Verifying Class Invariants in Concurrent Programs

Marina Zaharieva-Stojanovski and Marieke Huisman

University of Twente, the Netherlands

Abstract. Class invariants are a highly useful feature for the verification of object-oriented programs, because they can be used to capture all valid object states. In a sequential program setting, the validity of class invariants is typically described in terms of a *visible state semantics*, i.e., invariants only have to hold whenever a method begins or ends execution, and they may be broken inside a method body. However, in a concurrent setting, this restriction is no longer usable, because due to thread interleavings, any program state is potentially a visible state.
In this paper we present a new approach for reasoning about class invariants in multithreaded programs. We allow a thread to explicitly break an invariant at specific program locations, while ensuring that no other thread can observe the broken invariant. We develop our technique in a permission-based separation logic environment. However, we deviate from separation logic's standard rules and allow a class invariant to express properties over shared memory locations (the *invariant footprint*), independently of the permissions on these locations. In this way, a thread may break or reestablish an invariant without holding permissions to all locations in its footprint. To enable modular verification, we adopt the restrictions of Müller's ownership-based type system.

1 Introduction

In object-oriented programs, class invariants are typically used to express properties about the object's state that should hold throughout the object's life cycle. However, in practice it is often impossible to maintain the invariant continuously. For example, for an invariant that expresses a relation between fields x and y, $x == y$, when x is updated, y must also be updated, and both updates can not be done atomically. Therefore, invariant theory should provide for the possibility that a class invariant is temporarily *broken* at specific program parts.

In the sequential setting, the theory about invariant validity is well-developed; in essence, class invariants only have to hold in the program's *visible states*, i.e., in pre- and poststates of public methods [17]. In particular, if a class invariant I holds in a method's prestate, the method must end in a state satisfying I.

However, in the setting of multithreading programs, this approach can not be carried over directly. Due to possible interference between parallel threads, any program state may be *visible*. For example, when the field x in the invariant above is updated, any other thread might observe this change and the broken invariant. This problem is sometimes called a *high-level data race* [2].

Therefore, this paper defines an approach to define validity of class invariants in a multithreaded setting. Our approach supports explicit *breaking of invariants*, under the condition that other threads can not see that the invariant is broken. We build our technique on *permission-based separation logic* [4], using a Java-like language. However, in contrast to standard separation logic, we explicitly make a distinction between *state formulas*, which describe a property about the shared state, and *resource formulas*, which describe when a thread holds a permission to access a certain location. We ensure modular verification using the restrictions from *ownership-based type systems* [7].

Our approach works as follows. A class invariant is specified as a condition on the shared memory. For each class invariant, we maintain a token that indicates whether the class invariant can be inspected. This token can be split and combined: if a thread has the complete token, it can *break* the invariant; otherwise it can only *use* it. Breaking the invariant is done by executing a (specification-only) `unpack` statement. When a thread reestablishes the invariant, the token to inspect the invariant becomes available again for other threads to break or inspect the invariant. This behaviour is modeled by a (specification-only) `pack` statement. Thus, within the unpacked segment, a thread is free to do whatever it wants with the class invariant, as our verification approach ensures that no other thread can observe the invariant in parallel.

To guarantee that class invariants can be verified in a modular way, when a class invariant is broken, a thread is not allowed to obtain any new permissions anymore. In particular, if a thread requires a lock to change any of the fields associated to the invariant, it should obtain this lock before breaking the invariant. This requirement shows that there is close connection between the locking strategy and the functional invariant properties that can be maintained in an application. Further, it is important that with our approach, a thread does not need to have all access permissions that are associated with the invariant, but only the access permissions needed to break the invariant; all other variables are implicitly assumed to be unchanged. Moreover, our technique does allow creating new (helper) threads when an invariant is broken; however, these threads need to be finished and joined before the invariant is reestablished again.

The main contribution of this paper is a sound modular technique for verification of class invariants in multithreaded programs, which:

- is flexible and permissive, because it allows a thread to break an invariant without holding all permissions associated to the invariant property; and
- reveals the connection between locking policy and invariant properties that can be maintained.

The motivation and applicability of our approach is illustrated on several examples. Its implementation as part of the VerCors tool set is under development.

Outline We begin by introducing a short overview of permissions in separation logic, Sec. 2. Next, in Sec. 3 we present the main concepts of our approach, which is further formalised in Sec. 4. Sec. 5 reviews others approaches that tie in with our work. Finally, in Sec. 6 we summarise our work and discuss our future plans.

2 Background

This paper builds on Parkinson's work on separation logic for Java-like programs [21], and its extension by Haack *et al.* [11] for concurrency.

Separation logic [23] is an extension of Hoare Logic [12] for reasoning about separate parts of the heap. The base of this logic is the binary *separating conjunction* operation: $P{*}Q$ describes that P and Q hold for disjoint parts of the heap. O'Hearn shows that separation logic is also convenient for reasoning about multithreaded programs [19]. To allow parallel reads of the same data, basic separation logic is extended with *fractional permissions* [4]. Permission π is a value in the domain $(0, 1]$. At any point in time, a thread holds a number of permissions on locations. If a thread has a write permission for a certain location, i.e., the value 1, it is allowed to change this location. If a thread has a fractional permission, i.e., a fraction less than 1, then it may only read this location. Permissions can be split and combined, to change between read and write permissions. The soundness of this logic ensures that the sum of all threads' permissions for a certain location never exceeds 1, which guarantees data-race freedom. The predicate $\mathsf{Perm}(x.f, \pi)$ indicates that $x.f$ points to a location for which the actual thread has a permission π. Permission expressions are combined with the *separating conjunction* operation.

Parkinson adapts separation logic for object-oriented concepts in a Java-like language [21]. He proposes *abstract predicates* [20] to provide abstraction. Later, Haack *et al.* extended this logic to show how to reason about multithreaded Java-like programs [11] that include reentrant locks and dynamic thread creation. For each lock, a *resource invariant* is specified, i.e., an abstract predicate describing which permissions are stored in the lock. A newly created lock is still fresh and not ready to be acquired. The thread must first execute the commit command on the lock, which transfers the permissions from the thread to the lock and changes the lock's state to initialized. Any thread then may acquire the initialized lock to get the resource invariant (except for reentrant acquiring). Upon final release of the lock, the thread returns the resource invariant back to the lock.

3 Verification Methodology for Class Invariants

This section gives a conceptual understanding of our methodology, presented from two different aspects. First, we discuss how we model the *invariant protocol*, i.e., when an invariant may be assumed, and how it can be broken and reestablished. Then, we describe how our method supports modular verification.

3.1 Class Invariant Protocol

We assume that class invariants express properties over non-static class fields. Thus, a class invariant I defined in a class C is always associated with a particular object v of class C, we write $v.I$. We call the set of locations referred to by an invariant $v.I$ the *footprint of $v.I$*, denoted $\mathsf{fp}(v.I)$ (formally defined in Sec. 4).

Assuming a Class Invariant Our technique should guarantee absence of high-level data races; therefore, it should control access to the invariant's footprint. To provide this control, to every invariant $v.I$, we associate a special abstract predicate $\mathsf{holds}(v.I, 1)$, distributed as a token among the threads. The intuitive meaning of this predicate is the following: when a thread holds a predicate $\mathsf{holds}(v.I, \pi), \pi > 0$, it may assume that the invariant $v.I$ holds; if $\pi = 1$, the running thread may additionally break the invariant. The predicate might be divided among different threads by using the following equivalence:

$$\mathsf{holds}(v.I, \pi) * - * \mathsf{holds}(v.I, \pi/2) * \mathsf{holds}(v.I, \pi/2)$$

This approach guarantees that: 1) a class invariant $v.I$ is stable and all threads that hold a token $\mathsf{holds}(v.I, \pi)$ may rely on $v.I$'s correctness; or 2) *at most one* thread has the token $\mathsf{holds}(v.I, 1)$ and no other thread may assume $v.I$.

Breaking a Class Invariant Inspired by the work of Leino *et al.* [14], we explicitly specify the segment in the program where an invariant property might be violated: for an invariant $v.I$, specification command $\mathsf{unpack}(v.I)$ must be executed at the beginning of such a segment, and $\mathsf{pack}(v.I)$ at its end. The segment between both commands is called an *unpacked segment of v.I.* A special case is object initialisation: the program segment between the end of v's construction and the first execution of the $\mathsf{pack}(v.I)$ command is also $v.I$'s unpacked segment.

