Automating Regression Verification

Dennis Felsing†
dennis.felsing@student.kit.edu
Vladimir Klebanov†
klebanov@kit.edu
Philipp Rümmer‡
philipp.ruemmer@it.uu.se
Mattias Ulbrich†
ulbrich@kit.edu
Sarah Grebing†
sarah.grebing@kit.edu

†Karlsruhe Institute of Technology, Germany
‡Uppsala University, Sweden

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 automatic approach for regression verification that reduces the equivalence of two related imperative integer programs to Horn constraints over uninterpreted predicates. Subsequently, state-of-the-art SMT solvers are used to solve the constraints. We have implemented the approach, and our experiments show non-trivial integer programs that can now be proved equivalent without further user input.

Categories and Subject Descriptors
F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; D.2.4 [Software Engineering]: Software/Program Verification

Keywords
Regression verification; program equivalence; invariant generation; formal methods

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ASE'14, September 15–19, 2014, Västerås, Sweden
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642987.
kind of programs is poorly supported by existing automatic approaches, as these either require static (i.e., known at compile time) control flow [31], employ coarse abstractions on program computations [17,31], or are overly restrictive (e.g., require small bounds on loops or that equivalent unbounded loops have equivalent bodies) [26].

Our method works well whenever sufficiently "simple" coupling predicates over linear arithmetic exist that prove program equivalence. This is often the case in practice, as we argue throughout the paper. In particular, in Section 5 we demonstrate the effectiveness of our technique using a collection of small but non-trivial benchmarks.

In detail, the contributions of this paper are:

- A method for automatic regression verification for programs employing complex arithmetic on integer variables
- 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/).

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 formulas. The produced VC are in Horn normal form and are passed to an SMT solver for Horn constraints (such as Z3 [23] or ELDARICA [27]), as presented in Section 4. 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.

1.1 Illustration

Example 1. Consider the function \( g_1 \) 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 \( g_2 \) in Figure 2(b) computes essentially the same result, but it has been optimized to employ tail recursion.

\(^2\)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 substrings indicating the program to which they belong. We may also concurrently use the original identifiers without a subscript as long as the relation is clear from the context.
the illustrative example is discussed in Section 4, and can be solved by ELDRICA \cite{eldarica} in roughly three seconds, inferring the coupling predicate \( n_1 = n_2 \rightarrow r_1 + s_2 = r_2 \), where \( n_1 \) is the argument of \( g_1 \), \( n_2 \) and \( s_2 \) are the arguments of \( g_2 \), and \( r_1 \) 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.

2. PROGRAM EQUIVALENCE

This section introduces the considered programming language, and formalizes our notion of program equivalence in terms of Dijkstra’s weakest preconditions \cite{dijkstra}. 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 4.

The Programming Language.

We consider deterministic imperative programs with unbounded integer variables (mathematical integers), written in ANSI C notation. Determinism means that program runs starting in the same state also terminate in the same state. Sequential real-world programs are deterministic, provided that all variables are initialized before they are used (which can be efficiently checked by a compiler). Furthermore, we require that all considered programs terminate for all inputs; this can be checked with one of the existing termination techniques. This can be checked with one of the existing termination techniques. When the function summary \( f_1 \) and \( f_2 \) are initialized before they are used (which can be efficiently checked by a compiler). Furthermore, we require that all considered programs terminate for all inputs; this can be checked with one of the existing termination checkers for imperative programs, such as, e.g., \cite{termination,termination2}. To simplify presentation, we assume that every function ends with a return statement, and that return is always the last statement in a function. Further program features, for instance heap or arrays, are discussed in Section 5.2, but not the main focus of the present paper.

We also 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 \( \bar{a}_1 \) resp. \( \bar{a}_2 \), 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 \cite{dijkstra}. The weakest precondition predicate \( wp(P, \varphi) \) denotes the weakest condition that needs to hold before an execution of statement list\(^3\) \( P \) such that the execution terminates and the postcondition \( \varphi \) 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 \( \varphi \) holds after the execution of \( P \) if \( P \) terminates. Thus, the formula \( pre \rightarrow wlp(P, \psi) \) 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 formulas in pure first-order logic. Figure 3(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 3(a) is complete in the sense that every \( wlp \)-expression can be reduced to a pure first-order formula.

