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

Automating Regression Verification of Pointer Programs by Predicate Abstraction

Vladimir Klebanov · Philipp Rümmer · Mattias Ulbrich

Received: date / Accepted: date

Abstract Regression verification is an approach complementing regression testing with formal verification. The goal is to formally prove that two versions of a program behave either equally or differently in a precisely specified way. In this paper, we present a novel automated approach for regression verification that reduces the equivalence of two related imperative pointer programs to constrained Horn clauses over uninterpreted predicates. Subsequently, state-of-the-art SMT solvers are used to solve the clauses. We have implemented the approach, and our experiments show that non-trivial programs with integer and pointer arithmetic can now be proved equivalent without further user input.

1 Introduction

One of the main concerns during software evolution is to prevent the introduction of unwanted behavior, commonly known as *regressions,* when implementing new features, fixing defects, or during optimization. Undetected regressions can have severe consequences and incur high cost, in particular in late stages of development, or in software that is already deployed. Currently, the main quality assurance measure during software evolution is *regression testing* [3]. Regression testing uses a carefully crafted test suite to check that a modified version of a program is equivalent to the original one in relevant behavioral aspects.

Regression verification is a complementary approach that aims to achieve similar goals as regression testing with techniques from formal verification. This means

Vladimir Klebanov / Mattias Ulbrich
Institute for Theoretical Informatics
Karlsruhe Institute of Technology
Germany
E-mail: {klebanov, ulbrich}@kit.edu

Philipp Rümmer
Uppsala University
Sweden
E-mail: philipp.ruemmer@it.uu.se

establishing a formal proof of equivalence of the two program versions. 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, as testing has unique capabilities. Tests can, for instance, validate non-functional aspects of software (e.g., performance) or its interactions with the underlying software (and even hardware) layers. On the other hand, regression verification—especially if automated—is an attractive additional instrument of software quality assurance. If successful, it offers guaranteed full coverage, while not requiring additional expenses to develop and maintain a test suite.

At the same time, regression verification offers a more favorable pragmatics than the verification of functional properties of individual programs. For regression verification, one does not need to write and maintain complex formal specifications (which can be a significant bottleneck in the verification process). Furthermore, given two program versions that are both complex but similar to each other, much less effort is required to prove their equivalence than to prove that they satisfy an—also complex—functional specification. The effort for proving equivalence mainly depends on the difference between the programs and not on their overall size and complexity. Regression verification can exploit the fact that modifications are often local and only affect a small portion of a program.

A number of approaches and tools for regression verification exist already (see Section 8), but the majority of them are not automated, i.e., they require the user to supply inductive invariants (e.g., [9, 26, 38]). We present an approach and a tool for *automating* regression verification of imperative pointer programs. We use invariant generation techniques to infer sufficiently strong *coupling predicates*[1] between programs—and thus prove behavior equivalence. As we demonstrate in this paper, automation is possible in many cases where it was unavailable previously. If it fails (e.g., due to resource exhaustion), the user can still fall back to supplying the coupling predicates manually.

Our approach is targeted towards showing equivalence of programs with complex control flow and arithmetic on integers and pointers. This kind of programs is poorly supported by existing automation approaches, as these either require static (i.e., known at compile time) control flow [37], employ coarse abstractions on program computations [21,37], or are overly restrictive (e.g., require small bounds on loops or that equivalent unbounded loops have equivalent bodies) [33].

Our method works well whenever sufficiently "simple" coupling predicates exist that prove program equivalence. Simple means here that the inferred predicates are limited to linear arithmetic, and express heap properties following a certain quantification pattern. In Section 6 we demonstrate the effectiveness of our technique using a collection of small but non-trivial benchmarks.

To check larger systems in practice, the technique presented here would, of course, need to be embedded into a more general change analysis resp. incremental checking framework. An instance of such a framework is part of the RVT

[1] A coupling predicate is an inductive two-program invariant that relates the two programs throughout their execution. We are typically interested in coupling predicates that imply result equality upon termination of both programs.

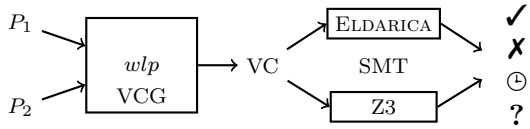

Fig. 1 Architecture of our approach

tool [21], where initially the smallest code block containing the change is checked for equivalence; only if its equivalence cannot be proven, one procedes with the block enclosing it, and so on. We plan to extend our approach and tool in this regard as part of future work.

In detail, the contributions of this paper are:

- A method for automating regression verification for imperative programs employing complex arithmetic on integer variables and pointers
- As part of the above, a method for computing efficient verification conditions for program equivalence
- A tool implementing the approach (available at `http://formal.iti.kit.edu/improve/reve1`).

This paper is an extension of our previous work [18] in two regards:

1. We introduce support for pointers to data stored on a heap and their manipulation. The corner stone of this development is inference of coupling predicates (of a certain shape) that contain quantifiers.
2. We improve the performance of the general abstraction-based inference procedure that we use for discovering coupling predicates by seeding the discovery with predicates that often occur during regression verification.

The architecture of our approach is shown in Figure 1 and can be described as follows: a frontend translates the two programs into efficient logical verification conditions (VC) for program equivalence using the algorithm presented in Section 3. The translation is completely automatic; the user does not have to supply the coupling predicates, loop invariants, or function summaries. Instead, placeholders for these entities are inserted into the VC formulae. The produced VC are in Horn normal form and are passed to an SMT solver for Horn constraints (such as Z3 [27] or ELDARICA [34]), as presented in Section 5. The solver tries to find a solution for the placeholders that would make the VC true. If the solver succeeds in finding a solution and thus inferring, among other things, a coupling predicate, the programs are equivalent. Alternatively, the solver may show that no solution exists (i.e., disprove equivalence) or time out; in case of programs with pointers, it is also possible that a solver terminates with an inconclusive result, since certain required transformation steps do not preserve completeness (Section 5.4).

1.1 Illustration

We begin with an example where programs operate on local variables only; an example with pointers will follow.

```
int g(int n) {
  int r = 0;

  if (n <= 0) {
    r = 0;
  } else {
    r = g(n-1) + n;
  }
  return r;
}
```

(a) basic version P_1

```
int g(int n, int s) {
  int r = 0;

  if (n <= 0) {
    r = s;
  } else {
    r = g(n-1, n+s);
  }
  return r;
}
```

(b) optimized version P_2

Fig. 2 Computing the n-th triangular number

Example 1 Consider the function $\mathtt{g}_1$ in Figure 2(a).[2] The function recursively computes the sum of integers in the interval $[1..n]$ (also known as the n-th *triangular number*). The function $\mathtt{g}_2$ in Figure 2(b) computes essentially the same result, but it has been optimized to employ tail recursion. The advantage of tail-recursive functions is that they can be executed without growing the stack. To enable this optimization, an accumulator parameter `s` has been added to the signature of $\mathtt{g}_2$ for collecting and passing on intermediate results. As another consequence, $\mathtt{g}_1$ performs summation from the end of the interval, while $\mathtt{g}_2$ starts from the beginning.

It is our goal to prove *automatically* using program verification technology that the two functions are equivalent in the sense that

$$\mathtt{g}_1(n) = \mathtt{g}_2(n,0) \text{ for any } n. \tag{1}$$

The *conceptually* simplest way to achieve this goal is to infer and compare a complete logical specification of each function, such as $\mathtt{g}_1(n) = \frac{n(n+1)}{2}$ for $n \geqslant 0$, and 0 otherwise. Yet, such a "brute-force" approach is clearly infeasible at the moment or in the foreseeable future.

Instead, we exploit the similarities between program versions and attempt to infer coupling predicates, i.e., a *relative* specification that only states how the executions of two versions relate to each other, but not what they compute in general. Of course, we demand that the coupling predicate is such that the two executions terminate in the same state, but in general, it will be a stronger assertion. To deal with unbounded recursion and loops, the predicate must be *inductive*: assuming that it holds at a key point of the computation (i.e., loop iteration, recursive call) should be sufficient to show that it also holds at the next key point. Conveniently, the complexity of such a relative specification is often proportional to the difference between the two program versions and not the program size.

Suppose, we want to calculate `g(5)`. We call $\mathtt{g}_1$`(5)` resp. $\mathtt{g}_2$`(5,0)`. The functions descend recursively, proceeding to compute $\mathtt{g}_1$`(4)+5` resp. $\mathtt{g}_2$`(4,5)`, and then $\mathtt{g}_1$`(3)+4+5` resp. $\mathtt{g}_2$`(3,9)`. At this point, one could suspect that at *every* recursion step

$$\mathtt{g}_1(n) + s = \mathtt{g}_2(n,s) \ . \tag{2}$$

[2] Our approach requires that the two programs which we prove equivalent have disjoint variable and function names. To distinguish equally named 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.

```
void memcpy(int *dst,
            int *src,
            int size) {
  int i = 0;
  while(i < size) {
    dst[i] = src[i];
    i++;
  }
}
```

(a) basic version P_1

```
void memcpy(int *dst,
            int *src,
            int size) {
  int *start = src;
  while(src - start < size) {
    *dst = *src;
    dst++;
    src++;
  }
}
```

(b) alternative version P_2

```
void memcpy(int *dst,
            int *src,
            int size) {
   src--;
   dst--;

   while(size > 0) {
      *++dst = *++src;
      size--;
   }
}
```

(c) alternative version P_3

Fig. 3 `memcpy()`: a function for copying memory

A simple induction proof can establish that our suspicion is indeed correct: assuming this relation for the callees in both functions allows us to prove the relation for the callers. Thus, the formula (2) is a valid coupling predicate. Fortunately, (2) also implies the desired equivalence for the top-level call (1) with $s = 0$. Indeed, if one knows or guesses the formula (2), then the fact that it is a valid coupling predicate and that it implies equivalence can be proved automatically with existing verification technology (cf., e.g., [9, 26, 38]).

In this paper, we show that it is actually in many cases possible to automatically *infer* coupling predicates that imply program equivalence. The Horn encoding of the VCs for the illustrative example is discussed in Section 5, and can be solved by Eldarica [34] in a fraction of a second, inferring the coupling predicate $n_1 = n_2 \rightarrow r_1 + s_2 = r_2$, where n_1 is the argument of $\mathtt{g}_1$, n_2 and s_2 are the arguments of $\mathtt{g}_2$, and r_i denote the respective return values. We note that the coupling predicate is linear even though the mathematical function computed by the two programs is non-linear.

Example 2 Consider a function `memcpy(int *dst, int *src, int size)`, which is a part of the standard C library. The function copies `size` bytes from the heap memory area pointed to by `src` to the heap memory area pointed to by `dst`. It is assumed that both areas do not overlap.

