A Symbolic Approach to Permission Accounting for Concurrent Reasoning

Marieke Huisman and Wojciech Mostowski
Formal Methods and Tools, University of Twente, Enschede, The Netherlands
Email: {m.huisman,w.mostowski}@utwente.nl

***Abstract*—Permission accounting is fundamental to modular, thread-local reasoning about concurrent programs. This paper presents a new, symbolic system for permission accounting. In existing systems, permissions are numeric value-based and refer to the current thread only. Our system is based on symbolic expressions that provide a view of permissions for all relevant threads in the scope of the permission *originator* – current thread or a lock. This enables: (a) better understanding of permission tracking for the specifier, (b) more natural specification of complex permission transfer scenarios, and (c) more efficient reasoning for verification tools (in particular, no reasoning about rational numbers is required). Our system is based on symbolic permission slicing to divide permissions between multiple owners, and on tracking the history of permission transfers by means of "I-owe-you" chains of permission owners. We axiomatised our permission system in the KeY verifier as well as in PVS, and proved correct with both tools a list of vital properties about our permissions. KeY is an interactive verification tool for Java and our primary target to employ our permission system. First results with the verification of concurrent Java programs using our permission system in KeY are also reported.**

***Keywords*-permission accounting; fractional permissions; interactive verification; formal specification; Java**

I. Introduction

Permission accounting [1] is the essential factor in thread-local reasoning about concurrent programs. In permission-based verification, programs are annotated with permission expressions specifying a thread's access rights to each memory location. Verification ensures that all memory accesses are protected by a corresponding permission: *full* permissions grant a write access, *partial* permissions grant only a read access, no permission prohibits access entirely. Soundness of the verification technique ensures that multiple threads are not allowed to collectively hold more than a full permission to one memory location at all times. This ensures that verified programs are free of data races, and that program specifications are stable, i.e., they cannot be invalidated by other threads. The main complication for this approach are synchronisation points, like locks or forked threads, where permissions are transferred between threads. The most commonly used approach [1] is to represent a permission as a rational fraction in the range $(0, 1]$ (or a related structure [2]), where 0 (resp. lack of permission specification) denotes no access, 1 denotes full access, and any fraction denotes read access. Upon synchronisation points, permissions can be split (by division) and recombined (by addition). The main challenge with fractions is to be able to combine read permissions back to a write (and *only* a write) permission where necessary, as this requires high-precision specifications and reasoning.

The main contribution of this paper is an alternative and more flexible *permission model*, which is *fully symbolic* and can be used to specify *complex synchronisation scenarios*. In essence, symbolic treatment of permissions is achieved by specifying *what kind of transfer* is applied to a permission and *between what parties*, instead of specifying *how much* of the permission is transferred. This relieves both the specifier and the verification tool of the need to, respectively, specify and reason about concrete fractions, which requires dedicated complex decision procedures in first-order reasoning [3]. To specify complex synchronisation scenarios such as Java threads with multi-join possibilities, and latches [4], the system tracks the *permission originators* in the permission expressions, which can be used to determine the permission return path. To handle all scenarios, under certain conditions, it is allowed to modify this return path. Section II provides two examples that illustrate these main characteristics of our permission system. The formal description of our permission system is provided in Sects. III and IV.

The context of our work is the VerCors project[1] [5], targeting the functional verification of concurrent data structures. The base for VerCors is our own version of separation logic with permissions [6]; we specify programs with Java Modeling Language (JML) [7]; and we provide tool support for this combination. For automated tool support we encode verification problems to the Silver language and use the Silicon verifier [8]. For interactive verification, we are currently adopting the KeY tool[2] [9], a user-friendly interactive verifier for Java, based on dynamic logic. This entails extending the KeY verifier with permission accounting, thus we formalised our new permission system in the KeY logic. We also proved some permission properties with KeY, but as KeY does not support induction over data types, we could not prove the properties in their most general form. Therefore, we also formalised our permissions in PVS [10] and proved the most general properties correct with PVS. These properties and their formalisation are described in Sects. V and VI.

[1]http://fmt.cs.utwente.nl/research/projects/VerCors/.
[2]http://www.key-project.org/.

978-1-4673-7148-3/15 $31.00 © 2015 IEEE
DOI 10.1109/ISPDC.2015.26

Note that our permission model itself is language-independent, it can be used for any concurrent programming language with some means of synchronisation. It has to be, however, appropriately integrated into the corresponding program logic. In Sect. VII we very briefly show how this is done for the dynamic logic used by KeY to verify Java programs and we discuss another example that also attempts to show the relationship between our permission system and the classical fractional approach. The theories and examples from this paper are available on-line [11].

II. Symbolic Permissions in a Nutshell

Some of the shortcomings associated with using fractions for permissions have been already described in the context of the Chalice verifier for an idealised concurrent language [12]. In essence, the problematic points are reasoning about rational fractions [3] and the necessity to provide concrete fractions (or relative amounts [13]) in specifications. Real programming languages bring further challenges, such as re-entrant locks and other complex synchronisation methods, like count-down latches [4] in Java. To provide an intuition how permissions can be treated symbolically, this section discusses two examples. The first one describes the most basic case of a permission transfer between a thread and a lock. The second one is a verification scenario with multi-joined threads. Although fractions can still be used to provide a sufficient specification for this second example, it suffers from two drawbacks that we demonstrate with this example. In both examples, we keep the specifications and programs down to a bare minimum to only show the essentials of symbolic permissions; realistic specifications have to account for many other aspects of concurrent verification, this is discussed more in Sect. VII.

Simple Read and Write Resource Locking: A basic scenario for permission-based reasoning is when a resource (access to a memory location, in Java an object field o.f) is guarded by a lock. Acquiring the lock transfers either a partial or a full permission from the lock to the locking thread and allows that thread to, respectively, read or write the location. Releasing the lock transfers the permission back to the lock. In Fig. 1 the lock l provides a write access to o.x and a read access to o.y. For the specification of fractional permissions we use Perm(l, p) propositions that state the amount of permission p assigned to a location l. As in concurrent separation logic [6], only locations that are provided by the specification can be used by the method. Otherwise, we use JML syntax [7] for specifications.

