Permission Specifications for Common Multithreaded Programming Patterns

Marieke Huisman, Clément Hurlin

PERMISSION SPECIFICATIONS FOR COMMON MULTITHREADED PROGRAMMING PATTERNS

MARIEKE HUISMAN AND CLÉMENT HURLIN

INRIA Sophia Antipolis, France
e-mail address: Marieke.Huisman@inria.fr

INRIA Sophia Antipolis, France
e-mail address: Clement.Hurlin@inria.fr

ABSTRACT. Multithreading is the next challenge for program verification. To support modular verification of multithreaded programs, one should know when data might be accessed or updated by the different threads in the system. We propose a permission-based annotation system that is designed to do exactly this, *i.e.* it specifies when a thread can read or write a variable. The annotation system ensures that threads have exclusive access to a variable whenever they have the possibility to write it, thus avoiding data races. Moreover, the annotation system allows to change permissions dynamically throughout the execution. The information from the permission annotations can be used for further verification of the program. This paper shows how the annotation system can be used to specify variable access in several typical multithreaded programming patterns.

1. INTRODUCTION

Because of increasing requirements on performance and reactivity, multithreading nowadays is unavoidable for programmers. However, multithreaded programs are notoriously difficult to write correctly. Therefore, it is important to have the means to establish a formal correctness statement for the application (it behaves as specified, does not contain security bugs etc.). However, just as writing multithreaded applications is more difficult than writing sequential ones, verifying multithreaded applications is also more complicated than verifying sequential ones.

A promising approach to verify multithreaded programs is to abstract the behaviour of threads by specifying what they *cannot* do. This can be considered as the worst-case behaviour of a thread. Then, to verify a particular thread, one can reduce all other threads to their abstractions, which makes it unnecessary to know the precise state each thread is in. Our annotation system for permissions does exactly this: it specifies *whether* a thread can potentially change or read a variable, without specifying *what* it reads or writes. In addition, our system also ensures that *(i)* if a thread can write to a location no other thread may write or read to this location simultaneously, and *(ii)* if a thread can read to a location no other thread can write to this location simultaneously. The annotation system has been designed to be as general as possible, and to handle a large class of

This work is funded in part by the IST programme of the EC, under the IST-FET-2005-015905 Mobius project and the French national research organisation (ANR), under the ANR-06-SETIN-010 ParSec project.

LOGICAL METHODS
IN COMPUTER SCIENCE

DOI:10.2168/LMCS-???

$$
\begin{array}{rcl}
\mathit{Permission} & ::= & \mathtt{permission}\ (\mathit{PermName}\,(\mathtt{=}\ \mathit{PermDecl}^{+})^{?}\,\mathtt{;})^{+} \\
\mathit{LockClause} & ::= & \mathtt{lock}\ \{\mathit{LockName}\}\ \mathtt{=}\ \mathit{PermDecl}^{+} \\
\mathit{PermDecl} & ::= & \mathit{PrimPermDecl} \mid \mathit{ObjPermDecl} \mid \mathit{LockPermDecl} \\
\mathit{PrimPermDecl} & ::= & \mathit{FieldName}\mathtt{:}\mathit{BasePerm} \\
\mathit{ObjPermDecl} & ::= & \mathit{FieldName}\mathtt{:}\mathit{BasePerm}\mathtt{(}\mathit{EncapsPermList}\mathtt{)} \\
\mathit{LockPermDecl} & ::= & \{\mathit{LockName}\} \\
\mathit{BasePerm} & ::= & \mathtt{\backslash split(}\mathit{Nat}\mathtt{)} \mid \mathtt{W} \mid \mathtt{R} \\
\mathit{EncapsPermList} & ::= & \mathit{EncapsPerm} \mid \mathit{EncapsPerm}\mathtt{,}\mathit{EncapsPermList} \\
\mathit{EncapsPerm} & ::= & \mathtt{\backslash split(}\mathit{PermName}\mathtt{,}\mathit{Nat}\mathtt{)} \mid \mathit{PermName}
\end{array}
$$

Figure 1: Grammar for permission declarations

programming patterns. It can be used to show the correctness of lock-free algorithms, algorithms based on locking, and mixtures of both.

