**Noname manuscript No.**
(will be inserted by the editor)

# Relational Program Reasoning Using Compiler IR

## Combining Static Verification and Dynamic Analysis

**Moritz Kiefer · Vladimir Klebanov ·
Mattias Ulbrich**

the date of receipt and acceptance should be inserted later

**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, by now, 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 that operates on LLVM IR. To extend the reach of the approach and to make it more efficient, we show how dynamic analyses can be employed to support and strengthen the static verification.

The effectiveness of the approach is demonstrated 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

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

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, by now, sufficiently well understood. For instance, in [13], 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 [20, 24] or Eldarica [32]) used to solve the constraints. Our current work follows the same principles.

Yet, the calculus in [13] 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.

**Incorporating dynamic analyses** Horn constraint solvers are a powerful technique to infer the predicates required in our approach. However, there are limits to their capabilities:

1. If the programs are not related enough, i.e., if there control-flow structures are too different, the required predicates are more involved and more difficult to infer.

2. They are limited to certain shapes of coupling predicates (essentially first order formulas over linear arithmetic)

To support Horn constraint solvers in their task, we have devised two techniques that make the resulting sets of Horn constraints easier to verify. The techniques exploit that we still have the original programs which can be evaluated/executed. They analyze dynamic data gathered during repeated execution of the programs. We thus combine dynamic and static analyses as the latter incorporates insight gained in the first.

**Contributions.** The main contribution of this paper is a method for automated relational program reasoning that is significantly more practical than [13] 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 [22] 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 plain 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.

In addition to the logical encoding, we present techniques that exploit dynamic data gathered from traces of recorded program executions. The information is used (a) to find program transformations which harmonize the loop structure between the compared programs, and (b) to efficiently infer loop invariant candidates.

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 [20,24] or Eldarica [32]). 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.

Listing 1: memchr(), dietlibc

```
#include <stddef.h>

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>

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);
}
```

**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 handling integers overflows correctly, more efficient heap data structure encodings, or dealing with general bit operations or floating-point arithmetic remain due to the limitations of the underlying solvers.

## 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 [15] and OpenBSD libc [29] are two mature implementations of the library. The diet libc (or dietlibc) [23] 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 or not 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.

The code shown here 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. For demonstration purposes, we have added invocations of the synthetic function `__mark()` into the loop bodies. These calls identify *synchronization points* in the execution of the two programs where their states are most similar. The numerical arguments to `__mark` serve to identify matching pairs of points. Synchronization points must be added such that all cycles in the control flow are broken, otherwise the tool will abort with an error message. In cases where the control flow structure between the two compared programs is similar enough (like in the example), the engine is able to infer the marks automatically. If the loop synchronization is not obvious, the user is able to manually annotate coupling synchronization like done in the example.

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]

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

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* (e.g., by Hawblitzel et al. [18,19]). Together, they fall into the class of *coupling predicates*, inductive assertions allowing us to deduce the desired relation upon program termination. In [13], 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.

[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.

## 3 Related Work

Our own previous work on relational verification of C programs [13] 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 [25] and SeaHorn [17]. 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 [31] framework for rapid prototyping of verifiers, a translator from the LLVM IR into the Boogie intermediate verification language (IVL) [3].

The term regression verification for equivalence checking of similar programs was coined by Godlin and Strichman [16]. 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.

Parallel to us, De Angelis et al. [9] developed another relational verification process based on Horn constraints. This work assumes that the two programs have been translated into constrained Horn clauses separately. The two Horn constraints—rather than control flow graphs as in LLRÊVE—are combined into a single Horn constraint that encodes the desired relational property.

Verdoolaege et al. [40,39] 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 [18] and later developed in [19]. The concept is implemented in the equivalence checker SYMDIFF [21], 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 [41] 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. [4] 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. In a later extension [5], they present a framework for asymmetric relational problems (in which traces may be universally or existentially quantified). Their implementation on top of Frama-C can also deal with more complex unwinding schemes like loop tiling.

Beringer [6] 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. Banerjee and Naumann [2] present a theoretical logical foundation for modeling relational problems with framing based on region logic. They liberalize coupling conditions even more than presented here by allowing user-specified semantic predicates (called alignment guards) which control the synchronization between the programs. It is not clear how alignment guards could be inferred automatically in our approach.

Ulbrich [38] introduces a framework and implementation for relational verification on an unstructured intermediate verification language (similar to Boogie). It also supports asymmetric relational verification and is 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.

Dynamic analyses have already been used for loop invariant discovery. The Daikon tool [11] uses user-specified patterns to identify invariant candidates for Java programs. The dynamic invariant generator DIG [28] infers from dynamically gathered data, amongst other kinds of invariants, algebraic equations as loop invariants. These approaches are similar to our techniques outlined in Sect. 5. However, these approaches have used dynamic analyses for the inference of functional rather than for relational loop invariant candidates. Since for relational verification (and for regression verification in particular) invariants can be expected to follow typical (application-independent) patterns, we are confident that pattern-driven invariant inference is as least as promising for relational as for functional cases.

Extending dynamic analyses by creating a counterexample driven refinement loops has been explored previously both for polynomial invariants [34] and as a general framework independent of the dynamic analyses and combined with a random search [33].

The use of program transformations to reduce differences between programs has been explored previously by Smith and Dill [35]. However, these transformations focus on bounded control flow and rewrite a small sequence of instructions while we focus on transforming unbounded loops. Barthe et al. [4] and Banerjee et al. [2] provide rule-based schemes for user-guided program weaving. Our approach strives to avoid user interaction such thatwe did not adapt these interactive strategies.

