

Cooperative Permissions for Reasoning About Aliased Objects

Kevin Bierhoff and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA, USA
{kevin.bierhoff, jonathan.aldrich}@cs.cmu.edu

Abstract. Maintaining object invariants is notoriously difficult when objects involved in invariants are aliased. Existing approaches achieve soundness in reasoning about object invariants by imposing structural constraints on object graphs, excluding many useful programs from being verified. This paper proposes a novel abstraction, cooperative permissions, for sound reasoning about aliased objects. Cooperative permissions describe not only what aliases are allowed to do, but also restrict possible effects to the referenced object through other aliases. Therefore, cooperative permissions are a local approach to keeping assumptions that aliases make about the referenced object globally consistent, allowing sound modular reasoning without imposing structural constraints. The paper illustrates with examples how cooperative permissions can be used to specify object behavior and gives an intuition of how such specifications could be verified.

1 Introduction

Maintaining object invariants is notoriously difficult when objects involved in invariants are aliased. The canonical example of a master timer that is displayed by an arbitrary number of clocks [4] illustrates the problems that arise. Each clock holds a reference to the aliased master. Clocks *depend* on the master to only increase its timer, but when no clocks are around then the master can be safely reset (figure 1).

Such dependencies are a well-known problem in sound modular program verification. A stream of existing work in the context of Spec# [3] and the JML [15, 11] achieves soundness by only allowing dependencies on *owned* objects: only owned objects can be included in invariants. However, this model requires a strict ownership hierarchy and rules out many useful programs like the clock example.

This paper proposes a novel abstraction, *cooperative permissions*, for sound modular verification of object behavior in the presence of aliasing. Permissions track precisely how objects can be accessed through different aliases. For example, an object might be modified through one distinguished reference but read through arbitrarily many other references [5]. The paper generalizes recent work on modular typestate verification of aliased objects [6]. Cooperative permissions include a *restriction predicate* that (temporarily) restricts how the referenced object can be modified through *any* reference. A restriction predicate acts like

an object invariant, but the restriction is not necessarily permanent and can be introduced by a client rather than the restricted object itself.

Therefore, cooperative permissions are a local approach to keeping assumptions that aliases make about the referenced object globally consistent, allowing sound modular reasoning without imposing structural constraints. The paper illustrates with examples how cooperative permissions can be used to specify object behavior and gives an intuition of how such specifications could be verified.

2 Depending on Aliased Objects

This section describes a running example that illustrates the challenges involved in reasoning about invariants that depend on aliased objects and summarizes some existing work in this area.

2.1 Running Clock Example

Objects can in their invariants *depend* on other objects. This happens whenever a field $o.f$ of an object o is dereferenced in *inter-object invariants* such as $o.f.g = o.h$ to guarantee that $o.f$'s field g is equal to the value of field $o.h$. We will say that in this situation, o is a *dependent* of $o.f$. Inter-object invariants are notoriously hard to maintain because the object referenced by $o.f$ might be manipulated without o 's knowledge. In other words, there might be *aliases* to $o.f$ that reference the same object.

Barnett and Naumann's canonical *clock* example illustrates the problem of multiple objects depending on a single aliased object (figure 1) [4]. A master timer `Master` keeps track of the time. Time can `tick` away until it is `reset`. `Clock` objects can be used to display a `Master`'s time. Many clocks can display the same time, and therefore each clock caches the time in a field t that it updates when it is redrawn. This reduces the number of updates between masters and their clocks. However, the clocks implicitly assume that their master's time monotonically increases. Therefore, resetting a master might break its clocks' invariants when the master time falls under the value cached in a clock. In other words, calling `reset` is only safe if no clocks are currently attached to the master.

2.2 Ownership, Friends and History Invariants

Modular verification of invariants that depend on other objects, such as the clock's invariant from figure 1, is notoriously challenging because of aliasing. Some approaches [13] simply ignore aliasing, leading to unsound verification results. More recently, variants of *ownership* have been proposed that allow sound verification of JML and `Spec#` specifications [3, 11]. The idea is that only one distinguished owner object can depend on objects it owns. The owner is also the only object through which owned objects can be modified. Objects can only read from and cannot depend on objects that they do not own.

```

class Master {
    int time;
    invariant time >= 0;

    ensures time = 0;
    Master() { time = 0; }

    requires n > 0;
    ensures time = old(time) + n;
    tick(int n) { time += n; }

    ensures time == 0;
    reset() { time = 0; }
}

class Clock {
    final Master m;
    int t;
    invariant t >= 0 && t <= m.time;

    ensures this.m == m && t == m.time;
    Clock(Master m)
    { this.m = m; t = m.time; }

    ensures t == m.time;
    void sync() { t = m.time; }

    void draw() { ... sync(); ... }
}

```

Fig. 1. Master timer and dependent clocks. Clocks require the master time to monotonically increase in order to independently synchronize their own with the master time.

