

Regression Verification

Benny Godlin¹ Ofer Strichman²

¹ CS, Technion, Haifa, Israel. bgodlin@cs.technion.ac.il

² Information Systems Engineering, IE, Technion, Haifa, Israel
ofers@ie.technion.ac.il

Abstract. Proving the equivalence of successive, closely related versions of a program has the potential of being easier in practice than functional verification, although both problems are undecidable. There are two main reasons for this claim: it circumvents the problem of specifying what the program should do, and in many cases it is computationally easier. We study theoretical and practical aspects of this problem, which we call *regression verification*.

1 Introduction

Proving the equivalence of successive, closely related versions of a program has the potential of being easier in practice than applying functional verification to the newer version against a user-defined, high-level specification. There are two reasons for this claim. First, it mostly circumvents the problem of specifying what the program should do. The user can take a no-action ‘default specification’ by which the outputs of the program should remain unchanged if the two programs are run with the same inputs. Second, as we show in this article, there are various opportunities for abstraction and decomposition that are only relevant to the problem of proving equivalence between similar programs, and these techniques make the computational burden less of a problem.

Both functional verification and program equivalence of general programs are undecidable. Coping with the former was declared in 2003 by Tony Hoare as a “grand challenge” to the computer science community [13]. Program equivalence can be thought of as a grand challenge in its own right, but there are reasons to believe, as indicated above, that it is a ‘lower hanging fruit’. The observation that equivalence is easier to establish than functional correctness is supported by past experience with two prominent technologies: *regression testing* – the most popular automated testing technique for software – and *equivalence checking* – the most popular formal verification technique for hardware. In both cases the reference is a previous version of the system. Equivalence checking also demonstrates the difference in the computational effort: it is computationally easier than model-checking, at least under the same assumption that we make here, namely that the two compared systems are mostly similar. One may argue, however, that the notion of correctness is weaker: rather than proving that a model is ‘correct’, we prove that it is ‘as correct’ as the previous version. In contrast one may argue that it can still expose functional errors since failing to comply

with the equivalence specification indicates that something is wrong with the assumptions of the user. In any case, it might be a feasible venue even in cases where the alternative of functional verification is not.

We call the problem of proving the equivalence of closely related programs *regression verification*. It can be useful wherever regression testing is useful, and in particular for guaranteeing backward compatibility. This statement holds even when the programs are *not* equivalent. Our system allows the user to define an ‘equivalence specification’ in which the compared values (e.g., the outputs) are checked only if a user-defined condition is met. For example, if a new feature – activated by a flag – is added to the program and we wish to verify that all previous features are unaffected, we condition the equivalence requirement with this flag being turned off. Backward compatibility can also be useful when introducing new performance optimizations or applying *refactoring* [5].

The idea of proving equivalence between programs is not new, and in fact preceded the idea of functional verification.¹ We delay a detailed survey of earlier work to later in this section, but let us just mention that as far as we know no one has focused so far on coping with this problem for general programs in real programming languages nor on how to exploit the fact that large parts of the code in the two compared programs is identical. Ideally the complexity of the solution should be dominated by the (semantic) difference between the two compared programs rather than on their size.

There are many different ways to define the notion of Input/Output equivalent programs (we defined six different notions of equivalence in [7]). Here we focus on the following definition:

Definition 1. *Partial equivalence of functions* *Two functions f and f' are said to be partially equivalent (p -equiv for short) if any two terminating executions of f and f' starting from the same inputs, return the same value.*

The problem of function (or program) equivalence according to this definition can be reduced to one of functional verification of a single program P rather easily: P should simply call the two compared functions consecutively with non-deterministic but equal inputs, and assert that they return the same output. The problem with this direct approach is that it makes no use of the expected similarity of the code. Rather, it solves a monolithic functional verification problem without decomposition. As we show in this article, the similar code structure can be beneficial exactly for this reason.

This article extends an earlier proceedings version [8]. The main difference is that here we generalize the technique to programs with mutual recursion, both in the description of the inference rule and in the decomposition algorithm ([8] only considered simple recursion). We also include a proof of correctness for the main decomposition algorithm. Finally, the inference rules are described here in a way that shows more clearly their resemblance to Hoare’s rule for recursive functions. We find this description clearer.

¹ In his 1969 paper about axiomatic basis for computer programming [11], Hoare points to previous works from the late 50’s on axiomatic treatment of the problem of proving equivalence between programs.

The article is structured as follows. Sect. 2 describes briefly an inference rule for proving the partial equivalence of recursive programs. This rule is obviously not complete, but turns out to be strong enough for proving partial equivalence in many realistic cases. In Sect. 3 we will present an algorithm for decomposing the equivalence proof – ideally to the granularity of pairs of functions. We report on some experiments in Sect. 4. Many details about our system are left out, as well as more references to earlier works. The interested reader may find them in the first author’s thesis [6]. The theoretical background on the inference rule that we use can be found also in [7].

Related work As mentioned earlier, the idea of proving equivalence between programs is not new. It is a rather old challenge in the theorem-proving community. A lot of attention has been given to this problem in the ACL2 community (see, e.g., [2, 16, 17]). These works are mostly concerned with program equivalence as a case study for using proof techniques that are generic (i.e., not specific for proving equivalence). We are not aware of such works that are targeted at programs that are mostly syntactically equivalent, which is the target of regression verification.

Attempts to build fully automatic proof engines for industrial programs concentrated so far, to the best of our knowledge, on very restricted cases. Specifically, they all focused on programs without dynamic memory allocation and with bounded loops (or loop-free). Feng and Hu, for example, considered the problem of proving equivalence of embedded code [3]. The main technique used in this line of work is to prove the equivalence of small segments of the code after unrolling loops some predefined number of times. Arons et al. [1] developed a tool in Intel for proving the equivalence of two versions of microcode, with the goal of proving backwards compatibility, but the programs were assumed to be loop-free.

