Regression Verification for Multi-Threaded Programs

Sagar Chaki[1] Arie Gurfinkel[1] Ofer Strichman[1,2]

[1] SEI/CMU, Pittsburgh, USA
[2] Technion, Haifa, Israel
chaki@sei.cmu.edu arie@cmu.edu ofers@ie.technion.ac.il

Abstract. Regression verification is the problem of deciding whether two similar programs are equivalent under an arbitrary yet equal context, given some definition of equivalence. So far this problem has only been studied for the case of single-threaded deterministic programs. We present a method for regression verification to establish partial equivalence (i.e., input/output equivalence of terminating executions) of multi-threaded programs. Specifically, we develop two proof-rules that decompose the regression verification between *concurrent* programs to that of regression verification between *sequential* functions, a more tractable problem. This ability to avoid composing threads altogether when discharging premises, in a fully automatic way and for general programs, uniquely distinguishes our proof rules from others used for classical verification of concurrent programs.

1 Introduction

Regression verification [4, 5] is the problem of deciding whether two similar programs are equivalent under an arbitrary yet equal context. The problem is parameterized by a notion of equivalence. In this paper, we focus on *partial equivalence* [4], i.e., input/output equivalence of terminating executions. Regression verification under partial equivalence – which we refer to simply as regression verification – is undecidable in general. However, in practice it can be solved in many cases fully automatically for *deterministic single-threaded programs*, even in the presence of loops, recursion and dynamic memory allocation. For example, the algorithm suggested in [5] progresses bottom-up on the call graphs of the two programs, and attempts to prove equivalence of pairs of functions while abstracting descendants that were already proved equivalent with uninterpreted functions. This algorithm is implemented in two tools – RVT [5] and SymDiff [9] – both of which output a list of provably equivalent function pairs.

The ability to perform regression verification adds several elements to the developer's toolbox: checking that no change has propagated to the interface after refactoring or performance optimization; checking backward compatibility; performing impact analysis (checking which functions may possibly be affected by a change, in order to know which tests should be repeated), and more.

Multithreaded (MT) programs are widely deployed, which makes the extension of regression verification to such programs an important problem. This task

is challenging for at least two reasons. First — since MT programs are inherently nondeterministic due to the scheduler, we need an appropriate notion of equivalence for nondeterministic programs. The standard definition of partial equivalence mentioned above is inadequate, since it implies that a nondeterministic program is not even equivalent to itself: given the same input, the program may produce different outputs.[1]

Second — while regression verification of sequential programs is broken down to proofs of I/O equivalence of pairs of functions, in the case of MT programs the behavior of functions is affected by other threads, which makes a similar decomposition to the level of functions much harder. Compositional verification methodologies [11, 7] and tools [6, 2] for MT programs target reachability properties of a *single* program, and decompose only to the level of individual *threads*. They are therefore not directly applicable to our problem.

In this paper we propose theoretical foundations for regression verification of multi-threaded recursive programs and address the above two challenges. First, we extend the definition of partial equivalence to non-deterministic programs. Second, assuming a bijective correspondence mapping between the functions and global variables of the two programs, we present two proof rules whose premises only require verification of sequential programs, at the granularity of individual functions. We prove that these rules are sound under our extended notion of partial equivalence. For the first rule, each premise verifies that a pair of corresponding functions generate the same observable behavior under an arbitrary yet equal environment. The second rule has premises that are weaker, but also computationally harder to discharge. Specifically, each premise verifies that a pair of corresponding functions generate the same observable behavior under an arbitrary yet equal environment that is consistent with some overapproximation of the other threads in the program. For both rules, each premise is discharged by verifying a sequential program. A key feature of our proof rules therefore is that they enable decomposition to the level of both threads and functions.

The rest of the article is structured as follows. In the next section we present our extended notion of partial equivalence. In Sec. 3 we list our assumptions about the input programs, and describe how they should be preprocessed for our procedure to work. In Sec. 4 we describe our first rule and prove its soundness. In Sec. 5 we present the second rule and prove its soundness. Finally, in Sec. 6, we conclude and describe some directions for future work.

2 Equivalence of multi-threaded programs

Let P be a multi-threaded program. P defines a relation between inputs and outputs, which we denote by $R(P)$. Let $\Pi(P)$ denote the set of terminating computations of P. Then:

$$R(P) = \{(\boldsymbol{in}, \boldsymbol{out}) \mid \exists \pi \in \Pi(P). \;\; \pi \text{ begins in } \boldsymbol{in} \text{ and ends in } \boldsymbol{out}\} \; .$$

[1] An indication of the difficulty of this problem is given by Lee's statement in [10], that "with threads, there is no useful theory of equivalence".

Definition 1 (Partial equivalence of nondeterministic programs). *Two nondeterministic programs P, P' are partially equivalent if $R(P) = R(P')$.*

We denote by $p.e.(P, P')$ the fact that P and P' are partially equivalent. Note that the definition refers to whole programs, and that by this definition every program is equivalent to itself. If loops and recursion are bounded this problem is decidable, as we show in the full version of this article [1]. Recall, however, that here we are concerned with the unbounded case, and with the question of how to decompose the verification problem to the granularity of threads and functions. This is the subject of this article.

3 Assumptions, preprocessing and mapping

We assume C as the input languages, with few restrictions that will be mentioned throughout this section. We generally refrain in this paper from discussing in detail issues that are also relevant to regression verification of sequential programs (e.g., issues concerning the heap, aliasing etc), because these are covered in earlier publications [4, 9].

The input program P is assumed to consist of a finite and fixed set of threads, i.e., no dynamic thread creation and deletion. A k-threaded program P is written as $f_1 \parallel \ldots \parallel f_k$ where, for $i \in [1..k]$, the i-th thread is rooted at f_i. The call graph of P is written as $\textsc{cg}(P)$. The call graph of a function f in P, denoted $\textsc{cg}(f)$, is the subgraph of P that can be reached from f. We assume that threads have disjoint call graphs.