Symbolic Permissions: To avoid fractions, we keep track of permission owners given by thread identifiers. In the context of Java, these are object references, but other identifiers could be used, e.g., integers. Just for this first example, consider our permissions to be lists of permission owners – *threads* or *locks*. New owners receive their permissions by either being added to this list (to gain shared access), or replacing existing owners (to fully take over access of another owner). When permissions are returned a reverse operation is applied. Each element in the list is a permission *slice*. Each permission slice gives partial access right to one owner and if all slices in the permission refer to the same owner, that owner has a full access right (i.e., identical permissions slices can be merged). Thus, in our approach the permission keeps a view of which other owners (threads) have potential rights associated with the permission.

```
class Client { ...
 l.lock();      // produces Perm(o.x, 1) and Perm(o.y, 1/2)
 o.x = o.y;     // write o.x, read o.y
 l.unlock();    // consumes Perm(o.x, 1) and Perm(o.y, 1/2)
 ... }

class Lock {
 //@ requires !locked; ensures locked;
 //@ ensures Perm(o.x,1) ** Perm(o.y,1/2);
 void lock();

 //@ requires Perm(o.x,1) ** Perm(o.y,1/2);
 //@ requires locked; ensures !locked;
 void unlock(); }
```

Figure 1. The use of a simple lock and its fractional-style specification.

For our example in Fig. 1, prior to locking, the view on the permissions to both o.x and o.y is that they completely belong to the lock object l. We denote this with the list [l]. Upon locking, the permission to o.x is completely transferred to the currently running thread ct, i.e., l is replaced by ct and the permission becomes [ct] meaning that the current thread fully owns this permission. The permission to o.y is transferred to ct only partially by first slicing the permission [l] into [l, l] and then transferring one of the slices to ct. This permission becomes [ct, l] meaning that the current thread owns one slice of this permission and that further permission transfers from the lock are still possible based on the l slice that the lock still owns. Upon unlocking, the permissions are returned to the lock by replacing the current thread object ct in the list with the lock object l, leaving the permission to o.x at [l] again, and the permission to o.y at [l, l], which is equivalent and can be merged into [l].

For the specification, using functional style expressions rather than separation logic style, through the respective postconditions we state how the permissions to o.x and o.y *change* when lock and unlock are called. We use one return function *retPerm* and two transfer functions *transPerm* and *transPermSplit* for a complete and slicing transfer, respectively. We define these functions formally in Sect. IV. They all take two (*from* and *to*) owners and the permission as arguments, and return a new permission. Figure 2 shows this.

Thread Multi-joining with Fractions: The second example illustrates the need to additionally keep track of to whom each permission slice is *owed*, i.e., the permission originators. Consider the two threads and the client code in Fig. 3.

```
    class Thread1 { Client c;
2     //@ requires Perm(c.a, 1/2);
      //@ ensures Perm(this.join, 1);
4     void start();

6     //@ requires Perm(this.join, p);
      //@ ensures Perm(c.a, p/2);
8     void join();

10    void run() {
        ... = c.a; // read c.a
12    } }
```

```
    class Thread2 { Client c; Thread1 t1;
2     //@ requires Perm(t1.join, 1/2);
      //@ ensures Perm(this.join, 1);
4     void start();

6     //@ requires Perm(this.join, p);
      //@ ensures Perm(c.a, p/4);
8     void join();

10    void run() {
       t1.join(); // get read access from t1
12      ... = c.a; // read c.a
    } }
```

```
    class Client { int a;
2     void main() {
        Thread t1 = new Thread1(this);
4       Thread t2 = new Thread2(this, t1);
        t1.start();
6       t2.start();
        ... = this.a; // read this.a
8       t1.join();
        t2.join();
10      this.a = ...; // write this.a
    } }
```

Figure 3. Multiply joined thread annotated with fractional-style permissions.

```
class Client {
 ...           // Perm(o.x), Perm(o.y) are [l]
 l.lock();     // Perm(o.x) becomes [ct], Perm(o.y) becomes [ct, l]
 o.x = o.y;    // [ct] → write access, [ct, ·] → read access
 l.unlock();   // Perm(o.x) becomes [l], Perm(o.y) becomes [l, l]
 ... }

class Lock {
 //@ ensures Perm(o.x) == transPerm(this, ct, \old(Perm(o.x)));
 //@ ensures Perm(o.y) == transPermSplit(this, ct, \old(Perm(o.y)));
 void lock();

 //@ ensures Perm(o.x) == retPerm(ct, this, \old(Perm(o.x)));
 //@ ensures Perm(o.y) == retPerm(ct, this, \old(Perm(o.y)));
 void unlock(); }
```

Figure 2. Simple lock specified with symbolic permissions.

Perm →	**this**.a	t1.join	t2.join
thread initialisation	1	—	—
t1.start();	$\frac{1}{2}$	1	—
t2.start();	$\frac{1}{2}$	$\frac{1}{2}$	1
read **this**.a (OK)	$\frac{1}{2}$	$\frac{1}{2}$	1
t1.join(); (p $= \frac{1}{2}$)	$\frac{1}{2}+\frac{1}{4}$	—	1
t2.join(); (p $= 1$)	$\frac{1}{2}+\frac{1}{4}+\frac{1}{4}$	—	—
write **this**.a (OK)	1	—	—

Figure 4. Verifying fractional permissions.

Here permissions are transferred upon thread forking (the start method) and thread joining. The code that is executed asynchronously by the forked thread is contained in the run method. In this example the Client class passes on a partial permission granting a read access for **this**.a to thread t1. The client itself maintains a read permission. It also allows thread t2 to get a read permission to **this**.a, but only transitively by allowing thread t2 to join thread t1 (by passing a special join permission) and effectively get the permission to read **this**.a from t1. After all the threads are joined (thread t1 twice, by the client and by t2) the client code is again allowed to write **this**.a by holding a complete permission to it.