While we assume fixed coupling points and then harmonize differences by applying program transformations discovered using dynamic analyses, Partush and Yahav [30] explores a dynamic inference of coupling points by the use of abstract interpretation. Specifically the approach tries to find points at which the difference between the programs is minimal.

*Translation validation*verifies that programs produced by an optimizing compiler are semantically equivalent to the input programs. While this problem also requires proving programs equivalent, existing approaches for translation validation typically try to exploit the fact that program differences were produced by compiler optimizations and thus have a specific form. In particular, most approaches target intra-procedural optimizations [43,42,36,27]. Some approaches also require that branch instructions in the input and the optimized program correspond to each other [43,42,27] and can thus only verify equivalence if there are no significant structural differences. The work by Zaks and Pnueli [43,42] and the work by Necula [27] both use relational invariants to deal with unbounded control flow but their invariants are limited to equalities. *Equality saturation* is a different technique used by Stepp et al. [36] for translation validation of LLVM programs. Equality saturation iteratively infers equalities based on built-in axioms until it can prove that both programs are in the same equivalence class. However, some of these axioms are specific to the optimizations found in LLVM so it is unclear if a set of axioms can be found that are suitable for verifying program equivalence in general. Tristan et al. [37] also target LLVM but their approach proceeds by creating a combined value graph of both programs based on their gated SSA representations. This value graph is then successively normalized based on a set of built-in rules until either no further normalization is possible. While this approach works well for some optimizations, the authors have not implemented it for other optimizations such as *instcombine* due to the large number of rules required. This suggests that finding a general set of rules that are suitable for verifying the equivalence of two programs is challenging.

## 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 (i.e., LLVM's select instructions which are similar to the ternary operator `?` in C). This step reduces the number of distinct paths through the program. The transformation is no guarantee against an exponential blowup of the number of paths of a program, but we have experienced that it kept the number of distinct paths manageable.
- inline function calls where desired by the user.

While further LLVM optimization passes might have positive effects on verification efficiency, they tend to modify the control flow graphs considerably, thus disturbing the annotated synchronization similarities between the programs. Since this may lead LLRêve astray, they have not been included in the implementation.

### 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.

### 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

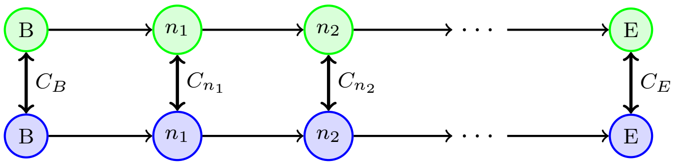


Fig. 1: Illustration of coupled control flow of two fully synchronized programs

$C_E$ 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:

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

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:

$$C_n(x_p, x_q) \land \pi(x_p, x_p') \land \Bigg(\bigwedge_{\substack{\langle n,\rho,n\rangle \\ \text{in } Q}} \forall x_q'.\, \neg\rho(x_q, x_q')\Bigg) \to C_n(x_p', x_q) \ . \tag{2}$$

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

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

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:

$$C_n(x_p, x_q) \land \pi(x_p, x_p') \land \rho(x_q, x_q') \to \text{false} \ . \tag{3}$$

### 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. 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 $\mathit{Sum}_f(x_p, x_q, r_p, r_q)$ 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.

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 $\phi$ and $\psi$ 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:

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

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 $\mathit{Pre}_f$ 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,

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

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.

[3] This restriction releases us from the need to create coupling predicates for arbitrary combinations of synchronization points. It has been of minor practical importance on the considered examples where a one-to-one mapping of synchronization points could easily be specified.

*Example.* To make the above clearer, let us look at the encoding of the program in Listing 3 when verified against itself. Let $C^{\mathbf{f}}_B(n_1, n_2)$ and $C^{\mathbf{f}}_E(r_1, r_2)$ 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 $\mathit{Sum}_g$ at the callsite but have to show that $\mathit{Pre}_g$ 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()`

$$C^{\mathbf{f}}_B(n_1, n_2) \wedge n_1^* = n_1{-}1 \wedge n_2^* = n_2{-}1 \rightarrow \mathit{Pre}_g(n_1^*, n_2^*)$$

$$C^{\mathbf{f}}_B(n_1, n_2) \wedge n_1^* = n_1{-}1 \wedge n_2^* = n_2{-}1 \wedge \mathit{Sum}(n_1^*, n_2^*, r_1, r_2) \rightarrow C^{\mathbf{f}}_E(r_1, r_2) \enspace .$$

To make sure that $\mathit{Pre}_g$ and $\mathit{Sum}_g$ are a faithful abstraction for $g$, we have a new constraint for $g$, which boils down to

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

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

**Proposition 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 [26] 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.