Ownership therefore imposes a structural constraint on object graphs: objects have to be arranged in a strict ownership hierarchy; otherwise, invariants cannot depend on other objects. While some programs can be structured in this way, other programs are excluded, such as the clock example. The master is not owned by any of its clocks (there could be only one clock), and it is likely not modified through its clocks but from somewhere else, maybe even from a different thread. On the other hand, clocks do depend on their master in their invariant, and resetting a master is in fact forbidden as long as there are clocks around.

Barnett and Naumann proposed the notion of *friends* to overcome ownership limitations [4]. Objects are aware of their friends and take effects on friends into account when changing state. In the clock example, **Master** would declare the **Clock** class as its friend. **Clock** declares the invariant it depends on, and clocks are explicitly attached and detached from masters. When verifying the **Master** implementation, additional proof obligations ensure that updates to the master maintain invariants of attached clocks. Resetting the clock requires that no clocks be attached as a pre-condition.

The notion of friends can handle the clock example. A disadvantage of friends, however, is that the object to be aliased has to be aware of its dependents. (1) It explicitly “grants” (i.e., declares) friendship with classes that depend on it. (2) It has to keep track of its friends. (3) It has to verify that modifications to its fields respect its friends’ invariants. Thus, taking an existing **Master** class and using it in clocks is not possible without changing the **Master** implementation and verifying the new implementation. Changes to **Clock**’s invariants require re-verification of **Master**.

Leino and Schulte overcome these disadvantages by introducing *history invariants* [17]. History invariants are reflexive and transitive predicates over tem-

porally ordered pairs of program states that describe how “newer” states relate to “older” states. For example, a monotonically increasing variable could be described as $\mathbf{old}(time) \leq time$, where \mathbf{old} conventionally refers to the “old” value of the $time$ field. If the master validates this history invariant, then clocks (and anybody else) can just depend on it. Masters no longer have to declare their friends, and they do not have to be re-verified when `Clock` invariants change. When verifying `Clock`, a separate proof obligation is discharged to verify that the clock’s invariant is never violated given the master’s history invariant.

While this solution is appealing, it does not work for our clock example from figure 1 because `reset` does not validate the history invariant: `time` decreases. (Moreover, verifying history invariants is an open problem [17].) Resetting the master can only be allowed if the master keeps track of the dependent clocks, or by implementing `Clock` so that it tolerates resetting. The latter approach is chosen by Leino and Schulte [17]; the former either requires the set of dependent clocks to become part of the master’s history invariant, or they have to be fixed up during `reset` like observers [14]. Another potential challenge with this approach is that the master has to declare the “right” history invariant. Arguably, the history invariant is not relevant for verifying `Master`; it is an arbitrary constraint imposed by its clocks. Changing the history invariant requires re-verifying `Master`; changing the master’s history invariant or any invariant in dependent objects requires discharging the aforementioned proof obligation again.

None of the mentioned solutions to resetting the master are very appealing, and the potential for not being able to use the `Master` because of a missing invariant seems quite unfortunate.

Our goal is to make the master independent from objects that depend on it, and to make it as oblivious from the way it is referenced from elsewhere as possible, while imposing no a priori constraints on program or heap structure. The impact on `Master` when writing `Clock` should be minimal. Independence of `Master` from its `Clocks` is precisely what their design seeks to accomplish—a model with multiple views [14]—and arguably would reduce complexity both of specifying and verifying the two classes because one can be developed without (for the most part) knowing about the other. The next section describes our approach for achieving these goals.

3 Cooperative Permissions

3.1 Restriction Predicates

In section 2.2 we observed that restricting the master’s time to monotonically increase is fundamentally a requirement imposed by the *clocks*; in fact, if there are no clocks, then there is no monotonicity restriction on the master necessary (trivially allowing it to reset). More generally, we would like to declare restrictions that a dependent imposes on an object it depends on *with the dependent*.