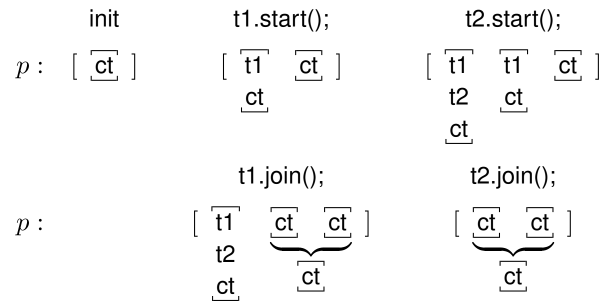
This scenario is specifiable with fractional permissions as shown in the annotations in Fig. 3. The essential part here is a so-called join token [14] – a fractional permission to join a thread that *captures* what part of the initially acquired permission should be returned to the joining thread. Splitting the join token between different threads allows these threads to join the same resource and acquire a corresponding partial permission to the resource depending on how much permission to the join token the joining thread has left. For instance, the specification of Thread1.start (lines 2–3 on the left in Fig. 3) states that upon forking, the thread transfers half of the permission to c.a. When the thread is joined (specification at lines 6–7) the corresponding part of this half is returned based on the current amount of the permission p to the join token. In particular, if the join token is not split (p is 1), the complete $\frac{1}{2}$ permission to c.a acquired on start is returned on join. The consistency of this permission flow is checked by verifying the run method with the start's precondition and join's postcondition.

When verifying the client code given the specifications of Thread1 and Thread2, all parametric permissions are assigned concrete values and the permission flow for the client is traced as shown in Fig. 4. The run method of Thread2 that also joins t1 is verified in a similar way.

Although fractional permissions still work for this example there are two limitations. The first problem is the need to specify concrete values. In the specification of Thread2.join one has to calculate the permissions to come up with $\frac{p}{4}$. This specification is not modular, in the sense that extending the client code might (in principle) invalidate this specification, and in fact, locally this value seems arbitrary. At this point, one would really like to specify the transfer of all *eligible* permissions. The second problem is the use of the join token to implicitly store and track the permission amounts to c.a. We only specified one actual memory location using a permission, and we used two entire join tokens of threads t1 and t2 to track this permission in our program. In effect, the information about one memory location is implicitly tracked in *three* different permissions, and the join tokens cannot be used to track other locations, unless they would have the same permission flow as c.a. If we wanted to include another memory location, say, c.b that would be first written by t1, then be written by t2, and then be written by the client again, we would have to start adding new join tokens for each new location.

Symbolic Permission Transfers and Debts: In our symbolic permissions we store all the information required to transfer and reconstruct a single permission within the permission expression itself, so that each resource can use a fully independent permission, i.e., permission flow is specified separately for each location and independently of other permissions. Compared to the simple lock in our first example, to allow for more general permission transfers we need to record the transfer history and introduce the notion of a debt transfer, as we explain in the following.

On initialisation, a new permission Perm(**this**.a) (let us abbreviate it by p) now becomes a two-dimensional list that contains only ct, denoted [[ct]]. It states that the permission contains only one slice and this slice belongs to the distinguished current thread ct only, the first *originator* of the permission, without any history of permission transfers. The inner list is used to keep track of the transfer history of each slice, i.e., an "I-owe-you" dependency chain. After executing our client program from Fig. 3, the permission p takes the following forms after each permission transferring call:



The inner lists, showed vertically, are single permission slices. After initialisation the originator ct is the most recent owner of the only slice in this permission, meaning that ct has a full permission to the location. After t1.start, the permission is split and one of the resulting slices is transferred from ct to t1. At this point, both threads hold only a read permission, i.e., they each own one slice and none of them owns all slices. Additionally we know that thread t1 *owes* its share to ct in case the return is requested. This debt replaces the join token from Fig. 3. The most interesting transformation of p happens when thread t2 is forked (t2.start). Instead of transferring any current share of p directly to t2, part of the debt that t1 has to ct is transferred from ct to t2. The current thread can do that because it is the holder of the debt. This transfer of debt effectively means, from the point of view of the originating current thread, that t2 now also has the potential right to join t1 to obtain a part of permission p from t1. This corresponds to the splitting of the join token specified in Fig. 3 (line 2 in the middle). Because the debt was split before this transfer, ct still maintains its right to join t1. This happens in the next step (t1.join) after which t1 is removed from the top of the middle slice – the permission is returned from t1 to ct. Permission p now has two identical $\boxed{\text{ct}}$ slices, which can be merged into one, depicted with an under-brace above. Finally, the specification of t2.join() should capture that thread t2 joins t1, and then returns its permission to ct. Consequently, the current thread regains full write permission p it started with in the first place, and can write to the associated memory location. Note that the current thread has also the right to read this location in between the t2.start()–t1.join() and t1.join()–t2.join() calls, because it owns one of the permission slices, $\boxed{\text{ct}}$, at these points, but not all of the slices.

As before, we can specify how p is changed upon thread forking and joining by applying permission transfer functions to p specifying the *from* and *to* threads of the transfer. Using these functions the start and join postconditions of threads t1 and t2 are specified as follows, where p is Perm(c.a):

$$\begin{array}{ll}
\text{t1.start():} & p == \mathit{transPermSplit}(\text{ct}, \text{t1}, \backslash\textbf{old}(p)); \\
\text{t2.start():} & p == \mathit{transPermDebtSplit}(\text{ct}, \text{t2}, \backslash\textbf{old}(p)); \\
\text{t1.join():} & p == \mathit{retPerm}(\text{t1}, \text{ct}, \backslash\textbf{old}(p)); \\
\text{t2.join():} & p == \mathit{retPerm}(\text{t2}, \text{ct}, \mathit{retPerm}(\text{t1}, \text{t2}, \backslash\textbf{old}(p)));
\end{array}$$

The specification for t1.start states that p is transferred from the current thread to t1 after first being split, like for o.y permission in the first example. Function $\mathit{transPermDebtSplit}$ specifies the transfer of the debt as explained above, after also splitting the permission into two slices first. Again, we define this function formally in Sect. IV.

The correctness of this reasoning is of course subject to also verifying the behaviour of both threads t1 and t2. In particular, for thread t1 we need to show that it does not modify permission p, and for thread t2 we need to show that it does indeed join thread t1. In the following we describe our permission data type more formally.

III. The Permission Data Type

We assume that all threads can be uniquely identified with values of some data type. In Java these are instances of the Thread class. We also assume that the distinguished current thread is uniquely identifiable in the set of all threads with ct as above. Furthermore, not only threads can hold ownership of memory locations, so we generally assume objects to be permission owners. In particular, locks (see first example) and other synchronisation objects can also hold permissions. Thus, the owners are simply Object references.