The $\mathsf{unpack}(v.I)$ command consumes the token $\mathsf{holds}(v.I, 1)$, and issues a predicate $\mathsf{unpacked}(v.I, 1)$ (*breaking token*). This token serves as a license for the thread to break the invariant $v.I$. Once all updates are done, the running thread must reestablish the validity of $v.I$ and call the $\mathsf{pack}(v.I)$ command, which trades the $\mathsf{unpacked}(v.I, 1)$ token for the $\mathsf{holds}(v.I, 1)$ token. The $\mathsf{unpack}(v.I)$ command is always followed by $\mathsf{pack}(v.I)$ within the same method and executed by the same thread. This thread is called *a holder* of the unpacked segment.

Lst. 1 illustrates the use of an unpacked segment: a class `Point`, represents a point lying on or above the line $y = -x$. Since method `move()` updates the fields x and y to which invariant I refers, these updates must happen within an unpacked segment of I. (The annotation safe at line 7 is discussed next.)

Restrictions to Unpacked Segments We showed how a thread obtains permission to modify an invariant footprint location $p.f$. Once $p.f$ is assigned, we say that $p.f$ is in a *critical state* until the end of the unpacked segment. More precisely:

Definition 1. *(Critical state of a location) Let $v.I$ be an invariant, $p.f$ a location, such that $p.f \in \mathsf{fp}(v.I)$, and let $p.f$ be assigned inside an unpacked segment of $v.I$. Then, any program execution state between the assignment and the end of the unpacked segment is a* critical state for *$p.f$.*

To prevent a thread to observe a broken invariant, a location in a critical state must not be publicly exposed. Therefore, within an unpacked segment we forbid the running thread to release permissions and make them accessible to other threads. Concretely, within an unpacked segment, we allow only safe

```
   class Point {
2    int x; int y;
   //@ invariant I : this.x + this.y >= 0;
4    //...constructors
   //@ requires holds(this.I,1) ∗ Perm(this.x,1) ∗ Perm(this.y,1);
6  //@ ensures holds(this.I,1) ∗ Perm(this.x,1) ∗ Perm(this.y,1);
   /∗@ safe @∗/ void move() {
8                 // the invariant I may now be assumed because of the holds token
   {holds(this.I,1) ∗ Perm(this.x,1) ∗ Perm(this.y,1) ∗ this.I}
10     //@ unpack(this.I);        // trades holds token for unpacked token
   {unpacked(this.I,1) ∗ Perm(this.x,1) ∗ Perm(this.y,1) ∗ this.I}
12       this.x = this.x − 1;      // the invariant I is broken
         this.y = this.y + 1;      // the invariant I can now be reestablished
14 {unpacked(this.I,1) ∗ Perm(this.x,1) ∗ Perm(this.y,1) ∗ this.I}
       //@ pack(this.I);           // trades unpacked token for holds token
16 {holds(this.I,1) ∗ Perm(this.x,1) ∗ Perm(this.y,1)}
   }}
```

Lst. 1. Unpacked segment of a class invariant

commands, i.e., commands that exclude any lock-related operation (acquiring, releasing or committing a lock). This means that all permissions used in the unpacked segment must be obtained before the segment begins. A safe command may call only safe methods, i.e., methods composed of safe commands only. These methods are specified with the optional modifier safe (see Lst. 1, line 7).

We allow forking a safe thread, i.e., threads with a safe $run()$ method, under the condition that the thread must be joined within the unpacked segment. We call these threads *local to the segment*. A safe thread may further fork other safe threads. The breaking token might be shared among all local threads of the unpacked segment, and thus, they might all update different locations of the invariant footprint in parallel. For this purpose, we define the following axiom:

$$\mathsf{unpacked}(v.I, \pi) \ast\!-\!\ast \mathsf{unpacked}(v.I, \pi/2) \ast \mathsf{unpacked}(v.I, \pi/2)$$

Lst. 2 shows a modified version of the `move` method (from Lst. 1) that can not be verified since acquiring/releasing a lock is used within the unpacked segment.

Object Initialisation In our language, object initialisation (the object constructor) is divided into two steps: 1) *object construction* creates an empty object v (all v's fields get a default value), and gives the running thread write permission for each of v's fields and a token $\mathsf{unpacked}(v.I, 1)$ for each invariant $v.I$. 2) the init method follows obligatorily after object construction, where object fields are initialised. Additionally, for every invariant $v.I$, the $\mathsf{pack}(v.I)$ is called by default at the end of the init method. Hence, at the end of v's initialisation, all v's invariants hold, and therefore, v is a valid object.

A verified program with our approach is free of *high-level data races*. This is expressed by the following theorem:

```
Lock lock; // resource invariant: Perm(x, 1) * Perm(y,1);
//@ requires holds(this.I,1);
//@ ensures holds(this.I,1);
void move(){
//@ unpack(this.I); //trades holds token for unpacked (breaking) token
 lock.lock(); //invalid call (permissions to x and y must be gained before unpacking)
 t.fork();      //another thread t may get half of the breaking token to modify x
 updateY();        //for updating y another method is called, which must be safe
 lock.unlock();   // invalid call, must happen after packing
 t.join();           //t is a safe thread, thus joining must be before packing
//@ pack(this.I);
}
```

Lst. 2. Restrictions to unpacked segments

Theorem 1. *(High-level data race freedom) If a value $p.f$ is in a critical state s of an unpacked segment S of an invariant $v.I$, then any thread that is neither holder nor a local thread of S can not access $p.f$.*

Proof. See Sec. 4.4.

As discussed initially, a thread that holds a token $\mathsf{holds}(v.I, \pi), \pi > 0$ may use the invariant $v.I$. This is justified by the following theorem:

Theorem 2. *(Use of a class invariant) An invariant $v.I$ holds in a program state in which the running thread t holds the predicate* $\mathsf{holds}(v.I, \pi)$*,* $\pi > 0$.

Proof. See Sec. 4.4.

Lst. 3 extends the program with the `Point` class (see Lst. 1) to show how a class invariant may be used for verifying a client class. The main thread creates initially a valid `Point` object s for which the invariant $s.I$ holds $(s.x + s.y >= 0)$ and obtains the token $\mathsf{holds}(s.I, 1)$ (lines 3,4). The thread then forks a set of new threads (lines 5-9), passing each of them a reference to s and part of the holds token. Each forked thread has a task to create a sequence of new points at specific locations calculated from the location of s (line 21). To prove that each new `Point` p is a valid object $(p.x + p.y >= 0)$ (line 24), each thread uses the class invariant $s.I$, which is guaranteed by the token $\mathsf{holds}(s.I, \pi)$.

To conclude, we summarise the rules that define the invariant protocol:

R1 (*Assuming*) A thread t may assume (use) a class invariant $v.I$ if t holds the predicate $\mathsf{holds}(v.I, \pi), \pi > 0$.
R2 (*Breaking*) A thread t may write on a location $p.f$ if apart from holding a write permission to $p.f$, it holds a breaking token $\mathsf{unpacked}(v.I, \pi), \pi > 0$ for each invariant $v.I$ that refers to $p.f$, *i.e.*, $p.f \in \mathsf{fp}(v.I)$.
R3 (*Reestablishing*) An invariant $v.I$ must have been reestablished when $\mathsf{pack}(v.I)$ is executed.
R4 (*Exchanging tokens*) The token $\mathsf{unpacked}(v.I, 1)$ is produced at v's construction; commands $\mathsf{unpack}(v.I)$ and $\mathsf{pack}(v.I)$ exchange the $\mathsf{holds}(v.I, 1)$ token for the $\mathsf{unpacked}(v.I, 1)$ token, and vice versa.

```
class DrawPoints {
 void create(){
  Point s = new Point (0, 0);
  //holds(s.I,1) is produced
 for (int k = 1; k<=10; k++){
   Task t = new Task(s, k);
   //each t gets part of holds token
   t.fork();
  }
  //join Task threads
 } }
```

```
class Task {
  Point s; int k;
//... constructors
//@ requires holds(s.I. π) ∗ ... ;
//@ ensures holds(s.I. π) ∗ ... ;
 void run(){
  for (int i = 1; i < 10; i ++) {
  int x = s.x+i; int y = s.y+ki;
//s.I holds (because of the holds token)
//use s.I to validate p.I
 Point p = new Point(x, y);
 draw(p);
} } }
```

Lst. 3. Using a class invariant for verifying a client class

3.2 Modular Verification

As a second step, we discuss the additional properties needed to support modular verification. In the prestate of the assignment to a location $p.f$, rule **R2** requires a breaking token for all invariants that refer to $p.f$. However, in the context (class) where the assignment happens, not all invariants in the program are known. To support modularity, the breaking token is only explicitly checked for the invariants of the object p. Additionally, it is guaranteed that this token is implicitly held for all other invariants. We use Müller's *ownership type system* [7], which is strongly connected to modular verification of invariants [18, 3, 16, 8].