We will call such imposed restrictions *restriction predicates*. We allow restriction predicates to be declared for any variable, and in particular for fields and

method parameters. Restriction predicates can be formed just like any other predicate and start from the object being restricted. Thus a restriction predicate $\mathbf{old}(time) \leq time$ declared on the `Clock`'s `m` field refers to `m`'s `time`. A first attempt at specifying `Clock` is therefore as follows (restrictions are written as $[r]$, where r is the restriction predicate):

```
class Clock {
  [old(time) <= time] Master m;
  int t;
  invariant t >= 0 && t <= m.time;
  ...
  Clock([old(time) <= time] Master m) { this.m = m; t = m.time; }
  ...
  void sync() { t = m.time; }
  ... }
```

The restriction predicate intuitively allows verifying `Clock`'s invariant: the master's time will never decrease, regardless of other aliases, and the clock preserves its own invariant.

3.2 Introducing and Dropping Restrictions

The challenge, of course, is to make sure that our restriction predicate is not violated. While we can make sure that the implementation of `Clock` does not modify its master, the master might be manipulated through other aliases. Basically we need to ensure that restrictions imposed by one alias are *respected* by all aliases to the referenced object. Thus, restriction predicates, even through declared locally, need to be enforced globally. In order to do so, we first need to identify where it is safe to introduce (and later drop) restrictions.

Our intuition is that restrictions for an object can be introduced (and dropped) if that object is not already aliased. In fact, aliases are not a problem as long as they cannot modify the object. Thus, the following sequence of (conceptual) events should be permitted for `Master` objects:

1. Create a new `Master` instance o .
2. Restrict o with $r_1 \equiv \mathbf{old}(time) \leq time$.
3. Create a clock c_1 with o as master.
4. Restrict o with another predicate r_2 .
5. Pass o to a thread that continuously updates `time`.

The reasoning behind this goes as follows: After step 1, o is the only reference to the newly created master object. Therefore, restricting o in step 2 is permitted. Passing o to a clock in step 3 creates an alias, but we know that a clock only reads from the master. Therefore, we can introduce another restriction in step 4: even though c_1 will not be aware of this additional restriction and therefore cannot depend on it, c_1 cannot violate r_2 because it never modifies o . After passing o to an update thread, we cannot introduce any more restrictions because the update thread would not be aware of them (the update thread modifies o).

This reasoning assumed that clocks do not modify their masters. In general, it seems to be necessary to somehow keep track of how different references can access aliased objects. We build on previous work [8, 5] by introducing several *access levels* that we encode as *permissions*:

- **unique** permissions give read/write access and imply that there are no aliases to the referenced object. This situation has also been described as “linearity”.
- **full** permissions give exclusive read/write access but allow the presence of read-only aliases.
- **pure** permissions only give read-only access and allow the possibility of aliases with read/write aliases. Thus, they are the counterparts of **full** permissions.

Now we can formalize our intuition about when restriction predicates can be introduced and dropped:

- A restriction predicate r for an object o can be introduced if
 - a **unique** or **full** permission for o is available and
 - r currently holds.
- A restriction predicate r for an object o can be dropped if a **unique** permission for o is available. r is still true after it was dropped as a restriction; however, later modifications of o might make r false.

We emphasize that a predicate has to hold in order to introduce it as a restriction. We will use the convention that predicates relating current to **old** values initially hold because they only constrain future modifications. We write $perm(r)$ for a permission $perm$ with restriction predicate r , where $perm$ can be **unique**, **full**, or **pure**. We simply assume that all object references are associated with a permission. Unrestricted permissions can be written with restriction predicate **true**, but we will usually just omit the predicate in this case.

3.3 Splitting and Joining Permissions

Permissions, because of the exclusive semantics of **unique** and **full**, cannot be freely duplicated. Instead, they can be *split* in such a way that they remain consistent. Some legal splitting rules are the following:

$$\begin{aligned} \text{unique}(r) &\Rightarrow \text{full}(r) \\ \text{full}(r) &\Rightarrow \text{full}(r), \text{pure}(r) \\ \text{pure}(r) &\Rightarrow \text{pure}(r), \text{pure}(r) \end{aligned}$$

Notice that restriction predicates are *always* preserved during splitting. This guarantees that they are respected by all references to an object. Likewise, permissions are split in such a way that the assumptions made by **full** and **unique** permissions about other permissions are validated.