A basic implementation of `memcpy` is given in Figure 3(a), but many implementations are possible—and indeed used—depending on the target machine architecture. Figure 3(c), for instance, shows an implementation optimized for the PowerPC architecture, where special CPU instructions exist for loading and stor-

ing data on the heap in conjunction with preincrement of pointers.[3] Furthermore, reformulating the loop with a comparison to zero as loop condition improves the branch prediction performance of the CPU.

It is now our goal to prove that the different `memcpy` implementations behave equally in the sense that the postcondition

$$heap_1 = heap_2 \tag{3}$$

holds, where *heap* is a state-dependent map from pointers represented as integers to the heap data they point to. Intuitively, the postcondition states that after the execution of the two programs, the respective heaps contain identical values at every location. More details on heap modeling will be given in Section 4.

Since the programs contain loops, in order to prove (3), we need an appropriate coupling predicate, i.e., an inductive invariant relating both loop executions and entailing the postcondition upon termination. It is easy to see that (3) itself has to be part of the coupling predicate. Yet, (3) alone is not sufficient, as it is not inductive: assuming just (3) will not let us show (3) after one iteration of the loops.

Indeed, a coupling predicate for the equivalence of P_1 and P_2 is

$$(3) \wedge dst_1 + i_1 = dst_2 \wedge src_1 + i_1 = src_2 \wedge size_1 - i_1 = size_2 - src_2 + start_2$$

while for the equivalence of P_1 and P_3, the following coupling predicate suffices:

$$(3) \wedge dst_1 + i_1 = dst_2 + 1 \wedge src_1 + i_1 = src_2 + 1 \wedge size_1 - i_1 = size_2$$

These solutions for the equivalence VCs can be found by Eldarica in under one second.

2 Program Equivalence

This section introduces the considered programming language, and formalizes our notion of program equivalence in terms of Dijkstra's weakest preconditions [15]. The resulting program equivalence condition can be reduced to a program-free verification condition by applying reduction rules for weakest preconditions. A set of reduction rules optimized for equivalence proofs is defined in Section 3. Automation of the procedure is discussed in Section 5.

The Programming Language We consider deterministic imperative programs with unbounded integer variables (mathematical integers) and pointers, written in ANSI C notation. Determinism means that two program runs starting in the same state also terminate in the same state. Sequential programs are deterministic, provided that all variables are initialized before they are used (which can be efficiently checked by a compiler).[4] Furthermore, we require that all considered programs terminate

[3] Our parser currently does not support preincrement expressions, so we rewrite the relevant code fragment as `dst++; src++; *dst = *src;` when verifying with our tool.

[4] Sometimes, it is desirable to model the behavior of a subroutine using nondeterminism (e.g., for library functions or memory allocation). This can be done in our approach by providing according specifications for the subroutines.

for all inputs; this can be checked with one of the existing termination checkers for imperative programs, such as, e.g., [17, 19]. For simplicity, we assume that every function has at most one `return` statement, and, if it has one, then `return` is always the last statement in the function. The supported language fragment features integer-pointers, pointer dereferences and pointer arithmetic. Higher-dimensional pointers (i.e., pointers to pointers) are currently disregarded, but they could be easily added into the presented framework.

Programs may be composed of several functions, yet we assume that all programs have a distinguished function that is the entry point of the program. The entry point of the programs in our examples below is clear from the context.

Syntactical Conventions For reasons of presentation, we require that the programs P_1 and P_2 checked for equivalence have disjoint sets of variables. To distinguish equally named variables from the two programs, we add subscripts indicating the program version (1 or 2) to which they belong. We also establish the syntactic convention that program inputs (i.e., formal function parameters) are designated as $\bar{i}_1$ resp. $\bar{i}_2$, returned result variables as r_1 resp. r_2, and the vectors of all variables occurring in the programs as $\bar{x}_1$ resp. $\bar{x}_2$.

Background: Weakest Precondition Calculus Our reasoning about programs is formulated in terms of Dijkstra's weakest precondition calculus [15]. The *weakest precondition* predicate $wp(P, \varphi)$ denotes the weakest condition that needs to hold before an execution of statement list[5] P such that the execution terminates and the postcondition φ holds in the final state. The termination requirement is often considered optional. Relinquishing it, one obtains the *weakest liberal precondition* predicate $wlp(P, \varphi)$, which only demands that φ holds after the execution of P *if* P terminates. Thus, the formula $pre \rightarrow wlp(P, post)$ has the same intuitive meaning as the Floyd-Hoare triple $\{pre\}P\{post\}$.

A weakest precondition calculus is a set of rules which allow the resolution of wp/wlp predicates into formulae in pure first-order logic. Figure 4(a) lists a calculus for the wlp predicate for the considered programming language; the rules are standard, except that, for technical reasons, our calculus performs rewriting from the *beginning* of the statement list to its end, while a presentation with rules operating in the opposite direction is more customary. Reduction in forward direction is more convenient, however, for identifying structural similarity between the programs whose equivalence is verified. The calculus in Figure 4(a) is *complete* in the sense that every wlp-expression can be reduced to a pure first-order formula.

The rules (**??**), (7) and (8) allow the direct resolution of assignments, conditional statements and return statements (remember that the latter may only appear at the end of function bodies). The rule (9) for while loops is parametrized by a *loop invariant* $I(\bar{x}_1)$, a formula which needs to hold before the loop and must be preserved by the loop body under assumption of the loop condition. Likewise, the rule (10) for a (recursive) invocation b `= f1(`$\bar{a}$`)` of the function `f1` is parametrized by a *function summary predicate* $S_{\texttt{f1}}(\bar{a}, b)$ that relates the arguments $\bar{a}$ to the result value assigned to variable b. When the function summary $S_{\texttt{f1}}$ is used as abstraction

[5] To simplify presentation, we will use the terms "statement list" and "program" interchangeably. The exact relation will be clear from the context.

for the behavior of $\mathtt{f}_1$, the correctness of the summary has to be justified globally by an additional verification condition

$$wlp(P_1,\, S_{\mathtt{f1}}(\bar{i}_1, r_1)) \ , \tag{4}$$

in which P_1 is the function body of $\mathtt{f}_1$.

The invariant rule (9) and the recursive invocation rule (10) may approximate loop or function behavior depending on the chosen invariant or function summary. In this case, the formula derived by applying the rules will still be a correct precondition, but not necessarily the weakest one. Even when approximating, finding suitable loop invariants and summaries is in general a difficult task.

Stating Program Equivalence We consider two statement lists (usually the bodies of two functions) P_1 and P_2 *equivalent,* in writing

$$pre \to P_1 \simeq P_2 \ ,$$

when they behave equally (return the same value) for all inputs for which the precondition *pre* holds. We lift this notion to whole programs, by defining it as equivalence of the two program entry functions. The precondition *pre*, which can speak about variables from both P_1 and P_2, makes our notion of equivalence *conditional*. It is also possible to relax the equality between results to some other specified relation, yielding *relational equivalence*.

These notions can be formalized using the *wlp* predicate introduced above. Since we assume that P_1 and P_2 have disjoint vocabulary, their code can simply be combined sequentially. We define:

$$pre \to P_1 \simeq P_2 := \forall \bar{i}_1, \bar{i}_2.\ (\bar{i}_1 = \bar{i}_2 \wedge pre \to wlp(P_1\,;P_2,\, r_1 = r_2)) \ . \tag{5}$$

This kind of construction is known as *self-composition* [10, 13]. The weakest *liberal* precondition predicate has been used in this definition, since we deliberately abstract from termination issues in this paper.

3 Efficient Conditions for Program Equivalence

At this point, one could in theory directly resolve the *wlp* predicate in (5) by applying the rules from Figure 4(a) to obtain a first-order verification condition for equivalence of P_1 and P_2. However, the sequential composition of the two programs would require that they be analyzed individually without exploiting structural similarities between them.

Instead, we devise additional rules for the *wlp* predicate for the case that the program code given as the first argument is composed of two pieces with disjoint vocabulary. The disjointness allows us to use rules that would be unsound otherwise, as the statements with disjoint data cannot interfere with each other. The additional rules make use of two forms of *coupling predicates* that relate the states of the compared programs: *mutual invariants* C*,* which describe reachable states of two loops in the respective programs, iterating in a synchronized manner, and *mutual function summaries* R that express the relative behavior of two functions in the programs. The result of applying the new rules is a much more efficient first-order verification condition for equivalence.

$$wlp(\texttt{x = t}\,;P,\,\varphi) \rightsquigarrow \text{let}^6\ x = t \text{ in } wlp(P,\,\varphi) \tag{6}$$

$$wlp(\texttt{if(}\psi\texttt{) } T \texttt{ else } E\,;P,\,\varphi) \rightsquigarrow \text{if } \psi \text{ then } wlp(T\,;P,\,\varphi) \text{ else } wlp(E\,;P,\,\varphi) \tag{7}$$

$$wlp(\texttt{return } r,\,\varphi) \rightsquigarrow \varphi \tag{8}$$

$$wlp(\texttt{while(}\psi\texttt{) } B\,;P,\,\varphi) \rightsquigarrow I(\bar{x}_1) \wedge \forall \bar{x}_1.\big(\quad (I(\bar{x}_1) \wedge \psi \rightarrow wlp(B,\,I(\bar{x}_1))) \\ \wedge\, (I(\bar{x}_1) \wedge \neg\psi \rightarrow wlp(P,\,\varphi))\big) \tag{9}$$

$$wlp(r \texttt{ = f}_1\texttt{(}\bar{t}\texttt{)}\,;P,\,\varphi) \rightsquigarrow \forall r.\ S_{\texttt{f}1}(\bar{t}, r) \rightarrow wlp(P,\,\varphi) \tag{10}$$

(a) Conventional *wlp* calculus rules

(11)

$$wlp(P_1 \,;;\, P_2,\,\varphi) \rightsquigarrow wlp(P_2 \,;;\, P_1,\,\varphi) \tag{12}$$

$$wlp(\texttt{return } r \,;;\, P_2,\,\varphi) \rightsquigarrow wlp(P_2,\,\varphi) \tag{13}$$

$$wlp(\texttt{while(}\psi_1\texttt{) } B_1\,;P_1 \,;;\, \texttt{while(}\psi_2\texttt{) } B_2\,;P_2,\,\varphi) \rightsquigarrow C(\bar{x}_1,\bar{x}_2) \wedge \forall \bar{x}_1,\bar{x}_2.\big(\\ (C(\bar{x}_1,\bar{x}_2) \wedge \psi_1 \wedge \psi_2 \rightarrow wlp(B_1 \,;;\, B_2,\, C(\bar{x}_1,\bar{x}_2)) \\ \wedge (C(\bar{x}_1,\bar{x}_2) \wedge \neg\psi_1 \wedge \psi_2 \rightarrow wlp(B_2,\, C(\bar{x}_1,\bar{x}_2))) \\ \wedge (C(\bar{x}_1,\bar{x}_2) \wedge \psi_1 \wedge \neg\psi_2 \rightarrow wlp(B_1,\, C(\bar{x}_1,\bar{x}_2))) \\ \wedge (C(\bar{x}_1,\bar{x}_2) \wedge \neg\psi_1 \wedge \neg\psi_2 \rightarrow wlp(P_1 \,;;\, P_2,\,\psi))\big) \tag{14}$$

$$wlp(r_1 \texttt{ = f}_1\texttt{(}\bar{t}_1\texttt{)}\,;P_1 \,;;\, r_2 \texttt{ = f}_2\texttt{(}\bar{t}_2\texttt{)}\,;P_2,\,\varphi) \rightsquigarrow \\ \forall r_1, r_2.\ R_{\texttt{f1/f2}}(\bar{t}_1, r_1, \bar{t}_2, r_2) \rightarrow wlp(P_1 \,;;\, P_2,\,\varphi) \tag{15}$$

(b) Additional *wlp* calculus rules for independent programs

Fig. 4 Weakest precondition calculus

In Figure 4(b), we present the additional rules. To make the composition of two programs with disjoint vocabulary explicit, we use ;; instead of ; as separator between them. Semantically, both are equivalent. In particular, it is always sound to replace $P_1 \,;;\, P_2$ with $P_1\,;P_2$. Conversely, it is sound to replace $P_1\,;P_2$ with $P_1 \,;;\, P_2$ whenever P_1 and P_2 have disjoint vocabulary.

