Regression Verification

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

Ofer Strichman
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*.

Categories and Subject Descriptors

D2.4 [**Software/program verification**]

General Terms

Verification

Keywords

Software verification, Equivalence checking

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 (or even only its return value) 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 [8]. 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*[1].

The idea of proving equivalence between programs is not new, and in fact preceded the idea of functional verification.[2] 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 gen-

DAC'09, July 26-31, 2009, San Francisco, California, USA

[1] Refactoring is a popular set of techniques for rewriting existing code for various purposes.

[2] In his 1969 paper about axiomatic basis for computer programming [6], Hoare points to previous works from the late 50's on axiomatic treatment of the problem of proving equivalence between programs.

eral 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 (six different definitions appear in [3]). Here we focus on the following definition:

Definition 1. Partial equivalence Two programs P_1 and P_2 are said to be *partially equivalent* if any two terminating executions of P_1 and P_2 starting from the same inputs, return the same value.

The problem of 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 programs (after possible renaming of the global variables) consecutively with nondeterministic 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.

Sect. 2 below describes briefly an inference rule that we introduced in [3] 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. Due to lack of space 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 [2]. The theoretical background on the inference rule that we use can be found also in [3].

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., [11, 12]). 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. 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* [14, 13], 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. A recent example of translation validation, from the synchronous language SDL to C, is by Haroud and Biere [5].

2. PARTIAL EQUIVALENCE

To prove partial equivalence we suggested in [3] to use an inference rule in the style of Hoare's rule for recursive invocation [7]. Hoare's rule is

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

where *proc*-body is the body of the procedure *proc*, in which the recursive call is ignored. 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 [7] 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 of the recursion.

Our rule (PROC-P-EQ) (for 'Procedures Partial Equivalence') for partial equivalence between functions[3] P_1 and P_2 has the same flavor. For the case of P_1, P_2 being recursive functions without calls to other functions, it can be stated as follows:

$$\frac{\begin{array}{l} in[\text{call } P_1] = in[\text{call } P_2] \rightarrow out[\text{call } P_1] = out[\text{call } P_2] \vdash \\ in[P_1 \text{ body}] = in[P_2 \text{ body}] \rightarrow out[P_1 \text{ body}] = out[P_2 \text{ body}] \end{array}}{in[\text{call } P_1] = in[\text{call } P_2] \rightarrow out[\text{call } P_1] = out[\text{call } P_2]} \quad (2)$$

Informally, this means that if assuming that the recursive calls maintain the *congruence* condition (i.e., equal inputs lead to equal outputs) enables us to prove this condition over the bodies of P_1 and P_2 (i.e., P_1, P_2 without the recursive calls), then the congruence relation holds for P_1,P_2. In [3] we proved that this rule is sound (in fact the rule in [3] generalizes the rule presented here to the case of mutual recursion). Although the soundness proof refers to an artificial language, it has most of the features of an imperative language such as C. In Sect. 3.2 we will elaborate further on this point.

A convenient method for checking the premise of rule (PROC-P-EQ) is to replace the recursive call with an *uninterpreted function* (see, e.g., chapter 3 in [10]), because by definition it maintains the congruence condition. After performing this replacement we say that the calling function is *isolated*. Denote by P^{UF} the isolated version of a function P. Rule (PROC-P-EQ) can be reformulated accordingly:

$$\frac{\vdash_{\mathcal{UF}} in[P_1^{UF}] = in[P_2^{UF}] \rightarrow out[P_1^{UF}] = out[P_2^{UF}]}{in[\text{call } P_1] = in[\text{call } P_2] \rightarrow out[\text{call } P_1] = out[\text{call } P_2]}\ , \quad (3)$$

where $\vdash_{\mathcal{UF}}$ is some sound inference system that can also reason about uninterpreted functions. The key observation in using this rule is that its premise 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 and shows how partial equivalence is proven.

[3]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.