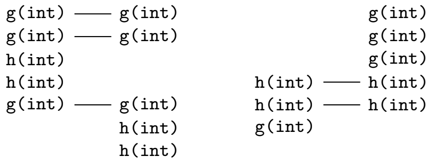
An additional feature is that the user can request to inline a specific call or all calls to a function with an `inline` pragma. The feature is especially important if the callee function contains a loop that should be synchronized with a loop in the caller function of the other program. The pragma can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

```
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

```
int f(int n) {
  int i = 0;
  while (i < n) {
    i++;
  }
  int r = i;
  return r;
}
```

Listing 4: Function `f`

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

Fig. 3: Iterative encoding of `f`

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

Fig. 4: Recursive encoding of `f`

**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, there are two possibilities: We support user-specified mutual summary annotations such that relational (and functional) properties about libraries can be used as assumptions during verification. Alternatively, if no relational contract is at hand, the two calls are abstracted using the canonical mutual summary $Sum_f : x_p = x_q \to r_p = r_q$ stating that equal inputs induce equal results. If a call cannot be matched, however, we have to use an 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.

[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.

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 by Hawblitzel et al. in [18]. 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 (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 suggested by Bjørner et al. in [8].

### 4.8 Assumptions

The presented regression verification approach strives to be as automatic as possible. To achieve this goal, we make simplifying assumptions about the programs.

Integer data types are not modeled as fixed-width bitvectors but as mathematical, unbounded integers. Our analyses are correct as long as no integer operation causes an overflow (or underflow). Likewise, for the analyses to be correct, programs must not show undefined behavior due to illegal memory accesses, division by 0, etc. Furthermore, our approach does not prove program termination, but silently assumes it.

While LLRêve focuses on regression verification, the above assumptions can be checked by other static analyses. For example, program termination can be checked by analyses such as by Falke et al. [12] or Giesl et al. [14].

Since LLRêve operates on compilation results produced by Clang, its verification results also apply to the intermediate code representation. Thus architecture-specific decisions made by the compiler play into the verification process (in particular fixing the bit-widths of integral types). Since we use the IR input rather on an abstract level (e.g., treating integers as mathematical integers), these decisions are less relevant. The verification results are faithful if one uses the same compilation framework to produce the executable code; then they are even closer to the actually executed code (as they make less assumptions about the compiler).

## 5 Exploiting Dynamic Analysis Data

The success of our relational verification approach relies on the capabilities of the Horn solvers to infer coupling predicates. Since this inference is expensive in practice, we developed two heuristic dynamic analysis techniques that improve the overall process in regard to both its efficacy and efficiency. Both techniques support static verification by exploiting data gathered during analysis of concrete program executions.

The first technique aims at harmonizing the iteration structure of loops between the two programs. It refactors two programs whose control flows are less correlated into two programs with more similar control flows. The second technique extracts interesting predicates based on the observed pairs of program traces and uses them as coupling predicate candidates for the general case. These candidates are produced from patterns and as algebraic equations.

### 5.1 Trace Collection

For the success of the dynamic approaches, it is crucial to collect suitable dynamic data from which the appropriate invariants can be extrapolated: In particular, the input values for trace collection must be well-chosen. The tool supports choosing the initial states either randomly or following a user-defined strategy. A fruitful strategy is taking the initial states from a test suite accompanying the program. The number of traces collected by LLRêve-dynamic is controlled by both global and per-trace resource limits. For instance, in the experiments presented below,

```
int dig10(int n) {
  int result = 1;
  n = n / 10;

  while (n > 0) {
    result++;
    n /= 10;
  }
  return result;
}
```

(a) Original version

```
int dig10(int n) {
  int result = 1;

  while (n > 0) {
    if (n < 10) return result;
    if (n < 100) return result+1;
    if (n < 1000) return result+2;
    if (n < 10000) return result+3;
    n /= 10000;
    result += 4;
  }
  return result;
}
```

(b) Optimized version

Fig. 5: Computing the number of digits in a decimal expansion of a non-negative number

efficiently detecting the best way to harmonize control flow was possible with ten trace pairs, while synthesizing coupling invariants took from two to 50 trace pairs.

To obtain trace data, we implemented a flexible special-purpose interpreter for the LLVM IR language. The alternative would have been to instrument the code and run it. However, this would have not have been flexible enough, as (1) the semantics of integers used in the instrumented code (bounded integers with fixed bit-width) would have been different from the one used by the constraint solvers (unbounded mathematical integers) resulting in inconsistencies, and (2) all traces would have needed to begin at the beginning of a program. The interpreter allows us also to start execution mid-program, which is needed to investigate counterexamples produced by a solver. The path that such a counterexample refers to starts at some synchronization point, which might not be located at the beginning of a program.

## 5.2 Harmonizing the Loop Iteration Structure

The rationale for harmonizing the loop iteration structure of two programs under comparison is the following: Relational verification based on coupling predicates works best if coupled states are similar. This similarity tends to increase if the number of iterations of a pair of corresponding loops in the two compared programs is equal (or almost equal).

We show how the differences in the iteration structure can be automatically reduced by applying code transformations to the two programs, namely loop peeling and loop unrolling. Specifically, we try to transform the programs in such a way that for each pair of corresponding loops the number of loop iterations is equal in both programs. If this goal is achieved, the constraint clauses for loosely synchronized loops can be removed, making the verification task easier for the Horn constraint solver.

### *5.2.1 Loop Unrolling*

The first harmonizing transformation that we support is *automatic loop unrolling*: Consider the two C functions given in Fig. 5. They both compute the number of digits in the decimal expansion of a non-negative integer $n$. Program (a) repeatedly divides by 10, while the optimized version (b) divides by 10,000 thus essentially reducing the number of expensive division operations by a factor of 4.

Yet, the equivalence of (a) and (b) is not immediately obvious. Regression verification as outlined earlier in this paper is theoretically possible but difficult due to the programs' different loop iteration structure. A coupling invariant would have to relate states where the loop in the second program has terminated, while the loop in the first one is still running. Such an invariant is more difficult or even impossible to infer *automatically* as it must encode significantly more of the functional aspects of the *individual* programs. In this case such an invariant would require using non-linear arithmetic.

If it were possible to compare four iterations of (a) against one of (b), a coupling predicate between the two programs would be easier to formulate and easier to automatically infer, as we have shown in [13] for a manually unrolled version. Requiring the user to specify the relationship of the loop iterations between the two compared programs would not tie in with our general idea of performing regression verification as automatically as possible. Hence, we compute this relationship heuristically by analyzing several execution traces of the two programs. We will come back to how to compute the number of times to unroll in Sect. 5.2.3.

In LLRêve, the loop unrolling transformation is carried out on the intermediate representation by duplicating basic blocks. On the level of C source code, the $k$-fold loop unrolling for a loop with a side-effect-free condition *cond* can be represented as follows:

$$\texttt{while}(\mathit{cond})\ \{\mathit{body}\} \overset{\textsc{unroll}(k)}{\Longrightarrow} \begin{array}{l}\texttt{while}(\mathit{cond})\ \{\\ \quad \left.\begin{array}{l}\texttt{if}(\mathit{cond})\ \{\mathit{body}\}\ \texttt{else break;}\\ \vdots \\ \texttt{if}(\mathit{cond})\ \{\mathit{body}\}\ \texttt{else break;}\}\end{array}\right\} k \text{ times}\end{array}$$

*5.2.2 Loop Peeling*

The second harmonizing transformation that we support is *loop peeling*, a technique often found in compiler optimization. It removes (i.e., peels) a constant number of iterations from a loop and places them before or after the loop. Since peeling iterations from the end of the loop generally requires non-trivial modifications of the loop condition, we only peel iterations from the beginning of a loop.

We demonstrate the effect of loop peeling by looking at the program in Listing 5, which computes the $n$-th triangular number $\sum_{k=1}^{n} k$. The first iteration of the loop actually has no effect, and the program is equivalent to its version where line 2 is replaced by `int i = 1`. But then, these two versions beginning with `i = 0` resp. `i = 1` are not strictly synchronized. The less efficient analysis for loosely coupled loops has to be used. Yet, if the first iteration of the loop in case of `i = 0` is peeled, then the loops *do* have the same number of iterations again.

Listing 5: An inefficient implementation for triangular numbers