We denote by $ReadParam(f)$ and $WriteParam(f)$ the set of parameters and global variables that are read and written-to, respectively, by functions in $\textsc{cg}(f)$. In general computing this information precisely is impossible, but over-approximations are easy to compute, while sacrificing completeness. For simplicity we will assume that these sets are given to us. Note that the intersection of these sets is not necessarily empty. A global variable is called *shared* if it is accessed by more than one thread. For simplicity, but without losing generality, *we consider all outputs of each function as if they were shared*, even if in practice they are local to a thread.

By convention x, x_1, x_2 etc. denote variables that are (really) shared (i.e., not outputs), t, t_1, t_2 etc. denote local variables, and exp denotes an expression over local variables. Primed symbols indicate that they refer to P'. Function names prefixed by `UF` denote uninterpreted functions. The signature and return type of these functions are declared implicitly, by the actual parameters and the variable at the left-hand side of the assignment, respectively. For example, if we use a statement of the form `t = UF_x(t1, t2)`, where `t,t1,t2` are integers, it means that we also declare an uninterpreted function `int UF_x(int, int)`.

3.1 Global preprocessing

We assume that all programs are preprocessed as follows:

- Loops are outlined [3] to recursive functions.

– Mutually recursive functions are converted [8] to simple recursion.
– Non-recursive functions are inlined.
– Auxiliary local variables are introduced to load and store shared variables explicitly such that: (i) a shared variable x only appears in statements of the form $t = x$ or $x = exp$, and (ii) every auxiliary variable is read once.
– If a function has a formal return value, it is replaced with an additional parameter sent to it by reference.

3.2 Mapping

We assume that after preprocessing, the target programs P and P' have the same number of threads and overall number of functions. Specifically, let $P = f_1 \parallel \ldots \parallel f_k$ and $P' = f'_1 \parallel \ldots \parallel f'_k$, and let the set of overall functions of P and P' be $\{g_1, \ldots, g_n\}$ and $\{g'_1, \ldots, g'_n\}$, respectively. We assume the following two mappings:

– A bijection $\phi_F : \{g_1, \ldots, g_n\} \mapsto \{g'_1, \ldots, g'_n\}$, such that $\forall i \in [1..k]\mathbf{.}\ \phi_F(f_i) = f'_i$ and furthermore, if $(g, g') \in \phi_F$, then
  - $\forall i \in [1..k]\mathbf{.}\ g \in \text{CG}(f_i) \iff g' \in \text{CG}(f'_i)$.
  - g and g' have the same prototype (list of formal parameter types).
  - Let p_i and p'_i denote the i-th parameter of g and g' respectively. Then $p_i \in \mathit{ReadParam}(g) \iff p'_i \in \mathit{ReadParam}(g')$ and $p_i \in \mathit{WriteParam}(f) \iff p'_i \in \mathit{WriteParam}(g')$.

  For convenience, we assume that $\forall i \in [1..n]\mathbf{.}\ \phi_F(g_i) = g'_i$.
– A bijection ϕ_G between the global variables of P and P', such that if $(v, v') \in \phi_G$, then
  - v and v' are of the same type,
  - v is a shared variable iff v' is a shared variable,
  - $\forall (g, g') \in \phi_F\mathbf{.}\ v \in \mathit{ReadParam}(g) \iff v' \in \mathit{ReadParam}(g')$ and $v \in \mathit{WriteParam}(g) \iff v' \in \mathit{WriteParam}(g')$.

Failure in finding the above two mappings dooms the proof. Note that the existence of ϕ_G implies that after preprocessing, P and P' also have the same number of global variables.

4 First Proof Rule

We now present our first proof rule for regression verification of P and P'. We begin with a specific transformation of a function f to a new function $\widehat{f}$, which we use subsequently in the premise of our proof rule.

4.1 Function transformation: from f to $\widehat{f}$.

Let $\mathit{ActualReadParam}(f)$ be the actual parameters and global variables sent respectively to the elements in $\mathit{ReadParam}(f)$. We construct $\widehat{f}$ from f via the

– Introduce a global counter `c` initialized to 0, and a list `out` of tuples $\langle Action, identifier, values \ldots \rangle$.
– *Read*: `t := x;` $\Rightarrow$ `t := UF_x(c); out += (R, "x"); c++;`
– *Write*: `x := exp;` $\Rightarrow$ `x := exp; out += (W, "x", exp); c++;`
– *Function call*: `foo`$(a_1, \ldots, a_m)$; $\Rightarrow$
`∀w ∈` $WriteParam$`(foo). w = UF`$_{foo_w}$$(ActualReadParam$`(foo));`
`out += (C, "foo",` $ActualReadParam$`(foo));`

Fig. 1. Constructing $\widehat{f}$ from f, for a function f in P. The operator "`+=`" appends an element to the end of `out`. Functions in P' are translated slightly differently (see text).

```
f(int t1) {
  int t2, t3 = 1;
  x = t1 + 1;
  t2 = x;
  foo(t3, &t1, &t2);
}
```

```
g(int i1,int *o1) {
  int t;
  if (i1 ≤ 0) {
    *o1 = 1;
  } else {
    g(i1 - 1, &t);
    *o1 = (*t) * i1;
} }
```

```
f̂(int t1) {
  int t2, t3 = 1, c = 0;
  x = t1 + 1;
  out+=(W,"x", t1 + 1); c++;
  t2 = UF_x(c);
  out+=(R, "x"); c++;
  t1 = UF_foo_t1(t3);
  t2 = UF_foo_t2(t3);
  out+=(C, foo, t3);
}
```

```
ĝ(int t1,int *o1) {
  int t, c = 0;
  if (i1 ≤ 0) {
    out+=(W, "o1", 1); c++;
  } else {
    t = UF_g_t(i1 - 1);
    out+=(C, g, i1 - 1);
    out+=(W, "o1", (*t) * i1); c++;
} }
```

Fig. 2. Example conversions of functions `f` and `g` to $\widehat{\texttt{f}}$ and $\widehat{\texttt{g}}$.

transformation described in Fig. 1. In the figure, $\Rightarrow$ indicates a specific transformation, with the left being the original code (in f) and the right being the new code (in $\widehat{f}$). The transformation of a function f' in P' is the same except that the elements in $WriteParam(f')$ are renamed to their counterparts in f according to the map ϕ_G, ensuring that f and $\phi_F(f)$ invoke the same uninterpreted function.