Another relevant line of research is concerned with *translation validation* [20, 18], the process of proving equivalence between a source and a target of a compiler or a code generator. The fact that the translation is mechanical allows the verification methodology to rely on various patterns and restrictions on the generated code. For example, translation validation for synchronous languages [18, 19] relies on the fact that the target C code has exactly one loop (corresponding to a *step* in the synchronous program) and hence the proof is conducted by induction over an expression which is derived from loop-free code. A recent example of translation validation, from the synchronous language SDL to C, is by Haroud and Biere [10]. It is based on a variation of Floyd’s method [4] for proving equivalence: it declares cutpoints in both programs (as in the original Floyd’s method, there should be at least one cutpoint in each loop), maps them between the two programs, and proves that two related cutpoints are equivalent with respect to the ‘observable’ variables if they are equivalent in the preceding pair of cutpoints. This method works in the boundary of a single function on each side and does not support general programs (e.g. recursive programs).

2 Proving partial equivalence

We begin by recalling Hoare’s rule for recursive invocation. Let $\{p\} \text{ call } f \{q\}$ be a Hoare triple where f is a recursive function and the `call` statement reminds us that we consider f with all possible values of actual parameters. The inference rule suggested by Hoare in [12] for proving this triple is:

$$\frac{\{p\} \text{ call } f \{q\} \vdash_H \{p\} f \text{ body } \{q\}}{\{p\} \text{ call } f \{q\}} \text{ (REC) ,} \quad (1)$$

where “ f body” is the body of f , in which the recursive call is ignored. In words, the only effect of the recursive call on the proof is that we assume that it maintains the (p, q) relation. This unintuitive rule was described by Hoare in [12] as follows: *The solution... is simple and dramatic: to permit the use of the desired conclusion as a hypothesis in the proof of the body itself.* The correctness of rule (REC) is proved by induction, where the base case corresponds to the base(s) of the recursion, namely the nonrecursive run(s) through the procedure.

2.1 Rule (PROC-P-EQ_s)

Our rule for partial equivalence between functions² f and f' has the same flavor as (REC). It applies to the special case of f, f' being recursive functions without calls to other functions. We call the rule (PROC-P-EQ_s), for ‘Procedures Partial Equivalence’, where the subscript s indicates that it applies only to the special case.

$$\frac{p\text{-equiv}(\text{call } f, \text{call } f') \vdash p\text{-equiv}(f \text{ body}, f' \text{ body})}{p\text{-equiv}(\text{call } f, \text{call } f')} \text{ (PROC-P-EQ}_s\text{) .} \quad (2)$$

Informally, this means that if assuming that the recursive calls are partially equivalent enables us to prove this condition over the bodies of f and f' (i.e., f, f' without the recursive calls), then f and f' are partially equivalent. In [7] we proved that this rule is sound. Although the soundness proof refers to an artificial abstract language, it has most of the features of an imperative language such as C. In Sect. 3.6 we will elaborate further on this point.

A convenient method for checking the premise of rule (PROC-P-EQ_s) is to replace the recursive call with an *uninterpreted function* because by definition instances of the same uninterpreted function are partially equivalent (this is guaranteed by the congruence axiom that defines such functions – see, e.g., chapter 3 in [15]). After performing this replacement we say that the calling

² We use the term ‘function’ here although the rule refers to procedures, i.e., functions that can have multiple outputs. In a languages such as C such multiple outputs can be returned by a function if they are gathered first into a single structure. In addition, global variables that are written to by a function can be modeled as part of its list of outputs.

function is *isolated*. Denote by f^{UF} the isolated version of a function f . Rule (PROC-P-EQ_s) can be reformulated accordingly:

$$\frac{p\text{-equiv}(f^{UF}, f'^{UF})}{p\text{-equiv}(\text{call } f, \text{call } f')} \quad (3)$$

Observe that the premise of this rule is decidable for a language with finite domains such as C because, recall, it contains no loops or recursive calls. The following example demonstrates the use of this rule.

Example 1. Consider the two functions in Fig. 1. Let U be the uninterpreted function such that calls to U replace the recursive calls to `gcd1` and `gcd2`. Figure 2 presents the isolated functions.

```

gcd1(int a, int b)      gcd2(int x, int y)
{ int g;                { int z;
  if (!b) g = a;        z = x;
  else {                if (y > 0)
    a = a%b;            z = gcd2(y, z%y);
    g = gcd1(b, a); }   return z;
  return g;             }
}

```

Fig. 1. Two functions to calculate GCD of two nonnegative integers.

```

gcd1(int a, int b)      gcd2(int x, int y)
{ int g;                { int z;
  if (!b) g = a;        z = x;
  else {                if (y > 0)
    a = a%b;            z = U(y, z%y);
    g = U(b, a); }     return z;
  return g;             }
}

```

Fig. 2. After isolation of the functions, i.e., replacing their function calls with calls to the uninterpreted function U .

To prove the partial equivalence of the two functions, we need to first translate them to formulas expressing their respective transition relations. A convenient way to do so is to use Static Single Assignment (SSA) (see, e.g., [15]). Briefly, this means that in each assignment of the form $x = \text{exp}$; the left-hand side variable x is replaced with a new variable, say x_1 . Any reference to x after this line and before x is assigned again is replaced with the new variable x_1 (this

is done in a context of a program without unbounded loops). In addition, assignments are guarded according to the control flow. After this transformation, the statements are conjoined: the resulting equation represents the computations of the original program.