Ownership-Based Types The ownership type system organises the objects in the heap in an *ownership tree*, where each object has one *owner* (either the root of the tree, or another object in the heap). We say that each ancestor of an object p in the tree is p's *transitive owner*. The position of the object p in the tree is determined on p's creation, with an attached required modifier from the set $\{\mathsf{rep}, \mathsf{peer}, \mathsf{rd}\}$ where: peer indicates that r has the same owner as the object this; rep specifies that r is owned by this, and rd(readonly) is any other relation. Additionally, the self modifier is used for references that point to the this object. An array a of object references has an additional modifier to define the relation of each element $a[i]$ with the this reference (see Lst. 4, line 2). When an object changes its context, for example, via transfer as a method parameter, the type of the new reference is determined by applying the *viewpoint adaptation* function ($\triangleright$), defined as:

$$r_1 \triangleright r_2 = \begin{cases} r_2 & \text{if } r_1 = \mathsf{self} \\ \mathsf{rep} & \text{if } r_1 = \mathsf{rep},\ r_2 = \mathsf{peer} \\ \mathsf{peer} & \text{if } r_1 = r_2 = \mathsf{peer} \\ \mathsf{rd} & \text{otherwise} \end{cases}$$

For example, if the this reference owns r, while r owns x, the type of the reference $r.x$ in the context of this is $\mathsf{rep} \rhd \mathsf{rep} = \mathsf{rd}$.

Additionally, the following discipline is imposed in the program: writing to a field $p.f$ or a call to a *non-pure* method (i.e. with side-effects) with a receiver p is forbidden when p has a modifier rd. In this way, each object controls all updates that happen in its transitively owned objects. This guarantees the following:

RO If a field $p.f$ is modified in a method m, for each transitive owner o of p, the call stack contains a method invocation where o is a receiver.

We require that all class invariants in the program are *ownership admissible*:

Definition 2. *A class invariant $v.I$ is ownership admissible if it expresses properties over fields $p_1.p_2...p_n.f$, where $n \geq 1$, $v == p_1$ and p_i is a rep field in the class of p_{i-1} $(i = 2..n)$.*

Verification Technique via Ownership Types Based on Def. 2, we observe the following: for a location $p.f$, an invariant $v.I$ may refer to $p.f$ only if $v == p$ or v is a transitive owner of p. Our verification technique suggests that before assigning to a location $p.f$, it is enough to require a breaking token only for the invariants of the object p $(p.I)$ that refer to $p.f$. If an invariant $v.I$, where v is a transitive owner of p, refers to $p.f$, then the rule **RO** ensures that assignment of $p.f$ is preceded by a method call where v is a receiver. To support modular verification, the check that the actual thread holds a breaking token for $v.I$ should therefore be a requirement of the method call where object v is a receiver. More precisely, we replace the rule **R2** listed above with the following two rules:

R2' A precondition for assigning a field $p.f$ requires a token $\mathsf{unpacked}(p.I, \pi)$ $(\pi > 0)$ for each invariant I of the object p that refers to $p.f$.
R2" A precondition for invoking a method m that assigns a field $p.f$ requires the token $\mathsf{unpacked}(\mathsf{this}.I, \pi)(\pi > 0)$ for each invariant I of the this object that refers to $p.f$.

To establish **R2"**, the contract of the called method m should provide information to the caller about the locations it assigns to. In permission-based separation logic, assigning to a location $p.f$ in m requires a write permission $\pi = 1$ for $p.f$. The caller can identify the locations assignable by m from the precondition formula $\mathsf{Pre_m}$: this is the set of locations for which $\mathsf{Pre_m}$ requires a write permission, denoted $\mathsf{wrt}(\mathsf{Pre_m})$ (see formal definition in Sec. 4). However, π might also be obtained by acquiring a lock during the execution of m. We ensure that this scenario is not possible. In particular, if a location $p.f$ is in the footprint of an invariant $v.I$, $p.f$ should not be protected by a lock object that is transitively owned by v, because this would mean that other threads might observe a broken invariant (see the example below). This restriction is imposed by the following rule (the formall definition of the functions used is presented later, see Sec. 4):

RL $\forall I \in \mathsf{inv}(C);\ \forall f \in \mathsf{relFld}(C);\ \mathsf{fld}(I) \cap \mathsf{fldResInv}(\mathsf{classOf}(f)) = \emptyset$

```
   class PointsSet {
2    rep rep Point[] points = new rep rep Point[100];
    //@ Invariant I₁:  (∀int i: 0 <=i<100) (points[i].x <= 10) ∗ (points[i].y <= 10);
4   //@ requires holds(this.I₁, 1) ∗ Perm(points[i].x, 1) ∗ Perm(points[i].y, 1)
    //@ ensures holds(this.I₁, 1) ∗ Perm(points[i].x, 1) ∗ Perm(points[i].y, 1)
6   void moveAt(int i) {
      //@ unpack(this.I₁);  // trades the holds token for unpacked token
8     if (points[i].y <= 9) {
   //required unpacked token for I₁ (as points[i].x, points[i].y ∈ wrt(Pre_move)∩fp(I₁))
10        points[i].move(); }
      //@ pack(this.I₁);  // trades the unpacked token for holds token
12  } }
```

Lst. 4. Modular verification

The rule is translated as: for any invariant I defined in a class C, and a field f *relevant* to C, the set of fields that appear in I is disjoint from the set of fields that appear in the resource invariant definition in the class of f. A field f is *relevant* to a class C if it may be expressed as a $p_1.p_2.,...p_n.f$, where p_1 is a rep field defined in C, and p_i is a rep or peer field in the class of $p_{i-1}, i = 2..n, n >= 1$.

In Lst. 4, we extend our program (from Lst. 1) to illustrate modular verification. Class `PointsSet` represents a set of points that lie within a predefined area. When calling the method `move()`(line 10), the caller provides a breaking token for its own invariants that `move()` might break (in this case invariant I_1). After the call to `move()`, invariant I_1 is reestablished (line 11), even though the actual thread has permissions to the ith array element only; our approach ensures that the other locations in fp(I_1) are stable until the end of the unpacked segment.

Fields x and y from class `Point` are relevant to the `PointsSet` class and used in I_1; hence, Rule **RL** forbids a lock that protects x and/or y to be transitively owned by a `PointsSet` object. This is necessary: if permissions to x and y could be obtained by a lock in `Point`, other threads might observe that I_1 is broken. To avoid this, the lock would have to be already acquired before the unpacked segments for I_1, but this would violate modularity. The example shows that the invariants that can be maintained strongly depend on the locking strategy used.

4 Formalisation

We formalise our approach using a Java-like concurrent language. The formalisation is mainly inspired by Haack *et al.* [11]. We concentrate on those points that are relevant for class invariants. For other concepts, e.g., those associated to locks, we only provide some basic intuition to make the paper self-contained.

4.1 Language

Fig. 1 shows the grammar of our language. With $\overline{x}$ we define sequences of x, while x? represents an optional x. A class is composed of fields, methods, predi-