Example 1. Consider the two functions in Fig. 1. Let H be the uninterpreted function to which we map `gcd1` and `gcd2`. Figure 2 presents the isolated functions.

```
gcd1(int a, int b)
{ int g;
  if (!b) g = a;
  else
    a = a%b;
    g = gcd1(b, a);
  return g;
}
```

```
gcd2(int x, int y)
{ int z;
  z = x;

  if (y > 0)
    z = gcd2(y, z%y);
  return z;
}
```

Figure 1: Two functions to calculate GCD of two positive integers.

```
gcd1(int a, int b)
{ int g;
  if (!b) g = a;
  else
    a = a%b;
    g = H(b, a);
  return g;
}
```

```
gcd2(int x, int y)
{ int z;
  z = x;

  if (y > 0)
    z = H(y, z%y);
  return z;
}
```

Figure 2: After isolation of the functions, i.e., replacing their function calls with calls to the uninterpreted function H.

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., [10]). Briefly, this means that in each assignment of the form `x = exp;` the left-hand side variable `x` is replaced with a new variable, say $\texttt{x}_1$. Any reference to `x` after this line and before `x` is assigned again is replaced with the new variable $\texttt{x}_1$ (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 `gcd1`, denoted T_{gcd_1}, is

$$\left(\begin{array}{ll} a_0 = a & \wedge \\ b_0 = b & \wedge \\ b_0 = 0 \rightarrow g_0 = a_0 & \wedge \\ (b_0 \neq 0 \rightarrow a_1 = (a_0 \% b_0)) \wedge (b_0 = 0 \rightarrow a_1 = a_0) & \wedge \\ (b_0 \neq 0 \rightarrow g_1 = H(b_0, a_1)) \wedge (b_0 = 0 \rightarrow g_1 = g_0) & \wedge \\ g = g_1 & \end{array}\right) \tag{4}$$

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

$$\left(\begin{array}{ll} x_0 = x & \wedge \\ y_0 = y & \wedge \\ z_0 = x_0 & \wedge \\ y_0 > 0 \rightarrow z_1 = H(y_0, (z_0 \% y_0)) & \wedge \\ y_0 \leq 0 \rightarrow z_1 = z_0 & \wedge \\ z = z_1 & \end{array}\right) \tag{5}$$

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

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

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

Now suppose that the two compared functions P_1, P_2 call other functions P_1^c, P_2^c, respectively. Rule (PROC-P-EQ) can still be used if one of the following holds:

1. If P_1^c, P_2^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 P_1^c, P_2^c and their descendants are not recursive then they can be inlined in their callers. The premise of rule (PROC-P-EQ) is then checked as before.

3. Otherwise, if some of the descendants of P_1^c, P_2^c are recursive but were proven partially equivalent then these descendants can be abstracted with uninterpreted functions. As in the previous case P_1^c, P_2^c can then be inlined into their callers.

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. REGRESSION VERIFICATION

Our Regression Verification Tool (RVT) is geared towards C programs and hence we begin with a brief description of CBMC [9], the underlying decision procedure that we use for checking the premise of rule (PROC-P-EQ) and its generalizations 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. This enables CBMC to symbolically characterize the full set of possible executions restricted by these bounds, by a decidable formula f. 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. Before this iterative process begins, RVT makes two preliminary steps.

Loops All loops in P_1 and P_2 are replaced with recursive functions. This process is described in [2].

Pairing A pairing *pair* is built by pairing functions and global variables between the two compared programs.

Pairing 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. Note that wrong pairing does not affect soundness: pairing is used for generating the verification conditions, and hence wrong pairing can only fail a proof. The pairing algorithm works top-down: it initially pairs global variables with the same name and type. It then pairs functions with the same name, return type, and prototype. Then, within paired functions that are also syntactically equivalent up to variable names, it attempts to pair elements that appear in isomorphic locations. If these elements were already paired it just checks that the pairing according to this function agrees with the previous one, and otherwise it issues a warning. This process is repeated until no new pairing is discovered.

We assume here that as a minimum this process results in a bijective pairing between all *recursive* functions. Without this condition fulfilled, RVT can only attempt to prove partial equivalence of subprograms rooted at paired functions that fulfill this condition.[4]

The input to the main algorithm is thus two recursive programs without loops and a pairing *pair*. We denote by $pair_f$ the pairs of functions in *pair*.

3.1 A bottom-up decomposition algorithm

The equivalence check in RVT, in its basic form, is presented in Algorithm 1. It is based on traversing bottom-up the call graphs of the two programs to be compared. This algorithm can be applied directly to two call graphs without loops of length larger than 1 (i.e., no mutual recursion). The more general case is considered in [2].