Rule (12) allows us to swap the two programs, thus enabling resolution of statements from both programs in an alternating fashion. The rule is sound since the statements of the two programs cannot possibly interfere; they have no common variables to refer to.

Together with the rules (**??**) and (7) of Figure 4(a), the swap rule allows us to resolve all statements but loops or recursion. These are the difficult cases since they require finding a suitable loop invariant or a function summary. The next two sections therefore introduce efficient rules for pair-wise loops and function calls. The *wlp* calculus can isolate the relevant loop pairs from within their programs even if they are embedded into enclosing conditionals or loops.

Proposition 1 (Soundness and Completeness)
Let Φ be a purely first-order formula derived from the condition $wlp(P_1 \,;;\, P_2, \varphi)$ by

[6] Here we use let expressions that the reader might be familiar with from functional programming. They allow reassigning existing variables, like let $x = x+1$ in $\ldots$. Variable substitutions are thus avoided and readabilty is improved.

rules from Figure 4(a) and (b). If the program $P_1 \,;\, P_2$ is started in a state satisfying the precondition Φ, and terminates, then φ holds in its final state. Furthermore, it is possible to choose suitable mutual invariants and summaries such that the derived formula is the weakest such precondition.

We give a justification for the validity of the proposition in the following.

3.1 While loops

We first consider equivalence of programs with loops, but without recursive function invocations. The loop rule for program equivalence is different from the rules discussed so far in that it talks about both programs at the same time and actually connects the two:

$$\begin{aligned} wlp(\texttt{while}(\psi_1)\ B_1 \,;\, P_1 \,;;\, \texttt{while}(\psi_2)\ B_2 \,;\, P_2,\, \varphi) \rightsquigarrow \\ C(\bar{x}_1, \bar{x}_2) \wedge \forall \bar{x}_1, \bar{x}_2 . \big(\\ (C(\bar{x}_1, \bar{x}_2) \wedge \psi_1 \wedge \psi_2 \rightarrow wlp(B_1 \,;;\, B_2,\, C(\bar{x}_1, \bar{x}_2)) \wedge \\ (C(\bar{x}_1, \bar{x}_2) \wedge \neg\psi_1 \wedge \psi_2 \rightarrow wlp(B_2,\, C(\bar{x}_1, \bar{x}_2))) \wedge \\ (C(\bar{x}_1, \bar{x}_2) \wedge \psi_1 \wedge \neg\psi_2 \rightarrow wlp(B_1,\, C(\bar{x}_1, \bar{x}_2))) \wedge \\ (C(\bar{x}_1, \bar{x}_2) \wedge \neg\psi_1 \wedge \neg\psi_2 \rightarrow wlp(P_1 \,;;\, P_2,\, \psi))\big) \ . \end{aligned}$$

The rule is parametrized by the mutual loop invariant $C(\bar{x}_1, \bar{x}_2)$, which is part of the coupling predicate that we are interested in. Unlike the invariant rule for a single program (9), which has two cases (loop condition holds or does not hold), this rule has four possible evaluations of the two loop conditions to consider.

For the justification of this rule, let us look at a particular reordering of the statements in the two loops. The central idea behind the rearrangement is that the two loops can be subject to a loop fusion resulting in the following program equivalence:

$$\begin{aligned} &\texttt{while}(\psi_1)\ B_1 \ ;;\ \texttt{while}(\psi_2)\ B_2 \quad \simeq \\ &\qquad\qquad \texttt{while}\ (\psi_1 \texttt{||} \psi_2)\ \{\ \texttt{if}(\psi_1)\ B_1\ \texttt{;}\ \texttt{if}(\psi_2)\ B_2\ \} \quad . \end{aligned} \tag{16}$$

Why is the single loop equivalent to the sequential execution of the separate loops? Running the two loops sequentially results in running the sequence of statements

$$(\underbrace{B_1, B_1, \ldots, B_1}_{n \text{ times}},\ \underbrace{B_2, B_2, \ldots, B_2}_{m \text{ times}}) \ ,$$

in which the first loop body B_1 is repeated n times followed by m repetitions of the second body B_2. Let w.l.o.g. the second loop be executed more often than the first in this schematic example (i.e., $m > n$). Due to disjoint vocabulary, loop body executions from different programs may be swapped. The run may hence be rearranged to

$$(\underbrace{B_1, B_2,\ B_1, B_2,\ \ldots, B_1, B_2}_{n \text{ times}},\ \underbrace{B_2, \ldots, B_2}_{m-n \text{ times}}) \tag{17}$$

without changing the semantics. One can make out m iterations now, of which the first n execute both loop bodies B_1, B_2, while the remaining $m - n$ rounds only

execute the second loop body B_2. The sequence (17) is a run for the fused loop from (16). It is the additional `if`-statements that ensure that bodies are only executed as often as they would be executed in a sequential execution. The disjunction in the guard ensures that the fused loop is iterated precisely as often as the maximum iterations of the individual loops.

Applying the traditional while *wlp* rule (9) to the fused loop from (16), has the same effect as applying the two-program rule (14). Since the traditional *wlp* calculus is sound and complete, our extension thus inherits these properties.

Mutual loop invariants are simpler than full functional invariants if the two programs are related. To show equivalence between a while loop and (a copy of) itself, for instance, the simple invariant $\bar{x}_1 = \bar{x}_2$ is sufficient regardless of what the loop computes.

Unlike for while loops, there is no special relational rule for coupled conditionals. Four cases have to be distinguished like for coupled loops, but since no invariant predicate is involved, this can also be achieved by consecutive appeals to the functional rule for conditionals.

3.2 Recursion

We now consider programs that have (recursive) function calls but no loops. Recursive calls of related functions in both programs can be abstracted by a single predicate, a *mutual function summary* (a term originated in [25]), that describes the relation between the arguments and result values of both invocations simultaneously and in relation to one another. The calculus rule to handle simultaneous function invocations is

$$wlp(r_1 \texttt{ = f}_1\texttt{(}\bar{t}_1\texttt{)} \,;\, P_1 \,;;\, r_2 \texttt{ = f}_2\texttt{(}\bar{t}_2\texttt{)} \,;\, P_2,\, \varphi) \rightsquigarrow \\ \forall r_1, r_2.\ R_{\texttt{f1/f2}}(\bar{t}_1, r_1, \bar{t}_2, r_2) \to wlp(P_1 \,;;\, P_2,\, \varphi)\ .$$

The rule is parametrized by the mutual function summary $R_{\texttt{f1/f2}}(\bar{t}_1, r_1, \bar{t}_2, r_2)$.

Abstracting function invocations with a mutual summary requires a (global) justification that the summary is a faithful abstraction, and we need to add the proof obligation

$$\forall \bar{i}_1, \bar{i}_2.\ wlp(P_1 \,;;\, P_2,\, R_{\texttt{f1/f2}}(\bar{i}_1, r_1, \bar{i}_2, r_2)) \tag{18}$$

to the verification conditions of equivalence. Here, P_1 and P_2 are the statement lists from the function bodies of the invoked functions $\texttt{f}_1$ and $\texttt{f}_2$.

The justification of rule (15) is as follows. Due to the disjointness of the program vocabulary, the statements in the rule can be reordered:

$$r_1\texttt{=f}_1\texttt{(}\bar{t}_1\texttt{)} \,;\, P_1 \,;;\, r_2\texttt{=f}_2\texttt{(}\bar{t}_2\texttt{)} \,;\, P_2 \quad \simeq \quad r_1\texttt{=f}_1\texttt{(}\bar{t}_1\texttt{)} \,;\, r_2\texttt{=f}_2\texttt{(}\bar{t}_2\texttt{)} \,;\, P_1 \,;\, P_2\ .$$

Condition (18) guarantees that $R_{\texttt{f1/f2}}(\bar{t}_1, r_1, \bar{t}_2, r_2)$ is a faithful abstraction of $P_1 \,;\, P_2$. Just as in the single-program case, it is thus sound to overapproximate the two recursive invocations with the mutual function summary.

As with mutual loop invariants, mutual function summaries are simpler than individual function summaries if the two programs are related. In case a recursive function is verified against (a copy of) itself, the simple mutual function summary $\bar{i}_1 = \bar{i}_2 \to r_1 = r_2$ can be used.

$$\begin{aligned} \mathit{select} &: \mathit{array} \times \mathit{Integer} \to \mathit{Integer} \\ \mathit{store} &: \mathit{array} \times \mathit{Integer} \times \mathit{Integer} \to \mathit{array} \end{aligned}$$

$$\forall a, i, j.\ \mathit{select}(\mathit{store}(a, i, v), j) = \begin{cases} v, & \text{if } i = j \\ \mathit{select}(a, j), & \text{otherwise.} \end{cases} \tag{19}$$

$$\forall a_1, a_2.\ (\forall i.\ \mathit{select}(a_1, i) = \mathit{select}(a_2, i)) \to a_1 = a_2 \tag{20}$$

Fig. 5 Theory of arrays [32]

Note that the same mutual summary $R_{\mathtt{f1/f2}}(\bar{t}_1, r_1, \bar{t}_2, r_2)$ is used for every occurrence of the pair $\mathtt{f}_1/\mathtt{f}_2$ of functions; this is in contrast to the coupling invariant rule (14) for loops, where it is possible to choose different mutual invariants $C(\bar{x}_1, \bar{x}_2)$ for every application. While our calculus could in principle be extended to support multiple mutual summaries per $\mathtt{f}_1/\mathtt{f}_2$ pair, the use of only a single such summary minimizes the number of required proof obligations (18).

4 Regression Verification for Programs with Pointers

To facilitate presentation, we have so far presented regression verification and our weakest precondition calculus for programs with integer variables only. We will now present the part of our method that is concerned with reading and writing heap data using pointers.