```
1  int tria(int n) {
2    int i = 0;
3    int x = 0;
4    while (i < n) {
5      x += i;
6      ++i;
7    }
8    return x;
9  }
```

**Algorithm 1** Deciding when to unroll and when to peel

```
Input List of count pairs $S_n = (a_1, b_1), \dots, (a_s, b_s)$ for synchronization point $n$

$ratio \leftarrow \frac{1}{s} \sum_{i=1}^{s} \frac{a_i}{b_i}$        // avg. ratio of the number of iterations
$factor \leftarrow \mathrm{round}(ratio + \beta)$        // rounding with bias $\beta$
if $factor \neq 1$ then
    UNROLL($factor$)
else
    $diff \leftarrow \max\{a_i - b_i : 1 \leq i \leq s\}$        // maximum difference in the number of iterations
    PEEL($diff$)
end if
```

Like loop unrolling, loop peeling in LLRÊVE is performed on LLVM basic blocks. On C source code level, the $k$-fold peeling operation can be represented as follows:

```
while(cond) {body} ==PEEL(k)==>
                   bool exit=false;
                   if(!exit && cond) { body } else exit=true;  \
                   ⋮                                            } k times
                   if(!exit && cond) { body } else exit=true;  /
                   while(!exit && cond) { body }
```

Note that instead of nesting $k$ if-statements, we introduce an additional variable `exit`. By means of this variable, it is possible to make every path through the transformed code reach the loop. This is important, since only then can the peeled program be coupled against the unpeeled version in which the loop is also always reached.

### *5.2.3 Deciding When to Unroll and When to Peel*

It remains to be explained how it is heuristically decided whether a loop should be left untouched, unrolled or peeled, and by how many iterations this should be done. The data used in this decision is gathered from traces obtained by interpreting both programs on the same list of $s$ test input values. The number of traversals through synchronization points is recorded and stored in a list $S_n = (a_1, b_1), \dots, (a_s, b_s)$ for each synchronization point $n$. Here, $a_i$ is the number of traversals through `__mark(n)` in the first program and $b_i$ in the second.

If the numbers of iterations is small, then it is difficult to decide whether the numbers differ by an additive offset (which would call for peeling) or whether there is a multiplicative factor between the numbers (in which case unrolling would be appropriate). To remove the false impression gained from such instances, we ignore data points where the number of iterations is below some threshold.

Algorithm 1 shows the procedure that decides if a program is unrolled or peeled. For simplicity of presentation, we assume w.l.o.g. that $a_i \geq b_i$, i.e., that the first program iterates through the synchronization point at least as often as the second program.

First the average proportion *ratio* between the number of iterations for the two programs is computed. In order to find an appropriate unrolling factor, this mean value needs to be rounded. Since the ratio of loop iterations cannot be expected to be a constant, the value will be below the desired unrolling factor. Therefore, we add a bias $0 < \beta < 1$ before rounding. Our experiments have shown that $\beta = 0.4$ (which means, e.g., that values between 2.1 and 3.1 are rounded to 3) is a good value.

Table 1 shows the ratios and differences for test inputs for the programs in Fig. 5 and Listing 5. In case of the optimization of computing digits in (a), the ratios are between 3 and 3.75, their average is 3.375. Rounding up is the appropriate thing to do here. In case (b) the quotient is very close to 1, so rounding up to 2 (resulting in loop unrolling) is not wise. Instead (since the factor is 1), the one loop iteration is peeled from the beginning of the loop.

| $i$ | $n$ | $a_i$ | $b_i$ | $\frac{a_i}{b_i}$ |
|---|---|---|---|---|
| 1 | $10^{12}$ | 12 | 4 | 3 |
| 2 | $10^{13}$ | 13 | 4 | 3.25 |
| 3 | $10^{14}$ | 14 | 4 | 3.5 |
| 4 | $10^{15}$ | 15 | 4 | 3.75 |

(a) iteration table for Fig. 5

| $i$ | $n$ | $a_i$ | $b_i$ | $\frac{a_i}{b_i}$ | $a_i - b_i$ |
|---|---|---|---|---|---|
| 1 | 10 | 10 | 9 | 1.11 | 1 |
| 2 | 100 | 100 | 99 | 1.01 | 1 |
| 3 | 1000 | 1000 | 999 | 1.001 | 1 |
| 4 | 10000 | 10000 | 9999 | 1.0001 | 1 |

(b) iteration table for Listing 5

Table 1: Relationship of the number of iterations for test inputs

### 5.3 Finding Coupling Invariant Candidates

The key to regression verification is that the coupling predicates need not formally capture what result the two programs compute but encode what the relationship between the intermediate results is. If the programs are similar enough (e.g., after a local bug fix), it can be expected that the coupling between the programs' states can be expressed in a fragment of the logic.

This allows us to follow the promising approach of limiting the search for possible coupling invariants to specific and simple forms which can be explored more efficiently. Our experiments show that this can improve the performance of regression verification, both on the benchmarks from the C standard library (shown in Table 2), and even more so on examples with loops (shown in Table 3). Being based on the analysis of a finite set of execution traces, these methods only produce possible *invariant candidates*; their verification is delegated to an SMT solver.