The algorithm progresses bottom-up on the call graphs, and updates the labels on the nodes to "Equivalent" or "Failed". The progress on the graph is made by a function NEXT-UNMARKED-PAIR() (not presented) which returns the next unmarked pair in $pair_f$, according to a BFS order on the reversed call graph of one of the two compared programs. This function aborts if either all pairs are already marked or it finds that the call-graphs are inconsistent (inconsistency means that there are two pairs of functions $\langle f, f' \rangle \in pair_f$ and $\langle g, g' \rangle \in pair_f$ such that f is a descendant of g but f' is an *ancestor* of g').

The equivalence check in line 5 is conducted by verifying, with CBMC, that various assertions hold in a single loop-free and recursion-free C program *check-block*(f, g) that RVT constructs (see below). CBMC returns TRUE if the assertions hold and FALSE otherwise. We call these checks 'semantic checks' to distinguish them from the syntactic checks in line 2. We now proceed by describing *check-block*(f, g).

Check-blocks Consider the maximal connected subDAG rooted at f that contains only functions that are unpaired or marked "Failed". Let S_f denote this set of functions (excluding f). S_g is defined similarly with respect to g. The program *check-block*(f, g) consists of the following elements:

1. The functions f,g and all functions in S_f, S_g, such that
   - Name collisions in global identifiers of the two programs are solved by renaming;
   - All calls to f, g are replaced with calls to $UF(f)$, $UF(g)$, respectively;
   - For all $\langle h_1, h_2 \rangle \in pair_f$ such that $h_1, h_2 \notin \{S_f \cup S_g\}$, calls to h_1, h_2 are replaced, respectively, with calls to $UF(h_1)$ and $UF(h_2)$. (Observe that the pair $\langle h_1, h_2 \rangle$ is marked "Equivalent").
2. The `main()` function, which consists of:
   - Assignment of nondeterministic but equal values to inputs of f and g;
   - Calls to f, g; and
   - Assertion that the outputs of f and g are equal.

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

- The check-block is nonrecursive. This is because when a recursive pair $\langle f, g \rangle \in pair_f$ is labeled "Failed" the algorithm aborts in line 8, and hence the code of f, g will not be part of future check-blocks.
- The code of each nonrecursive pair $\langle f, g \rangle \in pair_f$ that is labeled "Failed" 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 $\langle f, g \rangle \in pair_f$ that is proven to be equivalent does not participate in any subsequent check-block. It is replaced with uninterpreted functions in all subsequent semantic 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 paired isolated functions also proves their partial equivalence by rule (PROC-P-EQ).

Equivalence specification Since just proving the equivalence of the return value is insufficient in practice, RVT allows the user to specify the equivalence criterion with pairs of tuples of the form $\langle$*label*, *condition*, *expression*$\rangle$. It then adds assertions to the check-blocks that check that at the locations specified by the labels the expressions are equivalent every time the conditions hold.

Complexity Each pair in $pair_f$ is labeled at most once by either "Failed" or "Equivalent". Thus, if $n = |pair_f|$, which, in turn, cannot be larger than the number of functions, then the algorithm performs not more than n syntactic and n semantic checks.

Example 2. Consider the call graphs in Fig. 3. Assume that for $i = 1, \ldots, 6$ we have $\langle f_i, f_i' \rangle \in pair_f$, and that the functions marked by grey nodes in Fig. 3 are syntactically equivalent to their counterparts. We describe step by step the execution of Algorithm 1:

[4]RVT can also attempt to prove k-equivalence in this case, i.e., prove that there is no trace contradicting the equivalence which requires a recursion depth larger than k. The description of this feature is beyond the scope of this paper.

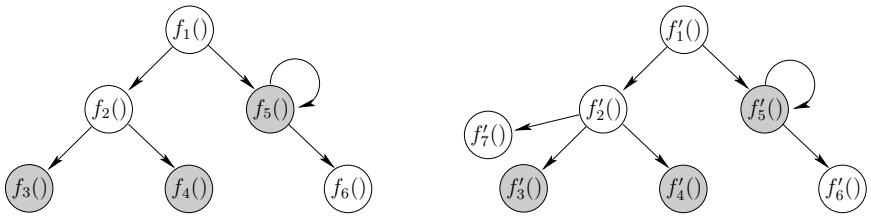

Figure 3: Two call graphs for Example 2. A node is grey if it is syntactically equivalent to its counterpart.