The SSA form of isolated `gcd1`, denoted T_{gcd_1} , is

$$\left(\begin{array}{l} a_0 = a \\ b_0 = b \\ b_0 = 0 \rightarrow g_0 = a_0 \\ (b_0 \neq 0 \rightarrow a_1 = (a_0 \% b_0)) \wedge (b_0 = 0 \rightarrow a_1 = a_0) \\ (b_0 \neq 0 \rightarrow g_1 = U(b_0, a_1)) \wedge (b_0 = 0 \rightarrow g_1 = g_0) \\ g = g_1 \end{array} \wedge \right). \quad (4)$$

The SSA form of isolated `gcd2`, denoted T_{gcd_2} , is

$$\left(\begin{array}{l} x_0 = x \\ y_0 = y \\ z_0 = x_0 \\ y_0 > 0 \rightarrow z_1 = U(y_0, (z_0 \% y_0)) \\ y_0 \leq 0 \rightarrow z_1 = z_0 \\ z = z_1 \end{array} \wedge \right). \quad (5)$$

The premise of rule (PROC-P-EQ_s) requires proving the validity of the following formula over nonnegative integers:

$$(a = x \wedge b = y \wedge T_{gcd_1} \wedge T_{gcd_2}) \rightarrow g = z. \quad (6)$$

Many theorem provers can prove such formulas fully automatically, and hence establish the partial equivalence of `gcd1` and `gcd2`. \square

2.2 Extensions to (PROC-P-EQ_s)

Now suppose that the two compared functions f, f' call other functions f_c, f'_c , respectively. Rule (PROC-P-EQ_s) can still be used if one of the following holds:

1. If f_c, f'_c were already proven to be equivalent then they can be replaced with uninterpreted functions. Such a replacement imposes an overapproximating abstraction. The soundness of the rule is maintained.
2. Otherwise, if f_c, f'_c and their descendants are not recursive then they can be inlined in their callers. The premise of rule (PROC-P-EQ_s) is then checked as before.
3. Otherwise, if some of the descendants of f_c, f'_c are recursive but were proven partially equivalent then these descendants can be abstracted with uninterpreted functions. As in the previous case f_c, f'_c can then be inlined into their callers.

2.3 A generalization: rule (PROC-P-EQ)

We now generalize (PROC-P-EQ_s) to mutually recursive functions. In the call graphs these appear as strongly connected components (SCCs) of size larger than one. We will focus on Maximal SCCs, or MSCCs. Nodes that are not part of any cycle in the call graph correspond to nonrecursive functions, and are called *trivial* MSCCs. Consider two nontrivial MSCCs m, m' in the two programs. Assume that the functions in m, m' do not have loops nor do they call functions outside of m and m' . Further assume that there is a bijective mapping map_f between the functions in m and m' . The generalization of (PROC-P-EQ_s) to mutually recursive functions is:

$$\frac{\forall(f, f') \in map_f. ((\forall(g, g') \in map_f. p\text{-equiv}(\text{call } g, \text{call } g')) \vdash p\text{-equiv}(f \text{ body}, f' \text{ body}))}{\forall(f, f') \in map_f. p\text{-equiv}(\text{call } f, \text{call } f')} \quad (\text{PROC-P-EQ}) . \quad (7)$$

The version of this rule with uninterpreted functions is defined next. For a function g , let $UF(g)$ be an uninterpreted function such that g and $UF(g)$ have the same prototype. We enforce $(g, g') \in map_f \Leftrightarrow UF(g) = UF(g')$. Let

$$f^{UF} \doteq f[g \leftarrow UF(g) \mid g \text{ is called in } f] .$$

The generalization of (3) is:

$$\frac{\forall(f, f') \in map_f. p\text{-equiv}(f^{UF}, f'^{UF})}{\forall(f, f') \in map_f. p\text{-equiv}(\text{call } f, \text{call } f')} . \quad (8)$$

Example 2. Consider the two small MSCCs in the top part of Fig. 3, where $map_f = \{(f_1, f'_1), (f_2, f'_2)\}$. According to (8) we need to prove partial equivalence of (f_1, f'_1) while replacing the calls to f_2, f'_2 with the same uninterpreted function ($U_2 = UF(f_2) = UF(f'_2)$, see middle drawing), and prove separately the partial equivalence of (f_2, f'_2) while replacing the calls to f_1, f'_1 with the same uninterpreted function ($U_1 = UF(f_1) = UF(f'_1)$, see bottom drawing). □

2.4 Extensions and relaxations of (PROC-P-EQ)

Now suppose that functions in m and m' do call functions outside of m and m' , respectively. All the exceptions listed in Sect. 2.2 for the case of simple recursive functions apply here (e.g., nonrecursive functions can be inlined). Further, suppose that there is no bijective mapping between the functions in m and m' , or that not all pairs in map_f are partially equivalent (or can be proven to be so). We now show that it still may be possible to prove the partial equivalence of *some* of the functions. The idea is to inline some of the functions in m and m' , as long as we do not create cycles. Formally, let $S \subseteq \{\langle f, f' \rangle \mid \langle f, f' \rangle \in map_f, f \in m, f' \in m'\}$ be a set of function pairs that satisfies:

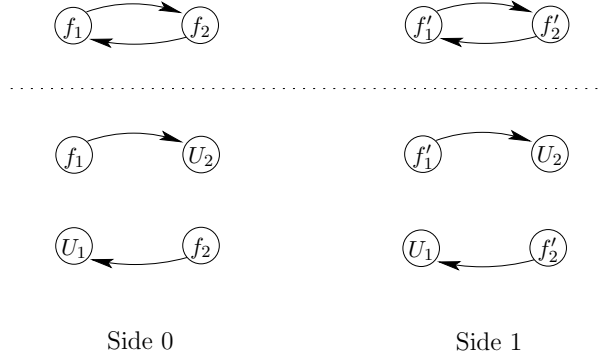


Fig. 3. Proving partial equivalence of mutual recursive functions.

- Every cycle in m contains a node f such that $\exists f'.(f, f') \in S$, and
- Every cycle in m' contains a node f' such that $\exists f.(f, f') \in S$.

The rule for proving the partial equivalence of the pairs in S is then:

$$\frac{\forall(f, f') \in S. p\text{-equiv}(f^{UF}, f'^{UF})}{\forall(f, f') \in S. p\text{-equiv}(\text{call } f, \text{call } f')} , \quad (9)$$

but here the definition of f^{UF} and f'^{UF} is slightly different than the definition in Sect. 2.1. Let m_S denote all the functions in m that participate in S , i.e., $m_S \doteq \{g \mid g \in m \wedge \exists g'.(g, g') \in S\}$. m'_S is defined similarly with respect to m' . f^{UF} is now created by replacing each call to a function $h \in m_S$ by a call to $UF(h)$, and inlining the rest. f'^{UF} is created similarly. Hence, the difference from the original definition of f^{UF} and f'^{UF} is that now the functions outside of $\{m_S \cup m'_S\}$ are inlined.