Programming Language We now incorporate pointers and pointer dereferences into the supported fragment of the programming language. That means that return values, function parameters, and local variables may now be of type `int *`, i.e., pointers to integer values on the heap.

We support the most important operations on pointers; in particular, the value at the address pointed to by `p` can be accessed using the expression `*p`, for both reading and writing. In addition, pointer arithmetic is possible: if `p` is a pointer expression, then `p + t` (with `t` an integer expression) is a pointer expression as well, and similarly `p - q` for two pointers `p` and `q` represents an integer expression. The array access notation `a[i]` can be used instead of `*(a+i)` both for reading and writing.

The Theory of Arrays On the logic side, we make use of McCarthy's theory of arrays to capture the semantics of memory access and modifications (Figure 5). The theory defines the operators *store* and *select*. Intuitively, $\mathit{store}(a, i, v)$ gives the array obtained by storing the value v at index i in array a, and $\mathit{select}(a, i)$ returns the value stored in array a at index i. The interplay of the operators is defined by the first axiom of the theory (Figure 5). Both Eldarica and Z3 natively support solving constraints over the theory of arrays, so that the same architecture as before can be used for regression verification for programs with arrays.

Modeling the Heap Our model of the heap captures the entire memory as a single one-dimensional array mapping pointers (integers) to the values stored on the heap (also integers). To model operations on the heap, we introduce an implicit

program variable *heap* of type *array* that holds the current memory state. Any access/modification of the heap is mapped to array operations on this program variable. For instance, a pointer dereference `*p` in the program corresponds to the selection operation $select(heap, p)$.

The wlp Calculus The *wlp* calculus in Figure 4 needs to be augmented by one rule which treats indirect assignment to a memory location and maps them to local variable assignments to the heap program variable using *store*:

$$wlp(\texttt{*x = t}\,;P,\,\varphi) \rightsquigarrow \text{let } heap = store(heap, x, t) \text{ in } wlp(P,\,\varphi) \tag{21}$$

In the presence of a heap, an invocation of a function may not only give rise to a return value but may also modify the heap memory. To model this fact within our framework, we assume that the *heap* variable is implicitly present as a parameter of each function, and that each function implicitly returns a tuple of *heap* and the explicit return value. This convention allows us to silently lift the rules (10) and (15) to heap-manipulating functions.

This modeling of the heap as a single array maintains both soundness and relative completeness of the approach.

Extended Notion of Program Equivalence In presence of heaps, it is no longer sufficient to ensure that the return values of the two functions are equal upon termination but it is necessary to also ensure that the heap has evolved equivalently as well, i.e., $heap_1 = heap_2$. This equality between maps can be reformulated in the manner that both heaps evaluate equally at all indexes according to the axiom of extensionality (20) in Figure 5. Hence, we refine the equivalence proof obligation (5) to

$$\begin{aligned} pre \to P_1 \simeq P_2 := \forall \bar{i}_1, \bar{i}_2.\ (\bar{i}_1 = \bar{i}_2 \wedge pre \to \\ wlp(P_1\,;P_2,\ r_1 = r_2 \wedge \forall x.\, select(heap_1, x) = select(heap_2, x))) \end{aligned} \tag{22}$$

in which x is an integer variable not occurring anywhere else.

5 Automating Equivalence Proofs

The application of the *wlp* rules in Figure 4 requires knowledge of specific predicates, namely loop invariants $I(\bar{i}_1, \bar{x}_1)$ in rule (9), mutual loop invariants $C(\bar{x}_1, \bar{x}_2)$ in (14), function summaries $S_{\texttt{f1}}(\bar{t}, s)$ in (10), and mutual function summaries $R_{\texttt{f1/f2}}(\bar{t}_1, r_1, \bar{t}_2, r_2)$ in (15). Together, those formulae represent the coupling predicate that witnesses program equivalence. Derivation of summaries and invariants is in general a complicated process and can require creativity and manual intervention. Thanks to the specialized *wlp*-rules for program equivalence, however, it is often possible to carry out equivalence proofs with comparatively simple predicates. In Section 1.1, for instance, it is possible to show the equivalence of programs computing non-linear functions with the help of just linear predicates; our experiments (Section 6) show that such simple predicates are sufficient for a wide range of realistic cases from regression verification.

We leverage recent methods for solving fixed-point constraints in order to compute required predicates without user intervention [23, 27, 34]. Such methods are

in principle incomplete, but they are effective for deriving predicates in practical cases arising from equivalence proofs.

5.1 Recursive Horn Clauses

In order to derive invariants and coupling predicates, verification conditions are represented in form of *Horn constraints* over (uninterpreted) relation symbols, including I, $C, S_{\mathtt{f1}}, R_{\mathtt{f}}$, and then solved with the help of model checking techniques like predicate abstraction and Craig interpolation. More generally, we fix a set $\mathcal{R}$ of uninterpreted fixed-arity *relation symbols,* and consider *Horn clauses* of the form $H \leftarrow \varphi \wedge B_1 \wedge \cdots \wedge B_n$, where:

- φ is a constraint over variables occurring in the clause; in our experiments, φ is always a formula in quantifier-free Presburger arithmetic, but extension to other theories (e.g., arrays) is possible;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of a symbol $p \in \mathcal{R}$ to first-order terms, or *false*.

H is called the *head* of the clause, $\varphi \wedge B_1 \wedge \cdots \wedge B_n$ the *body.* In case $\varphi = \mathit{true}$, we usually leave out φ and just write $H \leftarrow B_1 \wedge \cdots \wedge B_n$. First-order variables in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe of a first-order semantic structure. A set of Horn clauses HC over predicates $\mathcal{R}$ is called *solvable* if there is an interpretation of the predicates $\mathcal{R}$ as set-theoretic relations such the universal closure of every clause $h \in HC$ holds.

Example 3 (Example 1 continued) Figure 6 shows the equivalence VC for the programs from the illustration example (Figure 2) as Horn clauses. Here, R is the uninterpreted predicate symbol (placeholder) for the coupling predicate (mutual function summary) of $\mathtt{g}_1$ and $\mathtt{g}_2$ introduced by application of rule (15). The uninterpreted predicates $S_{\mathtt{g1}}$ and $S_{\mathtt{g2}}$ are the function summaries for the respective individual functions and are introduced by (10). Clauses with head *false* result from equivalence proof obligations (5), whereas the clauses with a head different from *false* are due to justification conditions (4) and (18).

5.2 Verification Conditions as Horn Clauses

For the encoding of verification conditions as Horn clauses, we assume that the set $\mathcal{R}$ contains symbols that can act as summaries for individual functions and function pairs (of appropriate arity), as well as relation symbols $I_1, I_2, I_3, \ldots$ and $C_1, C_2, C_3, \ldots$ to represent loop invariants:

$$\mathcal{R} \;=\; \{S_{\mathtt{f}}, R_{\mathtt{f1/f2}} \mid \mathtt{f}, \mathtt{f}_1, \mathtt{f}_2 \text{ functions}\} \cup \{I_1, I_2, I_3, \ldots, C_1, C_2, C_3, \ldots\} \;.$$

$$
\begin{array}{rcl}
\mathit{false} & \leftarrow & n_1 \leqslant 0 \wedge n_2 \leqslant 0 \wedge \\
& & n_1 = n_2 \wedge s_2 = 0 \wedge 0 \neq s_2 \\
\mathit{false} & \leftarrow & n_1 > 0 \wedge n_2 > 0 \wedge \\
& & n_1 = n_2 \wedge s_2 = 0 \wedge r_1 \neq r_2 \wedge \\
& & R(n_1 - 1, r_1, n_2 - 1, n_2 + s_2, r_2) \\
\mathit{false} & \leftarrow & n_1 > 0 \wedge n_2 \leqslant 0 \wedge \\
& & n_1 = n_2 \wedge s_2 = 0 \wedge r_1 \neq s_2 \wedge \\
& & S_{\mathtt{g1}}(n_2 - 1, r_1) \\
\mathit{false} & \leftarrow & n_1 \leqslant 0 \wedge n_2 > 0 \wedge \\
& & n_1 = n_2 \wedge s_2 = 0 \wedge 0 \neq r_2 \wedge \\
& & S_{\mathtt{g2}}(n_2 - 1, n_2 + s_2, r_2) \\
S_{\mathtt{g1}}(n_1, 0) & \leftarrow & n_1 \leqslant 0 \\
S_{\mathtt{g1}}(n_1, r_1 + n_1) & \leftarrow & n_1 > 0 \wedge S_{\mathtt{g1}}(n_1 - 1, r_1) \\
S_{\mathtt{g2}}(n_2, s_2, s_2) & \leftarrow & n_2 \leqslant 0 \\
S_{\mathtt{g2}}(n_2, s_2, r_2) & \leftarrow & n_2 > 0 \wedge S_{\mathtt{g2}}(n_2 - 1, n_2 + s_2, r_2) \\
R(n_1, 0, n_2, s_2, s_2) & \leftarrow & n_1 \leqslant 0 \wedge n_2 \leqslant 0 \\
R(n_1, r_1 + n_1, n_2, s_2, r_2) & \leftarrow & n_1 > 0 \wedge n_2 > 0 \wedge \\
& & R(n_1 - 1, r_1, n_2 - 1, n_2 + s_2, r_2)
\end{array}
$$

Fig. 6 Program equivalence VC as Horn clauses

We then consider the conjunction of the equivalence statement $\mathit{pre} \rightarrow P_1 \simeq P_2$ and the correctness of the summaries for all functions reachable from P_1 or P_2:

$$
\begin{array}{l}
\forall \bar{i}_1, \bar{i}_2.\ (\bar{i}_1 = \bar{i}_2 \wedge \mathit{pre} \rightarrow \mathit{wlp}(P_1 \mathbin{;;} P_2,\ r_1 = r_2)) \\
\wedge \bigwedge_{\substack{\mathtt{f}\ \mathrm{a} \\ \mathrm{function}}} \forall \bar{i}_{\mathtt{f}}.\ \mathit{wlp}(P_{\mathtt{f}},\ S_{\mathtt{f}}(\bar{i}_{\mathtt{f}}, r_{\mathtt{f}})) \\
\wedge \bigwedge_{\substack{\mathtt{f1},\mathtt{f2} \\ \mathrm{functions}}} \forall \bar{i}_{\mathtt{f1}}, \bar{i}_{\mathtt{f2}}.\ \mathit{wlp}(P_{\mathtt{f1}} \mathbin{;;} P_{\mathtt{f2}},\ R_{\mathtt{f1/f2}}(\bar{i}_{\mathtt{f1}}, r_{\mathtt{f1}}, \bar{i}_{\mathtt{f2}}, r_{\mathtt{f2}}))\ .
\end{array}
\tag{23}
$$

Intuitively, any valuation of the relation symbols $\mathcal{R}$ that makes (23) valid is a witness for the equivalence of P_1 and P_2, assuming *pre* holds initially.