Algorithm 1 A basic call-graph based algorithm for attempting to prove the partial equivalence of pairs of functions.

Procedure COMPARE()
input: Call graphs CG_1 and CG_2 and a pairing $pair_f$.
output: Marking of pairs in $pair_f$ with "Equivalent" or "Failed".

1: $\langle f, g\rangle$ = NEXT-UNMARKED-PAIR() ▷ Bottom-up. Aborts if none.
2: **if** $\langle f, g\rangle$ are syntactically equivalent and all their children (not including recursive calls) are marked by "Equivalent" **then**
3: Mark $\langle f, g\rangle$ by "Equivalent".
4: **else**
5: **if** CBMC(*check-block*(f, g)) **then** ▷ Semantic check
6: Mark $\langle f, g\rangle$ "Equivalent".
7: **else**
8: **if** f, g are recursive **then** abort.
9: Mark $\langle f, g\rangle$ "Failed".
10: goto line 1.

1. In line 3 pairs $\langle f_3, f'_3\rangle, \langle f_4, f'_4\rangle$ are marked "Equivalent".
2. The program *check-block*(f_2, f'_2) is sent to CBMC. Now $S_{f'_2} = \{f'_7\}$ and hence this program contains also the code of f'_7, whereas f_3, f'_3, f_4, f'_4 are replaced by uninterpreted functions. Assume that this semantic check fails. Then the pair $\langle f_2, f'_2\rangle$ is marked "Failed".
3. The program *check-block*(f_6, f'_6) is sent to CBMC. Assume that the check fails and hence the pair is marked "Failed".
4. The program *check-block*(f_5, f'_5) is sent to CBMC. Since $S_{f_5} = \{f_6\}$ and $S_{f'_5} = \{f'_6\}$, this program contains also f_6, f'_6. The recursive calls are replaced with uninterpreted functions(based on rule (PROC-P-EQ)). Assume that this time the check succeeds. Then $\langle f_5, f'_5\rangle$ is marked "Equivalent" in line 6.
5. The program *check-block*(f_1, f'_1) is sent to CBMC. Now $S_{f_1} = \{f_2\}$ and $S_{f'_1} = \{f'_2, f'_7\}$. Hence, the respective call subgraphs contain also f_2, f'_2 and f'_7, whereas $f_3, f'_3, f_4, f'_4, f_5, f'_5$ are replaced with uninterpreted functions. Assume this check succeeds. The algorithm marks $\langle f_1, f'_1\rangle$ with "Equivalent".

At this stage all function pairs are marked by either "Equivalent" or "Failed" and the algorithm terminates.

The soundness proof of algorithm 1 appears in [G08] and is omitted here due to space limitations.

3.2 Specific issues with C programs

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 a tree-like data structure with a bounded depth (see below) and with nondeterministic values. It then makes both p_1 and p_2 point to this tree. 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.

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`. Such a mechanism is not implemented yet in RVT, however, 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 dif-

ferent versions of these programs. In each case we specified simple equivalence criteria in the form of conditional expressions as explained before. 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 in versions 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 [4].

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.

6. 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] B. Godlin. Regression verification: Theoretical and implementation aspects. Master's thesis, Technion, Israel Institute of Technology, 2008.

[3] B. Godlin and O. Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.

[4] A. Groce, D. Kroening, and F. Lerda. Understanding counterexamples with explain. In *CAV*, pages 453–456, 2004.

[5] M. Haroud and A. Biere. SDL versus C equivalence checking. In *SDL Forum*, pages 323–338, 2005.

[6] C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[7] C. Hoare. Procedures and parameters: an axiomatic approach. *In Proc. Sym. on semantics of algorithmic languages*, (188), 1971.

[8] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[9] 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.

[10] D. Kroening and O. Strichman. *Decision procedures – an algorithmic point of view*. Theoretical computer science. Springer, May 2008.

[11] P. Manolios and M. Kaufmann. Adding a total order to acl2. In *The Third International Workshop on the ACL2 Theorem Prover*, 2002.

[12] P. Manolios and D. Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 2006. to appear.

[13] 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.

[14] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'08*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.