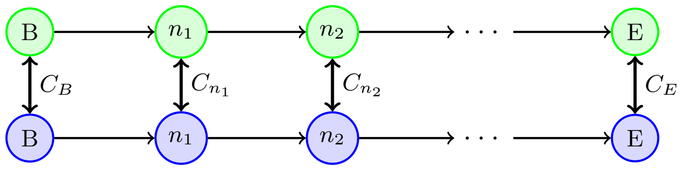

Fig. 1: Illustration of coupled control flow of two fully synchronized programs

4.3 Coupling and Coupling Predicates

Let in the following the two compared functions be called P and Q, and let x_p (resp. x_q) denote the local variables of P (resp. Q). Primed variables refer to post-states.

We assume that P and Q are related to each other, in particular that the control and data flow through the functions is similar. This means that we expect that there exist practicable *coupling predicates* describing the relation between corresponding states of P and Q. The synchronization points mark where the states are expected to be coupled. If a function were compared against itself, for instance, the coupling between two executions would be equality ranging over all variables and all heap locations. For the analysis of two different programs, more involved coupling predicates are, of course, necessary.

Formally, we introduce a coupling predicate $C_n(x_p, x_q)$ for every synchronization point index n. Note that these predicates have the variables of both programs as free variables. Two functions are considered coupled, if they yield *coupled traces* when fed with the same input values; coupled in the sense that the executions pass the same sequence of synchronization points in the CFG and that at each synchronization point, the corresponding coupling predicate is satisfied. See Fig. 1 for an illustration.

The coupling predicates C_B and C_E for the function entry and exit are special in that they form the *relational specification* for the equivalence between P and Q. For pure equivalence, C_B encodes equality of the input values and state, and C_E of the result value and output state. Variations like conditional or relational equivalence can be realized by choosing different formulas for C_B and C_E.

4.4 Coupling Predicates for Cyclic Control Flow

In the following, we outline the set of constraints that we generate for programs with loops. If this set possesses a model, i.e., if there are formulas making the constraint true when substituted for the coupling predicate placeholders C_i, then the programs fulfill their relational specification.

The first constraint encodes that every path leading from a synchronization point to the next satisfies the coupling predicate at the target point. Let $\langle n, \pi, m\rangle$ be a linear path in the CFG of P and $\langle n, \rho, m\rangle$ one for the same synchronization points for Q. For each such pair of paths, we emit the constraint:

$$C_n(x_p, x_q) \land \pi(x_p, x'_p) \land \rho(x_q, x'_q) \to C_m(x'_p, x'_q) \quad . \tag{1}$$

The above constraint only covers the case of *strictly synchronized* loops which are iterated equally often. Yet, often the number of loop iterations differs between revisions, e.g., if one loop iteration has been peeled in one of the programs. To accommodate that, we allow one program, say P, to loop at a synchronization point n more often than the other program.[2] Thus, P proceeds iterating the loop, while Q stutters in its present state. For each looping path $\langle n, \pi, n \rangle$ in P, we emit the constraint:

$$C_n(x_p, x_q) \wedge \pi(x_p, x'_p) \wedge \Big(\bigwedge_{\substack{\langle n,\rho,n \rangle \\ \text{in } Q}} \forall x'_q.\, \neg\rho(x_q, x'_q) \Big) \to C_n(x'_p, x_q) \ . \tag{2}$$

The second conjunct in the premiss of the implication encodes that P iterates from n to n, while the third captures that no linear path leads from n to n in Q from initial value x_q. The coupling predicate in the conclusion employs the initial values x_q, since we assume that the state of Q stutters.

Emitting (2) to accommodate loops that are not strictly synchronized adds to the complexity of the overall constraint and may in practice prevent the solver from finding a solution. We thus provide the user with the option to disable emitting (2), if they are confident that strict synchronization is sufficient.

Finally, we have to encode that the control flow of P and Q remains synchronized in the sense that it must not be possible that P and Q reach different synchronization points m and k when started from a coupled state at n.[3] For each path $\langle n, \pi, m \rangle$ in P and $\langle n, \rho, k \rangle$ in Q with $m \neq k$, $n \neq m$, $n \neq k$, we emit the constraint:

$$C_n(x_p, x_q) \wedge \pi(x_p, x'_p) \wedge \rho(x_q, x'_q) \to \text{false} \ . \tag{3}$$

4.5 Coupling Predicates for Function Calls

Besides at synchronization points that abstract loops or iteration in general, coupling predicates are also employed to describe the effects of corresponding function invocations in the two programs. To this end, matching pairs of function calls in the two CFGs are abstracted using mutual function summaries [13]. A heuristic used to match calls will be described later.