Our permission data type is a two-dimensional list. One of these dimensions is a list of owners that represent the current view of the history of ownership (subsequent originators) of a particular permission slice. A singleton list represents the initial owner, and a current new owner is added at the head of the list (or on top when viewed vertically as in our second example in Sect. II). No owner list should in principle be empty, the permissions are always initialised with at least one owner, the first originator. The data type defining permission owner lists is the following:

$$\mathit{OwnerList} ::= \mathit{emptyOwner} \mid \mathit{owner}(\mathit{Object}, \mathit{OwnerList}).$$

The other dimension of our permission data type stores complete permission slices. We use lists again; initially a permission consists of only one slice, meaning the complete permission belongs to the owner on the top of the owners list for that slice. In terms of access rights to some associated resource, such one slice means a full (write) access. When there are two or more slices of a permission, this indicates two or more read rights assigned to, possibly different, owners. If there are multiple slices, but the owner of all the slices is the same object, the permission is still a full permission. The exact definition of predicates establishing the read and the write permission is given shortly in the next section. The permission data type is defined as:

$$\mathit{Perm} ::= \mathit{emptyPerm} \mid \mathit{slice}(\mathit{OwnerList}, \mathit{Perm}).$$

To avoid notational confusion between owner's lists and slices we purposely use distinctive names for the two lists' constructors, i.e., $\mathit{emptyOwner}$ and owner for owner lists, and $\mathit{emptyPerm}$ and slice for permission slices. Furthermore, using dedicated list structures instead of generic ones also allows us to provide optimised all-in-one permission transfer functions, see next. However, generic lists could be also used to define our permissions, this is what we partly did in the PVS formalisation discussed later in Sect. VI.

An initial permission assigned to a freshly allocated memory location is a one-owner one-slice permission that belongs to the current thread: $\mathit{initFull} := \mathit{slice}(\mathit{owner}(\mathsf{ct}, \mathit{emptyOwner}), \mathit{emptyPerm})$. From this point on, the permission can be subjected to permission checks to establish access rights and permission transfers upon entering synchronisation points. Structure-wise, $\mathit{initFull}$ is the minimal expression that our permissions should take, i.e., there is always at least one slice and at least one owner of this slice.

IV. Queries and Commands on Permissions

Our first query function checks that a given object is an owner in the owners list. We do this for an arbitrary owner deep in the list, not only the current owner on the top of the list. This is to allow operating on the owner list below the top element to provide the ability to *transfer debts* as exemplified in Sect. II. The owner list can be changed only by the object that is the owner at the given depth. Thus, we define the predicate $\mathit{checkOwner}$:

$$\begin{array}{l}
\mathit{checkOwner} : \mathit{Object} \times \mathit{nat} \times \mathit{OwnerList} \to \mathit{Bool} \\
\mathit{checkOwner}(o, d, l) := \\
\quad l = \mathit{emptyOwner} \to \mathsf{false} \\
\quad l = \mathit{owner}(o', t) \to \\
\qquad \text{if } d = 0 \text{ then } o = o' \text{ else } \mathit{checkOwner}(o, d-1, t)
\end{array}$$

that checks the owner at a given index of the owner list. The definition is straightforward; the list is traversed to find the d position at which the element should be equal to the element o being looked for, while the empty list has no owner.

The $\mathit{readPerm}$ and $\mathit{writePerm}$ predicates check the type of access the given permission grants. The parameters for both predicates are the object that we check the access for and the permission expression. The checking is analogous to existential and universal quantification, respectively. For the read access we need to find at least one permission slice with the current owner equal to the object in question, for write access all slices need to belong to this object:

$$\begin{array}{l}
\mathit{readPerm} : \mathit{Object} \times \mathit{Perm} \to \mathit{Bool} \\
\mathit{readPerm}(o, p) := \\
\quad p = \mathit{emptyPerm} \to \mathsf{false} \\
\quad p = \mathit{slice}(l, p') \to \begin{array}{l} \mathit{checkOwner}(o, 0, l) \\ \vee\, \mathit{readPerm}(o, p') \end{array} \\
\mathit{writePerm} : \mathit{Object} \times \mathit{Perm} \to \mathit{Bool} \\
\mathit{writePerm}(o, p) := \\
\quad p = \mathit{emptyPerm} \to \mathsf{true} \\
\quad p = \mathit{slice}(l, p') \to \begin{array}{l} \mathit{checkOwner}(o, 0, l) \\ \wedge\, \mathit{writePerm}(o, p') \end{array}
\end{array}$$

To define the actual permission transfer functions we first define operations to add a new owner to a single permission slice and to return permission slices to their previous owners. As permission slices can be only mutated by the associated owners, these two functions are partial, i.e., guarded by the $\mathit{checkOwner}$ predicate. An owner o present anywhere in the list can insert a new owner o' above itself to redirect ownership returning to this other object o', i.e., o can transfer its ownership ($d = 0$) or debt ($d > 0$) to o'. When owners return their permissions, the current rightful owner is simply removed from the top of the owner list:

$$\begin{array}{l}
\mathit{insertOwner} : \\
\quad \mathit{Object} \times \mathit{Object} \times \mathit{nat} \times \mathit{OwnerList} \to \mathit{OwnerList} \\
\mathit{insertOwner}(o, o', d, l) := \quad [\text{when } \mathit{checkOwner}(o, d, l)] \\
\qquad d = 0 \to \mathit{owner}(o', l) \\
\qquad d > 0 \wedge l = \mathit{owner}(h, t) \to \\
\qquad\qquad \mathit{owner}(h, \mathit{insertOwner}(o, o', d-1, t))
\end{array}$$

$$\begin{array}{l}
\mathit{returnOwner} : \mathit{Object} \times \mathit{OwnerList} \to \mathit{OwnerList} \\
\mathit{returnOwner}(o, l) := \quad [\text{when } \mathit{checkOwner}(o, 0, l)] \\
\qquad l = \mathit{owner}(h, t) \to t
\end{array}$$