Example 1. Fig. 2 shows functions `f` and `g` and their translations to $\widehat{\texttt{f}}$ and $\widehat{\texttt{g}}$. Function `foo` called in `f` has three parameters, the first of which is only in $ReadParam$(`foo`) and the other two only in $WriteParam$(`foo`). The update of `x` in $\widehat{\texttt{f}}$ is not necessary in this case because it is not used, but it would have been used had `x` was an element of $ReadParam$(`foo`). Shifting our attention to `g`, this function computes the factorial of its input. It has two parameters, which are in $ReadParam$(`g`) and $WriteParam$(`g`), respectively. □

Intuition. Intuitively, the environment in which each thread operates is a stream of read and write (RW) instructions to shared variables. Let f and f' be two functions and let E and E' be environments generated by the *other* threads in their respective programs (P and P'). To prove the equivalence of f, f', we assume that E and E' are identical *but only if f and f''s interaction with them so far* has been equivalent. This assumption is checked as part of the premise of our rule as follows. Consider two shared variables x, x' in f, f', respectively. To emulate a possible preemption of f just before it reads x, it is sound to let it read a nondeterministic value. But since we want to assume that f and f' operate in the same environment, we want to ensure that if they are read at equal locations in their own RW stream, then they are assigned the *same* nondeterministic value. To this end, we replace each read of x and x' with the uninterpreted function call $UF_x(c)$. Since c is the current location in the RW stream and we use the same uninterpreted function $UF_x()$ for both x and x', we achieve the desired effect.

We prove that f and f' are observationally equivalent via the list `out`. For simplicity, we refer to the list `out` introduced during the construction of $\widehat{f}$ as $\widehat{f}.out$. In essence, the equality of $\widehat{f}.out$ and $\widehat{f'}.out$ implies that f and f' read (and write) the same values from (and to) the shared variables, and call the same functions with the same actual parameters, and in the same order. We now present our first proof rule formally.

4.2 The Proof Rule

We define the predicate $\delta(f)$ to be true if and only if the sequential program $VC_\delta(f)$, given below in pseudo-code, is valid (i.e., the assertion in $VC_\delta(f)$ is not violated) for all input vectors $\boldsymbol{in}$:

$$VC_\delta(f): \quad \widehat{f}(\boldsymbol{in}); \widehat{f'}(\boldsymbol{in}); rename(\widehat{f'}.out); assert(\widehat{f}.out = \widehat{f'}.out);$$

The function *rename* renames identifiers of functions and shared variables to their counterparts according to ϕ_F and ϕ_G, respectively. We omit some details on the construction (e.g., how $\boldsymbol{in}$ is generated when the signatures of f, f' include pointers), and verification of $VC_\delta(f)$. These details are available elsewhere [3], where similar programs are constructed for single-threaded programs. It should be clear, though, that validity of $VC_\delta(f)$ is decidable because there are no loops and (interpreted) function calls in $\widehat{f}$ and $\widehat{f'}$. Our first proof rule for partial equivalence of two MT programs P, P' is:

$$\frac{\forall i \in [1..n].\ \delta(f_i)}{p.e.(P, P')}\ . \tag{1}$$

Example 2. Consider the programs P in Fig. 3. For a fixed positive value of the shared variable `x`, `f1` computes recursively the GCD of `x` and the input argument `t`, if `t` > 0. The second thread `f2` changes the value of `x` to a nondeterministic value. `f1()` is assumed to be called from another function that first sets x

```
void f1(int td, int *o) {
  int t1, t2, t3;
  if (td <= 0) t2 = x;
  else {
    t1 = x;
    t3 = t1 % td;
    x  = td;
    f1(t3, &t2);
  }
  *o = t2;
}

void f2() {
  int t;
  x = t;
}
```

P

```
void f1'(int td, int *o') {
  int t1, t2;
  t2 = x';
  if (td > 0) {
    t1 = x';
    x' = td;
    f1'(t1 % td, &t2);
  }

  *o' = t2;
}

f2'() {
  int t;
  x' = t;
}
```

P'

Fig. 3. Two MT-programs for Example 2.

to some initial value (not shown here for simplicity). The program P' on the right does the same in a different way. We wish to check whether these two programs are partially equivalent. We assume that $\phi_F = \{(\texttt{f1},\texttt{f1'}), (\texttt{f2},\texttt{f2'})\}$, $\phi_G = \{(\texttt{x},\texttt{x'}),\ (\texttt{o},\texttt{o'})\}$. Note that in the construction we refer to `o,o'` as shared, although they are not, because of our convention that output variables are considered as shared. Also, we have $ActualReadParam(\texttt{f1}) = \{\texttt{t3},\ \texttt{x}\}$, $WriteParam(\texttt{f1}) = \{\texttt{o},\ \texttt{x}\}$, $ActualReadParam(\texttt{f1'}) = \{\texttt{t1}\ \%\ \texttt{td},\ \texttt{x'}\}$ and $WriteParam(\texttt{f1'}) = \{\texttt{o},\ \texttt{x'}\}$. Fig. 4 presents pseudo-code for $\delta(\texttt{f1})$. Note that the input `in` sent to $\widehat{\texttt{f1}}$ and $\widehat{\texttt{f1'}}$ is nondeterministic. $\delta(\texttt{f2})$ is trivial and not shown here. Both programs are valid, and hence by (1), $p.e.(P, P')$ holds. □

Note that when constructing $\widehat{f}$, we record both reads and writes in $\widehat{f}.out$. The following example shows that ignoring the order of reads makes (1) unsound.