Mutual function summaries. Let f_p be a function called from the function P, x_p denote the formal parameters of f_p, and r_p stand for the (optional) result returned when calling f_p. Assume that there is an equally named function f_q defined in the program of Q. A mutual summary for f_p and f_q is a predicate $Sum_f(x_p, x_q, r_p, r_q)$ that relationally couples the result values to the function arguments. If the function accesses the heap, the heap appears as an additional argument and return value of the function.

[2] The situation is symmetric with the case for Q omitted here.

[3] This restriction is of minor practical importance but releases us from the need to create coupling predicates for arbitrary combinations of synchronization points.

In our experiments, we found that it is beneficiary to additionally model an explicit relational precondition $\mathit{Pre}_f(x_p, x_q)$ of f. Although it does not increase expressiveness, the solvers found more solutions with precondition predicates present. We conjecture that the positive effect is related to the fact that mutual summary solutions are usually of the shape $\phi(x_p, x_q) \to \psi(r_p, r_q)$, and that making the precondition explicit allows the solver to infer ϕ and ψ separately without the need to infer the implication.

For every pair of paths $\langle n, \pi, m\rangle \in P$ and $\langle n, \rho, m\rangle \in Q$ that contain a single call to f, we emit the following additional constraint:

$$C_n(x_p, x_q) \wedge \pi(x_p, x'_p) \wedge \rho(x_q, x'_q) \to \mathit{Pre}_f(x^*_p, x^*_q) \ . \tag{4}$$

in which x^*_p and x^*_q denote the SSA variables used as the argument for the function calls to f. The constraint demands that the relational precondition Pre_f must be met when the callsites of f are reached in P and Q.

For every such pair of paths, we can now make use of the mutual summary by assuming $\mathit{Sum}_f(x^*_p, x^*_q, r_p, r_q)$. This means that for constraints emitted by (1)–(3), the mutual summary of the callsite can be added to the premiss. The augmented version of constraint (1) reads, for instance,

$$C_n(x_p, x_q) \wedge \pi(x_p, x'_p) \wedge \rho(x_q, x'_q) \wedge \mathit{Sum}_f(x^*_p, x^*_q, r_p, r_q) \to C_m(x'_p, x'_q) \ , \tag{5}$$

with r_p and r_q the SSA variables that receive the result values of the calls.

The mutual summary also needs to be justified. For that purpose, constraints are recursively generated for f, with the entry coupling predicate $C_B = \mathit{Pre}_f$ and exit predicate $C_E = \mathit{Sum}_f$.

The generalization to more than one function invocation is canonical.

Example. To make the above clearer, let us look at the encoding of the program in Listing 3 when verified against itself. Let $C^{\mathtt{f}}_B(n_1, n_2)$ and $C^{\mathtt{f}}_E(r_1, r_2)$ be the given coupling predicates that have to hold at the entry and exit of f. When encoding the function f, we are allowed to use Sum_g at the callsite but have to show that Pre_g holds. Thus we get the following constraints:

```
int f(int n) {
  return g(n-1);
}
int g (int n) {
  return n+1;
}
```

Listing 3: `f()` calling `g()`

$$C^{\mathtt{f}}_B(n_1, n_2) \wedge n^*_1 = n_1{-}1 \wedge n^*_2 = n_2{-}1 \to \mathit{Pre}_g(n^*_1, n^*_2)$$
$$C^{\mathtt{f}}_B(n_1, n_2) \wedge n^*_1 = n_1{-}1 \wedge n^*_2 = n_2{-}1 \wedge \mathit{Sum}(n^*_1, n^*_2, r_1, r_2) \to C^{\mathtt{f}}_E(r_1, r_2) \ .$$

To make sure that Pre_g and Sum_g are a faithful abstraction for g, we have a new constraint for g, which boils down to

$$\mathit{Pre}_g(n_1, n_2) \to \mathit{Sum}_g(n_1, n_2, n_1 + 1, n_2 + 1) \ .$$

At this point, the set of constraints is complete, and we can state the main result:

```
int f(int x) {
  if (x > 0) {
    x = g(x);
    x = g(x);
  }
  x = h(x);
  x = h(x);
  x = g(x);
  return x;
}
```

Program 1

```
int f(int x) {
  x = g(x);
  x = g(x);
  x = g(x);
  x = h(x);
  x = h(x);
  return x;
}
```

Program 2

```
g(int) —— g(int)
g(int) —— g(int)
h(int)
h(int)
g(int) —— g(int)
          h(int)
          h(int)
```

Matching for $x > 0$

```
          g(int)
          g(int)
          g(int)
h(int) —— h(int)
h(int) —— h(int)
g(int)
```

Matching for $x \leq 0$

Fig. 2: Illustration of function call matching

Theorem 1 (Soundness). *Let S be the set of constraints emitted by* (1)–(5).