The rules (5), (6) and (7) 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 (8) 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 (9) for a (recursive) invocation \( b = f_1(\bar{a}) \) of the function \( f_1 \) is parametrized by a function summary predicate \( S_{f_1}(\bar{a}, b) \) that relates the arguments \( \bar{a} \) to the result value assigned to variable \( b \). When the function summary \( S_{f_1} \) is used as abstraction for the behavior of \( f_1 \), the correctness of the summary has to be justified globally by an additional verification condition

\[
\text{wlp}(P_1, S_{f_1}(i_1, r_1)) \ ,
\]

in which \( P_1 \) is the function body of \( f_1 \).

The invariant rule (8) and the recursive invocation rule (9) 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

\[
pre \rightarrow P_1 \simeq P_2 \ ,
\]

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:

\[
\forall i_1, i_2. (i_1 = i_2 \land pre \rightarrow wlp(P_1 ; P_2, r_1 = r_2)) \ .
\]

\(^3\)To simplify presentation, we will use the terms “statement list” and “program” interchangeably. The exact relation will be clear from the context.
This kind of construction is known as self-composition [10, 11]. 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 (4) by applying the rules from Figure 3(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 more rules than would be sound 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.

In Figure 3(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 (10) 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 (5) and (6) of Figure 3(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 pairwise 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 $\Phi$ be a purely first-order formula derived from the condition $\text{wlp}(P_1;P_2,\varphi)$ by rules from Figure 3(a) and (b). If the program $P_1;P_2$ is started in a state satisfying the precondition $\Phi$, and terminates, then $\varphi$ 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:

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

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 (8), 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:

$$\text{while} (\psi_1) B_1 ; \text{while} (\psi_2) B_2 \simeq \text{while} (\psi_1 \lor \psi_2) \{ \text{if} (\psi_1) B_1 ; \text{if} (\psi_2) B_2 \} .$$

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

$$(B_1, B_1, \ldots, B_1, B_2, B_2, \ldots, B_2) ,$$

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

$$(B_1, B_2, B_1, B_2, \ldots, B_1, B_2, B_2, \ldots, B_2)$$

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 (15) is a run for the fused loop from (14). 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 (8) to the fused loop from (14), has the same effect as applying the two-program rule (12). 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.
the rule can be reordered:

\[ wlp(x = t; P, \varphi) \leadsto \text{let } x = t \text{ in } wlp(P, \varphi) \] (5)

\[ wlp(\text{if}(\psi) \ T \ \text{else } E; P, \varphi) \leadsto \text{if } \psi \text{ then } wlp(T; P, \varphi) \text{ else } wlp(E; P, \varphi) \] (6)

\[ wlp(\text{return } r, \varphi) \leadsto \varphi \] (7)

\[ wlp(\text{while}(\psi) \ B; P, \varphi) \leadsto I(\bar{x}_1) \land \forall \bar{x}_2, (I(\bar{x}_1) \land \psi \rightarrow wlp(B, I(\bar{x}_1))) \land (I(\bar{x}_1) \land \neg \psi \rightarrow wlp(P, \varphi))) \] (8)

\[ wlp(r = f_1(\bar{i}); P, \varphi) \leadsto \forall r. S_{t_1}(\bar{i}, s) \rightarrow wlp(P, \varphi) \] (9)

(a) Conventional \( wlp \) calculus rules

\[ wlp(P_1 || P_2, \varphi) \leadsto wlp(P_2 || P_1, \varphi) \] (10)

\[ wlp(\text{return } r || P_2, \varphi) \leadsto wlp(P_2, \varphi) \] (11)

\[ wlp(\text{while}(\psi_1) \ B_1; P_1 || \text{while}(\psi_2) \ B_2; P_2, \varphi) \leadsto C(\bar{x}_1, \bar{x}_2) \] (12)

\[ \land \forall \bar{x}_1, \bar{x}_2 (C(\bar{x}_1, \bar{x}_2) \land \psi_1 \land \psi_2 \rightarrow wlp(B_1 || B_2, C(\bar{x}_1, \bar{x}_2))) \]

\[ \land (C(\bar{x}_1, \bar{x}_2) \land \neg \psi_1 \land \psi_2 \rightarrow wlp(B_2, C(\bar{x}_1, \bar{x}_2))) \]

\[ \land (C(\bar{x}_1, \bar{x}_2) \land \psi_1 \land \neg \psi_2 \rightarrow wlp(B_1, C(\bar{x}_1, \bar{x}_2))) \]

\[ \land (C(\bar{x}_1, \bar{x}_2) \land \neg \psi_1 \land \neg \psi_2 \rightarrow wlp(P_2 || P_2, \varphi)) \] (13)