$cl \in \mathsf{Class}$	$::=$	$\mathsf{class}\ C\ \{fd*\ \ md*\ \ inv*\ \ pd*\}$
$fd \in \mathsf{Field}$	$::=$	$T f$
$md \in \mathsf{Method}$	$::=$	$spec\ T\ m(\overline{V}\ \overline{x})\{c\}$
$spec \in \mathsf{MethSpec}$	$::=$	$\mathsf{requires}\ F\ \mathsf{ensures}\ F\ \mathsf{pure?}\ \mathsf{safe?}$
$pd \in \mathsf{Predicate}$	$::=$	$\mathsf{pred}\ P = F_{\mathsf{res}}(P \neq res\_inv) \mid \mathsf{pred}\ res\_inv = F_{\mathsf{res}}$
$inv \in \mathsf{Invariant}$	$::=$	$\mathsf{Invariant}\ I : F_{\mathsf{inv}}$
$c \in \mathsf{Command}$	$::=$	v (return value or null in case of type void)
		$\mid T\ x; c \mid x = v; c \mid x = op(\overline{v}); c \mid x = v.f; c$
		$\mid x = \mathsf{new}\ \mathsf{rtype}\ C; c \mid (x = v.m(\overline{v}); c \mid \mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ c; c$
		$\mid v.f = v; c \mid v.\mathsf{lock}(); c \mid v.\mathsf{commit}(); c \mid v.\mathsf{unlock}(); c$
		$\mid v.\mathsf{fork}(); c \mid v.\mathsf{join}(); c \mid \mathsf{unpack}(v.I); c \mid \mathsf{pack}(v.I); c$
$F \in \mathsf{Formula}$	$::=$	$e \mid \mathsf{Perm}(v.f, \pi) \mid \pi.P \mid F \oplus F \mid (qt\ T\ \alpha)F$
		$\mid \mathsf{holds}(v.I, \pi) \mid \mathsf{unpacked}(v.I, \pi) \mid e.\mathsf{fresh}() \mid e.\mathsf{initialized}()$
$F_{\mathsf{res}} \in \mathsf{Formula}_{\mathsf{res}}$	$::=$	$e \mid \mathsf{Perm}(v.f, \pi) \mid \pi.P \mid F_{\mathsf{res}} \oplus F_{\mathsf{res}} \mid (qt\ T\ \alpha)(F_{\mathsf{res}}) \mid \mathsf{holds}(v.I, \pi)$
$F_{\mathsf{inv}} \in \mathsf{Formula}_{\mathsf{inv}}$	$::=$	$e_{inv} \mid (qt\ T\ \alpha)(F_{\mathsf{inv}}) \mid F_{\mathsf{inv}} \oplus F_{\mathsf{inv}}$
$e \in \mathsf{Exp}$	$::=$	$\pi \mid v.f \mid v \mid op(\overline{e})$
$e_{inv} \in \mathsf{Exp}_{\mathsf{inv}}$	$::=$	$v_1.v_2 ... v_n.f \mid op(\overline{e_{inv}})$
$T, U, V \in \mathsf{Type}$	$::=$	$\mathsf{void} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{perm} \mid (\mathsf{rtype},\ C)$
$rtype \in \mathsf{RefType}$	$::=$	$\mathsf{rep} \mid \mathsf{peer} \mid \mathsf{self} \mid \mathsf{rd}$
$\pi \in \mathsf{SpecVal}$	$::=$	$\alpha \mid v \mid 1 \mid \mathsf{split}(\pi)$ (1/2 of a fractional permission π)
$u, v, w \in \mathsf{Val}$	$::=$	$\mathsf{null} \mid n \mid b \mid o \mid x$

$\oplus \in \{*, \wedge, \vee\}$ $\quad op \in \mathsf{Op} \supseteq \{==, !, \wedge, \vee, \Rightarrow\}$ $\quad qt \in \{\exists, \forall\}$
$n \in \mathsf{int}$ $\quad b \in \{\mathsf{true}, \mathsf{false}\}$ $\quad x, y, z \in \mathsf{Variables}$ $\quad o, p \in \mathsf{ObjectId}$

Fig. 1. Language Syntax

cates, and class invariants. The special predicate res_inv is associated to a lock object, and is used to describe the resources that the lock protects. Methods may be declared as pure and/or safe, as explained below. The set of commands is extended with the specification commands $\mathsf{pack}(v.I)$ and $\mathsf{unpack}(v.I)$.

Specification Formulas We distinguish three types of specification formulas: i) *Standard formulas* F, expressed in permission-based separation logic and used to specify methods. Predicates holds and unpacked, and fresh and initialized are special tokens that describe the state of a class invariant or a lock, respectively. ii) *Resource invariant formulas* F_{res}, used to express the res_inv predicate. They are more restrictive than F: F_{res} must not use the special tokens unpacked, fresh and initialized.

iii) *State formulas* F_{inv}, first-order logic formulas, used to specify class invariants and describe properties over shared memory locations only. Thus, their syntax does not include the predicate $\mathsf{Perm}(v.f, \pi)$ or any of the special tokens.

Invariant Footprint We define the invariant footprint $\mathsf{fp}(v.I)$ by induction of the structure of $v.I$:

$$\mathsf{fp}(v_1.v_2..v_n.f) = \{v_1, v_1.v_2, ..., v_1...v_n.f\} \quad \mathsf{fp}(op(\overline{e_{inv}})) = \bigcup_{e \in \overline{e_{inv}}} \mathsf{fp}(e)$$
$$\mathsf{fp}(F_{\mathsf{inv}_1} \oplus F_{\mathsf{inv}_2}) = \mathsf{fp}(F_{\mathsf{inv}_1}) \cup \mathsf{fp}(F_{\mathsf{inv}_2}) \quad \mathsf{fp}((qt\ \alpha\, T)(F_{\mathsf{inv}})) = \bigcup_{v \in T \setminus \{\alpha\}} \mathsf{fp}(F_{\mathsf{inv}}[v/\alpha])$$

$$(\mathsf{new})\ \frac{\Gamma \vdash l : \mathsf{rtype}\ C \quad \mathsf{rtype} \in \{\mathsf{rep}, \mathsf{peer}, \mathsf{rd}\}}{\Gamma \vdash \mathsf{wf}(l = \mathsf{new}\ \mathsf{rtype}\ C)} \qquad (\mathsf{get})\ \frac{\Gamma \vdash v, f, l : T, U, U \quad f \in \mathsf{fld}(T^2)}{\Gamma \vdash \mathsf{wf}(l = v.f)}$$

$$(\mathsf{meth\_call})\ \frac{\begin{array}{c}\mathsf{md} ::= \mathsf{requires}\ F\ \mathsf{ensures}\ G\ \mathsf{safe}?\ T\ m(\bar{V}\ \bar{\imath})\{c\}\\ \Gamma \vdash u, \bar{w}, l : U, \bar{W}, T \quad U^1 \in \{\mathsf{peer}, \mathsf{rep}, \mathsf{self}\}\end{array}}{\Gamma \vdash \mathsf{wf}(l = u.m(\bar{w}))}$$

$$(\mathsf{meth\_call\_pure})\ \frac{\begin{array}{c}\mathsf{md} ::= \mathsf{requires}\ F\ \mathsf{ensures}\ G\ \mathsf{pure}\ \mathsf{safe}?\ T\ m(\bar{V}\ \bar{\imath})\{c\}\\ \Gamma \vdash u, \bar{w}, l : U, \bar{W}, T \quad U^1 \in \{\mathsf{peer}, \mathsf{rep}, \mathsf{self}, \mathsf{rd}\}\end{array}}{\Gamma \vdash \mathsf{wf}(l = u.m(\bar{w}))}$$

$$(\mathsf{set})\ \frac{\Gamma \vdash v, u, f : T, U, U \quad f \in \mathsf{fld}(T^2) \quad T^1 \in \{\mathsf{peer}, \mathsf{rep}, \mathsf{self}\}}{\Gamma \vdash \mathsf{wf}(v.f = u)}$$

$$(\mathsf{class})\ \frac{\begin{array}{c}\Gamma \vdash \mathsf{wf}(\mathsf{fd}*, \mathsf{md}*, \mathsf{pd}*, \mathsf{inv}*);\ \forall(\mathsf{Invariant}\ I) \in \mathsf{inv}*; \forall(Tf) \in \mathsf{relFld}(C);\\ \mathsf{fldResInv}(T^2) \cap \mathsf{fld}(I) = \emptyset\end{array}}{\Gamma \vdash \mathsf{wf}(\mathsf{cl} ::= \mathsf{class}\ C\ \mathsf{fd}*, \mathsf{md}*, \mathsf{pd}*, \mathsf{inv}*)}$$

$$(\mathsf{inv\_exp})\ \frac{\Gamma \vdash e_i : T_i (i = 1..n)\ T_i^1 \in \{\mathsf{self}, \mathsf{rep}\}}{\Gamma \vdash \mathsf{wf}(e_{\mathsf{inv}}) ::= e_1, e_2, .., e_n.f)}$$

Fig. 2. Type rules

Types A type of an object reference in our language is represented as a tuple $T = (\mathsf{rtype}, C)$. The first component, T^1, is a type modifier from the set $\mathsf{RefType} = \{\mathsf{rep}, \mathsf{peer}, \mathsf{self}, \mathsf{rd}\}$, while the second, T^2, represents the object's class. Consequently, two references pointing to the same object might have different reference types if they are in a different context. Fig. 4.1 shows the typing rules that represent constraints imposed by the ownership type system:

We use a function $\mathsf{df} : \mathsf{Type} \mapsto \mathsf{Val}$, which maps each type to a corresponding default value:

$$\mathsf{df}(\mathsf{rtype}\ C) \triangleq \mathsf{null} \qquad \mathsf{df}(\mathsf{bool}) \triangleq \mathsf{false} \qquad \mathsf{df}(\mathsf{int}) \triangleq 0 \qquad \mathsf{df}(\mathsf{void}) \triangleq \mathsf{null}$$