If the universal closure of S is satisfiable, then P and Q terminate in states with x'_p and x'_q satisfying $C_E(x'_p, x'_q)$ when they are executed in states with x_p and x_q satisfying $C_B(x_p, x_q)$ and both terminate.

Matching function calls. For treatment using mutual summaries, the function calls need to be combined into pairs of calls from both programs. Our goal is to match as many function calls between the two programs as possible. To this end, we look at any pair of possible paths from the two programs that start and end at the same synchronization points. For each path, we consider the sequence of invoked functions. To determine the optimal matching of function calls (i.e., covering as many calls as possible), an algorithm [20] for computing the longest common (not necessarily continuous) subsequence among the sequences is applied.

As an example, consider the functions in Fig. 2. There are no cycles in the control flow, so the only two synchronization points are the function entry and exit. In Program 1, there are two paths corresponding to $x > 0$ and $x \leq 0$ respectively. In Program 2, there is only a single path. That gives us two possible path pairs that we need to consider. The resulting longest matchings for the pairs are also shown in the figure. Matched calls are abstracted using mutual summaries, while unmatched calls have to be abstracted using conventional functional summaries.

An additional feature is that the user can request to inline a specific call or all calls to a function with an `inline` pragma. The feature is especially important if the callee function contains a loop that should be synchronized with a loop in the caller function of the other program. The pragma can also be used to inline some steps of a recursive call.

If a function's implementation is not available. A special case arises when there is a call from both programs to a function for which we do not have access to the sources. If such calls can be matched, we abstract the two calls using the canonical mutual summary $Sum_f : x_p = x_q \rightarrow r_p = r_q$ stating that equal inputs induce equal results. If a call cannot be matched, however, we have to use an

```
1  int f(int n) {
2    int i = 0;
3    while (i < n) {
4      i++;
5    }
6    int r = i;
7    return r;
8  }
```

Listing 4: Function `f`

$$
\begin{aligned}
&\forall n. rel_{in}(n) \to inv(0, n)\\
&\forall i, n. (i < n \land inv(i, n)) \to inv(i + 1, n)\\
&\forall i, n. (\neg(i < n) \land inv(i, n)) \to rel_{out}(i)
\end{aligned}
$$

Fig. 3: Iterative encoding of `f`

$$
\begin{aligned}
\forall n. rel_{in}(n) &\to inv_{pre}(0, n) \land\\
&\qquad (\forall r. inv(0, n, r) \to inv_f(n, r))\\
\forall i, n, r. (i < n \land inv_{pre}(i, n) \land inv(i + 1, n, r)) &\to inv(i, n, r)\\
\forall i, n. (\neg(i < n) \land inv_{pre}(i, n) &\to inv(i, n, i)\\
\forall n, r. (rel_{in}(n) \land inv_f(n, r)) &\to rel_{out}(r)
\end{aligned}
$$

Fig. 4: Recursive encoding of `f`

uninterpreted functional summary, losing all information about the return value and the resulting heap. In most cases, this means that nothing can be proved.[4]

4.6 Alternative Loop Treatment as Tail Recursion

When developing our method, we explored two different approaches to deal with iterative unstructured control flow.

The first one models a program as a collection of mutually recursive functions such that the function themselves do not have cyclic control flow. Loops must be translated to tail recursion. This aligns with the approach presented in [13]. It is attractive since it is conceptually simple allowing a unified handling of cyclic branching and function calls. However, our experiments have shown that for our purposes the encoding did not work as well as the one presented in Sect. 4.4 which handles loops using coupling predicates directly instead of by translation into tail recursion. A possible explanation for this observation could be that the number of arguments to the coupling predicates is smaller if (coupling) invariants are used. For these predicates, it suffices to use those variables as arguments which may be changed by the following code. The mutual summaries for tail recursion require more variables and the return values as arguments.