The two quoted functions are used in the top-level permission slicing and recombining functions used for permission transfer. Their signatures are the following:

$$\begin{array}{l}
\mathit{transferPerm} : \\
\quad \mathit{Bool} \times \mathit{Object} \times \mathit{Object} \times \mathit{nat} \times \mathit{Perm} \to \mathit{Perm} \\
\mathit{returnPerm} : \mathit{Object} \times \mathit{Object} \times \mathit{Perm} \to \mathit{Perm}
\end{array}$$

and they are formally defined in Fig. 5. The first parameter to transfer a permission specifies whether a permission slice should be first split into two before the transfer. This splitting

```
$\mathit{transferPerm}(s, f, t, d, p) :=$
  $p = \mathit{emptyPerm} \rightarrow \mathit{emptyPerm}$
  $p = \mathit{slice}(l, p') \rightarrow$
    $f = t \rightarrow p$
    otherwise $\rightarrow$
      $\mathit{checkOwner}(f, d, l) \rightarrow$
        $s = \mathsf{true} \rightarrow \mathit{slice}(\mathit{insertOwner}(f, t, d, l), p)$
        $s = \mathsf{false} \rightarrow \mathit{slice}(\mathit{insertOwner}(f, t, d, l),$
                                  $\mathit{transferPerm}(s, f, t, d, p'))$
      otherwise $\rightarrow \mathit{slice}(l, \mathit{transferPerm}(s, f, t, d, p'))$

$\mathit{returnPerm}(f, t, p) :=$
  $p = \mathit{emptyPerm} \rightarrow p$
  $p = \mathit{slice}(l, p') \rightarrow$
    $f = t \rightarrow p$
    otherwise $\rightarrow$
      $\mathit{checkOwner}(f, 0, l) \wedge \mathit{checkOwner}(t, 1, l) \rightarrow$
        $l = \mathit{owner}(f, l') \wedge p' = \mathit{slice}(l', p'') \rightarrow$
          $\mathit{slice}(\mathit{returnOwner}(f, l),$
                          $\mathit{returnPerm}(f, t, p''))$
        otherwise $\rightarrow \mathit{slice}(\mathit{returnOwner}(f, l),$
                          $\mathit{returnPerm}(f, t, p'))$
      otherwise $\rightarrow \mathit{slice}(l, \mathit{returnPerm}(f, t, p'))$
```

Figure 5. Complete definitions of $\mathit{transferPerm}$ and $\mathit{returnPerm}$.

differentiates between a complete or a partial transfer, as discussed in Sect. II. In the first case, the ownership of all slices is transferred – the current owner gives up its whole access right, whatever it is, to another object. In the second case one slice is split into two, and then only one is transferred to another object – the current owner retains a partial right and grants a partial right to another object. The two object parameters to $\mathit{transferPerm}$ are the *from* and *to* objects of the transfer. The integer parameter is the depth at which the transfer happens to allow for transferring debts as described above and in Sect. II. Finally, the function takes a permission and returns an accordingly modified one. The transfer function is an identity in two cases: when the object that requests the transfer does not have any rights in the permission, and when the *from* and *to* parameters are identical.

The $\mathit{returnPerm}$ function also only allows the current slice owners to return their rights. However, contrary to $\mathit{transferPerm}$, permissions are always returned completely, if possible; the current owner is obliged to give up rights to all currently owned slices to their originators. This is to ensure that no permissions are unnecessarily lost during returns. Upon return identical slices are merged together as in the examples in Sect. II.

The transfer functions from the examples in Sect. II correspond to the $\mathit{transferPerm}$ function as follows: $\mathit{transPermSplit}(f, t, p) \equiv \mathit{transferPerm}(\mathsf{true}, f, t, 0, p)$ and $\mathit{transPermDebtSplit}(f, t, p) \equiv \mathit{transferPerm}(\mathsf{true}, f, t, 1, p)$. The $\mathit{retPerm}$ function from Sect. II has a direct correspondence to $\mathit{returnPerm}$. Moreover, our definition of $\mathit{returnPerm}$ ensures that adjacent identical slices likely to appear after an earlier split transfer are merged into one, exactly as we showed in Sect. II where $[[\mathsf{ct}], [\mathsf{ct}]]$ was merged into $[[\mathsf{ct}]]$. This allows to keep the permission expressions short, which should improve reasoning. However, in general our functions are not guaranteed to always produce a completely simplified permission. This does not destroy the correctness of the system, unreduced expressions can still be queried to establish the associated access rights and can be subjected to further transfer operations. Hence, unreduced expressions may only hinder reasoning efficiency. In practice, however, most verification scenarios shall only involve a very limited number of threads or synchronisation objects, keeping the permission expression limited in size anyhow and maintaining efficiency in reasoning. For example, a typical synchronisation pattern with a simple write lock would involve one complete permission transfer over a full permission $[[\mathsf{l}]]$ and a subsequent permission return, as shown in our first example in Sect. II.

V. Permission Properties

The importance of permission accounting in concurrent reasoning is that threads verified with respect to permission specifications are guaranteed to be data-race free. But this only holds if the permissions themselves and their operations preserve certain properties. For instance, in fractional style permissions one has to ensure that no permissions greater than 1 in value are ever created, or that no two threads hold more than a full permission to one location. More generally, one has to ensure that no *deficit* (“negative” permission) or *surplus* (more than a write permission) rights are created when permissions are transferred.

The second vital aspect of permission properties is to support efficient reasoning and enable abstraction. All proved permission properties can be turned into lemmas and subsequently used for efficient verification. For abstraction, one can use facts like “transferring and returning a permission gives the original permission” in which case the actual permission can be left underspecified by only stating predicates that hold for it.

Our permission expressions are self-contained. In particular, objects involved in each permission transfer are stated and stored in the permission expression. Consequently, many of the properties we are interested in are easy to establish independently from a particular verification logic or context. In fact, some of the properties that we list below seem trivial. We only list the very crucial properties necessary to guarantee sound reasoning to establish data-race freedom and support basic abstraction from concrete expressions, but other auxiliary properties can be added, especially ones that can support more efficient reasoning. Hence, the properties that we concentrate on are the following:

1) Initial permission is a full access permission,
2) A write permission is also a read permission,
3) A write permission for one object grants no access for other objects,
4) A split-transfer of any permission to another object leaves the permission in a read access state for the original object, but not in a write access state,
5) Similarly, such a transfer gives the receiver a read, but not a write access,
6) A complete transfer of permissions strips the original owner of all rights,
7) Similarly, the receiver gets all the rights that the original object had,
8) Any debt transfer ($d > 0$) retains all access for all current permission owners,
9) Objects not involved in the transfer or return retain all their access rights,
10) Any transfer followed by a corresponding return retains all original rights.