The annotation system declares for each object one or more permission states, listing all permissions on an object that may exist at a particular point in time. If a thread can be shown to hold all permissions on a particular object, *i.e.* it has the exclusive access to the object, then the object's permission state can be changed. Method's preconditions specify which permissions are needed to call a method, while postconditions specify which permissions are given back to the caller. As mentioned above, the annotations should ensure that at each point in time *(i)* there exists at most one permission to write a variable, and *(ii)* if a thread has the permission to write a variable, then no other thread has the permission to read this variable.

Our annotation system is strongly based on Boyland's fractional permission system [2], but whereas Boyland's system is defined for a simple language with parallel composition, we extended it to a Java-like language with synchronisation, thread creation and joins[1]. Moreover, our annotation system supports proper encapsulation of the object state by introducing names for the different permissions.

This paper gives a quick overview of the syntax of our annotation system, described as an extension of JML [7]. The remainder of the paper shows how the annotations can be used to specify typical multithreaded programming patterns. In particular, we discuss the following patterns: Immutability, Lock Splitting, Confinement across methods, Worker threads, Fork/Join algorithms, and concurrent Subject/Observers. These patterns and their implementations are all taken or inspired by Lea's book on concurrency in Java [6] or the Design Pattern book [3].

2. Short Overview of the Annotations

A class can contain several permission and lock clauses. Figure 1 gives their grammar (where we use regular expression notation: $X^?$ for 0 or 1 X, X^* for 0 or more X, and X^+ for 1 or more X). A permission consists of a list of names and for each name, a (possibly empty) list of permission declarations. A permission declaration can specify a permission for a field of primitive type, a field of reference type, or it can be a special lock permission. A permission declaration for a primitive field contains the field name and a base permission. A permission declaration for a reference field, in addition, specifies the encapsulated permissions of the object referenced to. Finally, references can be declared to be special lock permissions.

[1] Notice that such a thread creation/join mechanism can simulate parallel composition, but parallel composition cannot simulate all programs using thread creation and joining.

$$\begin{array}{rcl}
\mathit{PermPred} & ::= & \texttt{\textbackslash split(}\mathit{Path}\texttt{.}\mathit{PermName}\texttt{,}\mathit{Nat}\texttt{)} \mid \texttt{\textbackslash part(}\mathit{Path}\texttt{.}\mathit{PermName}\texttt{)} \\
 & \mid & \texttt{\textbackslash has(}\mathit{Path}, \mathit{Nat}, \mathit{Nat}\texttt{)} \mid \texttt{\textbackslash hasnot(}\mathit{Path}\texttt{)} \\
 & \mid & \mathit{Path}\texttt{.}\mathit{PermName} \mid \mathit{PermPred} * \mathit{PermPred} \\
\mathit{Path} & ::= & \mathit{ArgAccess}^{?} \mid \mathit{ArgAccess}^{?}\texttt{.}\mathit{FieldAccess} \\
\mathit{ArgAccess} & ::= & \texttt{this} \mid \mathit{VarName} \\
\mathit{FieldAccess} & ::= & \mathit{FieldName} \mid \mathit{FieldAccess}\texttt{.}\mathit{FieldName}
\end{array}$$

Figure 2: Grammar for permission predicates

A base permission is of the form `\split(n)` (where `n` is a natural number). Its intuitive meaning is that there are at most 2^{n} threads accessing the field at the same time. Permissions can be split, using the equivalence `\split(n)` $\equiv$ `\split(n+1)*\split(n+1)`, where the `*` operator comes from separation logic [9]. For standard JML expressions, `*` and `&` are equivalent, but their meaning is slightly different for permission formulas. Intuitively, if P and Q are permission formulas, "$P*Q$ is valid w.r.t. to a heap h" means that h can be split in two disjoint heaps h_1 and h_2 such that "P is valid w.r.t. h_1" and "Q is valid w.r.t. h_2". For this paper, it is sufficient to understand `*` as `&`. The equivalence doubles the number of threads that can access the field. Notice that permission splitting is unbounded.