Example 3. Consider the 2-threaded programs P (left) and P' (right) shown in Fig. 5. Assume that all variables are initialized to 0, and `x3,x4` are the outputs. P' is identical to P other than the fact that the first two lines in `f1()` are swapped. Thus, if reads are not recorded in $\widehat{f}.out$, then $VC_\delta(f_1)$ and $VC_\delta(f_2)$ are both valid. Hence, our proof rule would imply that P and P' are partially equivalent. But this is in fact wrong, as we now demonstrate.

If `x4 = 1` at the end of P's execution, then the instruction `t2 = x1` in `f2()` must have happened after the instruction `x1 = 1` in `f1()`. Therefore `t1` reads the value of `x2` before it is updated by `f2()`, which means that `t1`, and hence `x3`, are equal to 0. Hence, at the end of any execution of P, `x4 = 1` $\Rightarrow$ `x3 = 0`.

On the other hand, in P', after the computation `x1 = 1; t2 = x1; x2 = 2; t1 = x2; x3 = t1; x4 = t2;` we have (`x4 = 1, x3 = 2`). Since this output is impossible in P, then $\neg p.e.(P, P')$. Hence, our proof rule would be unsound. □

```
void f1^ (int td, int *o) {
  int t1, t2, t3, c = 0;
  if (td <= 0) {
    //2 ▷ t2 = x;
    t2 = UF_x(c);
    out1 += (R, "x"); c++;
  } else {
    //2 ▷ t1 = x;
    t1 = UF_x(c);
    out1 += (R, "x"); c++;
    //3 ▷ x  = td;
    t3 = t1 % td;
    x  = td;
    out1 += (W, "x", td); c++;
    //3 ▷ t2 = f1(t3, &t2);
    t2 = UF_f1_o(t3, x);
    x  = UF_f1_x(t3, x);
    out1 += (C, f1, t3, x);
  }
  //1 ▷ *o = t2;
  out1 += (W, "*o", t2); c++;
}
```

```
void f1'^(int td, int *o') {
  int t1, t2, c = 0;
  //2 ▷ t2 = x';
  t2 = UF_x(c);
  out2 += (R, "x'"); c++;
  if (td > 0) {
    //2 ▷ t1 = x';
    t1 = UF_x(c);
    out2 += (R, "x'"); c++;
    //2 ▷ x' = td;
    x' = td;
    out2 += (W, "x'", td); c++;
    //3 ▷ f1'(t1 % td, &t2);
    t2 = UF_f1_o(t1 % td, x');
    x'  = UF_f1_x(t1 % td, x');
    out2 += (C, f1, t1 % td, x');
  }
  //1 ▷ *o' = t2;
  out2 += (W, "*o'", t2); c++;
}

main() {
  int in;
  f1^(in);  f1'^(in);
  rename(out2);
  assert(out1 == out2);
}
```

Fig. 4. For Example 2: Pseudo-code for $\delta(\texttt{f1})$, where $n \triangleright X$ denotes that the next n lines encode X.

The above example also demonstrates that even minor alterations in the order of reads and writes in a thread – alterations that do have any effect in a sequential program – lead to loss of partial equivalence. This leads us to believe that there is little hope for a rule with a significantly weaker premise than (1).

4.3 Definitions

In this section we present definitions used later to prove the soundness of (1).

Definition 2 (Finite Read-Write trace). *A finite Read-Write trace (or RW-trace for short) is a sequence $(A, var, val)^*$, where $A \in \{R, W\}$, var is a shared variable identifier and val is the value corresponding to the action A on var.*

By 'trace' we mean a finite RW trace, and RW is the set of all RW traces.

Definition 3 (Function semantics). *The* semantics *of a function f under input $\boldsymbol{in}$ is the set of traces possible in $f(\boldsymbol{in})$ under an arbitrary program environment and input.*

We denote by $[f(\boldsymbol{in})]$ the semantics of f under input $\boldsymbol{in}$.

```
f1() {
  t1 = x2;
  x1 = 1;
  x3 = t1;
}
```

```
f2() {
  t2 = x1;
  x2 = 2;
  x4 = t2;
}
```

```
f1'() {
  x1 = 1;
  t1 = x2;
  x3 = t1;
}
```

```
f2'() {
  t2 = x1;
  x2 = 2;
  x4 = t2;
}
```

Fig. 5. Example programs P (left) and P' (right). All variables are of integer type.

Definition 4 (Sequential consistency). *An interleaving t of traces $t_1, \ldots, t_n$ is* sequentially consistent *if when (W, var, v_1) is the last write action to var before a read action (R, var, v_2) in t, then $v_1 = v_2$.*

Let $\bowtie(t_1, \ldots, t_n)$ denote the set of sequentially consistent interleavings of $t_1, \ldots, t_n$. The extension to sets of traces $S_1, \ldots, S_n$ is given by:

$$\bowtie(S_1, \ldots, S_n) = \bigcup_{\boldsymbol{t} \in S_1 \times \cdots \times S_n} \bowtie(\boldsymbol{t}) \ .$$

Definition 5 (Program semantics). *Let $P = f_1 \parallel \ldots \parallel f_k$ be a program. The semantics of P, denoted by $[P]$, is the set of terminating traces defined by:*

$$[P] = \bigcup_{\boldsymbol{in}} \bowtie([f_1(\boldsymbol{in})], \ldots, [f_k(\boldsymbol{in})]) \ .$$

Example 4. Consider the functions `f1()` and `f2()` from Fig. 5. Let $\mathbb{Z}$ be the set of all integers. We have:

$$[\texttt{f1}] = \bigcup_{z \in \mathbb{Z}} \{\langle (R, \texttt{x2}, z), (W, \texttt{x1}, 1), (W, \texttt{x3}, z)\rangle\}$$
$$[\texttt{f2}] = \bigcup_{z \in \mathbb{Z}} \{\langle (R, \texttt{x1}, z), (W, \texttt{x2}, 2), (W, \texttt{x4}, z)\rangle\}$$