The rule indicates that we can relax the requirement of bijectiveness of map_f : it can be any 1-1 partial mapping that contains a set S as defined above. Note that S replaces map_f in both the premise and the consequent of (PROC-P-EQ). Hence the weaker premise has a weaker consequent. Ideally S should be as close to the original definition, namely bijective, because then more pairs of functions are proven to be partially equivalent.

In the next section we describe an algorithm that attempts to prove partial equivalence of general programs by traversing the call graphs bottom-up and replacing functions with their uninterpreted versions when possible, based on these generalizations.

3 A decomposition algorithm

Our Regression Verification Tool (RVT) is geared towards C programs and hence we begin with a brief description of CBMC [14], the underlying decision proce-

dure that we use for checking the premise of rule (PROC-P-EQ) and its extensions as described in the previous section. CBMC, developed by D. Kroening, is a bounded model checker for C programs that supports almost all of the features of ANSI-C. It requires from the user to define a bound k on the number of iterations that each loop in a given ANSI-C program is taken, and a similar bound on the depth of each recursion. It also provides functions that return non-deterministic values (e.g., `nondet_int()` and `nondet_float()`), which are used for modeling the possible inputs. This enables CBMC to symbolically characterize the full set of possible executions restricted by the user-defined bounds, by a decidable formula f (by default f is propositional, but CBMC can generate the verification condition in other decidable theories). The existence of a solution to $f \wedge \neg a$, where a is a user defined assertion, implies the existence of a path in the program that violates a . Otherwise, we say that CBMC established the k -correctness of the checked assertions. We use CBMC in a very restricted way, however: recall that the premise of rule (PROC-P-EQ) is over nonrecursive functions without loops, hence in our case $k = 1$.

RVT generates small loop-free and recursion-free C programs – each corresponds to a pair of functions that it attempts to prove equal – which it sends to CBMC for decision.

3.1 Preprocessing and mapping

Before this iterative process begins, RVT makes two preliminary steps.

Loops All loops in f and f' are replaced with recursive functions. This process is described in [6].

Mapping A (possibly partial) mapping map is built by pairing functions and global variables between the two compared programs. Mapping is done recursively, in a manner reminiscent of computing congruence closure. The algorithm works on the parse trees of both programs and pairs nodes, where a node can be either a variable, a function, or a type. Initially it maps global variables with the same name and type. It then maps functions with the same name, return type, prototype, and that their lists of global variables that they read and write to are mapped pairwise. Then, within mapped functions that are also syntactically equivalent up to variable names, it attempts to map elements that appear in isomorphic locations. If these elements were already mapped it just checks that the mapping according to this function agrees with the previous one, and otherwise it issues a warning. This process is repeated until no new mapping is discovered.

Note that wrong mapping does not affect soundness: it is used for generating the verification conditions, and hence wrong mapping can only fail a proof. We denote by map_f the partial mapping of functions resulting from this process. For simplicity of the presentation we will assume that the global variables accessed by a function are added at the end of its list of parameters, in a consistent order. This assumption simplifies the description of the algorithm later on.

The input for the main algorithm is thus two recursive programs without loops, and a mapping between the functions map_f .

3.2 A bottom-up decomposition algorithm

The equivalence check in RVT is presented in Algorithm 1. It is based on traversing bottom-up the call graphs of the two programs to be compared. In line 2 we inline all nonrecursive functions that are not mapped. In the next line we build the MSCC DAGs MD_1 and MD_2 from the call graphs of the input programs. An MSCC DAG corresponding to a program is simply the call graph of the program after collapsing its MSCCs into single nodes. In line 4 we attempt to build a bijective mapping map_m between the nodes of MD_1 and MD_2 , which is consistent with map_f . In other words, if $\langle m_1, m_2 \rangle \in map_m$, f is a function in m_1 and $\langle f, f' \rangle \in map_f$, then f' is a function in m_2 (and vice-versa). If such a mapping is impossible, the algorithm aborts. In practice one may run the algorithm bottom-up until reaching nonmapped MSCCs, but we omit this option here in order to keep the description simple.

In line 5 we begin the bottom-up traversal. We search for the next unmarked pair of MSCCs that its children pairs are already marked. If the selected pair $\langle m_1, m_2 \rangle$ is trivial, then we simply check in line 9 the equivalence of the two functions in m_1, m_2 . The function CHECK is described in Alg. 2. Otherwise, namely $\langle m_1, m_2 \rangle$ is not trivial, we proceed in line 11 by choosing nondeterministically a subset S of paired functions from m_1, m_2 that intersect all cycles in m_1 and m_2 (in graph-theoretic terms, the functions in S constitute a *feedback vertex set* of both m_1 and m_2). The algorithm can be determinized by, e.g., attempting all such sets.³ A good strategy, as implied by the discussion in the end of Sect. 2, is to give priority to larger sets, since the larger the set is, the more functions we prove to be partially equivalent. Further, larger sets imply less functions to inline, and hence the burden on the decision procedure is expected to be smaller. If one of the pairs in S cannot be proven to be equivalent, we must abort: neither $\langle m_1, m_2 \rangle$ or the SCCs above it can be proven equivalent by our algorithm. Otherwise we mark in line 14 all the functions that are paired in S as “Equivalent”. Finally, we mark $\langle m_1, m_2 \rangle$ as “Covered”, and continue to the next pair.

We now proceed by describing CHECK, which appears in Alg. 2. This function begins by checking whether the input functions happen to be syntactically equivalent and their children are also marked equivalent. If yes, it returns true. Otherwise it sends CBMC a nonrecursive, loop-free C program, which we call a *check-block*. Following is a description of this program.

Let D_f denote the maximal connected subDAG rooted at f that contains only functions that are unpaired or *not* marked “Equivalent”, but excluding f itself. $D_{f'}$ is defined similarly with respect to f' . The program *check-block* (f, f') consists of the following elements:

1. The functions f, f' and all functions in $D_f, D_{f'}$, such that

³ Although there can be an exponential number of them in the size of the MSCC, observe that large MSCCs in real programs are rare.