We explored two complementary types of invariant candidate generation:

1. *Polynomial invariants.* Polynomial invariants are algebraic equations over local integer variables. They do not include statements over heaps and cannot express inequalities. However, they require no additional input from the user (except for the maximal degree of the polynomials) and their computation is more efficient than that for equivalent patterns.
2. *User-provided patterns.* The user can provide a collection of formula templates whose instantiations are conditions over local variables and heaps. The patterns can express a subset of first order logic over integers (including equalities,

inequalities, quantifiers, . . . ). But they must be specified manually, and a large number of patterns can slow down the invariant inference process significantly.

Invariant candidates of the two techniques can canonically be combined into a single, stronger invariant candidate by composing them conjunctively.

The candidates identified by the proposed techniques are afterwards submitted to an SMT solver for verification. To this end, the coupling predicate symbol $C_n$ in the Horn constraints is replaced by the concrete candidate $\varphi_n$ obtained for synchronization point $n$. If the resulting formula (without uninterpreted predicates) is not satisfied, a counterexample witnessing this effect may be returned. The data from this counterexample can be used as an additional input value in another round of dynamic analysis. This refines the candidates from verification attempt to verification attempt. If the candidate cannot be made more precise within the considered fragment, but is still not yet sufficient, we can incorporate the information from dynamic analysis into the process of Horn constraint solving by adding constraints of the form $C_n \to \varphi_n$ to the set of generated Horn clauses. Thus, the dynamically gathered candidate can contribute as a nucleus for the solution of the coupling predicate in the static analysis.

As the counterexample may provide values for arbitrary synchronization points within the program (not necessarily the entry point), our LLVM interpreter is able to perform program simulation from any point within a program, using the memory state encoded in the counterexample.

### *5.3.1 Disjunctive Invariant Candidates*

Both invariant inference methods find a set of conditions whose conjunction holds on all analyzed execution traces. However, we have found that in many cases, coupling predicates need to distinguish between different cases within the invariant which corresponds to disjunctive combinations of invariants. We use the following heuristic approach to find disjunctive invariant candidates. The set of program states collected from execution traces is partitioned according to the path the programs have taken. Each partition is then analyzed separately to infer an invariant. Finally, the derived invariant candidates for the disjoint cases are combined disjunctively into a single invariant candidate for all cases.

Our implementation creates three separate invariants by distinguishing three cases: The execution of both programs is synchronized or one program has reached the end of a loosely synchronized loop while the other program continues looping. The latter is separated in two separate cases depending on which program is still looping.

### *5.3.2 Polynomial invariants*

A polynomial (or algebraic) constraint in $n$ integer variables $x_1, \ldots, x_n$ is an atomic formula of the form $\sum_{e_1,\ldots,e_n} a_{e_1,\ldots,e_n} x_1^{e_1} \cdot \ldots \cdot x_n^{e_n} = 0$ for natural-number exponents $e_i$. For the sake of comprehensibility, we limit the presentation here to polynomials of degree 1, i.e., to equations of the form $a_0 + \sum_{1 \le i \le n} a_i x_i = 0$. Our implementation does not have this restriction.

In search of one algebraic invariant (or several invariants) for a set of states obtained from the analyzed execution traces, we can put up linear equations constraining the coefficients of the desired polynomial. Given the pair of program states at a synchronization point in which the local variables (of both programs) $(x_1, \ldots, x_n)$ have the values $(c_1, \ldots, c_n) \in \mathbb{Z}^n$ the equation $a_0 + \sum_{1 \le i \le n} a_i c_i = 0$ is added to the system of constraints on the coefficients. Letting $c_{i,j}$ denote the value of variable $j$ in equation $i$, one obtains a set of linear equations on the coefficients:

$$\begin{pmatrix} 1 & c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ 1 & c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & c_{m,1} & c_{2,m} & \cdots & c_{m,n} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = 0 \ , \tag{6}$$

in which the matrix is the multivariate Vandermonde matrix[5] of order 1 for the data points given by the considered program states. The solution space, i.e., the coefficient such that the linear polynomial is an invariant, is the kernel of this matrix. The dimension of the kernel can be larger than one, if the variables are linearly dependent (which is often the case in relational verification).