For convenience we introduce `W` to abbreviate `\split(0)`: since $2^0 = 1$, this means that the thread has exclusive access to the field, and thus can be allowed to write to it. Any permission `\split(n)`, where `n` > 0, allows only to read, because there might be other threads accessing the field. We use `R` to abbreviate `\split(1)`, a basic read permission.

Encapsulated permissions are of the form `\split(p,n)`, meaning that the reference permission contains $(1/2^n)^{th}$ of the permission named `p`, from the object that the reference points to. We use the permission name `p` to abbreviate `\split(p,0)`, *i.e.* exclusive hold of the permission. As base permissions, encapsulated permissions can be split using the equivalence `\split(p,n)` $\equiv$ `\split(p,n+1)*\split(p,n+1)`.

For each lock permission declaration, the class contains a special lock clause. This specifies which permissions are obtained when the lock is acquired. Intuitively, at every point in the execution, a thread has certain permissions on an object. If a thread acquires a lock, it obtains the permissions that are held by that lock, and when it releases the lock, it returns these permissions to the lock. Fields that are declared as locks are required to be final.

To ensure that the annotations avoid data races, each permission should contain at most one write permission per field, and if a permission contains a write permission, it cannot contain read permissions on the same field. Notice that this only has to be guaranteed at the current class level: it is implicitly guaranteed to hold for all objects that are referenced to by fields in the class.

Method pre- and postconditions can be extended with permission predicates, see Figure 2 for their grammar (where a path denote 0 or more indirections beginning with `this` or a variable name, i.e., a sequence of the form `this.f1.....fn` or `v.f1.....fn`). Intuitively, a precondition specifies which permissions are necessary to execute a method, a postcondition specifies which permissions are returned to the caller[2].

[2] We do not consider exceptional postconditions in this paper. Typically, exceptional permission postconditions would coincide with the method's normal permission postconditions.

```
class Fraction{
  //@ permission rd = num : R, den : R;
  protected final long num, den;

  //@ ensures rd;
  public Fraction(long n, long d) { // ... normalize
    num = ...; den = ...   }

  //@ requires \part(rd) * \part(f.rd);
  //@ ensures  \part(rd) * \part(f.rd) * \result.rd;
  public Fraction plus(Fraction f) {
    return new Fraction(num * f.den + f.num * den, den * f.den);}}
```

Figure 3: Fragment of immutable class `Fraction`

Permission predicates can be base permissions on the fields of the object, or on the encapsulated fields of the object (named by a qualified expression, containing the name of the surrounding permission). Additionally, permission predicates can express whether a thread owns a lock or not: permission `\has(path,i,j)` means that the considered thread has acquired the lock on the object pointed to by `path` `j` times, and in addition at most `i` other threads have permission to acquire this lock. Further, we introduce to useful abbreviations: `\part(p)` and `\hasnot(path)`, defined in terms of JML/permission expressions. Expression `\part(p)` abbreviates `(\exists int n. \split(p,n) & n >= 0)`, *i.e.* it denotes a read permission on the fields contained in `p`. Expression `\hasnot(path)` indicates that a thread does not hold the lock pointed to by `path`: it abbreviates `(\exists int i.\has(path,i,0) & i >= 0)`. Other useful abbreviations could be imagined, *e.g.* to express that exactly the same permission is returned. More experience with writing permission annotations will show which abbreviations are useful.

To allow modular verification, standard JML uses modifies clauses. However, because permissions can be considered as modifies clauses (i.e., a write permission on a field in a precondition is similar to a modifies clause mentioning this field), we do not need to specify modifies clause in this paper.