VI. Tool Formalisation and Property Proofs

We formalised both our permission theory and properties in a formal language of two theorem provers and used the associated tools to prove the properties correct. The first prover is the KeY verification system for Java programs. The formalisation of the permission system in KeY is necessary anyhow, as KeY is the primary target to implement our permissions for verification of concurrent Java programs, see next section. We used the automated mode of KeY to show the correctness of parts of properties that can be established with pure first-order reasoning. However, full properties require structural induction proofs, which KeY cannot do in a methodological way. Thus, we also employed a prover more suited for this task, the Prototype Verification System (PVS) [10].

The KeY system is based on a first-order dynamic logic [15] tailored to Java [9], but for our permission theory itself the first-order base of the logic is sufficient. The sequent calculus of the KeY logic is defined in external files, that declare the logical sorts, function and predicate symbols that apply to the corresponding sorts, and rewrite rules that give the functions and predicates their meaning. The rules are essentially elaborate pattern-matching-based *find* and *add/replace* schemas for changing proof sequents. The KeY prover implements a very efficient proof engine to apply these rules in the effort to close proofs automatically.

Fig. 6 gives a snapshot of the KeY formalisation of our permissions. Only two rewrite rules that define the full access permission check are given, it should be clear how they use pattern-matching to modify the associated formulae. We also stated our permission properties (to a limited extent, see below) directly as first-order logic formulae and we used the core permission rules to prove the properties correct. Some of our properties have been already defined as lemma

```
\sorts{ Perm; OwnerList; } \functions { ... }

\predicates {
  readPerm(Object, Perm); writePerm(Object, Perm); }

\schemaVariables {
  \term Object o; \term Perm p; \term OwnerList ol; }

\rules {
  writePermSlice {
    \find(writePerm(o, slice(ol, p)))
    \replacewith(checkOwner(o, 0, ol) & writePerm(o, p)) };

  writePermEmpty {
    \find(writePerm(o, emptyPerm)) \replacewith(true) }; }
```

Figure 6. Snapshot of the KeY formalisation of symbolic permissions.

rules for verification of programs with KeY, however, the set of these lemma rules is not yet complete and we introduce them as required when working with new examples.

As an example, a property that states “a split transfer of an initial full permission from the current thread to another object revokes the write permission from the current thread” is formalised as the following KeY formula:

```
\forall Object o; (o != ct ->
    !writePerm(ct, transferPerm(TRUE, ct, o, 0, initFull)))
```

This is proved fully automatically with KeY. This property is a concrete instance of the more general property 4 above stated for any permission that is at least a read permission for the current thread. This general property requires a structural induction proof that is not possible with KeY, hence we turned to PVS, an interactive theorem prover for higher-order logic, to prove the properties in their most general form.

Similarly to KeY, special purpose data types have been defined in PVS to represent permissions and the properties as PVS lemmas. Due to space restrictions we do not quote these definitions, however, the complete PVS formalisation and all property proofs are available on-line [11]. Generally, all the properties are proven correct by structural induction on the form of the permission and subsequent unfolding of appropriate definitions. Noteworthy, properties 9 and 10 required well-founded induction to tackle the on-the-fly merging of slices when applying the $returnPerm$ function.

VII. Application in Reasoning about Concurrent Programs

So far we only discussed the permission expressions in isolation and stated that each such expression refers to some memory location of the program to be verified. To reason about actual programs we need to establish a connection between permission expressions and the memory model of the verification logic. Below we sketch how this is done for the KeY verifier and what needs to be considered in general. We then use this connection to verify another example that compares symbolic permissions with the fractional ones.

In the KeY logic, memory is represented as an explicit heap variable that maps objects' fields (i.e., memory locations) to their values. The heap program variable is special in the sense that it is subject to all matters associated with program memory change and framing. In particular, when proof obligations for establishing the correctness of a method contract are generated they include formulae ensuring that the framing conditions are satisfied. These formulae are quantifications over the memory locations on the heap following the dynamic frames approach [16].

The essence of adding support for permissions in KeY is to add a second *permission heap* that, instead of the program memory values, keeps the permissions to all memory locations that the program operates on. All the existing machinery of KeY for operating on the regular heap variable scales to the operation on an arbitrary, but fixed number of heaps simultaneously [17]. What remains is to lift this extension to the specification language JML*, KeY's version of JML [7]. Essentially, we do this by allowing one to state the heap variable that a given expression refers to explicitly in JML* specifications, with convenience expressions on top of it. A very simple example of this is the following:

```
//@ requires \writePerm(\perm(this.o));
//@ ensures this.o == p;
//@ assignable<heap> this.o;
//@ assignable<permissions> \nothing;
public void method(Object p) { this.o = p; }
```

The value for **this**.o on the permission heap (accessed with the \perm operator) has to be a write permission for the current thread (\writePerm). The assignable clauses state how the two heaps change. On the memory heap **this**.o is changed. The permission heap is unchanged as the program only uses the permission to **this**.o, but does not make any permission transfers for this location. Note that both assignable clauses are necessary as KeY, with or without the permission extension, always requires explicit dynamic frames, i.e., permissions are employed only to specify data non-interference and not implicit framing. Finally, no permissions for the object reference p are necessary as it is not a location. This example verifies automatically with KeY.

For the overall soundness of our reasoning we also have to show self-framing of specifications with respect to permissions, i.e., specifications should refer only to locations they have at least a read permission to. Conceptually this is rather straightforward in our approach and uses the same principle as showing data dependency contracts in KeY [16]. Shortly, one shows that a formula does not change its valuation when locations outside of the set of the dependency frame are anonymised. Here the dependency frame are the locations that we can show at least a read access for in a given context. In the short example above the dependency frame is **this**.o defined by the write permission in the precondition and none of the formulas in the specification mention other locations. The permission expression itself over **this**.o is self-framed by definition. We are in progress of working out the fine details of permission self-framing and related matters, completing the implementation in KeY, and writing a follow-up paper on this subject.