If one chooses any basis of the solution space, for each basis vector $(c_0 c_1 \ldots c_n)$ we create an algebraic constraint in the form of the equation $c_0 + c_1 * x_1 + \ldots + c_n * x_n = 0$. This equation holds on the examined execution traces. These constraints are then combined conjunctively to form the invariant candidate for the general case.

For performance reasons, the implementation can optionally limit the search space to polynomials of univariate summands (for each summand at most one $e_i$ differs from 0). In practice, linear summands are sufficient in most cases.

### *5.3.3 User-provided Patterns*

While polynomial invariants are the most important kind of invariants for equivalence checking and similar tasks, they have two shortcomings: (1) they are limited to equalities, and (2) they are limited to local integer variables and do not support array and heap accesses. Inequalities are nonetheless, in our experience, required in many practical instances of relational reasoning, even when the verification task is equivalence checking (let alone monotonicity checking). The importance of heap support is obvious, though its lack can, to some degree, be mitigated, if the relevant heap locations are known a priori. In this case, they can be treated as local variables and thereby made fit into the framework of polynomial invariants. This reduction does now work in the general case, though.

Hence, we deploy a second, more flexible source for invariant candidates alongside the polynomial engine. The main idea here is that candidates are built by instantiating a set of predefined patterns (formulas with free variables). The set of all possible instantiations in which the free variables are substituted by local program variables is the initial set of candidates. These instances are checked in the program states belonging to their corresponding synchronization points in the execution traces.

[5] Multiplication with a Vandermonde matrix evaluates a polynomial at a set of points and can be used to find interpolating polynomials. For a vector $(\alpha_1, ..., \alpha_m)$ of values, the $k$-th row of the univariate matrix of degree $n$ reads $\begin{pmatrix} 1 & \alpha_k & \alpha_k^2 & \ldots & \alpha_k^n \end{pmatrix}$ such that multiplication with the coefficient vector $(a_0 \ a_1 \ \ldots \ a_m)^t$ evaluates the polynomial $\sum_{i=0}^{m} a_i x^i$ for all $\alpha_i$. We use multivariate Vandermonde matrices of degree 1 here.

Table 2: Performance with different solvers for the libc benchmarks

| | | Run time with solver, seconds | | |
|---|---|---|---|---|
| Function | Source | Eldarica | Z3/duality | LLRêve-dynamic |
| memccpy | d/o | 0.736 | 0.123 | 0.099 |
| memchr | d/o | 0.338 | 0.080 | 0.073 |
| memmem | d/o | 1.03 | n/a | n/a |
| memmove | d/o | 14.96 | 0.223 | 0.189 |
| memrchr | g/o | 0.531 | 0.095 | 0.075 |
| memset | d/o | 0.387 | 0.103 | 0.080 |
| sbrk | d/g | 0.321 | 0.939 | n/a |
| stpcpy | d/o | 0.311 | 0.053 | 0.051 |
| strchr | d/g | t/o | 0.545 | t/o |
| strcmp | d/g | 0.901 | 0.093 | 0.108 |
| strncmp | g/o | 1.64 | 0.133 | t/o |
| strncmp | d/g | 1.869 | 0.196 | 0.076 |
| strncmp | d/o | 0.917 | 0.138 | 0.108 |
| strpbrk | d/o | t/o | 0.184 | 0.100 |
| strpbrk | d/g | 3.07 | 0.147 | 0.098 |
| strpbrk | g/o | 4.93 | 0.176 | 0.097 |
| swab | d/o | 10.33 | n/a | n/a |

d=dietlibc, g=glibc, o=OpenBSD libc.
t/o=timeout after 300 seconds.
n/a=unsupported because of external functions or custom preconditions
2 GHz i7-4750HQ CPU, 16 GB RAM.

The only limitation for the formulation of patterns is that any instantiation must be evaluatable given variable and heap assignments. Our current implementation supports comparisons on integers, array access and equality, quantification over bounded integer ranges and arbitrary Boolean combinations of these.

The surprisingly good performance of the "brute force" pattern method in practice can be explained by two factors. First, we have found that a small number of patterns is sufficient to verify large classes of examples, since the polynomial invariants already cover all equalities.[6] Second, the search space defined by trying all possible pattern instantiations is limited by the number of live program variables at a synchronization point (i.e., variables that are used in code reachable from this point), which is usually reasonably small.

## 6 Experiments

Our implementation of LLRêve and the dynamic extensions consists of ca. 17.3 KLOC of C++, building on LLVM version 3.9.0. It can be found online at `https://github.com/mattulbrich/llreve`.

In our experiments, we have proven equivalence across a sample of functions from three different libc implementations: dietlibc [23], glibc [15], and the OpenBSD libc [29]. Apart from the not yet automated placing of the synchronization marks, the proofs happen without user interaction. The average runtimes of the proofs are summarized in Table 2. One of the more complex examples, the function `memmove()`,

[6] For the libc benchmarks presented below, the patterns $heap_1 = heap_2$, $heap_1[\cdot] = \cdot$ and $heap_2[\cdot] = \cdot$ were sufficient. For the loop benchmarks, we used the patterns $\cdot \geqslant \cdot$, $\cdot > \cdot$ and $\cdot < 0$.