3. Annotations of Different Concurrency Patterns

This section shows how the annotation system presented above is particularly suited to specify several common concurrent programming patterns. The patterns that we present here are derived from Lea's pattern collection [6] and the Design Pattern book [3].

3.1. Immutability.

The first example is the immutability pattern. An object is said to be immutable if after initialisation it can never change its internal state (see also [4]). This is useful for multithreaded programming, because accesses to (initialised) immutable objects do not have to be protected by locks. Typical applications of immutable objects are abstract data types, value containers and shared state representations.

Figure 3 presents an example of an immutable object, representing the abstract data type of fractions (taken from [6, §2.1.1]). The annotations specify that the class has only one permission `rd`, in which both numerator (`num`) and denominator (`den`) can only be read. The constructor returns the full permission `rd` to its caller. These permissions can then freely be split, to give read access to the fraction object to any thread that requires this. To be able to compute with fractions (as

```
class Person{
    protected int age, salary;

    //@ permission d = {l1}, {l2};
    //@ lock {l1} = age : W;
    //@ lock {l2} = salary : W;

    final protected Object l1 = new Object();
    final protected Object l2 = new Object();

    //@ requires \part(d);
    //@ ensures  \part(d);
    public int getAge(){synchronized(l1) {return age;}}

    //@ requires \part(d);
    //@ ensures  \part(d);
    public void birthday(){
        synchronized(l1){synchronized(l2){age++; salary +=100;}}}

    //@ requires \part(d);
    //@ ensures  \part(d);
    public int getSalary(){synchronized(l2) {return salary;}}}
```

Figure 4: Fragment of class `Person`, illustrating the lock splitting pattern

does method `plus`), one needs to have some part of the `rd` permission. After `plus` has finished, a (possibly different) part of the `rd` permission is returned to the caller. In addition, the caller obtains the complete `rd` permission on the newly created object.

Notice that our permission system is also suitable to express *partial* immutability, where only some fields are immutable, or an object only is immutable during a part of the execution (e.g., before or after calling a certain method). To support the first case, a class can declare different permissions: one containing read permission declarations for all immutable fields, and one (or more) containing write permission declarations for the mutable fields. To support the second case, different permissions can be specified, so that the object can change its state from one permission to another. Methods that need the object to be immutable, require that the object is in the appropriate permission state, and thus that the appropriate read permissions are held.

3.2. **Lock Splitting.** To increase performance, it often is useful to associate locks only with certain functionalities of a class. Thus, different (groups of) fields are protected by different locks. Naturally, this makes it even more important to clearly specify which fields are protected by which lock (and to ensure that the application respects this).

Figure 4 gives a typical example of this lock splitting pattern [6, §2.4.2]. The class `Person` contains fields `age` and `salary`. It has a single permission `d`, which declares two lock permissions. All methods in the class require and ensure a fraction of this permission. Further the class contains two lock clauses: the first specifies that if a thread acquires lock `l1`, it obtains the permission to write the field `age`; similarly acquiring lock `l2` gives the permission to write `salary`. Notice that the method specifications only mention the permission `d`. However, the lock clauses ensure that any access to `age` or `salary` is protected by the appropriate locks.

```
class Point{
    public int x, y;
    //@ permission q = x : W, y : W; }

class Plotter{
    //@ ensures true;
    public void showNextPoint(){
        Point p = new Point();
        p.x = ...; p.y = ...;
        display(p); }

    //@ requires p.q;
    protected void display(Point p){ ... }}
```

Figure 5: Fragment of class `Plotter`, showing confinement across methods

Note that the `synchronized(..)` statements in class `Person` can be *reentrant* acquirements of `l1` or `l2`. A lock acquirement is said to be reentrant if the thread acquiring the lock already owns it. In this case, Java's semantics is such that the thread continues normally. To allow reentrant acquirement of locks, our system proceeds as follows: whenever a thread acquires a lock, the thread must show it does not already have the lock (i.e., the thread can gain the permissions inside the lock) or the thread must show it already has the permissions inside the lock. Thus, after acquiring a lock, one is sure that the permissions inside the lock have been transfered to (or were already owned by) the thread considered. For example, after executing the `synchronized(l1)` statement in `getAge`, a thread gains the permissions inside `l1` (i.e., write access to the field `age`) and can execute `return age;`.