The next step is the elimination of the *wlp* transformer from (23), by means of exhaustive application of the rules in Figure 4. When applying (10) or (15) to replace function calls $\mathtt{f}$, $\mathtt{f}_1, \mathtt{f}_2$ with the corresponding summary, the relation symbol $S_{\mathtt{f}}$ or $R_{\mathtt{f1/f2}}$ is inserted in the formula; similarly, when applying the loop rules (9) or (14), a fresh relation symbol I_k or C_k is introduced. We explain one possible strategy for applying the reduction rules below. Once application of the *wlp* rules to (23) has terminated, Horn clauses can be extracted from the reduct *VC* (a pure first-order formula), thanks to the following lemma:

Lemma 1 *Suppose VC resulted from exhaustive application of rules in Figure 4 to* (23)*. Then the clause normal form* VC_H *of VC is Horn.*

The clause normal form VC_H is derived by first distributing negations (*negation normal form*) in *VC*, then pulling out all universal quantifiers $\forall$ (*prenex normal form*), and finally transforming to *conjunctive normal form* [24]. To see that the

clause normal form VC_H is Horn, observe that (23) only contains *wlp* in positive positions, and that any two positive occurrences of relation symbols are separated by a conjunction; both properties are preserved by application of *wlp* rules, and entail that each clause in the clause normal form contains at most one positive relation symbol.

Reduction Strategy In some situations, it can happen that more than one rule in Figure 4 is applicable to a *wlp* expression, so that in principle more than one verification condition VC can be derived from (23). Different VCs can represent different ways to match up loops and corresponding function calls in the two programs checked for equivalence, and can therefore make the subsequent solving of the Horn constraints VC_H more or less difficult.

At the moment, we resolve such choice points using a greedy application strategy:

1. as long as possible, rules (**??**), (7), (8), (13) to eliminate assignments, conditionals, and return statements of the individual programs, possibly together with (12) to change the order of programs.
2. if no further rules from point 1 are applicable, try to use (14) or (15) for synchronous handling of loops or function calls; if this succeeds, go back to 1.
3. if no further rules from point 1 or 2 are applicable, use (9) or (10) to eliminate single loops or function calls; if this succeeds, go back to 1.

This strategy matches up loops and function calls in the order in which they occur in the considered programs. The strategy produces good results in our experiments, but can clearly be refined to take more sophisticated similarity measures into account. Further discussion is given in Section 6.

5.3 Solving Horn Clauses

A number of algorithms exist to solve the Horn clauses VC_H, including *predicate abstraction* [23, 34] and *property-directed reachability* (PDR, also known as IC3) implemented in Z3 [27]. The procedures attempt to construct a symbolic solution of VC_H in a decidable logic, for instance in (quantifier-free) Presburger arithmetic; such a solution maps every n-ary relation symbol in $\mathcal{R}$ to a symbolic predicate over n variables.

Example 4 (Example 3 continued) For the clauses in Example 3, the following predicates are found for the uninterpreted symbols:

$$\begin{aligned} R(n_1, r_1, n_2, s_2, r_2) &\mapsto (n_1 = n_2 \to r_1 + s_2 = r_2) \\ S_{\mathtt{g1}}(n_1, r_1) &\mapsto \mathit{true} \\ S_{\mathtt{g2}}(n_2, s_2, r_2) &\mapsto \mathit{true} \end{aligned}$$

which is the solution already discussed in Section 1.1. The function summaries $S_{\mathtt{g1}}$ and $S_{\mathtt{g2}}$ can be trivially chosen to be *true* since the Horn clauses in which they occur in the body are already valid without them.

In general, if it terminates, a Horn solver will produce one of two possible results: (i) a symbolic *solution* of the processed Horn clauses, or (ii) a concrete *counterexample tree* that witnesses that no solution of the Horn clauses exists. The leaves in a counterexample tree correspond to entry clauses (clauses without relation symbols in the body), the root of the tree to an assertion clause with head *false*; the counterexample shows that every attempt to satisfy the Horn clauses has to lead to one of the assertion clauses being violated.[7] Through additional bookkeeping and labeling, counterexamples can be translated back to runs of the programs P_1, P_2 that are checked for equivalence; the counterexample specifies the path taken through each program, as well as the values of all program variables.

We summarize by stating the correctness of our procedure, which is based on the following observation:

Lemma 2 *Regardless of the order in which wlp rules are applied, it is the case that* $pre \rightarrow P_1 \simeq P_2$ *holds if and only if* VC_H *is solvable (in the set-theoretic sense).*

Good strategies for the *wlp* rules are crucial to ensure termination of Horn solvers in practice. As illustrated in Section 1.1, it can (frequently) happen that linear expressions suffice to prove equivalence when searching for the right set of mutual invariants or post-conditions, while otherwise non-linear formulas might be necessary (and Horn solvers would likely diverge and time out).

For programs P_1, P_2 that are *not* equivalent—i.e., $pre \rightarrow P_1 \simeq P_2$ does not hold—the situation is simpler in the sense that there are always concrete counterexamples of VC_H witnessing non-equivalence. This is the case since there have to be concrete inputs for which P_1 and P_2 produce distinct results, and the executions of P_1 and P_2 could be translated to a counterexample of VC_H. A Horn solver might still fail to find counterexamples in practice, though in principle a simple enumeration strategy suffices to guarantee completeness for non-equivalence.

Theorem 1 (Correctness) *If a Horn solver applied to* VC_H *terminates, then one of the following holds:*

- *a solution is found for* VC_H*, and in this case the equivalence* $pre \rightarrow P_1 \simeq P_2$ *holds;*
- *a counterexample is found, and the programs are not equivalent.*

5.4 Horn Solving with Arrays

Horn solvers like Z3 or Eldarica directly support the datatype of arrays in Horn clauses to be solved, but require the user to specify the shape of solutions that should be derived. Arrays are typically handled with the help of the instantiation technique introduced in [12], which corresponds to a step-wise reduction to Horn clauses over simpler datatypes:

1. Literals $p(a_1, \ldots, a_n, t_1, \ldots, t_m)$ in Horn clauses that receive array expressions $a_1, \ldots, a_n$ as arguments are transformed to literals in which all arguments are

[7] Due to reasons of computability, sets of Horn clauses exist for which neither (i) nor (ii) can be returned: those are clauses that are solvable in a set-theoretic sense, but no solution can be expressed in the decidable language used for predicates. In such cases, usually non-termination occurs.

scalar (in our case, integer-valued), at the cost of introducing additional quantifiers. For instance, $p(a_1, \ldots, a_n, t_1, \ldots, t_m)$ can be replaced with the formula

$$\forall i_1, \ldots, i_n.\ p'(i_1, \mathit{select}(a_1, i_1),\ \ldots, i_n, \mathit{select}(a_n, i_n),\ t_1, \ldots, t_m) \qquad (24)$$

with a new relation symbol p' that observes the value of each array a_j at a universally quantified location i_j. Occurrences of p are uniformly replaced with the quantified p' in all Horn clauses.
Other replacement schemata are possible as well, for instance p' can be defined to read multiple locations of each array argument a_i, or the same index variable i_j can be used for multiple arrays. The transformation can therefore most meaningfully be done by the user, who often has domain knowledge about the schema necessary to solve a set of Horn clauses at hand.

2. The formulae obtained after the first reduction are not strictly Horn, since they can contain additional universally quantified literals in the body. Those quantified literals can be replaced with a conjunction of instances of the literal by means of e-matching [12,14]; for example, $\forall i.\ p'(i, \mathit{select}(a, i))$ can be replaced with the conjunction $p'(t_1, \mathit{select}(a, t_1)) \wedge p'(t_2, \mathit{select}(a, t_2))$ if $\mathit{select}(a, t_1)$ and $\mathit{select}(a, t_2)$ are relevant existing terms in a clause. After elimination of all universal quantifiers in this way, we again arrive at a set of Horn clauses, with the additional property that no relation symbol has array-valued arguments.
Solvers like Eldarica can perform this second reduction step automatically.
3. Finally, occurrences of the array operations *select* and *store* can be eliminated from all clauses using an Ackermann-like encoding, taking the array axioms (Figure 5) into account. For instance, a clause

$$q(i, \mathit{select}(\mathit{store}(a, j, 0), i), j) \leftarrow p(i, \mathit{select}(a, i))$$

can be replaced by the (equivalent) set of array-free clauses

$$q(i, x, j) \leftarrow p(i, x) \wedge i \neq j, \qquad q(i, 0, j) \leftarrow p(i, x) \wedge i = j\ .$$

Again, solvers like Z3 and Eldarica can perform this reduction automatically, producing Horn clauses that do not contain arrays any more and can be solved using methods like CEGAR (Section 7) or PDR [27].

The three transformations are *sound,* in the sense that solvability of the clauses produced in Step 3 implies solvability of the original clauses. Steps 1 and 2 are in general incomplete, however: even if the original clauses are solvable (over the theory of arrays), it can happen that the formulae after Step 1 or 2 do not have any solutions. Due to this overapproximation, the solver may return with a result of "unknown" in cases of spurious counter examples.

For the case of regression verification, we translate relation symbols with array-valued arguments to quantified literals in the style of (24), i.e., only one location is read for every array argument. This restricts the expressiveness in coupling predicates. Sortedness of an array can, e.g., not be be expressed since that would inherently involve reading two locations within one array (like $a[i] \leq a[i+1]$). For regression verification, this restriction is not severe, however, since the predicates inferred here are *relational* and our experiments show that in many cases access to one location suffices here even if the individual functional contracts would require reading more than one location.

5.5 Completeness Properties of the Approach

Our method exploits structural similarities between the compared programs, and can generally be expected to perform well when applied to programs with a high degree of similarity. In other situations, for instance when exchanging complete algorithms (e.g., replacing a bubble sort procedure with quicksort), or when changing the design of a system in a fundamental way, it is a lot less likely that an equivalence proof can be found automatically. In other words, the method works well whenever sufficiently "simple" coupling predicates exist that prove program equivalence, which is typical in the regression verification setting.

When abstracting from the concrete behaviour of Horn solvers (which are "best-effort," and might fail to discover the right predicates), we can identify several cases where our approach is complete, in the sense that the clauses produced in Section 5.4 are satisfiable. As a baseline, our procedure will always be able to prove that a program is equivalent to itself, by applying the greedy reduction strategy from Section 5.2, and choosing the coupling predicates $\bar{x}_1 = \bar{x}_2 \wedge (i_1 = i_2 \rightarrow \mathit{select}(\mathit{heap}_1, i_1) = \mathit{select}(\mathit{heap}_2, i_2))$. In addition, for this completeness property it is required that Step 2 in Section 5.4 introduces a sufficient set of instances of quantified literals, covering all accesses to the arrays modelling heap.