Apart from that, our implementation in KeY is fully functional, our second example from Sect. II can also be specified and verified with KeY. We have also specified and verified a more complex version of this example, a plotter with two filter threads from [18]. Both of these examples are available on-line [11]. Here we take the opportunity to describe yet another example, our version of the motivating read-write lock example from [13] where constraints are used to combine fractions with counting permissions and to mitigate the need to use concrete rational numbers (though reasoning about *symbolic* rational numbers is still required). This example shows some fine points of specifying with our permissions and the conceptual difference between our approach and the classical fractions approach.

The code and our specification for the read-write lock example from [13] is shown in Fig. 7. We slightly refactored the original code and in-lined the shared variable read and writing statements to save space, however, it preserves the crucial difficulty of the original example (the full example that uses delegated and specified read and write methods is available at [11]). The lock and unlock methods are assumed to have an implementation providing a simple exclusive access lock. Through the use of the rds field this is turned into a counting read-write lock to access or change the shared val field. When rds is strictly positive only reading is possible, when it is 0 writing can occur. The difficulty in the example comes from the fact that the lock essentially only protects the rds field itself, while val is read when the lock is actually not acquired but when rds is guaranteed to be positive (l. 21 in Fig. 7). Writing of val is done within the scope of the lock when rds is equal to 0 (l. 14).

In [13] the access to val is guarded by a lock with a $1 - \text{rds} * \epsilon$ fraction which by reference to rds provides sufficient permissions when reading and writing occurs. In our specification we provide the information on how the permissions are flowing depending on the value of rds when the lock is used. Upon acquiring the lock all currently available permissions to val and the complete permission to rds (also ords, see below) are transferred to the currently running thread (ls. 31–33). Upon release all permissions are returned to the lock (ls. 40–42), however, when the value of rds is noticed to have been strictly increased since the call to lock, a spliced part of the permission to val is transferred again to the current thread to enable reading (l. 43).

The remaining parts of the lock and unlock specification is additional book-keeping and consistency checks. In particular, the ords ghost field records the value of rds when the lock is acquired to establish the lock usage scenario as described above. We also state that when the lock is successfully acquired, then the permission to rds and ords completely belonged to the lock. Note that information about fields such as rds only can be given in the postcondition of lock (e.g., l. 29) when the lock is already acquired and a corresponding permission is present.[3] In the precondition of lock there are not sufficient permissions to specify anything about rds (for unlock the inverse situation occurs). In fact, we require that the calling thread does not have any access to rds which is expressing that the lock is not currently acquired by the current thread. A similar specification is to be found for methods doRead and doWrite and in particular in the loop invariant of doWrite which states that the lock is not acquired outside of the loop. Finally, the **diverges** clause specifies that doWrite may possibly not terminate.

VIII. Conclusions and Future Work

We discussed a symbolic permission system for concurrent reasoning that improves over the established fractions approach in at least two ways. First, we mitigate the need to reason about fractional numbers that is considered difficult in first-order reasoning [3]. Second, we introduced mechanisms that allow us to reason about complex permission flow scenarios where multiple threads and synchronisation objects are involved. This is illustrated by an example program where two different threads simultaneously join a single thread. With our new permissions, the KeY verifier is already able to verify concurrent programs, including the quoted example [11]. We attribute the relative ease of reasoning with our new permissions to the explicit approach, i.e., not hiding or assuming information in the verification context, like it is commonly done in separation logic-based verification methods [19]. Our explicit approach in verification has proven itself efficient also in our preceding work [17]. To enable fully flexible reasoning about arbitrary multi-threaded Java programs, we are currently working on the generation of the complete and sound proof obligations in KeY Java dynamic logic, and on optimising our permission system for lemma-based reasoning in KeY. Although already functional, the latter is particularly important to enable suitable abstractions in specifications with permissions.

Related Work: Apart from fractional style permissions first described in [1] the literature also describes counting permissions and tree permissions as alternatives. As described in [20], counting permissions simply use a counter instead of a fraction and are considered complementary to fractional permissions, which are not suitable for some synchronisation mechanism, like semaphores. We briefly related to the work on constrained abstract fractional permissions [13] in the previous section. Tree permissions [2] are also close to our work, in that they use a dedicated data type to abstract away irrelevant information. However, tree permissions are only a direct abstraction of fractions,

```
   public class ReadWrite {
 2   private int val;
     private int rds; //@ private instance ghost int ords;
 4
     //@ requires !\readPerm(\perm(rds));
 6   //@ ensures !\readPerm(\perm(rds));
     //@ assignable<heap,permissions> rds, ords, val;
 8   //@ diverges true;
     public void doWrite() {
10     boolean done = false;
       //@ loop_invariant !\readPerm(\perm(rds));
12     //@ assignable<heap,permissions> rds, ords, val;
       while(!done) {
14       lock(); if(rds==0){ val++; done=true; } unlock(); } }

16   //@ requires !\readPerm(\perm(rds));
     //@ ensures !\readPerm(\perm(rds));
18   //@ assignable<heap,permissions> rds, ords, val;
     public int doRead() {
20     lock(); rds++; unlock(); // read "lock"
       int r = val;
22     lock(); rds−−; unlock(); // read "unlock"
       return r; }
24
     //@ requires !\readPerm(\perm(rds));
26   //@ ensures \old(\readPerm(\perm(val))) ==> rds>0;
     //@ ensures \writePermObj(this, \old(\perm(rds)));
28   //@ ensures \writePermObj(this, \old(\perm(ords)));
     //@ ensures rds >= 0 && ords == rds && (rds == 0 ==>
         ↪ \writePermObj(this, \old(\perm(val))));
30   //@ ensures \readPermObj(this, \old(\perm(val)));
     //@ ensures \perm(rds) == \transPerm(false, this, \ct, 0,
         ↪ \old(\perm(rds)));
32   //@ ensures \perm(ords) == \transPerm(false, this, \ct, 0,
         ↪  \old(\perm(ords)));
     //@ ensures \perm(val) == \transPerm(false, this, \ct, 0,
         ↪ \old(\perm(val)));
34   //@ assignable<permissions> rds, ords, val;
     //@ assignable ords;
36   public native void lock();

38   //@ requires rds >= 0 && \writePerm(\perm(rds));
     //@ requires \writePerm(\perm(ords));
40   //@ ensures \perm(rds) == \retPerm(\ct, this,
         ↪ \old(\perm(rds)));
     //@ ensures \perm(ords) == \retPerm(\ct, this,
         ↪ \old(\perm(ords)));
42   //@ ensures \old(rds) <= \old(ords) ==> \perm(val) ==
         ↪ \retPerm(\ct, this, \old(\perm(val)));
     //@ ensures \old(rds) > \old(ords) ==> \perm(val) ==
         ↪ \transPerm(true, this, \ct, 0, \retPerm(\ct, this,
         ↪         \old(\perm(val))));
44   //@ assignable<heap,permissions> rds, ords, val;
     public native void unlock();
46 }
```