3.3. **Confinement Across Methods.** Thread locality, *i.e.* the case where an object is only accessible via a single thread, is another means to guarantee non-interference of other threads. A generalisation of this is the case where there is at most one thread at the time having a reference to an object. After passing an object to another method (which might cause it to be accessed by different threads), the calling method guarantees that it does not access it anymore. Thus, possible changes to the object do not change the correct behaviour of the method. This pattern is called *confinement across methods* [6, §2.3.1].

Figure 5 shows a typical example of this pattern. The basic `Point` class contains a single permission set that allows to write both its fields. In class `Plotter`, the method `showNextPoint` creates a new point class, and properly initialises it. After initialisation, it gives the newly created point to the display method (that will typically use special threads for doing the graphics). The fact that the object is given away, is made explicit by the method's postcondition: the caller of `showNextPoint` does not get back any permissions on the newly created point. Also the `display` method requires the full permission on the point, and does not return any permission.

Notice that other confinement patterns (confinement within a thread, confinement within an object) can also be expressed in a natural way with our annotation system.

3.4. **Worker Threads.** The next pattern that we discuss is the worker thread pattern; an important pattern for many industrial applications. A main thread prepares different tasks, and sends them off to a worker thread for computation. This ensures that the main thread is never blocked for a long time, and thus that the application can stay reactive. It is important that access to the task

```
class MainThread extends Thread{

    //@ permission before = wk : R (p), t : R (wr, regain);
    //@ permission after  = wk : R (p), t : R (wr);

    final private Task t;
    final private WorkerThread wk;

    //@ requires wk.p;
    //@ ensures  before;
    public MainThread(WorkerThread wk)
           {this.wk = wk; t = new Task();}

    //@ requires before;
    //@ ensures  after;
    public void run()
           {t.prepare(); wk.addTask(t); ...; if t.isDone(){...}}}
```

Figure 6: Fragment of class `ServerThread`, part of the worker thread pattern

```
class WorkerThread extends Thread{

    final private Vector<Task> tp; //task pool
    //@ permission p = {this};
    //@ lock {this} = tp : W (...); // encapsulate permissions from Vector

    //@ requires \part(p);
    //@ ensures  \part(p);
    public void run(){
        Task t;
        while(true){
          if(taskWaiting()){t = getTask(); t.doTheJob(); t.setIsDone();}}}

    //@ requires \part(p) * t.wr;
    //@ ensures  \part(p);
    public synchronized void addTask(Task t){tp.add(t);}

    //@ requires \part(p);
    //@ ensures  \part(p) * \result.wr;
    public synchronized Task getTask(){
        Task t = tp.elementAt(0); tp.removeElementAt(0); return t; }

    //@ requires \part(p);
    //@ ensures  \part(p);
    public synchronized boolean taskWaiting(){return tp.size != 0;}
```

Figure 7: Fragment of class `WorkerThread`, part of the worker thread pattern

```
class Task{
    volatile boolean done;

    //@ permission wr = ...;
    //@ permission regain;

    //@ requires wr;
    //@ ensures  wr;
    public void prepare(){}

    //@ requires wr;
    //@ ensures  wr;
    public void doTheJob(){ ... }

    //@ requires wr;
    public void setIsDone(){ ... }

    //@ requires regain;
    //@ ensures  \result ==> wr && !\result ==> regain;
    public boolean isDone(){ ... }}
```

Figure 8: Fragment of class `Task`, part of the worker thread pattern

is exclusive, *i.e.* once the main thread has send off the task, it should no longer access the fields involved in the computation. Only once it knows that the task has finished, it can be allowed to access these fields again.