To illustrate the two styles of encoding, we explain how the program in Listing 4 is encoded. For simplicity of presentation, we encode a safety property of a *single* program. The point where the invariant *inv* has to hold is the loop header on Line 3. rel_{in} is a predicate that has to hold at the beginning of f and rel_{out} is the predicate that has to hold when f returns. In the recursive encoding

[4] Alternatively, it would also be possible to trade soundness for completeness and, e.g., assume that such a call does not change the heap.

(Fig. 4), *inv* has three arguments, the local variables i and n and the return value r. In the iterative case (Fig. 3), the return value is not an argument, so *inv* only has two arguments. The entry predicate inv_{pre} over the local variables i and n has to hold at every "call" to *inv*. The reasoning for having such a separate predicate has already been explained in Section 4.5.

In the end, a combination of the two encodings proved the most promising: We apply the iterative encoding to the function whose exit and entry predicates have been given as relational specification explained in 4.3. All other functions are modeled using the recursive encoding. Mutual summaries depend, by design, on the input parameters as well as the output parameters whereas the relational postcondition C_E usually only depends on the output parameters. Using an iterative encoding for the other functions would require passing the input parameters through every predicate to be able to refer to them when establishing the mutual summary at the exit point. The advantage of an iterative encoding of having fewer parameters in predicates is thereby less significant, and we employ the recursive encoding. A special case arises when the toplevel function itself recurses. In this case, we encode it twice: first using the iterative encoding, which then relies on the recursive encoding for the recursive calls.

4.7 Modeling the Heap

The heap is modeled directly as an SMT array and the LLVM load and store instructions are translated into the select and store functions in the SMT theory of arrays. We assume that all load and store operations are properly aligned; we do not support bit operations or, e.g., accessing the second byte of a 32 bit integer. Struct accesses are resolved into loads and stores at corresponding offsets. The logical handling of constraints with arrays requires quantifier reasoning and introduces additional complexity. We handle such constraints following the lines of [6].

5 Experiments

Our implementation of the approach consists of ca. 5.5 KLOC of C++, building on LLVM version 3.8.0.

In our experiments, we have proven equivalence across a sample of functions from three different libc implementations: dietlibc [27], glibc [10], and the OpenBSD

Table 1: Performance with different solvers for the libc benchmarks

		Run time w/solver, seconds	
Function	Source	ELDARICA	Z3/DUALITY
memccpy	d/o	0.733	0.499
memchr	d/o	0.623	0.328
memmem	d/o	1.545	3.634
memmove	d/o	4.195	4.219
memrchr	g/o	0.487	1.082
memset	d/o	0.263	1.211
sbrk	d/g	0.439	0.630
stpcpy	d/o	0.203	0.241
strchr	d/g	48.145	13.705
strcmp	g/o	0.545	0.985
strcspn	d/o	17.825	t/o
strncmp	g/o	1.046	4.556
strncmp	d/g	2.599	7.971
strncmp	d/o	0.602	1.742
strpbrk	d/o	3.419	3.237
strpbrk	d/g	1.029	2.083
strpbrk	g/o	1.734	3.453
swab	d/o	4.032	0.709

d=dietlibc, g=glibc, o=OpenBSD libc.
t/o=timeout after 300 seconds.
2 GHz i7-4750HQ CPU, 16 GB RAM.

```
void *memmove(void *dst,
              const void *src,
              size_t count) {
  char *a = dst;
  const char *b = src;



  if (src != dst) {


    if (src > dst) {
      while (count--) {
        __mark(0);
        *a++ = *b++;
      }
    } else {
      a += count - 1;
      b += count - 1;
      while (count--) {
        __mark(1);
        *a-- = *b--;
      }
    }
  }


  return dst;
}
```

(a) dietlibc

```
void *memmove(void *dst0,
              const void *src0,
              size_t length) {
  char *dst = dst0;
  const char *src = src0;
  size_t t;
  if (length == 0 || dst == src)
    goto done;
  if ((unsigned long)dst <
      (unsigned long)src) {
    t = length;
    if (t) {
      do {
        __mark(0);
        *dst++ = *src++;
      } while (--t);
    }
  } else {
    src += length;
    dst += length;
    t = length;
    if (t) {
      do {
        __mark(1);
        *--dst = *--src;
      } while (--t);
    }
  }
done:
  return (dst0);
}
```

(b) OpenBSD libc

Fig. 5: `memmove()`

libc [21]. Apart from the not yet automated placing of the synchronization marks, the proofs happen without user interaction. The runtimes of the proofs are summarized in Table 1. One of the more complex examples, the function `memmove()`, is shown in Fig. 5. It demonstrates the use of nested `if`s, multiple loops with different loop structures (`while`/`do-while`) and `goto` statements.