A second case in which we can observe completeness of the procedure are programs with the same control structure and locally equivalent (though not necessarily identical) loop and function bodies. In this case, the same coupling predicates $\bar{x}_1 = \bar{x}_2 \wedge (i_1 = i_2 \rightarrow \mathit{select}(\mathit{heap}_1, i_1) = \mathit{select}(\mathit{heap}_2, i_2))$ can be chosen for the entry points of the bodies.

6 Implementation and Experiments

We have implemented our approach for a language close to a subset of ANSI C in a tool named Rêve. Program data includes local variables and function parameters of type `int`, which is interpreted as unbounded (i.e., mathematical) integers, as well as pointers of type `int*`. Bounded integers can be simulated by instrumenting programs with modulo operations, at the cost of increased reasoning complexity. Supported control structures are if-then-else and while statements, function calls and returns. For simplicity, the return statement must always be the last statement of a function and must return a local variable. Recursive function calls may not occur within the conditions of if or while statements. Checking conditional and relational equivalence of programs is supported.

The tool (i.e., the *wlp* calculus) is implemented in Standard ML. As Horn constraint solvers we used Z3 (unstable branch as of 2014-11-03, using both PDR and Duality solver engines) and Eldarica (version v1.1-rc)[8]. In all cases, the Eldarica tool is used for clausification, i.e., to convert the result of the *wlp* calculus into a set of Horn clauses over integers (eliminating arrays as explained in Section 5.4).

[8] `https://github.com/uuverifiers/eldarica`

6.1 Experiments

We have evaluated the effectiveness and performance of our tool on a collection of benchmarks. The benchmarks vary in size from 16–53 lines of code (for both programs together, excluding comments or empty lines) and are available with the tool at the URL given in the introduction. Benchmark results are summarized in Table 1.

We give performance results for RÊVE with three different Horn solvers and also for the Regression Verification Tool (RVT) by Strichman and Godlin [21] (cf. Section 8), also operated by us. Dash (−) denotes timeout at 30 seconds, and cross (✗) denotes that the tool terminates but cannot prove equivalence . All times have been measured on a machine with a 2.80 GHz Intel Core i7 860 CPU.[9]

The programs in the first group in Table 1 are recursive, while the ones in the second group contain loops. The third group is concerned with pointer programs. Benchmarks where the two programs were not equivalent are in the fourth group, and their names end with a bang (`!`). All other benchmarks contain equivalent programs; the ✗ outcome is in this case a false negative.

Benchmarks `limit1` to `limit3` were given by Strichman and Godlin as beyond the limits of their approach to regression verification. Benchmarks `barthe2-big` and `barthe2-big2` embed the benchmark `barthe2` into a larger program that is syntactically identical in both versions. We could not prove equivalent the `ackermann` benchmark, as the result of a recursive function call is used as the argument to another recursive function call. Furthermore, we originally could not prove the `limit1` benchmark, as two steps of the first loop are equivalent to one step of the second loop, an issue that we solve in Section 6.3 and illustrate with the larger `digits10` benchmark. The `triangular-mod` benchmark corresponds to the illustrating example instrumented with modulo operations to simulate integer overflow.

The two pointer programs `memcpy` and `propagate` serve as examples in this paper. A small optimization of a selection sort implementation is considered in `selsort`; the swapping of two arrays `swaparray` is an example of conditional equivalence since the two algorithms are only equivalent if the arguments do not alias. In `cocome`, new behavior is added to an existing piece of code and the new revision behaves equivalently under conditions. The other pointer programs are variations of smaller algorithms.

As far as we are aware, RVT does not supply additional information to assist the user in case of a failed proof attempt. While, in theory, the model checker underlying RVT produces a counterexample, such a counterexample can be spurious due to the fixed abstraction employed. The ELDARICA solver that we use, in contrast, returns a genuine counterexample for many failed proofs (cf. Section 5). We found these counterexamples useful in diagnosing problems with the programs, even though we currently do not translate these counterexamples into source code terms.

[9] There are several reasons for the timings being different to those we previously reported in [18]. These include: different hardware, newer versions of all tools, and in particular ELDARICA now used in client-server mode, which avoids JVM startup and warmup costs.

Table 1 Benchmark results

			Run time (seconds)				
	Benchmark	LOC	RVT	RÊVE+ Z3/PDR	RÊVE+ Z3/Duality	RÊVE+ ELDARICA	Source
Recursion	ackermann	30	0.8	0.07	–	3.38	[21]
	mccarthy91	22	0.77	0.02	0.78	0.10	[21]
	limit1	22	✗	–	–	–	[21]
	limit2	22	0.79	–	0.15	0.10	[21]
	limit3	24	✗	–	0.22	0.15	[21]
	add-horn	26	✗	–	0.02	0.17	
	triangular	23	✗	–	–	0.13	
	triangular-mod	53	✗	–	–	–	
	inlining	20	✗	–	–	0.20	
Loops	simple-loop	16	✗	0.04	0.33	0.10	
	loop	22	✗	–	0.05	0.09	
	loop2	22	✗	–	0.04	0.10	
	loop3	28	✗	–	0.24	0.17	
	loop4	22	✗	–	–	–	
	loop5	22	✗	–	–	–	
	while-if	22	✗	–	0.08	0.11	
	digits10	32	✗	–	1.12	1.26	[1]
	barthe	28	✗	–	0.13	0.12	[9]
	barthe2	22	✗	–	0.16	0.15	[9]
	barthe2-big	32	✗	–	–	0.29	
	barthe2-big2	42	✗	–	–	0.44	
	bug15	26	1.09	0.10	0.09	0.09	[21]
	nested-while	28	1.2	–	0.13	0.19	[21]
Pointers	memcpy-a	15		–	0.26	0.45	
	memcpy-b	17		–	0.35	0.45	
	fib	25		0.09	–	–	
	propagate	16		0.27	0.28	1.46	
	clearstr	18		0.01	0.21	0.68	
	findmax	24		0.08	0.10	0.53	
	cocome	24		–	0.50	1.09	
	selsort	34		–	–	–	
	swaparray	20		–	0.62	3.35	
Not equivalent	ackermann!	30	✗	0.07	0.08	0.24	
	limit1!	22	✗	0.02	0.03	0.07	
	limit2!	25	✗	0.62	0.67	10.72	
	add-horn!	28	✗	0.03	0.02	0.11	
	triangular-mod!	49	✗	3.01	–	–	
	inlining!	20	✗	0.01	0.03	0.13	
	loop5!	22	✗	0.02	0.05	0.08	
	barthe!	31	✗	1.20	1.29	16.4	
	nested-while!	28	✗	0.06	0.04	0.17	

"–"=timeout, "✗"=unknown

6.2 Discussion of the Experiments

Studying Table 1, we can make a number of observations. The RVT tool does not perform well on our selection of benchmarks, which is because RVT is designed for comparison of larger programs, but with fixed coupling predicates expressing that the two programs are in the same state. Such predicates are not sufficient for almost all of our examples. On the other hand, as noted previously, for larger programs, a combination of the two approaches would make sense.

The next three columns (RÊVE+X) show that the choice of Horn solver has drastic impact on the performance of our approach. Z3/PDR turns out to be a bad choice of back-end for RÊVE, while Z3/Duality and ELDARICA are effective at solving the Horn problems resulting from regression verification. Since Z3/PDR is otherwise known to be a fast and effective Horn solver, we hypothesize that the effect is due to limited ability of Z3/PDR to discover relational predicates (predicates relating multiple program variables), which are essential as coupling predicates. Z3/PDR is very efficient, however, for disproving equivalence (the last group of benchmarks in Table 1). The generally good performance of ELDARICA can be explained with the range of heuristics for solving fixed-point equations included in the tool [31].

Several of the benchmarks are hard for all tools. This includes `triangular-mod` benchmarks, indicating that an encoding of bit-vectors as integers leads to quite hard Horn problems, and is probably not an option for larger programs; more intelligent approaches will have to be developed in the future.

The `loop4` and `loop5` benchmarks are programs in which loops cannot easily be synchronized, because the iteration speed or iteration order of the loops is different. We discuss in the next section how this issue can be overcome, albeit at the price of manual intervention.

The `fib` benchmark is a program with non-linear computation (the Fibonacci sequence), and turns out to be easier for Z3/PDR than for the other Horn solvers, for reasons not apparent to us. For `selsort`, although the compared versions of selection sort are almost identical, a relatively complex quantified mutual invariant about arrays is necessary to prove equivalence; finding this invariant automatically is beyond the capabilities of all considered tools.

6.3 An Example for Loop Equivalence

We consider a real-world example from [1]. The program P_1 in Figure 6.3(a) computes the number of digits in the decimal expansion of `n` through a series of integer divisions by 10. The program P_2 in Figure 6.3(c) computes the same result but (asymptotically) about seven times faster.

```
int f(int n) {
  int r = 1;
  n = n/10;

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

(a) basic version P_1

```
int f(int n) {
  int r = 1;
  n = n/10;

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

(b) intermediate version P_1'

```
int f(int n) {
  int r = 1;
  int b = 1;
  int v = -1;

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

(c) optimized version P_2

The programs P_1 and P_2 shown above are reformulations of those given in [1] in order to comply with the input requirements of our tool. The `do-while` and `for` loops have been replaced by `while` loops. The boolean flag `b` and the temporary storage variable `v` in P_2 have been introduced to avoid premature returns from the function.

Fig. 7 Computing the number of digits (`digits10`) from [1]

```
void propagate(int *a, int n) {
  int k = 0;
  while(k < n) {
    a[k+1] = a[k];
    k++;
  }
}
```

(a) version Q_1