Fields and Invariants in a Class We define functions $\mathsf{fld}(C)$, $\mathsf{relFld}(C)$ and $\mathsf{inv}(C)$ to represent respectively a set of fields defined in a class C, the set of fields *relevant* to C, and the set of invariants in C. For a class C with a definition $\mathsf{class}\ C\{\ \overline{T}\ \overline{f}\ \mathsf{md}*\ \mathsf{pd}*\ \overline{\mathsf{Invariant}}\ \overline{I}\}$, we have:

$$\begin{array}{l}\mathsf{fld}(C) = (\overline{T}\ \overline{f})\\ \mathsf{relFld}(C) = \bigcup_{Tf | Tf \in \overline{Tf}, T^1 = \mathsf{rep}} \mathsf{relAux}(T^2) \text{ where}\\ \mathsf{relAux}(C) = \bigcup_{Tf | Tf \in \overline{Tf}, T^1 \in \{\mathsf{self}, \mathsf{rep}, \mathsf{peer}\}} (Tf) \cup \mathsf{relFld}(T^2)\\ \mathsf{inv}(C) = (\overline{\mathsf{Invariant}}\ \overline{I})\end{array}$$

Fields in a Resource Invariant Furthermore, a function $\mathsf{fldResInv}(C)$ is defined to represent the fields used in the special resource invariant predicate *res_inv* defined in a class C. The $\mathsf{fldResInv}(C)$ function is defined by induction of the structure of the *res_inv* formula:

$\mathsf{class}\ C\{\ \mathsf{fd}{*}\ \mathsf{md}{*}\ \mathsf{pred}\ res\_inv = F_{\mathsf{res}};\ \mathsf{pd}{*}\ \mathsf{inv}{*}\} \quad \Rightarrow \quad \mathsf{fldResInv}(C) = \mathsf{fld}(F_{\mathsf{res}})$

$$\mathsf{fld}(F_{\mathsf{res}}) = \begin{cases} \emptyset & \text{if } F_{\mathsf{res}} \in \{e, \mathsf{holds}(v.I, \pi)\} \\ (T\ f) & \text{if } F_{\mathsf{res}} ::= \mathsf{Perm}(e.f, \pi), f : T \\ \mathsf{fld}(F'_{\mathsf{res}}) & \text{if } F_{\mathsf{res}} ::= \pi.P(\mathsf{pred}\ P = F'_{\mathsf{res}}) \\ \mathsf{fld}(F_{\mathsf{res}_1}) \cup \mathsf{fld}(F_{\mathsf{res}_2}) & \text{if } F_{\mathsf{res}} ::= F_{\mathsf{res}_1} \oplus F_{\mathsf{res}_2} \\ \mathsf{fld}(F_{\mathsf{res}_1}) & \text{if } F_{\mathsf{res}} ::= (qt\ \alpha\, T)(F_{\mathsf{res}_1}) \end{cases}$$

Fields in a Class Invariant In a similar way, $\mathsf{fld}(I)$ function is defined by induction of the structure of a *State Formula*, F_{inv}: it represents the fields used in the definition of the invariant I.

$\mathsf{Invariant}\ I : F_{\mathsf{inv}} \quad \Rightarrow \quad \mathsf{fld}(I) = \mathsf{fld}(F_{\mathsf{inv}})$

$$\mathsf{fld}(F_{\mathsf{inv}}) = \begin{cases} (\overline{T_i\ e_i}) \cup (U\ f) & \text{if } F_{\mathsf{inv}} ::= e_1.e_2...e_n.f\ (e_i : T_i, i = 1..n; f : U) \\ \bigcup_{e_{\mathsf{inv}} \in \overline{e_{\mathsf{inv}}}} \mathsf{fld}(e_{\mathsf{inv}}) & \text{if } F_{\mathsf{inv}} ::= op(\overline{e_{\mathsf{inv}}}) \\ \mathsf{fld}(F_{\mathsf{inv}_1}) \cup \mathsf{fld}(F_{\mathsf{inv}_2}) & \text{if } F_{\mathsf{inv}} ::= F_{\mathsf{inv}_1} \oplus F_{\mathsf{inv}_2} \\ \mathsf{fld}(F_{\mathsf{inv}_1}) & \text{if } F_{\mathsf{inv}} ::= (qt\ \alpha\, T)(F_{\mathsf{inv}_1}) \end{cases}$$

Writable Locations Above, in Sec. 3.2, we introduced the $\mathsf{wrt}(F)$ function: it returns the set of locations for which the formula F expresses a write permission. Below, we show the formal definition of $\mathsf{wrt}(F)$. The auxiliary function $\mathsf{permL}(F,\ e.f)$ returns the value of the permission contained in F for the location $e.f$, while $\mathsf{locs}(F)$ returns the set of all locations in F.

$\mathsf{wrt}(F) = \{e.f \mid e.f \in \mathsf{locs}(F), \mathsf{permL}(F,\ e.f) \geq 1\}$

$$\frac{F \in \{e_1, \mathsf{holds}(v.I, \pi), \mathsf{unpacked}(v.I, \pi), e_1.\mathsf{fresh}(), e_1.\mathsf{initialized}()\}}{\mathsf{permL}(F,\ e.f) = 0}$$

$$\frac{F = \mathsf{Perm}(e.f, \pi)}{\mathsf{permL}(F,\ e_1.f_1) = \begin{cases} \pi, & \text{if } e_1.f_1 == e.f \\ 0, & \text{if } e_1.f_1 \neq e.f \end{cases}}$$

$$\frac{F = F_1 \oplus F_2}{\mathsf{permL}(F,\ e.f) = \begin{cases} \mathsf{permL}(F_1,\ e.f) + \mathsf{permL}(F_2,\ e.f), & \text{if } \oplus == * \\ \mathsf{max}(\mathsf{permL}(F_1,\ e.f), \mathsf{permL}(F_2,\ e.f)), & \text{if } \oplus == \wedge \\ \mathsf{min}(\mathsf{permL}(F_1,\ e.f), \mathsf{permL}(F_2,\ e.f)), & \text{if } \oplus == \vee \end{cases}}$$

$$\frac{F = (qt\ \alpha\, T)(F)}{\mathsf{permL}(F,\ e.f) = \mathsf{permL}(\bigwedge_{v \in T \setminus \{\alpha\}} F[v/\alpha],\ e.f)} \qquad \frac{F = \pi.P\ (\mathsf{pred}\ P = F')}{\mathsf{permL}(F,\ e.f) = \mathsf{permL}(F',\ e.f)}$$

$$\mathsf{locs}(F) = \begin{cases} \emptyset & \text{if } F \in \{e_1, \mathsf{holds}(v.I, \pi), \mathsf{unpacked}(v.I, \pi), \\ & \qquad e_1.\mathsf{fresh}(), e_1.\mathsf{initialized}()\} \\ e.f & \text{if } F ::= \mathsf{Perm}(e.f, \pi) \\ \mathsf{locs}(F_1) \cup \mathsf{locs}(F_2) & \text{if } F ::= F_1 \oplus F_2 \\ \bigcup_{v \in T \setminus \{\alpha\}} \mathsf{locs}(F[v/\alpha]) & \text{if } F ::= (qt\ \alpha\, T)(F) \\ \mathsf{locs}(\pi.F') & \text{if } F ::= \pi.P\ (\mathsf{pred}\ P = F') \end{cases}$$

Safe and Pure Commands Above, we introduced the notion of safe commands. For a safe command c the predicate $\mathsf{safe}(c, V)$ holds, where V is a set that keeps track of all identifiers of threads that are forked and expected to be joined. The V parameter is used to capture that threads forked within a safe command c, must also be joined within c. For a method m defined as $\mathsf{safe}\ T\ m(\overline{V}\ \overline{i})\ \{c\}$, the relation $\mathsf{safe}(m)$ holds iff $\mathsf{safe}(c, [])$ holds. A safe method is annotated with the optional modifier safe. We define inductively the set of safe commands:

$$\begin{array}{ll}
\mathsf{safe}(v,\ V) & \Leftrightarrow \mathsf{true}\\
\mathsf{safe}(c,\ V) & \Leftrightarrow \mathsf{false},\ \ \text{if } c \in \{v.\mathsf{lock}(), v.\mathsf{unlock}(), v.\mathsf{commit}()\}\\
\mathsf{safe}(c; c_1,\ V) & \Leftrightarrow \mathsf{safe}(c_1,\ V),\ \ \text{if } c \in \{T\ x,\ x = v,\ x = v.f,\ v.f = v,\\
 & \quad x = op(\overline{v}),\ \mathsf{new\ rtype}\ C,\ \mathsf{unpack}(v.I),\ \mathsf{pack}(v.I)\}\\
\mathsf{safe}(x = v.m(\overline{v}); c,\ V) & \Leftrightarrow \mathsf{safe}(m) \wedge \mathsf{safe}(c,\ V)\\
\mathsf{safe}(v.\mathsf{fork}(); c,\ V) & \Leftrightarrow \mathsf{safe}(c,\ V \cup \{v\})\\
\mathsf{safe}(v.\mathsf{join}(); c,\ V) & \Leftrightarrow \mathsf{safe}(c,\ V \setminus \{v\})\\
\mathsf{safe}(\mathsf{if}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c,\ V) & \Leftrightarrow \mathsf{safe}(c_1,\ []) \wedge \mathsf{safe}(c_2,\ []) \wedge \mathsf{safe}(c,\ V)
\end{array}$$