Splitting allows us to create “more” aliasing. For example, we can pass a “split off” permission as an argument into a method or constructor. This lets us specify `Clock` as follows:

```

class Clock {
  pure(olds(time) <= time) Master m;
  ...
  Clock(pure(olds(time) <= time) Master m) { this.m = m; ... }
  ... }

```

But what if we need to reduce aliasing? In order to *join* permissions we need some way of reference-counting permissions to the same object. A flexible way of doing so is to use *fractions* [7] to keep track of the number of permission splits. In order to use fractions, we simply associate a rational number q , where $0 < q \leq 1$, with each permission; **unique** permissions by definition have 1 as their fraction. Splitting distributes the given fraction among the two new permissions; joining puts them back together:

$$\begin{aligned}
\text{unique}(r, 1) &\iff \text{full}(r, 1) \\
\text{full}(r, q) &\iff \text{full}(r, q/2), \text{pure}(r, q/2) \\
\text{pure}(r, q) &\iff \text{pure}(r, q/2), \text{pure}(r, q/2)
\end{aligned}$$

We allow existential and universal quantification over fractions. All permissions in this paper are implicitly quantified. With this convention, we can specify **Master** and **Clock** using permissions as shown in figure 2.

We annotate permissions with modifiers **yields**, **borrow**s, and **capture**s. **yields** is used in constructors to indicate that a given permission is available after creating an object. **borrow**s indicates that a permission is needed and returned while **capture**s implies that a permission will be assigned to a field inside the method and is therefore not returned to the caller. Permissions written in front of methods and constructors apply to the receiver object *this*.

4 Verification Approach

This section sketches how code with cooperative permissions can be verified in a sound modular fashion. It follows previous work on tracking permissions for tpestate verification [6]. We ignore challenges of subtyping and inheritance as they are largely orthogonal [6].

Preserving restrictions. An important question is how modifications to objects can be verified in the presence of permissions. For example, how can we guarantee that **Master.tick()** observes its restriction predicate? As it turns out, we can reason about restrictions in the same way as object (and history) invariants. In the case of a “visible state” semantics as employed by the JML [15], this essentially means that restrictions need to hold after every operation. When employing the Boogie methodology [3], restrictions need to hold whenever objects are “packed”.

In **Master.tick()**, the restriction predicate is obviously preserved. However, one could ask what the exact semantics of **old** should be. We follow Leino and Schulte’s rules for history invariants and require predicates involving **old** to be reflexive and transitive. For a given pair of neighboring execution points where restriction predicates should hold, **old** refers to the earlier one [17].

```

class Master {
  int time;
  invariant time >= 0;

  yields unique;
  ensures time == 0;
  Master() { time = 0; }

  borrows full(old(time) <= time);
  requires n >= 0;
  ensures time == old(time) + n;
  void tick(int n) { time += n; }

  borrows full;
  ensures time == 0;
  void reset() { time = 0; }
}
class Clock {
  final pure(old(time) <= time) Master m;
  int t;
  invariant t >= 0 && t <= m.time;

  yields unique;
  ensures this.m == m && t == m.time;
  Clock(captures pure(old(time) <= time) Master m)
  { this.m = m; t = m.time; }

  borrows full;
  ensures t == m.time;
  void sync() { t = m.time; }
}

```

Fig. 2. Legal master and clock specifications with cooperative permissions. The master implementation is largely unchanged from figure 1.

Permission tracking. Figure 3 shows a sample snippet of code that creates a master with several clocks and then resets the master. It shows the permissions available *after* execution of each statement. As can be seen, each creation of a clock splits a pure permission off and passes it into the clock. The first clock causes the necessary restriction predicate to be introduced on m . Thus, splits and restrictions are applied as needed. Note that a method requiring a weaker restriction than currently available can only be called if we are allowed to drop the current restriction as discussed before. Likewise, a method requiring a stronger restriction can only be called if we are allowed to introduce a restriction.

An interesting issue is the last line in figure 3 that resets the master. Why is this legal? We mentioned earlier that masters can be reset when no clocks exist for them. Let us assume that neither $c1$ nor $c2$ are used later in the program

```

Master m = new Master(); // m : unique(true, 1)
m.tick(2);                // m : full(r, 1/4)
Clock c1 = new Clock(m); // m : full(r, 1/2), c1 : unique(true)
c1.display();            // m : full(r, 1/2), c1 : unique(true)
Clock c2 = new Clock(m); // m : full(r, 1/4), c1 : unique(true), c2 : unique(true)
m.tick(5);                // m : full(r, 1/4), c1 : unique(true), c2 : unique(true)
c2.display();            // m : full(r, 1/2), c1 : unique(true), c2 dead
c1.display();            // m : unique(true, 1), c1 and c2 dead
m.reset();                // m : unique(true, 1), c1 and c2 dead

```

Fig. 3. Sample client that can be verified based on the specifications in figure 2. Permissions for m available after each line of code are shown on the right. r abbreviates $\text{old}(time) \leq time$.