Figure 6 shows an implementation of a main thread, that prepares jobs for a worker thread. In our simple example, it has read-only references to a single worker thread and a single task object. It has two different permissions: `before` and `after`. The only difference between these two permissions is whether the server thread has a special `regain` permission on task `t`. The role of this `regain` permission will be explained below. The permissions on the worker thread and task objects are encapsulated in the permission state for the `ServerThread` object. Notice that since both `before` and `after` contain the full encapsulated permissions `p` and `wr` (explained below), the main thread cannot hold both the `before` and the `after` permission at the same time. Notice further that `ServerThread`'s constructor requires that its parameter has an appropriate permission on the worker thread. In contrast, since the task is a newly created object, no further requirements are necessary.

Figure 7 shows the implementation of a worker thread (inspired by [6, §4.1.4]). The worker thread has a single permission, which ensures that access to the task pool is protected by a lock. This avoids possible race conditions.

Figure 8 gives a possible implementation of tasks. It has a permission `wr` giving write access to the fields of the task (not further detailed here), and a special `regain` permission, mentioned above. This permission is only a marker, it does not give access to any of the fields. The holder of the `regain` permission is allowed to regain the `wr` permission, once the task is done. The constructor of `Task` returns this permission to the thread that constructed the object. This (or some other) thread can inspect whether the task is finished, and if so, it regains the write permission `wr` on the fields of the task, and the `regain` permission is destroyed.

The `run` method of the `ServerThread` first prepares the task, and then it puts it in the task pool of the worker thread. Notice that the specification of `addTask` in `WorkerThread`

```
class Fib extends FJTask{
  int number;
  //@ permission p = number : W;

  //@ ensures p;
  Fib(int n) { number = n; }

  //@ requires p;
  //@ ensures  p;
  public void run() {
    int n = number;
    if (n = 1) number = 1;
    else {Fib f1 = new Fib(n - 1);Fib f2 = new Fib(n - 2);
          coInvoke(f1, f2);
          number = f1.number + f2.number;}}}
```

Figure 9: Fragment of fork-join implementation of Fibonacci function

requires the `wr` permission on the task object, and does not return it, *i.e.* the server thread looses its permission to do something with the task. When the `WorkerThread` takes the task from the task pool (using `getTask`) it acquires the permission `wr` on the task, and thus it is able to do the job, and then set a volatile field[3] to signal that it is done. By executing the `setIsDone` method, it looses its permission to access the task. In the mean time, the `ServerThread` can continue with other things (*e.g.* react on other requests). When it needs the result of the task, it can inspect whether the task is finished, using the `isDone` method. For this it needs the special `regain` permission. If the `isDone` method returns true, the `regain` permission is destroyed, and instead the `wr` permission is returned (otherwise the `regain` permission is kept). Thus, implicitly the permission of `ServerThread` changes from `before` to `after`.

The use of the special `regain` permission ensures that the `wr` permission is not duplicated: once the `wr` permission has been returned by `isDone`, the `regain` permission is destroyed, and thus the `isDone` method cannot be called anymore for this task. A variant of this specification would be that the method would only return the `wr` permission, *if* the method was called with the `regain` permission (which is then destroyed). Alternatively, one could also imagine a specification where method `isDone` requires only a part of the `regain` permission and returns the corresponding part of the `wr` permission (see method `run` of the `Matrix` example in the next paragraph for such an example).

3.5. **Fork/Join Algorithms.** An important class of concurrent algorithms are fork/join algorithms – the concurrent variation of divide-and-conquer algorithms – *i.e.* each thread spawns off several other threads to do sub-computations. It waits for all these threads to finish, and then combines their results into a single result. We show how our annotation system can be used to show that the fork-join implementation in Figure 9 of the Fibonacci function [6, §4.4.1][4] does not contain data races.

[3] To avoid possible race conditions, and therewith unexpected behaviours.

[4] Even though, as pointed out by Lea, this is an unrealistic example, because there are much faster non-recursive solutions. However, because of its simplicity, it nicely illustrates the working of our annotation system.