```
void propagate(int *a, int n) {
  int k = 0;
  while(k < n) {
    a[k+1] = a[0];
    k++;
  }
}
```

(b) version Q_2

Fig. 8 Propagating the first value $a[0]$ to the entire array a (benchmark `propagate`)

This speedup is accomplished by reducing the strength of operations. The loop has been unrolled four times[10] *and* the majority of divisions have been replaced by pure comparisons.

Unsurprisingly, P_1 and P_2 cannot be proved equivalent automatically. To do so, the tool would in the least need to figure out the (very complex) relation between one iteration of the loop in P_1 and four iterations of the same loop. To overcome this barrier, the software engineer needs to supply to the tool the knowledge that an unrolling transformation took place. At the moment, we achieve this transfer by manually carrying out the unrolling on P_1 and producing the intermediate program P_1' shown in Figure 6.3(b). We then prove automatically that P_1' and P_2 are equivalent. Note that P_1' is still significantly different from P_2, as unrolling is not the only optimization that has been performed originally. The program P_1' still performs four times as many divisions as P_2. The if-conditions directly follow the divisions and depend on them, which slows the program down, while the four if-conditions in P_2 are all dependent on the same division result.

After slightly less than a second, Rêve with Eldarica succeeds in proving equivalence with the following automatically inferred coupling predicate:

$$\begin{aligned}&(b_2 = 1 \wedge r_1 = r_2 \wedge 10n_1 \leqslant n_2 \wedge n_2 \leqslant 10n_1 + 9)\\ \vee\ &(b_2 = 0 \wedge r_1 = v_2 \wedge n_2 \geqslant 10n_1 \wedge n_1 \leqslant 0)\ .\end{aligned}$$

Here, n_1 and r_1 denote the variables of P_1', and n_2, b_2, r_2 the variables of P_2. The variable b_2 indicates whether the loop will ($b_2 = 1$) or will not ($b_2 = 0$) be executed once more. The coupling predicate is hence a disjunction over these two cases: While the loop is iterated, r_1 and r_2 hold the same value and n_1 is one division by 10 ahead of n_2, i.e., $n_1 = n_2 \text{ DIV } 10$. Exactly this fact is expressed by the linear constraint $10n_1 \leqslant n_2 \wedge n_2 \leqslant 10n_1 + 9$. When the loop of P_2 has finished, its negated loop guard $n_1 \leqslant 0$ holds and the final results are stored in r_1 and v_2.

6.4 An Example for Pointer Program Equivalence

Consider the two programs from Figure 8 that both propagate the first value from the array to all n entries of the array. The original version in Figure 8(a) copies the previously written element $a[k]$ to the next location $a[k+1]$. The developer has decided to make the propagation of the first element $a[0]$ more explicit and

[10] Loop unrolling is a simple transformation, in which the loop body is replicated within the loop and guarded by the loop guard. This transformation preserves the semantics of the program.

changed the code to the one shown in Figure 8(b). They now want to make sure that the new revision is equivalent to the old one.

The functions are declared `void`; they do not return a result value, and the regression proof obligation boils down to showing that the heaps are equal after termination. A human specifier could come up with the manually crafted inductive coupling predicate[11]

$$k_1 = k_2 \wedge heap_1 = heap_2 \wedge select(heap_1, a_1 + k_1) = select(heap_2, a_2) \quad (25)$$

for the loops. The first two conjuncts state that the counting variables and the heaps are always equal between the revisions, and the third links the value of $a_1[k_1]$ in the first program to the value of $a_2[0]$ in the second program, thus implying that the propagation always writes the same value in both revisions.

In Section 5.4 we discussed that array arguments in Horn literals need to be transformed into integer arguments and that the coupling predicate must be in the shape of (24). And indeed, (25) can be equivalently formulated in this shape as

$$\forall i_1, i_2.\, k_1 = k_2 \wedge (i_1 = i_2 \rightarrow select(heap_1, i_1) = select(heap_2, i_2)) \wedge \\ (i_1 = a_1 + k_1 \wedge i_2 = a_2 \rightarrow select(heap_1, i_1) = select(heap_2, i_2))$$

which is also the invariant that RÊVE with ELDARICA finds automatically after slightly more than two seconds.

7 Tailoring Horn Solvers to Regression Verification

Using the encoding presented in the previous sections, in principle any Horn solver with support for relevant datatypes can be employed to perform regression verification. In practice, the scalability of the approach can be increased drastically by choosing solver parameters in a suitable way: as we observed, this applies in particular to the case of regression verification on relatively large programs, containing relatively few modifications, where it is beneficial to guide solvers towards existing simple solutions of the Horn clauses. In the following, we consider the case of Horn solvers that follow the "counterexample-guided abstraction refinement" (CEGAR) architecture in order to synthesize predicate abstractions of Horn clauses [23, 34]; similar optimizations apply to other algorithms.

7.1 Predicate Abstraction

Tools like ELDARICA [34] construct solutions of Horn clauses in disjunctive normal form by building an abstract reachability graph over a set of given predicates. When a counterexample is detected (a clause with consistent body literals and head *false*), a theorem prover is used to verify that the counterexample is genuine; spurious counterexamples are eliminated by generating additional predicates by means of Craig interpolation.

[11] For the ease of presentation, we left out the unmodified variables a_1, n_1, a_2, n_2 although they would appear in the predicate.

In order to define the concept of predicate abstraction, we assume that $\mathcal{R}$ is a set of relation symbols, and that for each predicate $p \in \mathcal{R}$ a vector $\bar{x}_p$ of formal argument variables has been fixed. Further, $\Pi : \mathcal{R} \to \mathcal{P}_{\text{fin}}(For)$ denotes a mapping from relation symbols $p \in \mathcal{R}$ to finite sets of formulae over the variables $\bar{x}_p$ used to approximate the relation symbol.

Given a set HC of Horn clauses, we define an *abstract reachability graph* (ARG) as a hyper-graph (S, E), where

- $S \subseteq \{(p, Q) \mid p \in \mathcal{R}, Q \subseteq \Pi(p)\}$ is the set of nodes, each of which is a pair consisting of a relation symbol and a set of predicates.
- $E \subseteq S^* \times HC \times S$ is a hyper-edge relation, with each edge being labelled with a clause. An edge $E(\langle s_1, \ldots, s_n \rangle, h, s)$, with $h = (H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n) \in HC$, implies that
  - $s_i = (p_i, Q_i)$ and $B_i = p_i(\bar{t}_i)$ for all $i = 1, \ldots, n$, and
  - $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$, where we write $Q_i[\bar{t}_i]$ for the conjunction of the predicates Q_i, with the formal arguments $\bar{x}_{p_i}$ replaced by the argument terms t_i.

An ARG (S, E) is called *closed* if the edge relation represents all Horn clauses in HC: for every clause $h = (H \leftarrow C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n)) \in HC$ and every sequence $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$ of nodes one of the following properties holds:

- $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \mathit{false}$, or
- there is an edge $E(\langle (p_1, Q_1), \ldots, (p_n, Q_n) \rangle, C, s)$ such that $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$.

Lemma 3 *A set HC of Horn clauses has a closed ARG (S, E) if and only if HC has a solution that can be expressed symbolically using formulae.*

A Horn solver proceeds starting from some initial mapping Π_0; in most cases, Π_0 will map every relation symbol to an empty set. The solver will then attempt to construct a closed ARG by means of fixed-point computation, which can either succeed (in which case a solution of the Horn clauses has been derived), or fail because some assertion clause $\mathit{false} \leftarrow C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n)$ is violated during the construction. In the latter case, a connected acyclic ARG fragment can be extracted that leads from entry clauses (clauses $H \leftarrow C$ without relation symbols in the body) to the violated assertion clause. The acyclic fragment represents a possible counterexample to the solvability of HC, and can be checked for *spuriousness* using classical SMT technology: if the counterexample is genuine, it has been shown that HC can in fact not be solved. Otherwise the ARG fragment can be translated to a Craig interpolation problem [34], and gives rise to new predicates and an extended predicate mapping Π_1. Subsequently, ARG construction is restarted, leading either to further counterexamples or eventually a closed ARG.

7.2 Seeding Predicate Abstraction for Regression Verification

Instead of starting from an empty predicate mapping Π_0, a set of meaningful initial predicates can be chosen as a seed. This can lead to a reduced number of refinement steps, or prevent sub-optimal predicates from being produced by interpolation. In the context of regression verification, where typically two programs are compared

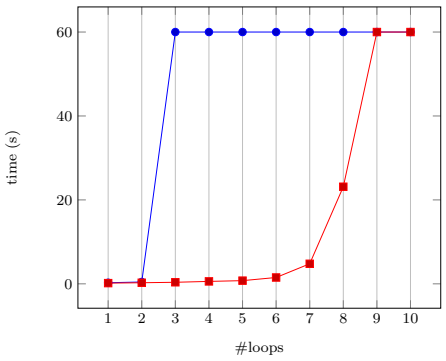

Fig. 9 Efficacy of abstraction seeding for regression verification of mostly identical programs

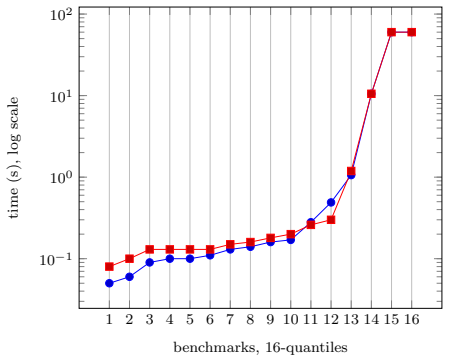

Fig. 10 Efficacy of abstraction seeding for regression verification of loop benchmarks from Section 6

that are to a large degree identical, likely coupling predicates are equations $y_1 = y_2$ that relate corresponding variables y_1 and y_2 of the programs to each other. It is meaningful to start the CEGAR process with a predicate mapping Π_{eq} that already contains such predicates; in particular, we can observe that the equivalence of a program with itself can in this way be shown without any refinement steps.

For every coupling loop invariant C with arguments $\bar{x}_1, \bar{x}_2$, and every coupling function summary $R_{\texttt{f1/f2}}$ with arguments $\bar{i}_1, r_1, \bar{i}_2, r_2$, we choose the initial predicates as follows:

$$\Pi_{eq}(C) = \{y_1 = y_2 \mid y_1 \text{ a variable in } \bar{x}_1, \text{ and } y_2 \text{ corresponding variable in } \bar{x}_2\}$$
$$\Pi_{eq}(R_{\texttt{f1/f2}}) = \{y_1 = y_2 \mid y_1 \text{ a variable in } \bar{i}_1, r_1, \text{ and } y_2 \text{ corresponding variable in } \bar{i}_2, r_2\} .$$

If an encoding of arrays as in (24) is used, it is usually meaningful to add *negated* equations for corresponding quantified variables in $i_1, \ldots, i_n$; this is because formulae $i_1 = i_2 \rightarrow \mathit{select}(a_1, i_1) = \mathit{select}(a_2, i_2)$ are likely coupling predicates in this case.

7.3 Efficacy of Abstraction Seeding

Figures 9 to 11 demonstrate the effects of abstraction seeding as presented above. We have evaluated seeding on three benchmark sets. The first set contains programs that are to a large extent identical. The set was created by beginning with the `barthe2` benchmark and progressively adding identical loops to both versions of the function. The biggest benchmark solvable within 60 seconds contains 8 loops and a total of 970 LOC. The results clearly demonstrate the efficacy of predicate seeding for regression verification of large programs with small changes (Figure 9).[12]

[12] As explained in the introduction, in practice, one would not be checking the equivalence of a large system all at once, though.

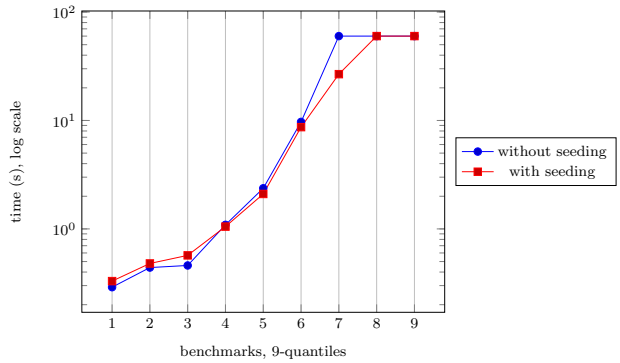

Fig. 11 Efficacy of abstraction seeding for regression verification of pointer benchmarks from Section 6

The second and the third benchmark sets correspond to the loop and the pointer benchmarks from Section 6 respectively. Figures 10 and 11 show that seeding can impose an insubstantial penalty (consider that the scale is logarithmic), but also can offer significant benefits in individual cases. This behavior is not surprising as the programs in these benchmark sets contain hardly any identical code and the initial predicates used as the seed are not as useful to express the required coupling predicates.

8 Related Work

Research on proving program equivalence is driven by a variety of applications, including security verification, compiler optimizations, backwards compatibility and refactoring, cryptographic algorithms, hardware design, and general-purpose regression verification.