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

\[ wlp(r_1 = f_1(\bar{i}_1); P_1 || r_2 = f_2(\bar{i}_2); P_2, \varphi) \leadsto \forall r_1, r_2. R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2) \rightarrow wlp(P_1 || P_2, \varphi). \]

The rule is parametrized by the mutual function summary \( R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2) \).

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

\[ \forall \bar{i}_1, \bar{i}_2. \ wlp(P_1 || P_2, R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2)) \] (16)

to the verification conditions of equivalence. Here, \( P_1 \) and \( P_2 \) are the statement lists from the function bodies of the invoked functions \( f_1 \) and \( f_2 \).

The justification of rule (13) is as follows. Due to the disjointness of the program vocabulary, the statements in the rule can be reordered:

\[ r_1 = f_1(\bar{i}_1); P_1 || r_2 = f_2(\bar{i}_2); P_2 \leadsto r_1 = f_1(\bar{i}_1); r_2 = f_2(\bar{i}_2); P_1 || P_2 \]

Condition (16) guarantees that \( R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2) \) is a faithful abstraction of \( P_2 \). 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 \( i_1 = i_2 \rightarrow r_1 = r_2 \) can be used.

Note that the same mutual summary \( R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2) \) is used for every occurrence of the pair \( f_1/f_2 \) of functions; this is in contrast to the coupling invariant rule (12) 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 \( f_1/f_2 \) pair, the use of only a single such summary minimizes the number of required proof obligations (16).

4. AUTOMATIC EQUIVALENCE PROOFS

The application of the \( wlp \) rules in Figure 3 requires knowledge of specific predicates, namely loop invariants \( I(\bar{i}_1, \bar{x}_1) \) in rule (8), mutual loop invariants \( C(\bar{x}_1, \bar{x}_2) \) in (12), function summaries \( S_{t_1}(\bar{i}, s) \) in (9), and mutual function summaries \( R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2) \) in (13). Together, those formulas 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 5) 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 fully automatically [19, 23, 27]. Such methods are in principle incomplete, but they are effective for deriving predicates in practical cases arising from equivalence proofs.

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 [21]), 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

\[ wlp(r_1 = f_1(\bar{i}_1); P_1 || r_2 = f_2(\bar{i}_2); P_2, \varphi) \leadsto \forall r_1, r_2. R_{t_1/\bar{t}_2}(\bar{i}_1, r_1, \bar{i}_2, r_2) \rightarrow wlp(P_1 || P_2, \varphi). \]

Figure 3: Weakest precondition calculus
Recursive Horn Clauses.

In order to derive invariants and coupling predicates, verification conditions are represented in form of Horn constrains over (uninterpreted) relation symbols, including \( I, C, S_1, R_t \), 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

\[
H \leftarrow \varphi \land B_1 \land \ldots \land B_n,
\]

where:

- \( \varphi \) is a constraint over variables occurring in the clause; in our experiments, \( \varphi \) 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, \( \varphi \land B_1 \land \ldots \land B_n \) the body. In case \( \varphi = true \), we usually leave out \( \varphi \) and just write \( H \leftarrow B_1 \land \ldots \land B_n \). 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 2. (Example 1 continued). Figure 4 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 \( g_1 \) and \( g_2 \) introduced by application of rule (13). The uninterpreted predicates \( S_{g_1} \) and \( S_{g_2} \) are the function summaries for the respective individual functions and are introduced by (9). Clauses with head false result from equivalence proof obligations (4), whereas the clauses with a head different from false are due to justification conditions (3) and (16).