Figure 7. The read-write example specified with symbolic permissions.

and by using only two tokens to mark the tree nodes, they only differentiate between the current thread and all other threads, i.e., they cannot identify each single thread separately. Moreover, transfer histories are not recorded causing similar problems as with rational fractions when specifying complex permission flows.

Not many implemented verification systems that we know of actually support permission-based reasoning. VeriFast [21] does, it is based on separation logic and imple-

[3]On the more conceptual level this is actually stating that lock may block execution – we can only talk about the lock state once lock has successfully returned.

ments fractional style permissions. Another such tool would be the Chalice system [22], which is based on implicit dynamic frames, and deals with permissions also in the fractional fashion providing strong, but limited means of permission abstraction [12]. Finally, our own automated VerCors toolset [5] implements our version of separation logic with permissions for Java [6] and uses Silicon [8] as a back-end verifier.

Acknowledgement

The work presented in this paper is supported by ERC grant 258405 for the VerCors project.

References

[1] J. Boyland, "Checking interference with fractional permissions," in *Static Analysis Symposium*, ser. LNCS, R. Cousot, Ed., vol. 2694. Springer, 2003, pp. 55–72.

[2] R. Dockins, A. Hobor, and A. W. Appel, "A fresh look at separation algebras and share accounting," in *7th Asian Symposium on Programming Languages and Systems*, ser. LNCS, Z. Hu, Ed., vol. 5904. Springer, 2009, pp. 161–177.

[3] X. B. Le, C. Gherghina, and A. Hobor, "Decision procedures over sophisticated fractional permissions," in *10th Asian Symposium on Programming Languages and Systems*, ser. LNCS, R. Jhala and A. Igarashi, Eds., vol. 7705. Springer, 2012, pp. 368–385.

[4] A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski, "Formal specifications for Java's synchronisation classes," in *Conference on Parallel, Distributed, and Network-Based Processing*, A. L. Lafuente and E. Tuosto, Eds. IEEE Computer Society, 2014, pp. 725–733.

[5] A. Amighi, S. C. Blom, M. Huisman, and M. Zaharieva-Stojanovski, "The VerCors project: Setting up basecamp," in *6th Workshop Programming Languages meets Program Verification*. ACM, 2012, pp. 71–82.

[6] C. Haack, M. Huisman, and C. Hurlin, "Reasoning about Java's reentrant locks," in *6th Asian Conference on Programming Languages and Systems*, ser. LNCS, G. Ramalingam, Ed., vol. 5356. Springer, 2008, pp. 171–187.

[7] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *SIGSOFT*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[8] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," ETH Zürich, Tech. Rep., 2014.

[9] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. LNAI. Springer, 2007, vol. 4334.

[10] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *11th International Conference on Automated Deduction (CADE)*, ser. LNAI, D. Kapur, Ed., vol. 607. Springer, June 1992, pp. 748–752.

[11] Symbolic Permissions On-line. http://wwwhome.ewi.utwente.nl/~mostowskiwi/permissions/.

[12] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers, "Abstract read permissions: Fractional permissions without the fractions," in *Verification, Model Checking, and Abstract Interpretation 2013*, ser. LNCS, R. Giacobazzi, J. Berdine, and I. Mastroeni, Eds., vol. 7737. Springer, 2013, pp. 315–334.

[13] J. Boyland, P. Müller, M. Schwerhoff, and A. J. Summers, "Constraint semantics for abstract read permissions," in *Formal Techniques for Java-like Programs (FTfJP)*. ACM, 2014.

[14] C. Haack and C. Hurlin, "Separation logic contracts for a Java-like language with fork/join," in *Algebraic Methodology and Software Technology*, ser. LNCS, J. Meseguer and G. Rosu, Eds., vol. 5140. Springer, 2008, pp. 199–215.

[15] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic Logic*. MIT Press, 2000.

[16] P. H. Schmitt, M. Ulbrich, and B. Weiß, "Dynamic frames in Java dynamic logic," in *Formal Verification of Object-Oriented Software Conference*, ser. LNCS, B. Beckert and C. Marché, Eds., vol. 6528. Springer, 2011, pp. 138–152.

[17] W. Mostowski, "A case study in formal verification using multiple explicit heaps," in *IFIP Joint International Conference on Formal Techniques for Distributed Systems*, ser. LNCS, D. Beyer and M. Boreale, Eds., vol. 7892. Springer, 2013, pp. 20–34.

[18] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski, "Verification of concurrent systems with VerCors," in *14th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Executable Software Models*, ser. LNCS, M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, and I. Schaefer, Eds., vol. 8483. Springer, 2014, pp. 172–216.

[19] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *17th IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, pp. 55–74.

[20] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson, "Permission accounting in separation logic," in *Principles of Programming Languages*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 259–270.

[21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "Verifast: A powerful, sound, predictable, fast verifier for C and Java," in *NASA Formal Methods*, ser. LNCS, vol. 6617. Springer, 2011, pp. 41–55.

[22] K. R. M. Leino, P. Müller, and J. Smans, "Verification of concurrent programs with Chalice," in *Foundations of Security Analysis and Design*, A. Aldini, G. Barthe, and R. Gorrieri, Eds. Springer, 2009, pp. 195–222.