```
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. 6: `memmove()`

is shown in Fig. 6. It demonstrates the use of nested `if`s, multiple loops with different loop structures (`while`/`do-while`) and `goto` statements. While equally named functions are implemented similarly in the different libraries, the control flow differs from implementation to implementation, which can be observed in fact that non-trivial coupling invariants need to be inferred by LLRÊVE for the proofs.

Table 3: Performance with different solvers for artificial loop benchmarks

| | Run time with solver, seconds | | |
|---|---|---|---|
| Function | ELDARICA | Z3/DUALITY | LLRÊVE-DYNAMIC |
| barthe | 0.092 | 0.154 | 0.087 |
| barthe2 | 0.099 | 0.140 | 0.095 |
| barthe2-big | 0.179 | 2.000 | 0.159 |
| barthe2-big2 | 4.300 | 2.900 | 0.158 |
| barthe2-big3 | 4.200 | t/o | 0.299 |
| break | 0.097 | 5.700 | 0.083 |
| break-single | 0.101 | 0.088 | 0.064 |
| bug15 | 0.068 | 0.059 | 0.036 |
| loop2 | 0.079 | 0.075 | 0.067 |
| loop3 | 0.108 | 0.320 | 0.054 |
| simple-loop | 0.088 | 0.265 | 0.038 |

t/o=timeout after 10 seconds. 2 GHz i7-4750HQ CPU, 16 GB RAM.

```
void send(short *to,
          short *from,
          int count) {
    if (count <= 0) {
        return;
    }
    do {
        *to = *from++;
    } while (__mark(0) &
             (--count > 0));
}
```

(a) before

```
void send(short *to,
          short *from,
          int count) {
  if (count <= 0) {
    return;
  }
  unsigned n = (count + 7) / 8;
  switch (count %
  case 0:
    do {
      *to = *from++;
    case 7: *to = *from++;
    case 6: *to = *from++;
    case 5: *to = *from++;
    case 4: *to = *from++;
    case 3: *to = *from++;
    case 2: *to = *from++;
    case 1: *to = *from++;
    } while (__mark(0) & (--n > 0));
  }
}
```

(b) after

Fig. 7: Optimizing a loop with Duff's device

Revisiting the `memchr()` example discussed in Section 2, the early implementation of `memchr()` in dietlibc is known to have contained a bug (Listing 6). 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 6: 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, e.g., by Beyene et al. [7], most solver implementations currently only support universally quantified clauses. The libc implementations, however, were sufficiently similar so that such preconditions were not necessary.

By discovering unroll factors using dynamic analysis we have been able to prove that applying a specific form of loop unrolling called *Duff's device* [10] does not change the result of the program. The original and the transformed program are shown in Fig. 7. To generate the execution traces random inputs have been used. For the invariant inference we have used Z3. Discovering and applying the unroll factor of 8 and proving the programs equivalent takes about 1.2s.

## 7 Conclusion

We have shown how the automated relational reasoning approach presented in [13] 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.

Moreover, we demonstrated how dynamic data gathered from recorded program traces can be used to make the static verification more efficient and effective.

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. Balliu, M., Dam, M., Guanciale, R.: Automating information flow analysis of low level code. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14, pp. 1080–1091. ACM (2014)
2. Banerjee, A., Naumann, D.A., Nikouei, M.: Relational logic with framing and hypotheses. In: 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India, pp. 11:1–11:16 (2016). DOI 10.4230/LIPIcs.FSTTCS.2016.11. URL http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2016.11
3. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: 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, pp. 364–387. Springer-Verlag (2006)
4. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: M. Butler, W. Schulte (eds.) Proceedings, 17th International Symposium on Formal Methods (FM), *Lecture Notes in Computer Science*, vol. 6664, pp. 200–214. Springer (2011)
5. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Logical Foundations of Computer Science, International Symposium, LFCS 2013, San Diego, CA, USA, January 6-8, 2013. Proceedings, pp. 29–43 (2013). DOI 10.1007/978-3-642-35722-0_3. URL http://dx.doi.org/10.1007/978-3-642-35722-0_3
6. Beringer, L.: Relational decomposition. In: Proceedings of the 2nd International Conference on Interactive Theorem Proving (ITP), *Lecture Notes in Computer Science*, vol. 6898, pp. 39–54. Springer (2011)
7. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: N. Sharygina, H. Veith (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Proceedings, *Lecture Notes in Computer Science*, vol. 8044, pp. 869–882. Springer (2013)
8. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified Horn clauses. In: F. Logozzo, M. Fähndrich (eds.) Static Analysis - 20th International Symposium, SAS 2013, Proceedings, *Lecture Notes in Computer Science*, vol. 7935, pp. 105–125. Springer (2013)
9. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Relational verification through Horn clause transformation. In: Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, pp. 147–169 (2016). DOI 10.1007/978-3-662-53413-7_8. URL http://dx.doi.org/10.1007/978-3-662-53413-7_8

