

Dynamic Owicki-Gries Reasoning using Ghost Fields and Fractional Permissions

Bart Jacobs *Frank Piessens*

Report CW 549, June 2009



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Dynamic Owicki-Gries Reasoning using Ghost Fields and Fractional Permissions

Bart Jacobs *Frank Piessens*

Report CW 549, June 2009

Department of Computer Science, K.U.Leuven

Abstract

In this short note, we show how ghost fields and fractional permissions can be used to prove fork-join patterns. The approach is a simple port of Owicki and Gries's auxiliary variables-based approach to the setting with dynamic threads, locks, and objects.

Dynamic Owicki-Gries Reasoning using Ghost Fields and Fractional Permissions

Bart Jacobs* Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
 {bart.jacobs,frank.piessens}@cs.kuleuven.be

```

x := cons(0);
l := cons(1);
s1 := cons(0);
s2 := cons(0);
fork (
  acquire(l);
  [x] := [x] + 1;
  release(l);
  release(s1)
);
fork (
  acquire(l);
  [x] := [x] + 1;
  release(l);
  release(s2)
);
acquire(s1);
acquire(s2);
assert [x] = 2
    
```

Figure 1. A simple fork-join example. The `acquire` and `release` operations treat the memory cell at the specified address as a semaphore; the cell’s value indicates the number of permits available. Semaphore `l` is used as a lock to protect `c`; `s1` and `s2` are used by each forked thread to signal completion.

Abstract

In this short note, we show how ghost fields and fractional permissions can be used to prove fork-join patterns. The approach is a simple port of Owicki and Gries’s auxiliary variables-based approach to the setting with dynamic threads, locks, and objects.

1. A Fork-Join Example

Consider the program in Figure 1. A main thread allocates a shared variable `x`, initialized to zero, and forks two threads that each increment `x` by one. After waiting for both threads to finish, the main thread asserts that the shared variable’s value equals 2.

A version of this program that uses static variables, static threads and static locks was proven by Owicki and Gries (Owicki and Gries 1976). They used *auxiliary variables*, variables introduced only for verification, as follows. An auxiliary variable is associated with each of the two threads that increment the value at `x`. We call this auxiliary variable the contributor thread’s contribution variable. The semaphore invariant for `l` states that the value at `x` is the sum of the contribution variables. Each contributor thread’s

postcondition states that its contribution equals 1. This allows the main thread to conclude that the value at `x` equals 2.

To allow this proof, Owicki and Gries proposed the following rules for which variables may be used where.

- A variable may be used in the proof outline (including the postcondition) of a given thread only if that variable is not modified by any other thread.
- A variable may be used in a resource (lock, semaphore, ...) invariant only if that variable is modified by a thread only when that thread holds the resource.

It follows that the contribution variables may be used in their contributor thread’s proof outlines as well as in the semaphore invariant of `l`, provided that they are updated only by their corresponding thread and only while they hold the semaphore.

The Owicki-Gries approach requires a syntactic check of which variables are read and written in which threads, and while holding which locks. However, in modern programming languages and practices, threads, locks, and variables may be aliased, so a syntactic check is not possible.

This problem is overcome easily by using separation logic with *ghost fields* and *fractional permissions* (Bornat et al. 2005). Ghost fields (or ghost memory cells) are regions of allocated memory used only for verification. It can be checked easily using separation logic that inserting such memory allocations and accesses does not influence the behavior of the program. Fractional permissions enable read-only sharing of regions of memory and, crucially, information about the contents of those regions of memory.

In the dynamic setting, read-shared ghost fields take the place of auxiliary variables; the rules on fractional permissions take the place of the syntactic Owicki-Gries checks.

The program of Figure 1 may be verified using this approach as shown in Figure 2.

We used the proof rules for threads and semaphores shown in Figure 3. The predicate `sema(π, s, z, I)` denotes permission to acquire and release semaphore `s`. The semaphore’s invariant is `I`; at any point in time, the semaphore owns as many instances of `I` as indicated by its current value (i.e., its number of permits). The positive real number `π` denotes the fraction of the total permission for the semaphore; total permission is needed to destroy the semaphore and retrieve both the semaphore’s memory and any remaining instances of the invariant. `z` denotes a contribution to the total number of permits. At any point in time, the sum of the `z` values for all outstanding permissions equals the semaphore’s value. This makes it possible to determine the final remaining number of permits when destroying the semaphore.

The proof uses the following property of fractional permissions:

$$(\exists v \bullet x \xrightarrow{1/2} v * P(v)) * x \xrightarrow{1/2} v_0 \Rightarrow x \mapsto v_0 * P(v_0)$$