(this can be checked with a simple live variable analysis). Because we hold `unique` permissions to the clocks we can safely conclude that the two clocks are *dead* at this point: they could be garbage-collected. When this happens they “release” the master permissions they hold. Thus from `c1` we regain $m : \text{pure}(r, 1/2)$ and from `c2` we regain $m : \text{pure}(r, 1/4)$. Joining all permissions for the master yield a $\text{unique}(r, 1)$ permission. Dropping the restriction predicate now allows us to call `reset`. An alternative approach would be an explicit method on `Clock` that disconnects the clock from its master and returns the master permission. Either way, by joining permissions we can ultimately regain the ability to reset the master because we “know” that no clocks depend on it any more. This is a major benefit we get from tracking fractions in permissions.

Of course, this reasoning is only valid because of `Clock`’s specification. It guarantees that (1) a clock’s m field initially captures the passed-in master permission and (2) that m cannot subsequently change. We defer a more formal treatment of these arguments to future work.

5 Extensions for Data Groups

A practical concern one might have when using cooperative permissions as described in section 3 is scalability. Restriction predicates might accumulate for objects that are highly aliased and need to respect various different restrictions imposed by dependent objects.

If restrictions refer to different fields, there is an elegant solution to this problem: separate permissions for data groups. Data groups [16] are used to group related fields. By convention, every field f trivially forms its own data group with the same name, and a special data group *all* encompasses all fields defined.

To achieve better scalability, we can track permissions to individual data groups. For example, when creating a `Clock`, we can track separate permissions for its fields t and m . Restrictions can then be introduced separately for these two permissions. However, restriction predicates must only refer to fields inside the permission’s data group.

We can write permissions for data groups as $perm(g, r, q)$, where g is the data group the permission applies to, and $perm$, r , and q are defined as before. Splitting and joining happens independently as well. The formal treatment of breaking a permission for a “larger” data group (such as *all*) into smaller ones (e.g., for individual fields) is discussed in previous work [6].

6 Related Work

This paper generalizes our recent work on verifying tpestate protocols using permissions [6] by augmenting permissions with restriction predicates. Compared to related work on modular program verification (see section 2), cooperative permissions impose no a priori structural constraints on object graphs and minimize the impact of object dependencies on the depended-upon object.

Other approaches to verification of temporal properties either operate as global analyses [2, 12] or impose restrictions on object aliasing, typically a relaxed form of linearity [19] based on alias types [18] or AliasJava [1] (e.g., [10, 9]). Cooperative permissions, albeit linear themselves, allow modular verification while giving a great deal of aliasing flexibility.

7 Conclusions and Future Work

Reasoning about invariants that depend on aliased objects is notoriously difficult. This paper proposes a novel abstraction, cooperative permissions, for sound reasoning about aliased objects without imposing a priori constraints on object graphs. Cooperative permissions are a local approach to keeping assumptions that aliases make about the referenced object globally consistent.

While cooperative permissions seem to reduce the impact of dependencies on depended-upon objects in the canonical “clock” example, we still needed to reflect the `Clock` dependency in the specification of `Master.sync`. This is because a restriction predicate must hold at all times while a post-condition only has to be established upon method return. In future work we hope to bridge this gap, possibly by adding a promise to method specifications that nothing “unexpected” happens between method entry and exit.

Another area of future work is the combination of permissions with ownership models as commonly used in modular program verification [3, 11]. Without restriction predicates, our full permissions are quite similar to a distinguished object owner.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 311–330, Nov. 2002.

2. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth SPIN Workshop*, pages 101–122, May 2001.
3. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.
4. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *International Conference on Mathematics of Program Construction*, pages 54–84. Springer, July 2004.
5. K. Bierhoff. Iterator specification with tpestates. In *5th International Workshop on Specification and Verification of Component-Based Systems*, pages 79–82. ACM Press, Nov. 2006.
6. K. Bierhoff and J. Aldrich. Modular tpestate checking for aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2007. To appear.
7. J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.
8. J. T. Boyland and W. Retert. Connecting effects and uniqueness with adoption. In *ACM Symposium on Principles of Programming Languages*, pages 283–295, Jan. 2005.
9. W.-N. Chin, S.-C. Khoo, S. Qin, C. Popeea, and H. H. Nguyen. Verifying safety policies with size properties and alias controls. In *International Conference on Software Engineering*, pages 186–195, May 2005.
10. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.
11. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
12. S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective tpestate verification in the presence of aliasing. In *ACM International Symposium on Software Testing and Analysis*, pages 133–144, July 2006.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 234–245, May 2002.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
15. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
16. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998.
17. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *European Symposium on Programming*, 2007.
18. F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, pages 366–381. Springer, 2000.
19. P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.