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:

\[
\mathcal{R} = \{ S_1, R_{1/2}, | f, f_1, f_2 \text{ functions} \} \cup
\{ I_1, I_2, I_3, \ldots, C_1, C_2, C_3, \ldots \}.
\]

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

\[
\forall \tilde{t}_1, \tilde{t}_2. (\tilde{t}_1 = \tilde{t}_2 \land pre \rightarrow wlp(P_1 \upharpoonright P_2, r_1 = r_2)) \land
\bigwedge_{f \text{ a function}} wlp(P_1, S_f(\tilde{t}_r, r_1)) \land
\bigwedge_{r_1, r_2 \text{ functions}} wlp(P_1, P_2, R_{1/2}(r_1, r_2)) \tag{17}
\]

Intuitively, any valuation of the relation symbols \( \mathcal{R} \) that makes (17) 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 (17), by means of exhaustive application of the rules in Figure 3. When applying (9) or (13) to replace function calls \( f, f_1, f_2 \) with the corresponding summary, the relation symbol \( S_f \) or \( R_{1/2} \) is inserted in the formula; similarly, when applying the loop rules (8) or (12), a fresh relation symbol \( I \) or \( C \) is introduced. We explain one possible strategy for applying the reduction rules below. Once application of the \( wlp \) rules to (17) 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 3 to (17). 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 all universal quantifiers \( \forall \) (prenex normal form), and finally transforming to conjunctive normal form [20]. To see that the clause normal form \( VC_H \) is Horn, observe that (17) only contains \( wlp \) in positive positions, and that any two positive occurrences of relation symbols are separated by a conjunct; 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 3 is applicable to a \( wlp \) expression, so that in principle more than one verification condition \( VC \) can be derived from (17). Different \( VC \)s 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 (5), (6), (7), (11) to eliminate assignments, conditionals, and return statements of the individual programs, possibly together with (10) to change the order of programs.

2. if no further rules from point 1 are applicable, try to use (12) or (13) 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 (8) or (9) 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 5.

**Solving Horn Clauses.**

A number of algorithms exist to solve the Horn clauses $VC_H$, including predicate abstraction [19, 27] and property-directed reachability (PDR, also known as IC3) implemented in Z3 [23]. 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 $R$ to a symbolic predicate over $n$ variables.

**Example 3** (Example 2 continued). For the clauses in Example 2, the following predicates are found for the un-interpreted symbols:

\[
R(n_1, r_1, n_2, s_2, r_2) \Rightarrow (n_1 = n_2 \rightarrow r_1 + s_2 = r_2)
\]

\[
S_{g1}(n_1, r_1) \Rightarrow true
\]

\[
S_{g2}(n_2, s_2, r_2) \Rightarrow true
\]

which is the solution already discussed in Section 1.1. The function summaries $S_{g1}$ and $S_{g2}$ 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. 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. It is important to note that the procedure is correct independently of the order in which $wlp$ rules are applied for translating (17) to $VC_H$; in particular, counterexamples are always genuine, and point to an actual case of non-equivalence. Good strategies when applying the rules can, however, improve efficiency and prevent non-termination of the Horn solver.

**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 considered equivalence $pre \rightarrow P_1 \simeq P_2$ holds;
- a counterexample is found, and the programs are not equivalent.

5. **IMPLEMENTATION AND EXPERIMENTS**

**Implementation.**

We have implemented our approach for a language close to a subset of ANSI C in a tool named RÊVE. Program data is limited to local variables and function parameters of type `int`, which is interpreted as unbounded (i.e., mathematical) integers. 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 2013-11-27) and ELDARICA (as of 2014-04-16).

**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) and are available with the tool at the URL given in the introduction. Benchmark results are summarized in Table 1. We also give results from the only automatic tool that is directly comparable to ours (due to scope, cf. Section 6), the Regression Verification Tool (RVT) by Strichman and Godlin [17].

The programs in the first group in Table 1 are recursive, while the ones in the second group contain loops. Benchmarks where the two programs were not equivalent are in the third group, and their names end with a bang (!). All other benchmarks contain equivalent programs; the `k` 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 `limit` benchmark, as two steps of the first loop are equivalent to one step of the second loop, an issue that we solve in the next section and illustrate with the larger `digits10` benchmark.