Class Fib contains a permission p, that allows to write the field number[5]. The implementation of the method coInvoke from the class FJTask is such that it implicitly behaves as f1.fork(); f2.fork(); f2.join(); f1.join();. Since the fork method starts a new thread, that will execute its run method, the precondition of the run method is propagated to also be the precondition of the fork method. Further, each runnable object is supposed to contain an implicit join permission, that is returned to the creator of the class (and can be passed around). The join method has the following (implicit) specification:

```
requires  join;
ensures   Q_run;
  also
requires  true;
ensures   true;
```

where Q_run is the specified postcondition of the method run (notice that to get even more precision and flexibility, we could specify that only a fraction of the join permission is required, and that exactly this fraction of the permissions in run's postcondition are returned). Thus, when a thread creates objects f1 and f2, it obtains the permission to write their number fields, plus a special join permission. Within coInvoke, the permission on number is given to the forked threads, and after joining, they regain the permission to access number – however, they have lost the implicit join permission.

As a more complex example, we sketch a program where we split and recombine permissions. Suppose we have a class Matrix with permission wr, that allows to write the elements in the matrix (but not to change the shape of the matrix), and with appropriate method annotations[6].

```
class Matrix{
  final int[][] elems;
  //@ permission wr: w = elems : R[R[W]]; ...
```

To be able to initialise the matrices, we need write permission on the elements. However, if we suppose that we only need read access to do the matrix computations, we can have the following programming pattern (where we have matrices a and b):

```
  while(true){
    initialise(a,b);
    // split matrix permissions for different threads
    coInvoke(...); // do fork-join matrix computations
    // recombine fractioned permissions into complete permission }
```

provided that the run method for the matrix computation is specified as follows:

```
requires \part(a.wr) * \part(b.wr);
ensures  \part(a.wr) * \part(b.wr) &
         (\forall int n.\old(\split(a.wr, n)) = \split(a.wr, n)) &
         (\forall int n.\old(\split(b.wr, n)) = \split(b.wr, n));
```

This specifies that a fraction of the wr permission on a and b is needed for the thread to start, and that after joining the finished thread, exactly the same permission is given back. It is crucial here that exactly the same fraction of the permission is given back: this allows to conclude that after joining all threads, the main thread holds the complete wr permission again, and thus that the matrices can be re-initialised.

[5] Notice that we do not require number to be volatile, in contrast with Lea's implementation. With our annotation system, and knowledge of the new Java Memory Model [8], we can show that there will be no data races, thus there is no need for this variable to be volatile.

[6] Where the grammar of permission declarations is extended to array declarations in the obvious way.

```
class Subject{
    //@ permission p = {this};
    //@            r = v : W;
    //@ lock {this}  = v : W, obs : W (\split(q,1));

    Observer obs; int v;

    /*@ requires (\exists int i,j. \has(this,i,j) & i >= 0 & j > 0) *
      @          obs.s == this;
      @ ensures  (\exists int i,j. \has(this,i,j) & i >= 0 & j > 0) *
      @          obs.s == this;
      @*/
    public void notifyObs(){ obs.update(); }

    /*@ requires (\exists int i,j. \has(this,i,j) & i >= 0 & j > 0) *
      @          obs.s == this;
      @ ensures  (\exists int i,j. \has(this,i,j) & i >= 0 & j > 0) *
      @          obs.s == this;
      @*/
    public void setState(int v){
        this.v = v;
        notifyObs();}

    //@ requires \part(r);
    //@ ensures  \part(r);
    int getState(){ return v; }}
```

Figure 10: Fragment of class `Subject`

3.6. **Concurrent Subject Observer.** The last pattern we discuss is the subject observer pattern; a typical example of object-oriented programming [3, §5]. In this pattern, a single subject is observed by several observers: when the subject's state changes, the observers are notified, so that they can update their internal representation of the subject's state. Figures 10 and 11 show a fragment of an implementation of this pattern. For clarity of presentation, we only have one observer per subject.