Now consider the program $P = \texttt{f1} \parallel \texttt{f2}$. Assume that all global variables are initialized to 0 Then, we have:

$$\begin{aligned}[P] = \{\ & \langle (R, \texttt{x2}, 0), (W, \texttt{x1}, 1), (W, \texttt{x3}, 0), (R, \texttt{x1}, 1), (W, \texttt{x2}, 2), (W, \texttt{x4}, 1)\rangle, \\ & \langle (R, \texttt{x1}, 0), (W, \texttt{x2}, 2), (W, \texttt{x4}, 0), (R, \texttt{x2}, 2), (W, \texttt{x1}, 1), (W, \texttt{x3}, 2)\rangle, \\ & \langle (R, \texttt{x2}, 0), (R, \texttt{x1}, 0), (W, \texttt{x1}, 1), (W, \texttt{x3}, 0), (W, \texttt{x2}, 2), (W, \texttt{x4}, 0)\rangle, \ \ldots\}\end{aligned}$$

□

Let $[\widehat{f}(\boldsymbol{in})]$ denote the possible values of $\widehat{f}.out$ under input $\boldsymbol{in}$. We now show how $[f(\boldsymbol{in})]$ is obtained from $[\widehat{f}(\boldsymbol{in})]$, by recursively expanding all function calls. For conciseness from hereon we frequently omit $\boldsymbol{in}$ from the notations $[\widehat{f}(\boldsymbol{in})]$ and $[f(\boldsymbol{in})]$, i.e., we write $[\widehat{f}]$ and $[f]$ instead.

Definition 6 (Finite Read-Write-Call trace). *A finite Read-Write-Call trace (or RWC trace for short) is a sequence $\{(A, var, val) \cup (C, f, a_1, \ldots, a_k)\}^*$, where A, var and val are the same as RW traces, f is a function, and $a_1, \ldots, a_k$ are values passed as arguments to f.*

For an RWC trace t, we write $CS(t)$ to mean the set of functions appearing in t, i.e.,:

$$CS(t) = \{f \mid (C, f, \ldots) \in t\} .$$

Expanding a function call requires to map it to the traces of the called function. For this purpose we define a function $\mu : CS(t) \mapsto 2^{RW}$ (recall that RW is the set of all traces). Then, $\mathbf{inline}(t, \mu) \subseteq RW$ is defined as follows: a trace t' belongs to $\mathbf{inline}(t, \mu)$ iff t' is obtained by replacing each element $(C, f, \ldots)$ in t with an element of $\mu(f)$.

Definition 7 (Bounded semantics of a function). *The* bounded semantics *of a function f is its RW traces up to a given recursion depth. Formally:*

$$[f]^0 = [\widehat{f}] \cap RW$$

and for $i > 0$,

$$[f]^i = \bigcup_{w \in [\widehat{f}]} \mathbf{inline}(w, \mu_f^i) \cap [f], \textit{ where}$$
$$\mu_f^i(f) = [f]^{i-1} \textit{ and } \forall g \neq f \textit{ called by } f\textbf{.}\ \mu_f^i(g) = [g] .$$

Less formally, at a recursive call (i.e., when $g = f$), μ_f^i inlines a trace of f that involves fewer than i recursive calls of f, and at a nonrecursive function call (i.e., when $g \neq f$) it inlines an arbitrary trace of g. Observe that the semantics of a function can be defined as a union of its bounded semantics:

$$[f] = \bigcup_{i \geq 0} [f]^i . \tag{2}$$

Example 5. Recall the factorial function g from Fig. 2. Then we have:

$$[\widehat{\mathsf{g}}] = \{\langle (W, \mathtt{o1}, 0!)\rangle\} \cup \bigcup_{z \in \mathbb{Z} \wedge z > 0} \langle (C, \mathsf{g}, z-1), (W, \mathtt{o1}, z!)\rangle$$
$$\forall i \geq 0\textbf{.}\ [\mathsf{g}]^i = \bigcup_{0 \leq z \leq i} \{\langle (W, \mathtt{o1}, 0!), (W, \mathtt{o1}, 1!), \ldots, (W, \mathtt{o1}, z!)\rangle\}$$
$$[\mathsf{g}] = \bigcup_{z \geq 0} \{\langle (W, \mathtt{o1}, 0!), (W, \mathtt{o1}, 1!), \ldots, (W, \mathtt{o1}, z!)\rangle\}$$

□

4.4 Soundness

We now prove the soundness of (1) in three stages:

1. In Theorem 1 we prove that for any function f, the following inference is sound:
$$\frac{\forall g \in \textsc{cg}(f)\textbf{.}\ \delta(g)}{\forall \boldsymbol{in}\textbf{.}\ [f(\boldsymbol{in})] = [f'(\boldsymbol{in})]} .$$
This establishes the connection between the premise of (1) and the equal semantics of mapped threads.

2. Then, in Theorem 2 we prove that the following inference is sound:

$$\frac{\forall i \in [1..k]\ \forall \boldsymbol{in}.\ [f_i(\boldsymbol{in})] = [f'_i(\boldsymbol{in})]}{[P] = [P']}.$$

   This establishes the connection between the equivalence of semantics of individual threads, and the equal semantics of their composition.
3. Finally, in Theorem 3 we prove that $[P] = [P'] \Rightarrow p.e.(P, P')$, which is the desired conclusion.

Theorem 1. *For any function f, the following inference is sound:*

$$\frac{\forall g \in \textsc{cg}(f).\ \delta(g)}{\left(\forall i \geq 0\ \forall \boldsymbol{in}.\ [f(\boldsymbol{in})]^i = [f'(\boldsymbol{in})]^i\right)\ \wedge\ \forall \boldsymbol{in}.\ [f(\boldsymbol{in})] = [f'(\boldsymbol{in})]}\ .$$

Proof. Note that, owing to (2), the left conjunct in the consequent implies the right one. Hence it suffices to prove the former.

Let $L(f)$ be the number of nodes in $\textsc{cg}(f)$. The proof is by simultaneous induction on i and $L(f)$, for an arbitrary input $\boldsymbol{in}$.