Revisiting the `memchr()` example discussed in Section 2, the early implementation of `memchr()` in dietlibc is known to have contained a bug (Listing 5). In case of a found character, the return value is one greater than expected. Unsurprisingly, this implementation cannot be proven equivalent to any of the other two, and LLRÊVE produces a counterexample. While counterexamples in the presence of heap operations in the program can be spurious (in the absence of heap operations, counterexamples are always genuine), in this case, the counterexample does demonstrate the problem.

```
void* memchr(const void *s,
             int c,
             size_t n) {
  const char* t=s;
  int i;
  for (i=n; i; --i)
    if (*t++==c)
      return (char*)t;
  return 0;
}
```

Listing 5: Bug in `memchr()`

An interesting observation we made was that existentially quantified preconditions might potentially be necessary, such as requiring the existence of a null byte terminating a string. While techniques for solving existentially quantified Horn clauses exist [5], most solver implementations currently only support universally quantified clauses. The libc implementations, however, were sufficiently similar so that such preconditions were not necessary.

6 Conclusion

We have shown how the automated relational reasoning approach presented in [8] can be taken in its applicability from a basic fragment to the full C language standard w.r.t. the control flow. In this work, LLVM played a crucial rule in reducing the complexity of a real-world language. We have successfully evaluated our approach on code actually used in production and were able to prove automatically that many string-manipulation functions from different implementations of libc are equivalent.

Acknowledgments

This work was partially supported by the German National Science Foundation (DFG) under the IMPROVE APS project within the priority program SPP 1593 “Design For Future – Managed Software Evolution”.

References

1. Musard Balliu, Mads Dam, and Roberto Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, pages 1080–1091. ACM, 2014.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO’05, pages 364–387. Springer-Verlag, 2006.
3. Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In Michael Butler and Wolfram Schulte, editors, *Proceedings, 17th International Symposium on Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2011.
4. Lennart Beringer. Relational decomposition. In *Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP)*, volume 6898 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2011.
5. Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. Solving existentially quantified Horn clauses. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 869–882. Springer, 2013.
6. Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On solving universally quantified Horn clauses. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 105–125. Springer, 2013.

7. Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'12)*, pages 261–277, Berlin, Heidelberg, 2012. Springer.
8. Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 349–360. ACM, 2014.
9. Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination proofs with AProVE. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference (RTA 2004), Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 210–220. Springer, 2004.
10. GNU C library. `https://www.gnu.org/software/libc/`, 2016.
11. Benny Godlin and Ofer Strichman. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 466–471. ACM, 2009.
12. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification (CAV), Proceedings*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.
13. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Mutual summaries: Unifying program comparison techniques. In *Proceedings, First International Workshop on Intermediate Verification Languages (BOOGIE)*, 2011. Available at `http://research.microsoft.com/en-us/um/people/moskal/boogie2011/boogie2011_pg40.pdf`.
14. Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 282–299. Springer, 2013.
15. Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 157–171, Berlin, Heidelberg, 2012. Springer-Verlag.
16. Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.
17. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04. IEEE Computer Society, 2004.
18. Kenneth McMillan and Andrey Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, 2013.
19. Florian Merz, Stephan Falke, and Carsten Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, VSTTE'12, pages 146–161, Berlin, Heidelberg, 2012. Springer-Verlag.
20. Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

21. OpenBSD libc. `http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/`, 2016.
22. Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 106–113, 2014.
23. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for Horn-clause verification. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 347–363, Berlin, Heidelberg, 2013. Springer-Verlag.
24. Mattias Ulbrich. *Dynamic Logic for an Intermediate Language: Verification, Interaction and Refinement.* PhD thesis, Karlsruhe Institute of Technology, June 2013. `http://nbn-resolving.org/urn:nbn:de:swb:90-411691`.
25. Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11:1–11:35, 2012.
26. Sven Verdoolaege, Martin Palkovic, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Experience with widening based equivalence checking in realistic multimedia systems. *J. Electronic Testing*, 26(2):279–292, 2010.
27. Felix von Leitner. diet libc. `https://www.fefe.de/dietlibc/`, 2016.
28. Yannick Welsch and Arnd Poetzsch-Heffter. Verifying backwards compatibility of object-oriented libraries using Boogie. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 35–41. ACM, 2012.