Godlin and Strichman [20–22] present an approach for automating general-purpose regression verification. In this approach, loops in the programs are transformed to recursive procedures, and matching recursive calls are abstracted by an uninterpreted function. The equivalence of functions (that no longer contain recursion) is then checked by the CBMC model checker. In our vernacular, the approach can be described as an attempt to verify equivalence with the fixed coupling predicate $\overline{i}_1 = \overline{i}_2 \rightarrow r_1 = r_2$ for every related pair of recursive functions. This abstraction imposes the limitation that function calls with different arguments or a different number of recursions of two matching recursive functions are not supported. The technique is implemented in the RVT tool and supports a subset of ANSI C.

A part of the approach by Godlin and Strichman [21] is a technique to deal with programs manipulating tree-shaped heap structures. A key part of this technique involves feeding both program versions inputs that point to isomorphic nondeterministic structures of a limited depth. A conservative sound depth can be syntactically inferred from the function bodies by counting the number of dereferences. This approach does not work in presence of pointer arithmetic, though, which is given in our case.

Verdoolaege et al. [36, 37] have developed an automation 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. Initially, dataflow analysis is applied to build a dependence graph abstraction of each of the two programs. Then the equivalence hypothesis for outputs is propagated through the graphs towards the inputs, in a manner resembling verification condition generation. The static control flow requirement means that the control flow of the program must be known already at compile time. Furthermore, arithmetical operations in the loop/function bodies are abstracted. Addition is, e.g., replaced by an associative and commutative uninterpreted function. The abstraction prevents proving equivalence of such programs as `x=x+1; x=x+1;` and `x=x+2;`.

Barthe et al. [9] 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 existing verification technology (the WHY tool, in that particular case). The verification process still requires user-supplied annotations though.

Almeida et al. [2] have verified the correctness of the OpenSSL implementation of the RC4 cipher w.r.t. a reference implementation. The authors use self-composition of programs together with interactively verified lemmas about particular program transformations and optimizations.

Sinz and Post [33] prove equivalence of two AES cipher implementations by means of bounded model checking. The approach unrolls resp. inlines all loops and recursive calls. Such reasoning is only feasible if the program admits small bounds on loops or depth of recursive calls. In the case of AES, a complete unrolling of the main loop was not possible, so the authors proved equivalence of loop bodies instead.

Backes et al. [5] propose to leverage slicing and impact analysis to improve scalability of regression verification. The idea is to subject both program versions to a dependency analysis, then to remove the code present in both versions that has no data or control dependencies on the introduced change, and to apply an existing technique (e.g., bounded symbolic execution) to show equivalence of the reduced programs.

Mutual function summaries have been prominently put forth by Hawblitzel et al. in [25] and later developed in [26]. The concept is implemented in the equivalence checker SYMDIFF [29], where the user supplies the mutual summary, and the verification conditions are discharged by BOOGIE. Loops are encoded as recursion. The BCVERIFIER tool for proving backwards compatibility of Java class libraries by Welsch and Poetzsch-Heffter [38] has a similar pragmatics.

In [30], the SYMDIFF tool was combined with the Houdini invariant generation algorithm to infer coupling predicates for regression verification of memory safety properties. Houdini attempts to construct a consistent set of inductive conjunctive invariants by "brute-force" elimination from a pool of candidates built by instantiating a user-specified template. Time performance data is not reported in [30].

The *Replace Isomorphism with Equality method* presented in [39] supports automatic equivalence proving for programs with heaps and memory allocations. It can

deal with out-of-order coupling of call sites and heap equivalence up to isomorphism (similar to [11] for non-interference). The tool uses a fixed mutual summary and verifies equivalence using Z3 via an encoding to Boogie.

Banerjee and Naumann [6,7] study equivalence of Java-like programs from the perspective of data encapsulation. They develop a programming discipline and a static analysis ensuring that changes in an object-oriented data structure's implementation are confined and cannot affect its clients other than through specified public methods.

Several relational program logics (e.g., [4, 8, 35]) have been developed for security applications. Proving in these logics requires user-supplied inductive invariants.

A large body of work also exists on equivalence checking of hardware logic circuits; see [28] for an overview. The approaches fall into two major groups. One group builds the product machine of two circuits and exhaustively traverses the state space to ensure that the corresponding outputs of the two circuits are identical in every reachable state. The other group recognizes that the incremental nature of the design process induces structural similarity between the circuit variants under verification and tries to exploit them. The techniques to do so include functional equivalences, indirect implications, permissible functions, and others (see e.g., [16]).

9 Conclusion and Future Work

In this paper, we have presented a novel approach that uses invariant inference techniques for automating regression proofs for two imperative pointer programs. To this end, the two versions of the program are transformed into Horn clauses over uninterpreted predicate symbols. These clauses constrain equivalence-witnessing coupling predicates that connect the states of the two programs at key points. A Horn constraint solver is used to find a solution for the coupling predicates, if one exists.

The approach is implemented and we have demonstrated its effectiveness on pointer programs with non-trivial arithmetic and control flow. Future work includes support of further programming language constructs, as well as improvements to scalability, e.g., by combination with other regression verification techniques.

Acknowledgments

This work was partially supported by the German National Science Foundation (DFG) under the IMPROVE project within the priority program SPP 1593 "Design For Future – Managed Software Evolution", and by the Swedish Research Council.

References

1. Alexandrescu, A.: Three optimization tips for C++ (2012). A presentation at Facebook NYC. Available at `www.facebook.com/notes/facebook-engineering/three-optimization-tips-for-c/10151361643253920`

2. Almeida, J., Barbosa, M., Sousa Pinto, J., Vieira, B.: Verifying cryptographic software correctness with respect to reference implementations. In: M. Alpuente, B. Cook, C. Joubert (eds.) Formal Methods for Industrial Critical Systems, *Lecture Notes in Computer Science*, vol. 5825, pp. 37–52. Springer Berlin / Heidelberg (2009)
3. Ammann, P., Offutt, J.: Introduction to Software Testing, first edn. Cambridge University Press, New York, NY, USA (2008)
4. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '06, pp. 91–102. ACM, New York, NY, USA (2006)
5. Backes, J., Person, S., Rungta, N., Tkachuk, O.: Regression verification using impact summaries. In: E. Bartocci, C. Ramakrishnan (eds.) Model Checking Software, *Lecture Notes in Computer Science*, vol. 7976, pp. 99–116. Springer Berlin Heidelberg (2013)
6. Banerjee, A., Naumann, D.A.: Ownership confinement ensures representation independence for object-oriented programs. J. ACM **52**(6), 894–960 (2005)
7. Banerjee, A., Naumann, D.A.: State based ownership, reentrance, and encapsulation. In: Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05, pp. 387–411. Springer-Verlag, Berlin, Heidelberg (2005)
8. Barthe, G., Crespo, J., Grégoire, B., Kunz, C., Zanella Béguelin, S.: Computer-aided cryptographic proofs. In: L. Beringer, A. Felty (eds.) Interactive Theorem Proving, *Lecture Notes in Computer Science*, vol. 7406, pp. 11–27. Springer Berlin Heidelberg (2012)
9. 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)
10. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA, pp. 100–114. IEEE Computer Society (2004)
11. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: G. Gupta, R. Peña (eds.) 23rd International Symposium on Logic-Based Program Synthesis and Transformation, (LOPSTR 2013), pp. 15–32. Dpto. de Systemas Informáticos y Computation, Universidad Complutense de Madrid, TR-11-13 (2013)
12. 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, Seattle, WA, USA, June 20-22, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7935, pp. 105–125. Springer (2013). DOI 10.1007/978-3-642-38856-9_8
13. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Proceedings of the Second International Conference on Security in Pervasive Computing, SPC'05, pp. 193–209. Springer-Verlag, Berlin, Heidelberg (2005)
14. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM **52**(3) (2005)
15. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8), 453–457 (1975). DOI 10.1145/360933.360975. URL `http://doi.acm.org/10.1145/360933.360975`
16. van Eijk, C.: Sequential equivalence checking based on structural similarities. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **19**(7), 814–819 (2000)
17. 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-Verlag, Berlin, Heidelberg (2012)
18. 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)
19. 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)
20. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Inf. **45**(6), 403–439 (2008)

21. Godlin, B., Strichman, O.: Regression verification. In: Proceedings of the 46th Annual Design Automation Conference, DAC '09, pp. 466–471. ACM (2009)
22. Godlin, B., Strichman, O.: Regression verification: proving the equivalence of similar programs. Software Testing, Verification and Reliability **23**(3), 241–258 (2013). DOI 10.1002/stvr.1472
23. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, pp. 405–416. ACM (2012)
24. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
25. 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`
26. 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, Lake Placid, NY, USA, June 9-14, 2013. Proceedings, *Lecture Notes in Computer Science*, vol. 7898, pp. 282–299. Springer (2013)
27. 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)
28. Huang, S.Y., Cheng, K.T.: Formal Equivalence Checking and Design DeBugging. Kluwer Academic Publishers, Norwell, MA, USA (1998)
29. 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)
30. Lahiri, S.K., McMillan, K.L., Sharma, R., Hawblitzel, C.: Differential assertion checking. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 345–355. ACM (2013)
31. Leroux, J., Rümmer, P., Subotic, P.: Guiding Craig interpolation with domain-specific abstractions. Acta Inf. **53**(4), 387–424 (2016). DOI 10.1007/s00236-015-0236-z
32. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28 (1962)
33. Post, H., Sinz, C.: Proving functional equivalence of two AES implementations using bounded model checking. In: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, ICST '09, pp. 31–40. IEEE Computer Society (2009)
34. 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)
35. Scheben, C., Schmitt, P.H.: Efficient self-composition for weakest precondition calculi. In: C.B. Jones, P. Pihlajasaari, J. Sun (eds.) Proceedings, 19th International Symposium on Formal Methods (FM), *Lecture Notes in Computer Science*, vol. 8442, pp. 579–594. Springer (2014)
36. 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
37. 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)
38. 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)
39. Wood, T., Drossopoulou, S., Lahiri, S.K., Eisenbach, S.: Modular verification of procedure equivalence in the presence of memory allocation. In: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, pp. 937–963 (2017). DOI 10.1007/978-3-662-54434-1_35