Base Case: Suppose $i = 0$ and $L(f) = 1$, which means that f and f' do not have function calls. In this case the inference holds by construction of $\delta(f)$, because the RW traces in $\widehat{f}(\boldsymbol{in})$ are exactly those in $[f(\boldsymbol{in})]^0$, and $\delta(f)$ implies that $\widehat{f}$ and $\widehat{f'}$ generate the same RW trace given the same input.

Inductive step: Suppose $i = n$ and $L(f) = l$ and suppose that the theorem holds for all $i < n$ and for all $L(f) < l$. Consider any $t \in [f(\boldsymbol{in})]^i$ and let $\widehat{t} \in [\widehat{f}(\boldsymbol{in})]$ such that $t \in \textbf{inline}(\widehat{t}, \mu^i_f)$. Now define:

$$\mu^i_{f'}(f'(\boldsymbol{in})) = [f'(\boldsymbol{in})]^{i-1} \text{ and } \forall g' \neq f' \text{called by } f'.\ \mu^i_{f'}(g'(\boldsymbol{in})) = [g'(\boldsymbol{in})]\ .$$

By the inductive hypothesis, we know that $\forall g \in \textsc{cg}(f).\ \mu^i_f(g(\boldsymbol{in})) = \mu^i_{f'}(g'(\boldsymbol{in}))$. Therefore, $t \in \textbf{inline}(\widehat{t}, \mu^i_{f'})$. Using the same argument as in the base case, we know that $\widehat{t} \in [\widehat{f'}(\boldsymbol{in})]$. Therefore, from the definition of $[f'(\boldsymbol{in})]^i$, we know that $t \in [f'(\boldsymbol{in})]^i$. Since t is an arbitrary element of $[f(\boldsymbol{in})]^i$, we conclude that $[f(\boldsymbol{in})]^i \subseteq [f'(\boldsymbol{in})]^i$. The same argument applies if we swap f and f'. Thus, $[f'(\boldsymbol{in})]^i \subseteq [f(\boldsymbol{in})]^i$ and, therefore, $[f(\boldsymbol{in})]^i = [f'(\boldsymbol{in})]^i$. This result holds for all inputs, since we did not rely on any particular value of $\boldsymbol{in}$. □

Theorem 2. *The following inference is sound:*

$$\frac{\forall i \in [1..k]\ \forall \boldsymbol{in}.\ [f_i(\boldsymbol{in})] = [f'_i(\boldsymbol{in})]}{[P] = [P']}\ .$$

Proof. By definitions 3, 4, and 5, we know that:

$$[P] = \bigcup_{\boldsymbol{in}} \bowtie([f_1(\boldsymbol{in})], \ldots, [f_k(\boldsymbol{in})]) = \bigcup_{\boldsymbol{in}} \bigcup_{\boldsymbol{t} \in [f_1(\boldsymbol{in})] \times \cdots \times [f_k(\boldsymbol{in})]} \bowtie(\boldsymbol{t}) \text{ and,}$$

$$[P'] = \bigcup_{\boldsymbol{in}} \bowtie([f'_1(\boldsymbol{in})], \ldots, [f'_k(\boldsymbol{in})]) = \bigcup_{\boldsymbol{in}} \bigcup_{\boldsymbol{t'} \in [f'_1(\boldsymbol{in})] \times \cdots \times [f'_k(\boldsymbol{in})]} \bowtie(\boldsymbol{t'})\ .$$

But since for $i \in [1..k]$ and for all input $\boldsymbol{in}$ $[f_i(\boldsymbol{in})] = [f'_i(\boldsymbol{in})]$, t and t' range over the same sets of trace vectors. Hence $[P] = [P']$. □

We now prove the soundness of the first rule.

Theorem 3. *Proof rule (1) is sound.*

Proof. Let $P = f_1 \parallel \ldots \parallel f_k$ and $P' = f'_1 \parallel \ldots \parallel f'_k$. From the premise of the proof rule, and Theorem 1, we know that:

$$\forall i \in [1..k]\ \forall \boldsymbol{in}.\ [f_i(\boldsymbol{in})] = [f'_i(\boldsymbol{in})]\ .$$

Therefore, by Theorem 2, we know that $[P] = [P']$. Observe that for any input $\boldsymbol{in}$ and output $\boldsymbol{out}$, $(\boldsymbol{in}, \boldsymbol{out}) \in R(P)$ iff $\exists t \in [P]$ starting from $\boldsymbol{in}$ and ending with $\boldsymbol{out}$. Recall that all the outputs of P, P' are assumed to be through shared variables. It is clear then, that if $[P] = [P']$ then for a given input they have the same set of outputs. Hence, we reach the desired conclusion. □

4.5 The value of partial success

Since (1) requires $\delta(f)$ to hold for all functions, it is interesting to see if anything is gained by proving that it holds for only some of the functions. Recall that Definition 1 referred to whole programs. We now define a similar notion for a function f with respect to the program P to which it belongs. By $R(f)$ we denote the I/O relation of f with respect to P. Formally, $R(f)$ is the set of all pairs $(\boldsymbol{in}, \boldsymbol{out})$ such that there exists a computation of P (including infinite ones) in which there is a call to f that begins with $ReadParam(f) = \boldsymbol{in}$ and ends with $WriteParam(f) = \boldsymbol{out}$.

Definition 8 (Partial equivalence of functions in MT programs). *Two functions f and f' are partially equivalent in their respective nondeterministic programs if $R(f) = R(f')$.*

Denote by $p.e.(f, f')$ the fact that f and f' are partially equivalent according to Definition 8. Now, suppose that for some function f, $\forall g \in \text{CG}(f).\ \delta(g)$. Then, Theorem 1 implies that $\forall \boldsymbol{in}.\ [f(\boldsymbol{in})] = [f'(\boldsymbol{in})]$. Considering the main goal of regression verification – providing feedback about the impact of changes to a program – this is valuable information. It implies that the observable behavior of f, f' can only be distinguished by running them in different environments. Note that this does not imply that f, f' are partially equivalent according to Definition 8, since they may have different I/O relation under the environments provided by P and P' respectively. On the other hand it is stronger than I/O equivalence of f, f' under arbitrary but equivalent environments, because it makes a statement about the entire observable behavior and not just the outputs.