Algorithm 1 A bottom-up decomposition algorithm for proving the partial equivalence of pairs of functions.

```

1: function PROVE(Programs  $P, P'$ , map between functions  $map_f$ )
2:   Inline nonrecursive nonmapped functions;
3:   Generate MSCC DAGs  $MD_1, MD_2$  from the call graphs of  $P, P'$ ;
4:   If possible, generate a bijective map  $map_m$  between nontrivial nodes in  $MD_1$ 
      and  $MD_2$  that is consistent with  $map_f$  (it is desirable but not necessary to
      add pairs of trivial nodes to  $map_m$ ). Otherwise abort.
5:   while  $\exists \langle m_1, m_2 \rangle \in map_m$  that is uncovered and its children are “Covered” do
6:     Choose such a pair  $\langle m_1, m_2 \rangle$ ;
7:     if  $m_1, m_2$  are trivial then
8:       Let  $f_1, f_2$  be the functions in  $m_1, m_2$ , respectively;
9:       if CHECK( $f_1, f_2$ ) then mark  $f_1, f_2$  as “Equivalent”;
10:    else
11:      Select nondeterministically a set of function pairs  $S \subseteq \{\langle f, f' \rangle \mid \langle f, f' \rangle \in$ 
         $map_f, f \in m_1, f' \in m_2\}$  that intersect all cycles in  $m_1$  and  $m_2$ ;
12:      for all  $\langle f, f' \rangle \in S$  do
13:        if  $\neg$ CHECKr( $f, f', S$ ) then abort;
14:        for all  $\langle f, f' \rangle \in S$  do mark  $f, f'$  as “Equivalent”;
15:      Mark  $\langle m_1, m_2 \rangle$  as “Covered”.

```

- Name collisions in global identifiers of the two programs are solved by renaming;
- All calls to f, f' are replaced with calls to $UF(f)$ ($=UF(f')$), respectively;
- For all $\langle h_1, h_2 \rangle \in map_f$ such that $h_1, h_2 \notin \{D_f \cup D_{f'}\}$, calls to h_1, h_2 are replaced with calls to $UF(h_1)$ ($=UF(h_2)$). (Observe that the pair $\langle h_1, h_2 \rangle$ must be marked “Equivalent”).
- 2. The `main()` function, which consists of:
 - Assignment of nondeterministic but equal values to inputs of f and f' ;
 - Calls to f, f' ; and
 - Assertion that the outputs of f and f' are equal.

Following are several notes on the definition of *check-block* (f, f'):

- check-block is guaranteed to be nonrecursive. This is because when PROVE fails to prove the equivalence of MSCCs it aborts in line 13, and hence recursive functions that are not proven equivalent will never be part of future check-blocks.
- The code of each nonrecursive pair $\langle f, f' \rangle \in map_f$ that could not be proven equivalent is included when checking the equivalence of their parents, and possibly more ancestors, until reaching a provably equivalent pair or reaching the roots. We call this process *logical inlining*, since it is equivalent to inlining but is more faithful to the program’s original structure. This enables RVT to prove equivalence in case, for example, that some code was moved from the parent to the child, but together they still perform the original computation.

- The code of a pair $\langle f, f' \rangle \in \text{map}_f$ that is proven to be equivalent does not participate in any subsequent check-block. It is replaced with uninterpreted functions in all subsequent checks, or disappears altogether if some ancestor pair is also marked “Equivalent” in each of its paths to the roots of the related subprograms.
- The replacement of recursive calls of paired functions with uninterpreted functions corresponds to isolation (see Sect. 2). Recall that proving equivalence of mapped isolated functions also proves their partial equivalence by rule (PROC-P-EQ).

Algorithm 2 A function called by PROVE for checking the equivalence of two input nonrecursive functions. `check-block` is a C program defined in the main text.

```

1: function CHECK(function  $f$ , function  $f'$ )
2:   if  $f$  and  $f'$  are syntactically equivalent and all their children are marked “Equivalent” then
3:     return true;
4:   return CBMC (check-block ( $f, f'$ ));

```

Algorithm 3 A function called by PROVE for checking the equivalence of two input functions that are part of MSCCs. `check-blockr` is a C program defined in the main text.

```

1: function CHECKr(function  $f$ , function  $f'$ , set of pairs  $S$ )
2:   if  $f$  and  $f'$  are syntactically equivalent and all their children are either marked “Equivalent” or in  $S$  then
3:     return true;
4:   return CBMC (check-blockr ( $f, f', S$ ));

```

The function `CHECKr`, described in Alg. 3, is similar to `CHECK`, and can be seen as its generalization to the case that $S \neq \emptyset$. It also begins by checking for syntactical equivalence, but permits children of the checked functions to be in S rather than being marked equivalent. If this check fails, it calls `CBMC` with a program that is identical to `check-block` (f, f') as defined above, except the following difference in the definition of D_f and $D_{f'}$. Recall that D_f and $D_{f'}$ include all nodes in the subDAG under f and f' respectively that are unpaired or not marked “Equivalent”. But here, since f and f' are recursive (they are part of MSCCs), their descendants form general graphs rather than DAGs. We redefine D_f and $D_{f'}$ so they do *not* include functions from S , which forces them to be nonrecursive. More formally, Let D_f^r be the set of functions in the maximal connected graph descending from f which does *not* include

- f itself,
- functions in S , and
- functions that are marked “Equivalent”.

D_f^r is defined similarly with respect to f' . Hence D_f^r and $D_{f'}^r$ are nonrecursive. Replace $D_f, D_{f'}$ with $D_f^r, D_{f'}^r$ in the definition of check-block to get check-block^r.

3.3 Examples

In this section we demonstrate Alg. 1 with two examples. The first focuses on programs with simple recursion only, and the second on the more general case.

Example 3. Consider the call graphs in Fig. 4. Assume that for $i = 1, \dots, 6$ we have $\langle f_i, f'_i \rangle \in \text{map}_f$, and that the functions marked by grey nodes in Fig. 4 are syntactically equivalent to their counterparts.