10. Duff, T.: Explanation, please! Online posting. Available at https://www.lysator.liu.se/c/duffs-device.html (1988)
11. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007). URL https://homes.cs.washington.edu/~mernst/pubs/daikon-tool-scp2007.pdf
12. Falke, S., Kapur, D., Sinz, C.: Termination analysis of imperative programs using bitvector arithmetic. In: Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments (VSTTE'12), pp. 261–277. Springer, Berlin, Heidelberg (2012)
13. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, pp. 349–360. ACM (2014)
14. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Automated termination proofs with AProVE. In: V. van Oostrom (ed.) Rewriting Techniques and Applications, 15th International Conference (RTA 2004), Proceedings, *Lecture Notes in Computer Science*, vol. 3091, pp. 210–220. Springer (2004)
15. GNU C library. https://www.gnu.org/software/libc/ (2016)
16. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference, DAC '09, pp. 466–471. ACM (2009)
17. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: D. Kroening, C.S. Pasareanu (eds.) Computer Aided Verification (CAV), Proceedings, *Lecture Notes in Computer Science*, vol. 9206, pp. 343–361. Springer (2015)
18. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: 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
19. Hawblitzel, C., Kawaguchi, M., Lahiri, S.K., Rebêlo, H.: Towards modularly comparing programs using automated theorem provers. In: M.P. Bonacina (ed.) Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7898, pp. 282–299. Springer (2013)
20. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing, SAT'12, pp. 157–171. Springer-Verlag, Berlin, Heidelberg (2012)
21. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SymDiff: A language-agnostic semantic diff tool for imperative programs. In: Proceedings of the 24th International Conference on Computer Aided Verification, CAV'12, pp. 712–717. Springer-Verlag, Berlin, Heidelberg (2012)
22. Lattner, C., Adve, V.: 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)
23. von Leitner, F.: diet libc. https://www.fefe.de/dietlibc/ (2016)
24. McMillan, K., Rybalchenko, A.: Computing relational fixed points using interpolation. Tech. Rep. MSR-TR-2013-6, Microsoft Research (2013). URL http://research.microsoft.com/apps/pubs/default.aspx?id=180055
25. Merz, F., Falke, S., Sinz, C.: 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, pp. 146–161. Springer-Verlag, Berlin, Heidelberg (2012)
26. Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica **1(2)**, 251–266 (1986)
27. Necula, G.C.: Translation validation for an optimizing compiler. SIGPLAN Not. **35**(5), 83–94 (2000). DOI 10.1145/358438.349314. URL http://doi.acm.org/10.1145/358438.349314
28. Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Dig: A dynamic invariant generator for polynomial and array invariants. ACM Trans. Softw. Eng. Methodol. **23**(4), 30:1–30:30 (2014). DOI 10.1145/2556782. URL http://doi.acm.org/10.1145/2556782
29. OpenBSD libc. http://cvsweb.openbsd.org/cgi-bin/cvsweb/src/lib/libc/ (2016)
30. Partush, N., Yahav, E.: Abstract semantic differencing via speculative correlation. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14, pp. 811–828. ACM (2014). DOI 10.1145/2660193.2660245

31. Rakamaric, Z., Emmi, M.: 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, pp. 106–113 (2014). DOI 10.1007/978-3-319-08867-9_7. URL http://dx.doi.org/10.1007/978-3-319-08867-9_7
32. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-clause verification. In: Proceedings of the 25th International Conference on Computer Aided Verification, CAV'13, pp. 347–363. Springer-Verlag, Berlin, Heidelberg (2013)
33. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: A. Biere, R. Bloem (eds.) 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, pp. 88–105. Springer International Publishing, Cham (2014). DOI 10.1007/978-3-319-08867-9_6. URL http://dx.doi.org/10.1007/978-3-319-08867-9_6
34. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: M. Felleisen, P. Gardner (eds.) Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, pp. 574–592. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). DOI 10.1007/978-3-642-37036-6_31. URL http://dx.doi.org/10.1007/978-3-642-37036-6_31
35. Smith, E.W., Dill, D.L.: Automatic formal verification of block cipher implementations. In: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design, FMCAD '08, pp. 6:1–6:7. IEEE Press, Piscataway, NJ, USA (2008). URL http://dl.acm.org/citation.cfm?id=1517424.1517430
36. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: Proceedings of the 23$^{rd}$ international conference on Computer Aided Verification, pp. 737–742. Springer-Verlag, Berlin, Heidelberg (2011). URL http://www.cs.cornell.edu/~ross/publications/eqsat/
37. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. SIGPLAN Not. **46**(6), 295–305 (2011). DOI 10.1145/1993316.1993533. URL http://doi.acm.org/10.1145/1993316.1993533
38. Ulbrich, M.: Dynamic logic for an intermediate language: Verification, interaction and refinement. Ph.D. thesis, Karlsruhe Institute of Technology (2013). http://nbn-resolving.org/urn:nbn:de:swb:90-411691
39. Verdoolaege, S., Janssens, G., Bruynooghe, M.: Equivalence checking of static affine programs using widening to handle recurrences. ACM Trans. Program. Lang. Syst. **34**(3), 11:1–11:35 (2012). DOI 10.1145/2362389.2362390
40. Verdoolaege, S., Palkovic, M., Bruynooghe, M., Janssens, G., Catthoor, F.: Experience with widening based equivalence checking in realistic multimedia systems. J. Electronic Testing **26**(2), 279–292 (2010)
41. Welsch, Y., Poetzsch-Heffter, A.: Verifying backwards compatibility of object-oriented libraries using Boogie. In: Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP '12, pp. 35–41. ACM (2012)
42. Zaks, A., Pnueli, A.: Covac: Compiler validation by program analysis of the cross-product. In: FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings, pp. 35–51 (2008). DOI 10.1007/978-3-540-68237-0_5. URL http://dx.doi.org/10.1007/978-3-540-68237-0_5
43. Zaks, A., Pnueli, A.: Program analysis for compiler validation. In: Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '08, pp. 1–7. ACM, New York, NY, USA (2008). DOI 10.1145/1512475.1512477. URL http://doi.acm.org/10.1145/1512475.1512477