While partial results are useful, our first rule prevents us from proving $p.e.(P, P')$ if even for one function g, $\delta(g)$ is false. Our second rule is aimed at improving this situation.

```
void f1(int *o)
{
  int t = x;
  if (t < 0)
    t = -t;
  *o = t;
}
```

```
void f2(int i)
{
  int t = i;
  if (t < 0)
    t = -t;
  x = t;
}
```

```
void f1'(int *o')
{
  int t = x';

  *o' = t;
}
```

```
void f2'(int i)
{
  int t = i;
  if (t < 0)
    t = -t;
  x' = t;
}
```

Fig. 6. The programs `f1` ‖ `f2` and `f1'` ‖ `f2'` are partially equivalent, but since the equivalence of `f1` and `f1'` depend on the *values* generated by `f2` and `f2'` (specifically, it depends on the fact that these functions update the shared variable with a positive value), δ(`f1`) is false, which falsifies the premise of (1). On the other hand rule (4) proves their equivalence.

5 Second Proof Rule

The premise of our second rule, like the first one, is observable equivalence of pairs of functions under equal environments. However, unlike the first rule, the environments are not arbitrary, but rather consistent with the other threads in the program. This enables us to prove equivalence of programs like the ones in Fig. 6. Note that the functions `f1` and `f1'` are equivalent only if their respective environments always write non-negative values to `x` and `x'`.

5.1 Recursion-Bounded Abstraction

As mentioned, for our second rule, when checking the equivalence of f and f', we want to restrict their inputs from the environment to those that are actually produced by the other threads. In general this is of course impossible, but we now suggest an abstraction based on the observation that a bound on the number of reads of shared variables in any execution of $\widehat{f}$ can be computed, since it does not contain loops and interpreted function calls. Let $B(\widehat{f})$ denote this bound.

Given a thread rooted at f_q, and a bound b, we construct its *recursion-bounded abstraction* f_q^b, which overapproximates that thread, by transforming each recursive function $g \in \text{CG}(f_q)$ according to the scheme shown in Fig. 7. The key idea is to bound the number of recursive calls, and make each of them start from a nondeterministic state (this is achieved with `havoc_vars`) . This emulates b calls to g that are not necessarily consecutive in the call stack.

To understand what this construction guarantees, we define the following: let W denote the set of all possible sequences of writes to shared variables that can be observed in an execution of f_q. A b-sequence is a sequence of b or less elements from $s \in W$ that is consistent with the order of s. For example, if $W = \langle (x,1),(x1,2),(x,2),(x1,1)\rangle$ and $b = 2$, then some b-sequences are $\langle (x,1),(x1,1)\rangle$, $\langle (x1,2),(x,2)\rangle$, $\langle (x,1)\rangle$ etc. We now claim without proof that:

Claim 1 *Every b-sequence of f_q can also be observed in some execution of f_q^b.*

This fact guarantees that the recursion-based abstraction allows a function $\widehat{f}$ to interact with f_q^b in any way it can interact with f_q, if $b \geq B(\widehat{f})$.

```
bool rec_flag_g = 0; int rec_count_g = 0;
g^b() {
    assume(rec_count_g < b); ++rec_count_g;
    if (rec_flag_g) havoc_vars();
    The rest is the same as ĝ , except that:
    1. the RWC trace is recorded in a list out_q common to all g ∈ CG(f_q).
    2. a recursive call to g() is replaced by: rec_flag_g = 1; g^b(); rec_flag_g = 0;
}
```

Fig. 7. To derive the recursion-bounded abstraction f_q^b of a thread root-function f_q, we replace each $g \in \text{CG}(f_q)$ with g^b as described here. Although g^b is still recursive, the `assume` statement in the beginning guarantees that only b calls are made, which makes reachability decidable.

5.2 The Proof Rule

Let $f \in \text{CG}(f_i)$ and $b = B(\widehat{f})$. Define the predicate $\Delta(f)$ as being true iff the following sequential program, $VC_\Delta(f)$, is valid for all input vectors $\boldsymbol{in}$:

$$\begin{aligned} VC_\Delta(f): \quad & f_1^b(\boldsymbol{in}); \ldots; f_{i-1}^b(\boldsymbol{in}); \widehat{f}(\boldsymbol{in}); f_{i+1}^b(\boldsymbol{in}); \ldots; f_k^b(\boldsymbol{in}); \\ & \mathit{check\_assumption}(\widehat{f}.\mathit{out}, i); \\ & \widehat{f'}(\boldsymbol{in}); \mathit{assert}(f.\mathit{out} == f'.\mathit{out}); \end{aligned} \tag{3}$$

Here $\widehat{f}$ is constructed from f as before (see Sec. 4.1). The implementation of *check_assumption* is shown in Fig. 8. The goal of this function is to constrain the values of shared variables read by $\widehat{f}$ to the last value written by either $\widehat{f}$ or some other thread. We assume that the array `w` used in lines 6 and 11 is initialized to 0, emulating a case that the variable read in line 11 is the initial value (in C global variables are initialized to 0 by default). Furthermore, it guarantees (through lines 13–15) that the values are read in the same order that they are produced by the environment, while allowing skips. The function $\mathit{last}(\mathit{var}, \mathit{tid}, \mathit{loc})$ that is invoked in line 16 returns the index of the last write to *var* in thread *tid* at or before location *loc*. More information is given in the comments and caption.