The `Subject` class has a permission `p` that allows to lock the considered subject. Once a subject is locked, write access to its state (`v`) and permission `q` on the observer are granted. Class `Subject` imposes client-side locking: to call a method on a subject, one must lock it beforehand. For example, `notifyObs`'s precondition indicates that to call this method, a thread has to hold the lock on the subject: `\has(this,i,j)` with `j` greater than 0 is required.

The method described in paragraph 3.2 to handle reentrance works also well in this pattern. For example, when the `synchronized(this)` statement is executed in method `update` of class `Observer`, our system makes sure that the observer was not already locked (in this case, the permissions inside the observer is transferred to the executing thread), or that the thread considered already has the permissions inside the lock (i.e., write access to the field `cache`). Thus, after acquiring the observer a thread can execute `cache = v;`.

Note that a thread can lock the subject and the observer if it has permission `p` on the subject, while another thread can lock the observer if it has a part of permission `q` of this observer. Two different threads can simultaneously hold permission `p` and a part of permission `q`, because the subject only has half of the permission `q` on its observer (see annotation `lock {this} = ..., obs : W`

```
class Observer{
    //@ permission q = {this}, sub : R ();
    //@ lock {this}  = cache : W;

    final Subject sub; int cache;

    //@ ensures q * s == sub;
    public Observer(Subject sub){ this.sub = sub; }

    //@ requires \part(q) * \part(sub.r);
    //@ ensures  \part(q) * \part(sub.r);
    void update(){
        int v = sub.getState();
        synchronized(this){ cache = v; }}

    //@ requires \part(q);
    //@ ensures  \part(q);
    public synchronized int getCache(){ return cache; }}
```

Figure 11: Fragment of class `Observer`

(`\split(q,1));` in class `Subject`). Thus, one can write a program where one thread updates the subject and the observer, while another thread simultaneously inspects the observer (to update a GUI for example).

Finally, to notify the observer, one has to know that its underlying subject (field `sub`) is the subject where `notifyObserver` is called: this is specified (using standard JML) in `notifyObs`'s precondition by `obs.s == this`. Because of this equality, the permission `\split(v,0)` (which is contained in `\has(this,i,j)` at `notifyObs`'s entry) can be used to show that the precondition of `update` is satisfied.

4. Conclusion & Future Work

We have shown how our annotation system for permissions can be used to specify several common multithreaded programming patterns. The specifications are intuitive to understand, and at the same time highly expressive. They allow to prove absence of race conditions, and moreover the annotations can be used as auxiliary information for the further verification of the program: if we can deduce from the permission annotations that other threads cannot interfere at certain program points, then we do not have to consider the possible interleavings at these points (see [5] for details).

Currently, we have implemented a run-time checker for the annotation system, and we are working on the development of a static verification method. For this, we will extend an existing translation of JML-annotated programs into BoogiePL [1] with information about the permissions of the current threads. We will then use a verification condition generator for BoogiePL to generate appropriate proof obligations.

Acknowledgements. It is our pleasure to dedicate this paper to Henk Barendregt. Henk has been one of the PhD supervisors of the first author, Marieke Huisman. During this period, Henk has taught me about the importance of looking at known facts with a different and fresh mindset, in

order to establish new connections. This is one of the lessons that I now try to teach to my own PhD students. Happy birthday!

References

[1] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, 2005.

[2] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In R. De Nicola, editor, *European Symposium on Programming*, Lecture Notes in Computer Science, pages 347–362. Springer-Verlag, 2007.

[5] M. Huisman and C. Hurlin. The stability problem for verification of concurrent object-oriented programs. In *VAMP 2007: Proceedings of the 1st International Workshop on Verification and Analysis of Multi-threaded Java-like Programs*, 2007. Technical Report ICIS-R07021, Radboud University Nijmegen.

[6] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns (Second Edition)*. Addison-Wesley, Boston, MA, USA, 1999.

[7] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[8] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.

[9] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, Copenhagen, Denmark, July 2002. IEEE Press.