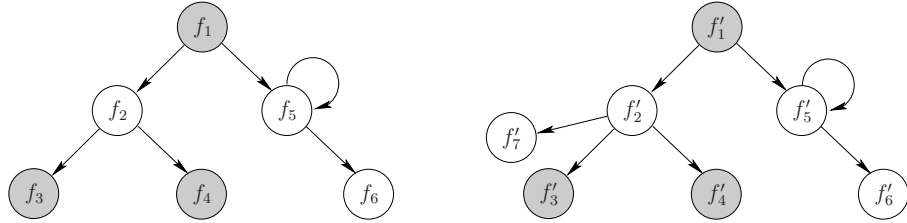


Fig. 4. Two call graphs for Example 3. A node is grey if it is syntactically equivalent to its counterpart.

We describe step by step the execution of Algorithm 1. Initially, in line 2, we inline f'_7 into f'_2 . The iterations that follow are listed below: \square

We continue with an example of programs with mutual recursion.

Example 4. Consider the call graphs that are presented in Fig. 5. Assume that

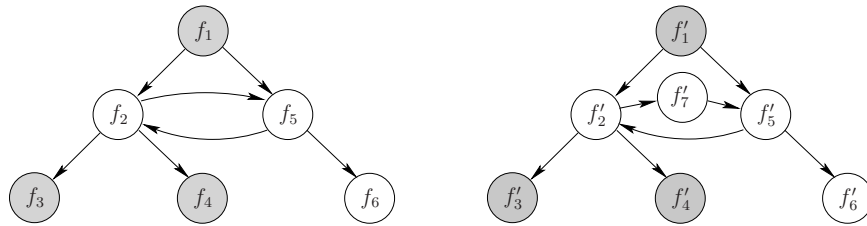


Fig. 5. Two call graphs for Example 4

It. Pair	Description	Res.
1. $\langle f_3, f'_3 \rangle$	Since f_3, f'_3 are nonrecursive we call CHECK in line 9, which returns TRUE based on a syntactic check.	✓
2. $\langle f_4, f'_4 \rangle$	Same as with $\langle f_3, f'_3 \rangle$.	✓
3. $\langle f_6, f'_6 \rangle$	Handled by CHECK, which calls CBMC. Assume that CHECK returns FALSE.	×
4. $\langle f_2, f'_2 \rangle$	Handled by CHECK, which calls CBMC. check-block contains f_2, f'_2 (recall that f'_7 is inlined into f'_2), and the calls to f_3, f'_3, f_4, f'_4 are replaced by calls to uninterpreted functions. Assume that this check fails, and hence CHECK returns FALSE.	×
5. $\langle f_5, f'_5 \rangle$	Handled by CHECK ^r . The only choice for S is $\langle f_5, f'_5 \rangle$. In this case $D_{f_5}^r = \{f_6\}$ and $D_{f'_5}^r = \{f'_6\}$, hence check-block ^r contains f_5, f'_5, f_6, f'_6 , and the recursive calls are replaced with uninterpreted functions. Assume that the check succeeds.	✓
6. $\langle f_1, f'_1 \rangle$	Handled by CHECK which calls CBMC. check-block contains f_1, f'_1, f_2, f'_2 (f'_7 is inlined into f'_2), and the calls to $f_3, f'_3, f_4, f'_4, f_5, f'_5$ are replaced by calls to uninterpreted functions.	✓

for $i = 1, \dots, 6$ we have $\langle f_i, f'_i \rangle \in \text{map}_f$. As in the previous example syntactically equivalent functions are marked by grey nodes. The nodes of the MSCC DAGs that are generated in line 3 are (listed bottom-up, left-to-right): $MD_1 = \{\{f_3\}, \{f_4\}, \{f_6\}, \{f_2, f_5\}, \{f_1\}\}$ and $MD_2 = \{\{f'_3\}, \{f'_4\}, \{f'_6\}, \{f'_2, f'_5, f'_7\}, \{f'_1\}\}$. The MSCC mapping map_m in line 4 is naturally derived from map_f . The first three iterations, over $\langle f_3, f'_3 \rangle$, $\langle f_4, f'_4 \rangle$ and $\langle f_6, f'_6 \rangle$, are the same as in the previous example. Next comes the nontrivial SCCs. Assume that in line 11 we choose $S = \{\langle f_2, f'_2 \rangle, \langle f_5, f'_5 \rangle\}$. This is more than strictly necessary for breaking the cycles, but recall that we wish to prove partial equivalence for as many pairs as possible. The iterations from there on are listed below. We use \checkmark^c to denote that the equivalence check succeeded, but the functions are not yet marked as equivalent:

It. Pair	Description	Res.
4. $\langle f_2, f'_2 \rangle$	Handled by CHECK ^r , which calls CBMC. check-block ^r contains f_2, f'_2, f'_7 . Calls to f_3, f'_3, f_4, f'_4 and f_5, f'_5 are replaced with calls to uninterpreted functions. Assume that this check succeeds.	\checkmark^c
5. $\langle f_5, f'_5 \rangle$	Handled by CHECK ^r , which calls CBMC. check-block ^r contains f_5, f'_5, f_6, f'_6 , and the calls to f_2, f'_2 are replaced with calls to uninterpreted functions. Assume that the check succeeds. (In line 14 $\langle f_2, f'_2 \rangle$ and $\langle f_5, f'_5 \rangle$ are marked “Equivalent”).	\checkmark^c
6. $\langle f_1, f'_1 \rangle$	Handled by CHECK. The syntactic check returns TRUE. (All MSCCs are marked “Covered” and the algorithm terminates).	✓

What would happen had we chose in line 11 $S = \{\langle f_2, f'_2 \rangle\}$? In that case check-block^r would contain $f_2, f'_2, f'_7, f_5, f'_5, f_6, f'_6$, and calls to f_3, f'_3, f_4, f'_4 and f_2, f'_2 would be replaced with calls to uninterpreted functions. If this check would

have succeeded, then only $\langle f_2, f'_2 \rangle$ would be marked as equivalent. Not only that this would not prove the equivalence of $\langle f_5, f'_5 \rangle$, but it also would complicate the next step: when checking the pair $\langle f_1, f'_1 \rangle$, we would need to include also f_5, f'_5, f_6, f'_6 , as they were not marked equivalent. Calls to $\langle f_2, f'_2 \rangle$ would be replaced with calls to uninterpreted functions. This example motivates our suggested determinization: try large S first. \square