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

```

{emp}
x := cons(0);
c1 := cons(0); c2 := cons(0);
l := cons(1);
{x ↦ 0 * c1 ↦ 0 * c2 ↦ 0 * l ↦ 1}
{(l ↦ 1 * (c1 1/2 ↦ 0 * c2 1/2 ↦ 0 * x ↦ 0)) * c1 1/2 ↦ 0 * c2 1/2 ↦ 0}
{(l ↦ 1 * IL) * c1 1/2 ↦ 0 * c2 1/2 ↦ 0}
application of rule SEMA-CREATE-1
{L * c1 1/2 ↦ 0 * c2 1/2 ↦ 0}
s1 := cons(0); s2 := cons(0);
two applications of rule SEMA-CREATE-0
{L * c1 1/2 ↦ 0 * c2 1/2 ↦ 0 * S1 * S2}
fork (
  {1/2L * c1 1/2 ↦ 0 * 1/2S1}
  acquire(l);
  {
    {sema(1/2, l, -1/2, IL) * IL * c1 1/2 ↦ 0 * 1/2S1}
    {
      {sema(1/2, l, -1/2, IL) * 1/2S1 *
        {∃v1, v2 • c1 1/2 ↦ v1 * c2 1/2 ↦ v2 * x ↦ v1 + v2 * c1 1/2 ↦ 0}
        {sema(1/2, l, -1/2, IL) * 1/2S1 *
          {∃v2 • c1 ↦ 0 * c2 1/2 ↦ v2 * x ↦ v2}
        }
      }
    }
  }
  [c1] := 1;
  [x] := [x] + 1;
  {
    {sema(1/2, l, -1/2, IL) * 1/2S1 *
      {∃v2 • c1 ↦ 1 * c2 1/2 ↦ v2 * x ↦ 1 + v2}
    }
    {sema(1/2, l, -1/2, IL) * 1/2S1 *
      {∃v1, v2 • c1 1/2 ↦ v1 * c2 1/2 ↦ v2 * x ↦ v1 + v2 * c1 1/2 ↦ 1}
    }
  }
  release(l);
  {1/2L * c1 1/2 ↦ 1 * 1/2S1}
  release(s1)
);
{1/2L * c2 1/2 ↦ 0 * 1/2S1 * S2}
fork (
  {1/2L * c2 1/2 ↦ 0 * 1/2S2}
  acquire(l); [x] := [x] + 1; release(l); release(s2)
);
{1/2S1 * 1/2S2}
acquire(s1);
{1/2L * c1 1/2 ↦ 1 * 1/2S2 * true}
acquire(s2);
{1/2L * c1 1/2 ↦ 1 * 1/2L * c2 1/2 ↦ 1 * true}
{L * c1 1/2 ↦ 1 * c2 1/2 ↦ 1 * true}
application of rule SEMA-DESTROY-1
{IL * c1 1/2 ↦ 1 * c2 1/2 ↦ 1 * true}
{c1 ↦ 1 * c2 ↦ 1 * x ↦ 2 * true}
assert [x] = 2
where IL = ∃v1, v2 • c1 1/2 ↦ v1 * c2 1/2 ↦ v2 * x ↦ v1 + v2
and L = sema(1, l, 1, IL)
and 1/2L = sema(1/2, l, 1/2, IL)
and Si = sema(1, si, 0, 1/2L * ci 1/2 ↦ 1)
and 1/2Si = sema(1/2, si, 0, 1/2L * ci 1/2 ↦ 1)

```

Figure 2. Proof of the example of Figure 1.

$$\frac{\text{SEMA-CREATE-0}}{s \mapsto 0} \quad \frac{\text{SEMA-CREATE-1}}{s \mapsto 1 * I}$$

$$\frac{}{\text{sema}(1, s, 0, I)} \quad \frac{}{\text{sema}(1, s, 1, I)}$$

ACQUIRE
 $\{\text{sema}(\pi, s, z, I)\} \text{acquire}(s) \{\text{sema}(\pi, s, z - 1, I) * I\}$

RELEASE
 $\{\text{sema}(\pi, s, z, I) * I\} \text{release}(s) \{\text{sema}(\pi, s, z + 1, I)\}$

$$\frac{\text{SEMA-SPLIT}}{0 < \pi_1 \quad 0 < \pi_2} \quad \frac{}{\text{sema}(\pi_1 + \pi_2, s, z_1 + z_2, I)}$$

$$\frac{}{\text{sema}(\pi_1, s, z_1, I) * \text{sema}(\pi_2, s, z_2, I)}$$

$$\frac{\text{SEMA-MERGE}}{0 < \pi_1 \quad 0 < \pi_2} \quad \frac{\text{SEMA-DESTROY-1}}{\text{sema}(1, s, 1, I)}$$

$$\frac{\text{sema}(\pi_1, s, z_1, I) * \text{sema}(\pi_2, s, z_2, I)}{\text{sema}(\pi_1 + \pi_2, s, z_1 + z_2, I)} \quad \frac{}{s \mapsto 1 * I}$$

$$\frac{\text{FORK}}{\{P\} c \{\text{true}\}} \quad \frac{}{\{P * Q\} \text{fork}(c) \{Q\}}$$

Figure 3. Proof rules for semaphores and threads, used in the proof of Figure 2.

This reflects the fact that fractional permissions enable not only resource sharing, but information sharing as well.

Note: we do not worry about resource disposal in this example. In order to be able to dispose all resources, we need to be able to pass the forked threads' permissions on s_1 and s_2 to the main thread via s_1 and s_2 themselves. This requires using named semaphore invariants; for simplicity, we use in-line invariants in the example proof.

2. Conclusion

In this short note, we showed how to port the Owicki-Gries approach for verifying fork-join patterns to the dynamic setting, using ghost fields and fractional permissions.

While we are not aware of existing work that shows this, a survey of and comparison with existing work remains to be done.

References

- Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proc. POPL*, volume 40 of *SIGPLAN Not.*, pages 259–270, 2005.
- Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.