Our method uses also the notion of pure commands, i.e., commands that do not make any changes to the shared state. Pure methods are composed of pure commands and specified with the optional modifier pure. Below, we define inductively the set of pure commands:

$$\begin{array}{lll}
\mathsf{pure}(v) & \Leftrightarrow \mathsf{true} & \\
\mathsf{pure}(c) & \Leftrightarrow \mathsf{false}, & \text{if } c \in \{v.\mathsf{lock}(), v.\mathsf{unlock}(), v.\mathsf{commit}()\\
 & & \quad v.\mathsf{fork}(), v.\mathsf{join}(),\ v.f = v,\ \mathsf{new\ rtype}\ C\}\\
\mathsf{pure}(c; c_1) & \Leftrightarrow \mathsf{pure}(c_1), & \text{if } c \in \{T\ x,\ x = v,\ x = v.f,\ x = op(\overline{v}),\\
 & & \quad \mathsf{unpack}(v.I),\ \mathsf{pack}(v.I)\}\\
\mathsf{pure}(\mathsf{if}\ v\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2; c) & \Leftrightarrow \mathsf{pure}(c_1) \wedge \mathsf{pure}(c_2) \wedge \mathsf{pure}(c) & \\
\mathsf{pure}(x = v.m(\overline{v}); c) & \Leftrightarrow \mathsf{pure}(m) \wedge \mathsf{pure}(c) & \\
\mathsf{pure}(m), (\mathsf{pure}\ T\ m(\overline{V}\ \overline{i})\ \{c\}) & \Leftrightarrow \mathsf{pure}(c) &
\end{array}$$

4.2 Hoare Triples

Fig. 3 shows the Hoare triples relevant to our approach (for the complete list of rules see [11]). We use: $\circledast_i F_i$ to abbreviate a separation conjunction of all formulas F_i; $\mathsf{PointsTo}(v.f, \pi, w)$ to abbreviate $\mathsf{Perm}(v.f, \pi) \wedge v.f == w$; functions $\mathsf{fld}(C)$ and $\mathsf{inv}(C)$ to represent respectively the set of fields and invariants in the class C; $\mathsf{df}(T)$ for the default value of type T; $\mathsf{wrt}(F)$ for the set of locations for which F expresses a write permission.

The rule (New) shows that construction of object v produces an unpacked token for each invariant of v, and a write permission for each field of v. Rules (Set) and (MethCall) encode **R2'** and **R2"** (see Sec. 3.2); they ensure that the breaking token is a condition for breaking the invariant $v.I$. Rules (Pack) and (Unpack) describe the invariant protocol and encode **R3** and **R4** (see Sec. 3.1). Finally, the rule (RuleInv) shows that the token $\mathsf{holds}(v.I, \pi)$ provides the actual thread the right to use the invariant $v.I$ (as justified by Theorem 2 in Sec 3.1).

(New)
$$\begin{array}{c}\{\mathsf{true}\}\\ v = \mathsf{new\ rtype}\ C\\ \{\circledast_{T f \in \mathsf{fld}(C)} \mathsf{PointsTo}(v.f, 1, \mathsf{df}(T^1)) * \circledast_{I \in \mathsf{inv}(C)} \mathsf{unpacked}(v.I, 1)\}\end{array}$$

(Set)
$$\frac{v : V}{\begin{array}{c}\{v \neq \mathsf{null} * \mathsf{PointsTo}(v.f, 1, u) * \circledast_{I \in \mathsf{inv}(V^2), v.f \in \mathsf{fp}(v.I)} \mathsf{unpacked}(v.I, \pi)\}\\ v.f = w;\\ \{\mathsf{PointsTo}(v.f, 1, w) * \circledast_{I \in \mathsf{inv}(V^2), v.f \in \mathsf{fp}(v.I)} \mathsf{unpacked}(v.I, \pi)\}\end{array}}$$

(MethCall)
$$\frac{\mathsf{md} ::= \mathsf{requires}\ F\ \mathsf{ensures}\ F'\ \mathsf{safe?}\ \mathsf{pure?}\ T\ m(\overline{U}\ \overline{u})\{c\} \quad \mathsf{this} : V}{\begin{array}{c}\{u \neq \mathsf{null} * F * \circledast_{I \in \mathsf{inv}(V^2), \mathsf{wrt}(F) \cap \mathsf{fp}(\mathsf{this}.I) \neq \emptyset} \mathsf{unpacked}(\mathsf{this}.I, \pi)\}\\ x = u.m(\overline{i})\\ \{\exists\ T\alpha)(\alpha == x * F') * \circledast_{I \in \mathsf{inv}(V^2), \mathsf{wrt}(F) \cap \mathsf{fp}(\mathsf{this}.I) \neq \emptyset} \mathsf{unpacked}(\mathsf{this}.I, \pi)\}\end{array}}$$

(Unpack)
$$\{\mathsf{holds}(v.I, 1)\}\ \mathsf{unpack}(v.I)\{\mathsf{unpacked}(v.I, 1) * v.I\}$$

(Pack)
$$\{\mathsf{unpacked}(v.I, 1) * v.I\}\ \mathsf{pack}(v.I)\ \{\mathsf{holds}(v.I, 1)\}$$

(RuleInv)
$$\frac{\{\mathsf{holds}(v.I, \pi) * v.I\}\ c\ \{F\}}{\{\mathsf{holds}(v.I, \pi)\}\ c\ \{F\}}$$

Fig. 3. Hoare triples

4.3 Semantics

We define a program state as: $st \in \mathsf{State} = \mathsf{Heap} \times \mathsf{ThreadPool} \times \mathsf{LockTable}$. A Heap models the shared memory: $h \in \mathsf{Heap} = \mathsf{ObjId} \mapsto \mathsf{Type} \times (\mathsf{FieldId} \mapsto \mathsf{Value})$. The ThreadPool component describes all threads that operate on the heap: $ts \in \mathsf{ThreadPool} = \mathsf{ObjId} \mapsto \mathsf{Thread}$, where each thread contains its own local memory and a command to execute, $t \in \mathsf{Thread} = \mathsf{Stack} \times \mathsf{Cmd}$. The LockTable expresses for every lock whether it is free, or it is acquired by a thread a certain number of times: $l \in \mathsf{LockTable} = \mathsf{ObjId} \mapsto \mathsf{free} \uplus (\mathsf{ObjId} \times \mathbb{N})$. Operationally, the two specification commands $\mathsf{unpack}(v.I)$ and $\mathsf{pack}(v.I)$ are no operations. The small-step operational semantics of the other commands is standard, see [11].

Semantics of Formulas The specification formulas are interpreted using the semantics relation $\Gamma \vdash \mathcal{E}, \mathcal{R}, s \models F$, which expresses validity of the formula F in a type environment Γ, a predicate environment $\mathcal{E}$ and a stack s, given a resource $\mathcal{R}$. Type environment Γ is a partial function of type $\mathsf{ObjId} \cup \mathsf{Var} \mapsto \mathsf{Type}$ that maps each object or variable to its type, while $\mathcal{E}$ maps each predicate symbol to an appropriate relation that represents its definition. For details see [11].