3.4 Controlling the abstraction level

The definition of check-block^r entails that every call to a function in S is replaced with a call to an uninterpreted function. This is not always necessary. Replacing functions with uninterpreted functions entail less computational effort, but also loses precision, and hence may fail proofs. Some control over the abstraction level is possible even for a given set S , because the only thing we need to guarantee is that check-block^r is nonrecursive, and that every pair of functions that we assume to be partially equivalent is eventually proven to be so. Consider once again Example 4, where we chose $S = \{\langle f_2, f'_2 \rangle, \langle f_5, f'_5 \rangle\}$. When checking $\langle f_5, f'_5 \rangle$, we replaced the calls to f_2, f'_2 with uninterpreted functions because $\langle f_2, f'_2 \rangle \in S$. Instead we can include these functions and f'_7 in check-block^r , and replace the calls to f_5, f'_5 with calls to uninterpreted functions.

It is obvious that the actual functions matter for choosing the most refined abstraction. Without this information a reasonable heuristic is to maximize the number of calls to interpreted functions, while still breaking all cycles.

3.5 Correctness of PROVE

Theorem 1. (Correctness of PROVE) *Functions that are marked “Equivalent” by Alg. 1 are partially equivalent to their counterpart in map_f .*

Proof. We give a proof sketch, based on the soundness of (PROC-P-EQ) and its generalization in Eq. (9), and on the soundness of CBMC itself. The program sent to CBMC clearly checks for partial equivalence; Hence the correctness argument for the case we invoke CBMC must show that the abstraction induced by replacing functions with uninterpreted functions is a conservative one.

Consider the current covering (the result of executing line 15) of MD_1 nodes right after line 15. Let d denote the current covering ‘depth’, i.e., the largest distance of a covered node from any leaf node in MD_1 . The proof is by induction on d .

Base: When $d = 1$, m_1 is a leaf. There are two cases to consider, corresponding to the lines in which the marking of functions with “Equivalent” is done:

- m_1 and m_2 are trivial nodes, and the single function that m_1 contains, which we denote by f , is marked “Equivalent”. It must have been marked by CHECK in line 9. CHECK returns TRUE if f and its counterpart are either syntactically equivalent or proven to be equivalent by CBMC. In the latter, check-block does not include calls to uninterpreted functions.

- m_1 (or m_2) is not trivial, and a subset of functions corresponding to S of line 11 is marked “Equivalent”. The condition for the marking in line 14 corresponds exactly to the premise of (9).

Step: Now assume that the theorem is correct for d , and consider $d + 1$. For a function f marked “Equivalent” in level $d + 1$, such that $\langle f, f' \rangle \in \text{map}_f$, we will prove that f and f' are indeed partially equivalent. The proof relies on the induction hypothesis, which implies that descendant functions of f and f' that are marked “Equivalent” are indeed partially equivalent to their counterparts in map_f . As in the base case, we consider separately the two lines in which the marking of functions with “Equivalent” is done:

- f and f' constitute trivial MSCC nodes m_1 and m_2 , respectively. f must have been marked by CHECK in line 9. CHECK returns TRUE in two cases:
 - f and f' are syntactically equivalent and their children are marked “Equivalent”. Hence f is indeed partially equivalent to f' .
 - Otherwise, CBMC must have returned TRUE. check-block contains, in addition to f and f' , either inlining of nonrecursive functions or calls to uninterpreted functions. The former clearly preserves correctness. The latter is a conservative abstraction of the original functions, which, recall, are partially equivalent by the induction hypothesis. Hence f and f' are partially equivalent.
- m_1 (or m_2) is not trivial, and a subset of functions corresponding to S is marked “Equivalent” in line 14. Let $\langle f, f' \rangle \in \text{map}_f$ be any of the pairs in S . CHECK^r (on line 3) returns TRUE in two cases:
 - f and f' are syntactically equivalent and their children are either marked “Equivalent” or in S . The former children are partially equivalent by the induction hypothesis. As for the latter children, f can only be marked “Equivalent” if all the pairs in S passed CHECK^r, which by (9) means that they are indeed partially equivalent. Hence in both cases, the fact that f and f' are marked “Equivalent” implies that they indeed are.
 - Otherwise, CBMC must have returned TRUE. check-block^r possibly contains, in addition to f and f' :
 - * Inlining of nonrecursive functions (outside of m_1 and m_2).
 - * Inlining of functions in m_1, m_2 that are not part of S , where calls to S functions are replaced with uninterpreted functions. The argument made above for syntactic checks in which S functions are replaced with uninterpreted functions apply here as well: f can only be marked “Equivalent” if all the pairs in S passed CHECK^r, which by (9) means that they are indeed partially equivalent. Hence the fact that f and f' are marked as partially equivalent implies that they indeed are.
 - * Calls to uninterpreted functions that abstract functions outside of m_1 and m_2 . By the induction hypothesis, these are indeed partially equivalent, and hence the calls to uninterpreted functions is a conservative abstraction.

□

3.6 Dynamic data structures

RVT works on C (reference ANSI C99) programs, although not all features are supported. A major issue in applying rule (PROC-P-EQ) to C programs is that of dynamic data structures. Recall that deciding formulas with uninterpreted functions requires the comparison pair-wise of the arguments with which such functions are called, and a similar comparison of their outputs. If some of these arguments are pointers, such a comparison is meaningless. In this section we briefly describe RVT's method of treating pointer arguments of functions and dynamic data structures.