The `triangular-mod` benchmark corresponds to the illustrating example instrumented with modulo operations to simulate integer overflow.

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 4). We found these counterexamples useful in diagnosing problems with the programs, even though we currently do not translate these counterexamples into source code terms.

5.1 An Example for Loop Equivalence

We consider a real-world example from [1]. The program $P_1$ in Figure 5(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 5(c) computes the same result but (asymptotically) about seven times faster. This speedup is accomplished by reducing the strength of operations. The loop has been unrolled four times\(^5\) 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 5(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 which, slows the program down, while the four if-conditions in $P_2$ are all dependent on the same division result.

After 11.3 seconds, RÊVE with Eldarica succeeds in proving equivalence with the following automatically inferred coupling predicate:

$$(b_2 = 1 \land r_1 = r_2 \land 10n_1 \leq n_2 \land n_2 \leq 10n_1 + 9) \lor (b_2 = 0 \land r_1 = r_2 \land n_1 \leq 0).$$

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., $n_1 = n_2 \div 10$. Exactly this fact is expressed by the linear constraint $10n_1 \leq n_2 \land n_2 \leq 10n_1 + 9$. When the loop of $P_2$ has finished, its negated loop guard $n_1 \leq 0$ holds and the final results are stored in $r_1$ and $v_2$.

5.2 Discussion

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>RÊVE</th>
<th>RÊVE+23</th>
<th>RÊVE+ Eldarica</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ackermann</td>
<td>30</td>
<td>0.8</td>
<td>–</td>
<td>–</td>
<td>[17]</td>
</tr>
<tr>
<td>mccarthy91</td>
<td>22</td>
<td>1.1</td>
<td>1.8</td>
<td>1.7</td>
<td>[17]</td>
</tr>
<tr>
<td>limit1</td>
<td>22</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td>[17]</td>
</tr>
<tr>
<td>limit2</td>
<td>22</td>
<td>X</td>
<td>–</td>
<td>5.2</td>
<td>[17]</td>
</tr>
<tr>
<td>limit3</td>
<td>24</td>
<td>X</td>
<td>–</td>
<td>4.5</td>
<td>[17]</td>
</tr>
<tr>
<td>add-horn</td>
<td>26</td>
<td>X</td>
<td>–</td>
<td>4.3</td>
<td></td>
</tr>
<tr>
<td>triangular</td>
<td>23</td>
<td>X</td>
<td>–</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>triangular-mod</td>
<td>53</td>
<td>X</td>
<td>–</td>
<td>–</td>
<td></td>
</tr>
<tr>
<td>inlining</td>
<td>20</td>
<td>X</td>
<td>–</td>
<td>5.7</td>
<td></td>
</tr>
<tr>
<td>Loops</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple-loop</td>
<td>16</td>
<td>0.8</td>
<td>0.1</td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>22</td>
<td>X</td>
<td>–</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>loop2</td>
<td>22</td>
<td>X</td>
<td>–</td>
<td>3.2</td>
<td></td>
</tr>
<tr>
<td>loop3</td>
<td>28</td>
<td>X</td>
<td>–</td>
<td>5.4</td>
<td></td>
</tr>
<tr>
<td>loop4</td>
<td>22</td>
<td>X</td>
<td>–</td>
<td>27.4</td>
<td></td>
</tr>
<tr>
<td>loop5</td>
<td>26</td>
<td>X</td>
<td>–</td>
<td>26.4</td>
<td></td>
</tr>
<tr>
<td>while-if</td>
<td>22</td>
<td>X</td>
<td>–</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>digits10</td>
<td>32</td>
<td>X</td>
<td>–</td>
<td>11.3</td>
<td>[1]</td>
</tr>
<tr>
<td>barthe</td>
<td>28</td>
<td>X</td>
<td>–</td>
<td>4.8</td>
<td>[9]</td>
</tr>
<tr>
<td>barthe2</td>
<td>22</td>
<td>0.5</td>
<td>10.2</td>
<td>3.9</td>
<td>[9]</td>
</tr>
<tr>
<td>barthe2-big</td>
<td>32</td>
<td>1.6</td>
<td>–</td>
<td>5.7</td>
<td></td>
</tr>
<tr>
<td>barthe2-big2</td>
<td>42</td>
<td>1.7</td>
<td>–</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>bug15</td>
<td>26</td>
<td>1.0</td>
<td>0.1</td>
<td>1.8</td>
<td>[17]</td>
</tr>
<tr>
<td>nested-while</td>
<td>28</td>
<td>1.5</td>
<td>–</td>
<td>5.2</td>
<td>[17]</td>
</tr>
<tr>
<td>Not equivalent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ackermann!</td>
<td>30</td>
<td>X</td>
<td>0.1</td>
<td>4.9</td>
<td></td>
</tr>
<tr>
<td>limit1!</td>
<td>22</td>
<td>X</td>
<td>0.0</td>
<td>1.4</td>
<td></td>
</tr>
<tr>
<td>limit2!</td>
<td>25</td>
<td>X</td>
<td>0.7</td>
<td>10.9</td>
<td></td>
</tr>
<tr>
<td>add-horn!</td>
<td>28</td>
<td>X</td>
<td>0.1</td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>triangular-mod!</td>
<td>49</td>
<td>X</td>
<td>0.5</td>
<td>28.1</td>
<td></td>
</tr>
<tr>
<td>inlining!</td>
<td>20</td>
<td>X</td>
<td>0.1</td>
<td>2.7</td>
<td></td>
</tr>
<tr>
<td>loop5!</td>
<td>22</td>
<td>X</td>
<td>0.0</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>barthe!</td>
<td>31</td>
<td>X</td>
<td>1.8</td>
<td>21.9</td>
<td></td>
</tr>
<tr>
<td>nested-while!</td>
<td>28</td>
<td>X</td>
<td>0.1</td>
<td>5.2</td>
<td></td>
</tr>
</tbody>
</table>