Our second proof rule for partial equivalence of two MT programs P, P' is:

$$\frac{\forall i \in [1..n].\ \Delta(f_i)}{p.e.(P, P')}\ . \tag{4}$$

Example 6. Rule (4) proves the equivalence of the programs in Fig. 6, whereas rule (1) fails because $\delta($`f1`$)$ is false. □

5.3 Soundness of the Proof Rule

Let f and g be functions such that $g \in \text{CG}(f)$ and let $t \in [f]$ be a RW trace. Consider all computations of f that run through g and whose observable behavior

```
check_assumption(list out, thread-id i) {
  int cf[k] = {0,..,0};      // location in 'out_j' for j ≠ i
  for(; q < |out|; ++q) {    // recall that out is f.out
    if(out[q] == (C,...)) continue;      // skipping function calls
    if(out[q] == (W,...)) {      // suppose out[q] == (W, "x", v))
6:    w['x'] = v;      // storing the written value
    } else {      // suppose out[q] = (R, "x")
        j = *;      // j is the thread from which we will read x
        assume (j ∈ {i | 1 <= i <= k,  thread i already wrote to x});
        if (j == i)      // reading x from f itself
11:       assume(UF_f_x(q) == w['x']);      // enforcing x = last written value
        else {      // reading x from another thread
13:       oldcf = cf[j];
14:       cf[j] = *;      // nondet jump
15:       assume(oldcf <= cf[j] < |out_j|);
16:       ll = last("x", j, cf[j]);      // last location ≤  cf[j] in out_j
                                         // in which x was written to
17:       assume(UF_f_x(q) == out_j[ll]);      // enforcing x to a value
                                                // written-to by thread j
} } } }
```

Fig. 8. Pseudocode of *check_assumption*(). This function enforces the value that was read into a shared variable (through a call to an uninterpreted function) be equal to the last value it wrote or to a value written to this variable by some other thread. Lines 13–15 guarantee that the values are read in the same order that they are produced while allowing skips. The lists `out_1` ... `out_k` correspond to the lists mentioned in Fig. 7.

is t. Their subcomputations in g have corresponding subcomputations in $[\widehat{g}]$. Let $[\widehat{g}]_t$ denote this set of subcomputations. The following claim, which follows from Claim 1, will be useful to prove the soundness of our proof rule.

Claim 2 *Let $f_1, \ldots, f_k$ be functions and let $t_1, \ldots, t_k$ be traces such that $\forall i \in [1..k]$. $t_i \in [f_i]$ and $\bowtie (t_1, \ldots, t_k) \neq \emptyset$. Then, $\forall i \in [1..k]$. $\forall g \in \text{CG}(f_i)$. $\forall \widehat{t} \in [\widehat{g}]_{t_i}$, there exists an execution of the program:*

$$f_1^b(\boldsymbol{in}); \ldots; f_{i-1}^b(\boldsymbol{in}); \widehat{g}(\boldsymbol{in}); f_{i+1}^b(\boldsymbol{in}); \ldots; f_k^b(\boldsymbol{in});$$
$$\mathit{check\_assumption}(i);$$

such that at the end of the execution $\widehat{g}.out = \widehat{t}$.

Theorem 4. *Inference rule (4) is sound.*

Proof. Falsely assume that P and P' are not partially equivalent despite the validity of the premise. This means that $\exists t \in [P] \setminus [P']$, which in itself implies $\exists t \in \bowtie([f_1], \ldots, [f_k]) \setminus \bowtie([f'_1], \ldots, [f'_k])$. Since $t \in \bowtie([f_1], \ldots, [f_k])$, we know that $\exists t_1, \ldots, t_k$ such that $\forall i \in [1..k]$. $t_i \in [f_i]$ and $t \in \bowtie(t_1, \ldots, t_k)$.

Since $t \notin \bowtie([f'_1], \ldots, [f'_k])$, there exists at least one index $i \in [1..k]$ such that $t_i \notin [f'_i]$. This implies that there must be at least one function $g \in \text{CG}(f_i)$ for

which $\exists \widehat{t} \in [\widehat{g}]_{t_i}.\ \widehat{t} \notin [\widehat{g'}]$. By Claim 2 there exists an execution e of the program:

$$f_1^b(\boldsymbol{in}); \ldots; f_{i-1}^b(\boldsymbol{in}); \widehat{g}(\boldsymbol{in}); f_{i+1}^b(\boldsymbol{in}); \ldots; f_k^b(\boldsymbol{in});$$
$$check\_assumption(i);$$

such that at the end of the execution $\widehat{g}.out = \widehat{t}$. But since $\Delta(g)$ is valid, then $\widehat{g}.out = \widehat{g'}.out$, and hence $\widehat{t} \in [\widehat{g'}]$ — a contradiction. □

6 Conclusion and future work

We proposed theoretical foundations for extending regression verification to multi-threaded programs. We defined a notion of equivalence of nondeterministic programs, and presented two proof rules for regression verification of general multi-threaded programs against this notion of equivalence. The premises of the rules are defined by a set of sequential programs (one for each function), whose validity is decidable and expected to be relatively easy to check.

One of the main areas for further investigation is to improve completeness. One direction is to use reachability invariants to strengthen the inference rules, similar to those found by THREADER [6] for the case of property verification. Also, note that we did not consider locks at all, and indeed without locks it is very hard to change a program and keep it equivalent. We therefore expect that integrating synchronization primitives into our framework will also assist in making the rules more complete. Finally, adding support for reactive programs and dynamic thread creation are also important avenues for further work.

References

1. Full version at `ie.technion.ac.il/∼ofers/publications/vmcai12_full.pdf`.
2. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *9th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, pages 331 – 346, 2003.
3. Benny Godlin . Regression verification: Theoretical and implementation aspects. Master's thesis, Technion, Israel Institute of Technology, 2008.
4. Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403 – 439, 2008.
5. Benny Godlin and Ofer Strichman. Regression verification. In 46^{th} *Design Automation Conference (DAC)*, 2009.
6. Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV'11*, pages 412–417.
7. Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.
8. Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *LOPLAS*, 2(1-4):151–164, 1993.
9. Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research, 2010.
10. Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
11. Susan S. Owicki and David Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.*, 6:319–340, 1976.