Whereas in nonpointer variables the comparison is between values, in the case of pointer variables the comparison should be between the data structures that they point to. A dynamic data structure can be represented as a graph whose vertices are structs and edges are the pointers that point from one struct to the other. We call such graph a *pointer-element graph*. We make a simplifying assumption that all dynamic structures that are passed to a function through pointer arguments or globals are in the form of trees, i.e., aliasing within dynamic structures and between function arguments is not allowed. We define equality of structures as follows:

Definition 2. (Iso-equal structures) *Two structures are iso-equal if their pointer-element graphs are isomorphic and the values at structs related by the isomorphism are equal.*

Let p_1, p_2 be paired pointer variables that are arguments to the functions that we wish to compare. RVT generates two iso-equal tree-like data structures with a bounded depth (see below) and with nondeterministic values (including possible null values for pointers). It then makes p_1 and p_2 point to these trees. This guarantees that the input structure is arbitrary but equivalent up to a bound, and under the assumption that on both sides it is a tree. A similar strategy is activated when we compare p_1 and p_2 that point to an output of the compared functions – the output structures are compared up to the same bound assuming they are trees.

What should be the bound on this tree? Recall that the code of the related subprograms that we check (the check-block) does not contain loops or recursion, and hence there is a bound on the maximal depth of the items this code can access in any dynamic data structure that is passed to the roots of the related subprograms. It is possible, then, to compute this bound, or at least overestimate it, by syntactic analysis. For example, searching for code that progresses on the structure such as `n = n -> next` for a pointer `n`. However, such a mechanism is not implemented yet in RVT and it relies instead on a user-defined bound.

4 Experiments

We tested RVT on several synthetic and limited-size industrial programs and attempted to prove equivalent different versions of these programs. We tested RVT with the following sets of programs.

Random programs We used a random program generator to create several dozen recursive programs of different sizes. The user specifies the probability to generate each type of variable, block, or operator. Variables can be global, local or formal arguments of functions. Types can be basic C types, structures or pointers to such types. Small differences between the two versions of each program are introduced in random places. We used this program to generate random yet executable C programs with up to 20 functions and thousands of lines of code. When the random versions are equivalent, RVT proves them to be partially-equivalent relatively fast, ranging from few seconds to 30 minutes. On non-equivalent versions, on the other hand, attempts to prove partial equivalence may run for many hours or run out of memory.

Industrial programs The small industrial programs that we tried are:

TCAS (Traffic Alert and Collision Avoidance System) is an aircraft conflict detection and resolution system used by all US commercial aircraft. We used a 300-line fragment of this program that was also used in [9].

MicroC/OS The core of MicroC/OS which is a low-cost priority-based preemptive real time multitasking operating system kernel for microprocessors, written mainly in C. The kernel is mainly used in embedded systems. The program is about 3000 lines long.

Matlab examples Parts of engine-fuel-injection simulation in Matlab which was generated in C from engine controller models. The tested parts contain several hundreds lines of code and use read-only arrays.

All these tests exhibit the same behavior as the random programs above. For equivalent programs, semantic-checks are very fast, proving equivalence in minutes. We did not encounter a case in which partially equivalent programs cannot be proven to be so due to the incompleteness of (PROC-P-EQ).

Recall that in the process of semantic checks, paired functions that cannot be proven equivalent are (logically) inlined. Our experience was that in such cases the proof becomes too hard: the decision procedure runs for hours or even fails to reach a decision at all. In some examples the bottleneck is the use of operators that burden the SAT solver, such as multiplication (*), division (/) and modulo (%) over integers. A simple solution in such cases is to *outline* these operators (i.e., take them out to a separate function). The reason is that RVT proves the equivalence of these separate functions syntactically and then replaces them with uninterpreted functions, which reduces the computation time dramatically.

5 Summary

We started the introduction by mentioning Tony Hoare's grand challenge, namely that of functional verification, and by mentioning that proving equivalence is a grand challenge in its own right, although an easier one. In this work we started exploring this direction in the context of C programs, and reported on our prototype tool RVT with which we were able to prove the equivalence of several small industrial programs. Our technique can be improved in several dimensions,

such as strengthening rule (PROC-P-EQ) with automatically generated invariants and finding more opportunities for making the verification conditions easier to decide. Investigating such opportunities for object-oriented code is another big challenge.

To summarize, the main contribution of this article is a method for an automatic, incremental proof, based on isolating functions from their callees and abstracting them with uninterpreted functions. This method keeps the verification conditions decidable and small relative to the size of the input programs. The initial syntactic checks and the decomposition mechanism helps meeting our goal of keeping the complexity sensitive to the changes rather than to the original size of the compared programs.

References

1. T. Arons, E. Elster, L. Fix, S. MadorHaim, M. Mishaeli, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, , and L. D. Zuck. Formal verification of backward compatibility of microcode. In *CAV05*, volume 3576 of *LNCS*. Springer, 2005.
2. S. Craciunescu. Proving the equivalence of clp programs. In *ICLP*, pages 287–301, 2002.
3. X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In *EMSOFT*, pages 307–316, 2005.
4. R. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.
5. M. Fowler. <http://www.refactoring.com>.
6. B. Godlin. Regression verification: Theoretical and implementation aspects. Master’s thesis, Technion, Israel Institute of Technology, 2008.
7. B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.
8. B. Godlin and O. Strichman. Regression verification. In *46th Design Automation Conference (DAC)*, 2009.
9. A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *CAV*, pages 453–456, 2004.
10. M. Haroud and A. Biere. SDL versus C equivalence checking. In *SDL Forum*, pages 323–338, 2005.
11. C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.
12. C. Hoare. Procedures and parameters: an axiomatic approach. In *Proc. Sym. on semantics of algorithmic languages*, (188), 1971.
13. T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
14. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
15. D. Kroening and O. Strichman. *Decision procedures – an algorithmic point of view*. Theoretical computer science. Springer, May 2008.
16. P. Manolios and M. Kaufmann. Adding a total order to acl2. In *The Third International Workshop on the ACL2 Theorem Prover*, 2002.
17. P. Manolios and D. Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 2006. to appear.

18. A. Pnueli, M. Siegel, and O. Shtrichman. Translation validation for synchronous languages. In *Proc. 25th Int. Colloq. on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 235–246. Springer, 1998.
19. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. Technical report, SACRES and Dept. of Comp. Sci., Weizmann Institute, Apr. 1997.
20. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'08*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.