LOC=non-empty, non-comment lines of code in both programs together. Dash (−) denotes timeout at 600 seconds, cross (X) denotes that the tool terminates but cannot prove equivalence. All times have been measured on a 2.5 GHz Intel Core2 Quad machine, using only one core.

\(^5\)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.
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.

\[ \text{int } f(\text{int } n) \{ \\
\text{int } r = 1; \\
n = n/10; \\
\text{while (}n > 0\text{) \{ \\
r++; \\
n = n / 10; \\
\text{if (}n > 0\text{) \{ \\
r++; \\
n = n / 10; \\
\text{if (}n > 0\text{) \{ \\
r++; \\
n = n / 10; \\
\}} \\
\}} \\
\}} \\
\}} \\
\text{return } r; \\
\} \]

(a) basic version $P_1$

\[ \text{int } f(\text{int } n) \{ \\
\text{int } r = 1; \\
\text{int } b = 1; \\
\text{int } v = -1; \\
\text{while (}b != 0\text{) \{ \\
\text{if (}n < 10\text{) \{ v = r; b = 0; \} \\
\text{else if (}n < 100\text{) \{ v = r+1; b = 0; \} \\
\text{else if (}n < 1000\text{) \{ v = r+2; b = 0; \} \\
\text{else if (}n < 10000\text{) \{ v = r+3; b = 0; \} \\
\}} \\
r = \text{result } + 4; \\
\}} \\
\text{return } v; \\
\} \]

(b) intermediate version $P'_1$

\[ \text{int } f(\text{int } n) \{ \\
\text{int } r = 1; \\
n = n/10; \\
\text{while (}n > 0\text{) \{ \\
r++; \\
n = n / 10; \\
\text{if (}n > 0\text{) \{ \\
r++; \\
n = n / 10; \\
\}} \\
\}} \\
\}} \\
\text{return } r; \\
\} \]

(c) optimized version $P_2$

Figure 5: Computing the number of digits (digits10) from [1]

...
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 [26] 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 [21] and later developed in [22]. The concept is implemented in the equivalence checker SYMDiff [25], 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-Heftner [32] has a similar pragmatics.

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., [4,8,28]) 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 [24] 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., [29]).

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel approach that uses invariant inference techniques to automatically conduct regression proofs for two imperative integer 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 integer programs with non-trivial arithmetic and control flow. Future work includes an extension to programs with arrays and heap structures, as well as development of more fine-grained coupling schemes.

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.

8. REFERENCES