The resource $\mathcal{R}$ is an abstraction of a program state represented by an 8-tuple, $\mathcal{R} = (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{U}, \mathcal{T})$, where each component describes part of the state: i) h represents the heap: $\mathsf{ObjId} \mapsto \mathsf{Type} \times (\mathsf{FieldId} \mapsto \mathsf{Val})$ ii) $\mathcal{P}$ is a permission table that stores permissions to object fields from the heap ($\mathsf{ObjId} \times \mathsf{FieldId} \mapsto [0, 1]$); iii) $\mathcal{J}$ is a join table ($\mathsf{ObjId} \mapsto [0, 1]$), where $\mathcal{J}(t)$ represents how much of the postcondition of a thread t is given to other forked threads; iv) $\mathcal{L}$ is an abstraction of the lock table, which maps each thread to the set of locks that it holds; v) $\mathcal{F}$ keeps a set of fresh locks; vi) $\mathcal{I}$ keeps a set of initialized locks; vii) $\mathcal{U}$ keeps the parts of the unpacked tokens for each invariant; and analogously viii) $\mathcal{T}$ keeps the holds tokens. Both components $\mathcal{U}$ and $\mathcal{T}$ are defined as functions $\mathsf{ObjId} \times \mathsf{InvId} \mapsto [0,1]$.

We define a *compatibility* binary relation (#) and a *resource joining operation* ($*$) over resources. Compatibility ensures that two different threads always observe the abstract state as two compatible resources, $\mathcal{R}\#\mathcal{R}'$: the object fields that are common for the heaps in $\mathcal{R}$ and $\mathcal{R}'$ are mapped to the same value; the sum of permissions for a location in $\mathcal{R}$ and $\mathcal{R}'$, or the sum of the parts of the special tokens (holds and unpacked) for an invariant in both resources never exceeds 1; etc. The intuitive meaning of the operation $\mathcal{R} * \mathcal{R}'$ is joining (summing) both resources. For example, $\mathcal{R} * \mathcal{R}'$ contains all permissions from both resources or all tokens from both resources. The definition of the # and $*$ is component-wise. We give the formal definitions for the structure $(\#, *)$ for the components $\mathcal{U}$ and $\mathcal{T}$, while for the others we refer to [11].

$$\begin{aligned} \mathcal{U}\#\mathcal{U}' &\Leftrightarrow \forall i \in \mathsf{dom}(\mathcal{U}) \cap \mathsf{dom}(\mathcal{U}').\, \mathcal{U}(i) + \mathcal{U}'(i) \leq 1 \quad (\mathcal{U} * \mathcal{U}')(i) = \mathcal{U}(i) + \mathcal{U}'(i) \\ \mathcal{T}\#\mathcal{T}' &\Leftrightarrow \forall i \in \mathsf{dom}(\mathcal{T}) \cap \mathsf{dom}(\mathcal{T}').\, \mathcal{T}(i) + \mathcal{T}'(i) \leq 1 \quad (\mathcal{T} * \mathcal{T}')(i) = \mathcal{T}(i) + \mathcal{T}'(i) \end{aligned}$$

Below we define that the specification formula $\mathsf{holds}(v.I, \pi)$ holds for a resource $\mathcal{R}$ if the part of the holds token for the invariant $v.I$ in $\mathcal{R}$ is at least π. The validity of the $\mathsf{unpacked}(v.I, \pi)$ formula is defined analogously. The semantics of a class invariant $v.I$ is expressed as a validity of the representation formula of $v.I$, i.e., F_{inv}.

$$\begin{aligned} \Gamma \vdash \mathcal{E}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{U}, \mathcal{T}), s \models \mathsf{holds}(v.I, \pi) &\Leftrightarrow \quad \mathcal{T}(v.I) \geq \pi \\ \Gamma \vdash \mathcal{E}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{U}, \mathcal{T}), s \models \mathsf{unpacked}(v.I, \pi) &\Leftrightarrow \quad U(v.I) \geq \pi \\ \Gamma \vdash \mathcal{R} = \mathcal{E}, (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{U}, \mathcal{T}), s \models v.I(I = F_{\mathsf{inv}}) &\Leftrightarrow \Gamma \vdash \mathcal{E}, \mathcal{R}, s \models F_{\mathsf{inv}} \end{aligned}$$

As our language contains *state formulas*, not all locations in the partial heap must be 'framed' by a positive permission (unlike in standard permission-based separation logic). For a sound resource $\mathcal{R} = (h, \mathcal{P}, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{U}, \mathcal{T})$ we require:

$$\begin{aligned} &\forall p \in \mathsf{dom}(h), f \in \mathsf{dom}(h(p)_2), \mathcal{P}(p, f) > 0 \, \vee \\ &\qquad (\exists v.I \in \mathsf{dom}(\mathcal{T}) \; p.f \in \mathsf{fp}(v.I) \wedge (\mathcal{T}(v.I) > 0 \vee \mathcal{U}(v.I) > 0) \end{aligned}$$

The rule states that if a location $p.f$ is not protected by a read permission ($\mathcal{P}(p, f) = 0$), then it must be protected by (a part of) the holds or unpacked token ($\mathcal{T}(v.I) > 0 \vee \mathcal{U}(v.I) > 0$), for an invariant $v.I$ that refers to $p.f$. This ensures that the location $p.f$ is stable and might not be modified by other threads.

4.4 Soundness

Below we present the proof of Theorem 1 and Theorem 2, both discussed in Sec. 3.1.

We use notations $s < s'$ (or $s > s'$) to express that program state s precedes (or follows) program state s'.

Lemma 1. *In a state s, where the running thread is t, we have:*

(i) *if t holds a predicate* holds($v.I, \pi$), $\pi > 0$*, then s is preceded by at least one execution of the command* pack($v.I$).
(ii) *if t holds a predicate* holds($v.I, \pi$), $\pi > 0$*, then s is not an internal state of an unpacked segment of $v.I$.*
(iii) *if t holds a predicate* unpacked($v.I, \pi$), $\pi > 0$*, s is an internal state of an unpacked segment of $v.I$.*

Proof. All statements follow from the defined invariant protocol:

(i) the predicate holds($v.I, \pi$) is produced only by executing the pack($v.I$) command. This ensures that s is preceded by a pack($v.I$) command.
(ii) the predicate holds($v.I, \pi$) is produced for the first time at the end of the object v's initialisation, when the initial unpacked segment of $v.I$ is finished. Afterwards, when an unpacked segment of $v.I$ starts, the predicate holds($v.I, \pi$) is consumed, and might only be obtained back when the unpacked segment finishes. Therefore, in any state s within the unpacked segment of $v.I$, no thread holds a predicate holds($v.I, \pi$).
(iii) The predicate unpacked($v.I, \pi$) is produced in a poststate of an unpack($v.I$) command, or a poststate of object v's construction. In both cases s is an internal state of an unpacked segment of $v.I$. Once the segment finishes (with the execution of the pack($v.I$) command), the unpacked($v.I, 1$) predicate is lost and might not be obtained until a new unpacked segment of $v.I$ starts. Therefore, no thread might hold unpacked($v.I, \pi$) predicate in a state s that is not an internal state of $v.I$'s unpacked segment.

Lemma 2. *If s is a prestate of an assignment $p.f' = w$, where the running thread is t, for any invariant $v.I$ that is dependent on $p.f'$, s is an internal state of an unpacked segment of $v.I$.*

Proof. We will prove that in the state s, there is a thread that holds the predicate unpacked($v.I, \pi$), $\pi > 0$. Then from Lemma 1(iii), we will directly conclude that s is an internal state of an unpacked segment of $v.I$.

From the definition of admissible invariant (Def. 2) (Sec. 3.2), since $v.I$ is dependent on the assigned location $p.f'$, then $p.f'$ is represented as $p_1.p_2...p_n.f$, where $n >= 1$, $v = p_1$ and p_i is owner of $p_{i+1}, i = 1..n-1$. We consider two cases:

(i) $n = 1$, then $p_1 = p = v$. From the Hoare triple (Set), in the prestate of $p.f' = w$, the running thread holds the predicate unpacked($v.I, \pi$), $\pi > 0$.
(ii) $n > 1$, then v is a transitive owner of p. We denote with m the method of execution of the assignment $p.f' = w$. From the rule **RO** (Sec. 3.2), for each p_i, the path to the invocation of m is preceded by a method call for which p_i is a receiver. Denoting m_x as a method with receiver x, we define the path of m's invocation as:

$$m_{p_1} \; ... \; m_{p_2} \; ... \; m_{p_n} ... \; m, \text{ where } n \geqslant 1,$$

where each method m_{p_i} is called by a method with a receiver p_{i-1}. Note that between m_{p_i} and $m_{p_{i+1}}$ there might be other method calls where the receiver object is a peer of p_i.
In the state s (which is an internal state of the method m), the running thread holds a write permission $\pi = 1$ for $p.f$'. Let s_1 be the prestate of the invocation of the method m_{p_2}. We distinguish 3 cases:
(a) the permission π is required in the precondition of the method m_{p_2}. Then, the triple (MethCall) ensures that in s_1 the running thread holds the predicate $\mathsf{unpacked}(p_1.I, 1)$, where $p_1 = v$. This predicate is not lost until the execution of the $\mathsf{pack}(v.I)$ command. Since the syntax ensures that $\mathsf{pack}(v.I)$ happens after the method m_{p_2} is finished, then, in the state s there is still a thread that holds a predicate $\mathsf{unpacked}(v.I, \pi)$.
(b) The permission π is obtained between the states s_1 and s by acquiring a lock l and obtaining its lock invariant, the *res_inv* predicate. Then, the field $f \in \mathsf{fldResInv}(L)$, where L is the class of l. Because acquiring the lock (the call to $l.\mathsf{lock}()$ method) happens after the state s_1, the object l must be descendant of $p_1 = v$. Therefore, the field $l \in \mathsf{relFld}(C)$, where C is the class of v. Since $f \in \mathsf{fld}(I)$ and thus, $f \in \mathsf{fld}(I) \cap \mathsf{fldResInv}(L)$, this contradicts the rule **RL** (Sec. 3.2), which requires $\mathsf{fld}(I) \cap \mathsf{fldResInv}(L) = \emptyset$.
(c) The location $p_1.p_2...p_n.f$ does not exist in the state s_1 and permission π to it is obtained in a state s_2, $s_1 < s_2 < s$, by creating the object p_n. In this case, at least one reference $p_i, i = 2..n$ exists in s_1, which is assigned in a state s_3, where $s_1 < s_3 < s_2$. In the state s_3, the running thread holds write permission for p_i. Since the location $p_i \in \mathsf{fp}(v.I)$, permission to p_i is not obtained by a lock between the states s_1 and s_3 (this follows from the discussion in the previous case). Therefore, a permission to p_i must be held in the state s_1. Then, the Hoare triple (MethCall) ensures that in the prestate s_1 of the invocation of the method m_{p_2}, the running thread holds the predicate $\mathsf{unpacked}(v.I, 1)$.

Theorem 1 *(High-level data race freedom) If a value $p.f$ is in a critical state s of an unpacked segment S of an invariant $v.I$, then any thread that is neither holder nor a local thread of S can not access $p.f$.*

Proof. From the definition of a critical state (Def. 1), $p.f \in \mathsf{fp}(v.I)$ and $p.f$ is assigned by a thread t within the segment S. Let s' be the prestate of this assignment and therefore, $s' < s$. From the Hoare triple (Set), the thread t holds write permission, $\pi = 1$, for the location $p.f$ in the state s', and thus, in s' no other thread holds any read permission on $p.f$. Further, each command c within the segment S is safe, and therefore: for all states of execution between s' and s, either t keeps the permission π, or t transfers (part of) π to another thread that is local to the segment S. Within S, no part of the permission π could leak to a thread $t' \neq t$ if t' is not local to S; thus, $p.f$ is not accessible by t'.

Theorem 2 *(Use of a class invariant) An invariant $v.I$ holds in a program state in which the running thread t holds the predicate* $\mathsf{holds}(v.I, \pi)$*, $\pi > 0$.*

Proof. Let s be a program state in which t holds a predicate (token) $\mathsf{holds}(v.I, \pi)$. We assume that s_1 is the latest program state that is a prestate of a $\mathsf{pack}(v.I)$ command, for which $s_1 < s$. Existence of s_1 follows directly from Lemma 1(i). The precondition of $\mathsf{pack}(v.I)$ (see Hoare triple (Pack)) ensures that $v.I$ holds in the state s_1. We prove by contradiction that $v.I$ can not be broken in any state between s_1 and s.

Let $v.I$ be broken in a state s_2, $s_1 < s_2 < s$. Since validity of the invariant $v.I$ might be changed only by assignment operation $p.f = w$, where $p.f \in \mathsf{fp}(v.I)$, then s_2 is a poststate of $p.f = w$. From Lemma 2, we conclude that the prestate of $p.f = w$, and thus s_2 also, is an internal state of an unpacked segment of $v.I$. Then, Lemma 1.ii ensures that in s_2 no thread holds the token $\mathsf{holds}(v.I, \pi)$. Since in the state s the running thread holds $\mathsf{holds}(v.I, \pi)$, this token must have been obtained by executing the $\mathsf{pack}(v.I)$ command between s_2 and s. Let s_3 is the prestate of this $\mathsf{pack}(v.I)$ command. Then $s_1 < s_2 < s_3 < s$, which contradicts the assumption that s_1 is the latest prestate of $\mathsf{pack}(v.I)$ which preceeds s.

5 Related Work

The early work on verification of class invariants in sequential programs [17, 15] is unsound for more complex data structure, for example if an invariant captures properties over different objects. Later, Poetzsch-Heffter [22] and Huizing *et al.* [13] presented sound techniques that do not restrict the invariant definition or the program itself; however, both approaches are not modular.

Müller *et al.* [18] propose two sound techniques for modular reasoning: the *ownership technique* and the less restrictive *visibility technique*. Both concepts, as well as Lu *et al.*'s modular technique [16], are designed for ownership-based type systems. These techniques are captured in Drossopoulou *et al.*'s abstract unified framework [9]. Although it is stated that this abstract framework should be suitable to model class invariants in a concurrent setting, the framework has never been applied on a concrete verification technique for concurrent programs.

Weiß models class invariants with a boolean model field *inv* [24]. Their validity is checked only on demand. Specifications use *inv* explicitly where needed, while $\mathsf{this}.inv$ is implicitly generated in each method pre- and postcondition.

We are not aware of much work done on verification of class invariants for multithreaded programs. Comparable to our approach is Jacobs *et al.*'s technique [14] for verifying multithreaded programs with class invariants, using the *Boogie methodology* [3] for sequential programs. However, this technique allows a thread to break an invariant of an object only if it completely owns this object. Instead, with our technique, breaking a class invariant is independent of permissions on heap memory. This ensures a broader applicability of our technique.

A different approach for modular verification of object invariants in concurrent programs is proposed by Cohen [6], implemented in VCC [5]. Each object is assigned a two-state invariant expressing the required relation between any two consecutive states of execution that has to be respected by every state update in the program. Modular verification of multithreaded programs with class invariants is also supported by the static checker Calvin [10]. However, both methodologies do not allow breaking of a class invariant in the program.

6 Conclusion and Future Work

We introduced a sound and modular approach for verifying class invariants in multithreaded Java-like programs in a permission-based separation logic setting. We do, however, deviate from the standard rules in separation logic: we impose that class invariants may express properties only over state and thus, their definition is free of permission expressions. We allow a thread to explicitly break an invariant, and we ensure that no other thread can observe the invalidated object's state. Moreover, breaking and reestablishing an invariant is allowed without holding all permissions associated to the invariant. This makes our technique broadly applicable. To achieve modularity, we restrict our technique to ownership-based type systems only. The method requires simple specifications support.

For future work, we plan to integrate our technique in the VerCors tool [1], and to use it to verify data structures from the *java.util.concurrency* package. We plan to extend the concept to support class inheritance, to allow more permissive invariants with model methods and/or abstract predicates, to allow more fine-grained permission handling. as well as to support *history constraints*.

Acknowledgments We thank Christian Haack and Stefan Blom for their useful feedback. This work was supported by ERC grant 258405 for the VerCors project.

References

1. A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors project: setting up basecamp. In *PLPV*, pages 71–82, 2012.
2. C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4):207–227, 2003.
3. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.
5. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.
6. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.
7. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
8. W. Dietl and P. Müller. Object ownership in program verification. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming*, LNCS. Springer-Verlag, 2012.

9. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *Types, Logics and Semantics for State*, 2008.
10. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.
11. C. Haack, M. Huisman, C. Hurlin, and A.Amighi. Permission-based separation logic for Java, 201x. Conditionally accepted for LMCS.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
13. K. Huizing and R. Kuiper. Verification of object oriented programs using class invariants. In *FASE*, pages 208–221, 2000.
14. B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM*, pages 137–147, 2005.
15. B. Liskov and J. Guttag. *Abstraction and specification in program development.* MIT Press, Cambridge, MA, USA, 1986.
16. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *ECOOP*, pages 202–226, 2007.
17. B. Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.
18. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.
19. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
20. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *Principles of programming languages (POPL '08)*, pages 75–86. ACM, 2008.
21. M. J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, Nov. 2005.
22. A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* PhD thesis, Habilitation thesis, Technical University of Munich, 1997.
23. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on LICS 2002*, pages 55–74. IEEE Computer Society, 2002.
24. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, 2011.